# Contents

1 GLib Overview .......................... 1
   1.1 Compiling the GLib package .................. 1
   1.2 Cross-compiling the GLib package ............ 4
   1.3 Compiling GLib Applications .................. 5
   1.4 Running GLib Applications .................... 6
   1.5 Changes to GLib .......................... 8
   1.6 Regular expression syntax ................... 9
   1.7 Mailing lists and bug reports ............... 30

2 GLib Fundamentals ....................... 33
   2.1 Version Information ....................... 33
   2.2 Basic Types ................................ 35
   2.3 Limits of Basic Types ........................ 40
   2.4 Standard Macros .......................... 44
   2.5 Type Conversion Macros .................... 48
   2.6 Byte Order Macros ......................... 50
   2.7 Numerical Definitions ..................... 60
   2.8 Miscellaneous Macros ...................... 63
   2.9 Atomic Operations ........................ 73

3 GLib Core Application Support ........... 77
   3.1 The Main Event Loop ....................... 77
   3.2 Threads .................................. 104
   3.3 Thread Pools ............................ 131
   3.4 Asynchronous Queues ....................... 136
   3.5 Dynamic Loading of Modules ................. 143
   3.6 Memory Allocation ......................... 147
   3.7 IO Channels ................................ 154
   3.8 Error Reporting ........................... 171
   3.9 Message Output and Debugging Functions .... 178
   3.10 Message Logging ......................... 182

4 GLib Utilities .......................... 189
   4.1 String Utility Functions .................... 189
   4.2 Character Set Conversion .................... 213
   4.3 Unicode Manipulation ....................... 223
   4.4 Base64 Encoding .......................... 248
   4.5 Data Checksums ............................ 251
   4.6 Internationalization ....................... 255
   4.7 Date and Time Functions .................... 260
   4.8 Random Numbers ........................... 276
   4.9 Hook Functions ............................ 281
   4.10 Miscellaneous Utility Functions .......... 291
   4.11 Lexical Scanner .......................... 304
   4.12 Automatic String Completion ............... 317
   4.13 Timers .................................. 320
   4.14 Spawning Processes ....................... 322
   4.15 File Utilities ............................ 322
   4.16 URI Functions ............................ 343
   4.17 Shell-related Utilities .................... 346
   4.18 Commandline option parser .................. 347
   4.19 Glob-style pattern matching ............... 361
   4.20 Perl-compatible regular expressions ........ 364
   4.21 Simple XML Subset Parser ................. 386
List of Figures

3 GLib Core Application Support
   3.1 States of a Main Context .................................................. 81

4 GLib Utilities
   4.1 Conversion between File Name Encodings ................................. 215
List of Tables

1 GLib Overview
   1.1 Metacharacters outside square brackets ........................................... 10
   1.2 Metacharacters inside square brackets ............................................. 10
   1.3 Non-printing characters ................................................................. 10
   1.4 Non-printing characters ................................................................. 11
   1.5 Generic characters ........................................................................... 12
   1.6 Generic character types ..................................................................... 12
   1.7 Property codes .................................................................................... 15
   1.8 Simple assertions ............................................................................... 16
   1.9 Posix classes ....................................................................................... 19
   1.10 Option settings .................................................................................. 19
   1.11 Abbreviations for quantifiers ............................................................. 21
Chapter 1

GLib Overview

GLib is a general-purpose utility library, which provides many useful data types, macros, type conversions, string utilities, file utilities, a main loop abstraction, and so on. It works on many UNIX-like platforms, Windows, OS/2 and BeOS. GLib is released under the GNU Library General Public License (GNU LGPL).

The general policy of GLib is that all functions are invisibly threadsafe with the exception of data structure manipulation functions, where, if you have two threads manipulating the same data structure, they must use a lock to synchronize their operation.

1.1 Compiling the GLib package

Name

Compiling the GLib Package – How to compile GLib itself

Building the Library on UNIX

On UNIX, GLib uses the standard GNU build system, using autoconf for package configuration and resolving portability issues, automake for building makefiles that comply with the GNU Coding Standards, and libtool for building shared libraries on multiple platforms. The normal sequence for compiling and installing the GLib library is thus:

```
./configure
make
make install
```

The standard options provided by GNU autoconf may be passed to the configure script. Please see the autoconf documentation or run `./configure --help` for information about the standard options.

The GTK+ documentation contains further details about the build process and ways to influence it.

Dependencies

Before you can compile the GLib library, you need to have various other tools and libraries installed on your system. The two tools needed during the build process (as differentiated from the tools used in when creating GLib mentioned above such as autoconf) are `pkg-config` and GNU make.

- `pkg-config` is a tool for tracking the compilation flags needed for libraries that are used by the GLib library. (For each library, a small `pc` text file is installed in a standard location that contains the compilation flags needed for that library along with version number information.) The version of `pkg-config` needed to build GLib is mirrored in the `dependencies` directory on the GTK+ FTP site.

- The GTK+ makefiles will mostly work with different versions of `make`, however, there tends to be a few incompatibilities, so the GTK+ team recommends installing GNU `make` if you don’t already have it on your system and using it. (It may be called `gmake` rather than `make`.)
GLib depends on a number of other libraries.

- The **GNU libiconv library** is needed to build GLib if your system doesn’t have the `iconv()` function for doing conversion between character encodings. Most modern systems should have `iconv()`, however many older systems lack an `iconv()` implementation. On such systems, you must install the libiconv library. This can be found at: [http://www.gnu.org/software/libiconv](http://www.gnu.org/software/libiconv).

  If your system has an `iconv()` implementation but you want to use libiconv instead, you can pass the `--with-libiconv` option to configure. This forces libiconv to be used.

  Note that if you have libiconv installed in your default include search path (for instance, in `/usr/local/`), but don’t enable it, you will get an error while compiling GLib because the `iconv.h` that libiconv installs hides the system `iconv`.

  If you are using the native `iconv` implementation on Solaris instead of libiconv, you’ll need to make sure that you have the converters between locale encodings and UTF-8 installed. At a minimum you’ll need the SUNWuiu8 package. You probably should also install the SUNWciu8, SUNWhiu8, SUNWju8, and SUNWku8 packages.

- The **libintl library from the GNU gettext package** is needed if your system doesn’t have the `gettext()` functionality for handling message translation databases.

- A thread implementation is needed, unless you want to compile GLib without thread support, which is not recommended. The thread support in GLib can be based upon several native thread implementations, e.g. POSIX threads, DCE threads or Solaris threads.

- GRegex uses the **PCRE library** for regular expression matching. The default is to use the internal version of PCRE that is patched to use GLib for memory management and Unicode handling. If you prefer to use the system-supplied PCRE library you can pass the `--with-pcre=system` option to configure, but it is not recommended.

- The optional extended attribute support in GIO requires the `getxattr()` family of functions that may be provided by glibc or by the standalone libattr library. To build GLib without extended attribute support, use the `--disable-xattr` configure option.

- The optional SELinux support in GIO requires libselinux. To build GLib without SELinux support, use the `--disable-selinux` configure option.

### Extra Configuration Options

In addition to the normal options, the `configure` script in the GLib library supports these additional arguments:

```
```

- **--enable-debug** Turns on various amounts of debugging support. Setting this to ‘no’ disables `g_assert()`, `g_return_if_fail()`, `g_return_val_if_fail()` and all cast checks between different object types. Setting it to ‘minimum’ disables only cast checks. Setting it to ‘yes’ enables runtime debugging. The default is ‘minimum’. Note that ‘no’ is fast, but dangerous as it tends to destabilize even mostly bug-free software by changing the effect of many bugs from simple warnings into fatal crashes. Thus `--enable-debug=no` should not be used for stable releases of GLib.

- **--disable-gc-friendly and --enable-gc-friendly** By default, and with `--disable-gc-friendly` as well, Glib does not clear the memory for certain objects before they are freed. For example, Glib may decide to recycle GList nodes by putting them in a free list. However, memory profiling and debugging tools like Valgrind work better if an application does not keep dangling pointers to freed memory (even though these pointers are no longer dereferenced), or invalid pointers inside uninitialized memory. The `--enable-gc-friendly` option makes Glib clear memory in these situations.
When shrinking a GArray, Glib will clear the memory no longer available in the array: shrink an array from 10 bytes to 7, and the last 3 bytes will be cleared. This includes removals of single and multiple elements.

When growing a GArray, Glib will clear the new chunk of memory. Grow an array from 7 bytes to 10 bytes, and the last 3 bytes will be cleared.

The above applies to GPtrArray as well.

When freeing a node from a GHashTable, Glib will first clear the node, which used to have pointers to the key and the value stored at that node.

When destroying or removing a GTree node, Glib will clear the node, which used to have pointers to the node’s value, and the left and right subnodes.

Since clearing the memory has a cost, --disable-gc-friendly is the default.

--disable-mem-pools and --enable-mem-pools Many small chunks of memory are often allocated via collective pools in GLib and are cached after release to speed up reallocations. For sparse memory systems this behaviour is often inferior, so memory pools can be disabled to avoid excessive caching and force atomic maintenance of chunks through the g_malloc() and g_free() functions. Code currently affected by this:

- GList, GSList, GNode, GHash allocations. The functions g_list_push_allocator(), g_list_pop_allocator(), g_slist_push_allocator(), g_slist_pop_allocator(), g_node_push_allocator() and g_node_pop_allocator() are not available
- GMemChunks become basically non-effective
- GSignal disables all caching (potentially very slow)
- GType doesn’t honour the GTypeInfo n_preallocs field anymore
- the GBSearchArray flag G_BSEARCH_ALIGN_POWER2 becomes non-functional

--disable-threads and --enable-threads Do not compile GLib to be multi thread safe. GLib will be slightly faster then. This is however not recommended, as many programs rely on GLib being multi thread safe.

--with-threads Specify a thread implementation to use.
- ‘posix’ and ‘dce’ can be used interchangeable to mean the different versions of Posix threads. configure tries to find out, which one is installed.
- ‘none’ means that GLib will be thread safe, but does not have a default thread implementation. This has to be supplied to g_thread_init() by the programmer.

--disable-regex and --enable-regex Do not compile GLib with regular expression support. GLib will be smaller because it will not need the PCRE library. This is however not recommended, as programs may need GRegex.

--with-pcre Specify whether to use the internal or the system-supplied PCRE library.
- ‘internal’ means that GRegex will be compiled to use the internal PCRE library.
- ‘system’ means that GRegex will be compiled to use the system-supplied PCRE library.

Using the internal PCRE is the preferred solution:

- System-supplied PCRE has a separated copy of the big tables used for Unicode handling.
- Some systems have PCRE libraries compiled without some needed features, such as UTF-8 and Unicode support.
- PCRE uses some global variables for memory management and other features. In the rare case of a program using both GRegex and PCRE (maybe indirectly through a library), this variables could lead to problems when they are modified.
--disable-included-printf and --enable-included-printf By default the configure script will try to auto-detect whether the C library provides a suitable set of printf() functions. In detail, configure checks that the semantics of snprintf() are as specified by C99 and that positional parameters as specified in the Single Unix Specification are supported. If this not the case, GLib will include an implementation of the printf() family. These options can be used to explicitly control whether an implementation fo the printf() family should be included or not.

--disable-visibility and --enable-visibility By default, GLib uses ELF visibility attributes to optimize PLT table entries if the compiler supports ELF visibility attributes. A side-effect of the way in which this is currently implemented is that any header change forces a full recompilation, and missing includes may go unnoticed. Therefore, it makes sense to turn this feature off while doing GLib development, even if the compiler supports ELF visibility attributes. The --disable-visibility option allows to do that.

--disable-gtk-doc and --enable-gtk-doc By default the configure script will try to auto-detect whether the gtk-doc package is installed. If it is, then it will use it to extract and build the documentation for the GLib library. These options can be used to explicitly control whether gtk-doc should be used or not. If it is not used, the distributed, pre-generated HTML files will be installed instead of building them on your machine.

--disable-man and --enable-man By default the configure script will try to auto-detect whether xslt-proc and the necessary Docbook stylesheets are installed. If they are, then it will use them to rebuild the included man pages from the XML sources. These options can be used to explicitly control whether man pages should be rebuilt used or not. The distribution includes pre-generated man pages.

--disable-xattr and --enable-xattr By default the configure script will try to auto-detect whether the getxattr() family of functions is available. If it is, then extended attribute support will be included in GIO. These options can be used to explicitly control whether extended attribute support should be included or not. getxattr() and friends can be provided by glibc or by the standalone libattr library.

--disable-selinux and --enable-selinux By default the configure script will auto-detect if libselinux is available and include SELinux support in GIO if it is. These options can be used to explicitly control whether SELinux support should be included.

1.2 Cross-compiling the GLib package

Name

Cross-compiling the GLib Package – How to cross-compile GLib

Building the Library for a different architecture

Cross-compilation is the process of compiling a program or library on a different architecture or operating system then it will be run upon. GLib is slightly more difficult to cross-compile than many packages because much of GLib is about hiding differences between different systems.

These notes cover things specific to cross-compiling GLib; for general information about cross-compilation, see the autoconf info pages.

GLib tries to detect as much information as possible about the target system by compiling and linking programs without actually running anything; however, some information GLib needs is not available this way. This information needs to be provided to the configure script via a "cache file" or by setting the cache variables in your environment.

As an example of using a cache file, to cross compile for the "MingW32" Win32 runtime environment on a Linux system, create a file ‘win32.cache’ with the following contents:

```
glib_cv_long_long_format=I64

glib_cv_stack_grows=no
```

Then execute the following commands:

```
PATH=/path/to/mingw32-compiler/bin:$PATH
chmod a-w win32.cache  # prevent configure from changing it
./configure --cache-file=win32.cache --host=mingw32
```

The complete list of cache file variables follows. Most of these won’t need to be set in most cases.
Cache file variables

`glib_cv_long_long_format=\[ll/q/I64\]` Format used by `printf()` and `scanf()` for 64 bit integers. "ll" is the C99 standard, and what is used by the 'trio' library that GLib builds if your `printf()` is insufficiently capable. Doesn’t need to be set if you are compiling using trio.

`glib_cv_stack_grows=\[yes/no\]` Whether the stack grows up or down. Most places will want "no", A few architectures, such as PA-RISC need "yes".

`glib_cv_working_bcopy=\[yes/no\]` Whether your `bcopy()` can handle overlapping copies. Only needs to be set if you don’t have `memmove()`. (Very unlikely)

`glib_cv_sane_realloc=\[yes/no\]` Whether your `realloc()` conforms to ANSI C and can handle NULL as the first argument. Defaults to "yes" and probably doesn’t need to be set.

`glib_cv_have_strlcpy=\[yes/no\]` Whether you have `strlcpy()` that matches OpenBSD. Defaults to "no", which is safe, since GLib uses a built-in version in that case.

`glib_cv_va_val_copy=\[yes/no\]` Whether va_list can be copied as a pointer. If set to "no", then `memcpy()` will be used. Only matters if you don’t have `va_copy()` or `__va_copy()`. (So, doesn’t matter for GCC.) Defaults to "yes" which is slightly more common than "no".

`glib_cv_rtldglocal_broken=\[yes/no\]` Whether you have a bug found in OSF/1 v5.0. Defaults to "no".

`glib_cv_uscore=\[yes/no\]` Whether an underscore needs to be prepended to symbols when looking them up via `dlsym()`. Only needs to be set if your system uses `dlopen()`/`dlsym()`.

`ac_cv_funcposix_getpwuid_r=\[yes/no\]` Whether you have a getpwuid_r function (in your C library, not your thread library) that conforms to the POSIX spec. (Takes a 'struct passwd **' as the final argument)

`ac_cv_func_nonposix_getpwuid_r=\[yes/no\]` Whether you have some variant of `getpwuid_r()` that doesn’t conform to to the POSIX spec, but GLib might be able to use (or might segfault.) Only needs to be set if `ac_cv_funcposix_getpwuid_r` is not set. It’s safest to set this to "no".

`ac_cv_funcposix_getgrgid_r=\[yes/no\]` Whether you have a getgrgid_r function that conforms to the POSIX spec.

`glib_cv_use_pid_surrogate=\[yes/no\]` Whether to use a `setpriority()` on the PID of the thread as a method for setting the priority of threads. This only needs to be set when using POSIX threads.

`ac_cv_func_printf_unix98=\[yes/no\]` Whether your `printf()` family supports Unix98 style `%N$` positional parameters. Defaults to "no".

`ac_cv_func_vsnprintf_c99=\[yes/no\]` Whether you have a `vsnprintf()` with C99 semantics. (C99 semantics means returning the number of bytes that would have been written had the output buffer had enough space.) Defaults to "no".

## 1.3 Compiling GLib Applications

### Name

Compiling GLib Applications – How to compile your GLib application

### Compiling GLib Applications on UNIX

To compile a GLib application, you need to tell the compiler where to find the GLib header files and libraries. This is done with the pkg-config utility.

The following interactive shell session demonstrates how pkg-config is used (the actual output on your system may be different):

```bash
$ pkg-config --cflags glib-2.0
-I/usr/include/glib-2.0 -I/usr/lib/glib-2.0/include
$ pkg-config --libs glib-2.0
-L/usr/lib -lm -lglibus-2.0
```

If your application uses threads or GObject features, it must be compiled and linked with the options returned by the following pkg-config invocations:

```bash
$ pkg-config --cflags --libs gthread-2.0
$ pkg-config --cflags --libs gobject-2.0
```
CHAPTER 1. GLIB OVERVIEW

1.4. RUNNING GLIB APPLICATIONS

If your application uses modules, it must be compiled and linked with the options returned by one of the following pkg-config invocations:

```
$ pkg-config --cflags --libs gmodule-no-export-2.0
$ pkg-config --cflags --libs gmodule-2.0
```

The difference between the two is that gmodule-2.0 adds `--export-dynamic` to the linker flags, which is often not needed.

The simplest way to compile a program is to use the "backticks" feature of the shell. If you enclose a command in backticks (not single quotes), then its output will be substituted into the command line before execution. So to compile a GLib Hello, World, you would type the following:

```
$ cc `pkg-config --cflags --libs glib-2.0` hello.c -o hello
```

If you want to make sure that your program doesn’t use any deprecated functions, you can define the preprocessor symbol `G_DISABLE_DEPRECATED` by using the command line option `-DG_DISABLE_DEPRECATED=1`.

The recommended way of using GLib has always been to only include the toplevel headers `glib.h`, `glib-object.h`, `gio.h`. Starting with 2.17, GLib enforces this by generating an error when individual headers are directly included. To help with the transition, the enforcement is not turned on by default for GLib headers (it is turned on for GObject and GIO). To turn it on, define the preprocessor symbol `G_DISABLE_SINGLE_INCLUDES` by using the command line option `-DG_DISABLE_SINGLE_INCLUDES`.

1.4 Running GLib Applications

Name

Running GLib Applications – How to run and debug your GLib application

Running and debugging GLib Applications

Environment variables

GLib inspects a few of environment variables in addition to standard variables like `LANG`, `PATH` or `HOME`.

`G_FILENAME_ENCODING`
This environment variable can be set to a comma-separated list of character set names. GLib assumes that filenames are encoded in the first character set from that list rather than in UTF-8. The special token `"@locale"` can be used to specify the character set for the current locale.

`G_BROKEN_FILENAMES`
If this environment variable is set, GLib assumes that filenames are in the locale encoding rather than in UTF-8. `G_FILENAME_ENCODING` takes priority over `G_BROKEN_FILENAMES`.

`G_MESSAGES_PREFIXED`
A list of log levels for which messages should be prefixed by the program name and PID of the application. The default is to prefix everything except `G_LOG_LEVEL_MESSAGE` and `G_LOG_LEVEL_INFO`.

`G_DEBUG`
If GLib has been configured with `--enable-debug=yes`, this variable can be set to a list of debug options, which cause GLib to print out different types of debugging information.

`fatal_warnings` Causes GLib to abort the program at the first call to `g_warning()` or `g_critical()`. This option is special in that it doesn’t require GLib to be configured with debugging support.

`fatal_criticals` Causes GLib to abort the program at the first call to `g_critical()`. This option is special in that it doesn’t require GLib to be configured with debugging support.

`gc-friendly` Newly allocated memory that isn’t directly initialized, as well as memory being freed will be reset to 0. The point here is to allow memory checkers and similar programs that use bohem GC alike algorithms to produce more accurate results. This option is special in that it doesn’t require GLib to be configured with debugging support.
resident-modules All modules loaded by GModule will be made resident. This can be useful for tracking memory leaks in modules which are later unloaded; but it can also hide bugs where code is accessed after the module would have normally been unloaded. This option is special in that it doesn’t require GLib to be configured with debugging support.

bind-now-modules All modules loaded by GModule will bind their symbols at load time, even when the code uses %G_MODULE_BIND_LAZY. This option is special in that it doesn’t require GLib to be configured with debugging support.

The special value all can be used to turn on all debug options. The special value help can be used to print all available options.

G_SLICE
This environment variable allows reconfiguration of the GSlice memory allocator.

always-malloc This will cause all slices allocated through g_slice_alloc() and released by g_slice_free1() to be actually allocated via direct calls to g_malloc() and g_free(). This is most useful for memory checkers and similar programs that use Bohem GC alike algorithms to produce more accurate results. It can also be in conjunction with debugging features of the system’s malloc implementation such as glibc’s MALLOC_CHECK_=2 to debug erroneous slice allocation code, allthough debug-blocks usually is a better suited debugging tool.

default-blocks Using this option (present since GLib-2.13) engages extra code which performs sanity checks on the released memory slices. Invalid slice adresses or slice sizes will be reported and lead to a program halt. This option is for debugging scenarios. In particular, client packages sporting their own test suite should always enable this option when running tests. Global slice validation is ensured by storing size and address information for each allocated chunk, and maintaining a global hash table of that data. That way, multi-thread scalability is given up, and memory consumption is increased. However, the resulting code usually performs acceptably well, possibly better than with comparable memory checking carried out using external tools. An example of a memory corruption scenario that cannot be reproduced with G_SLICE=always-malloc, but will be caught by G_SLICE=debug-blocks is as follows:

```c
void *slist = g_slist_alloc(); /* void* gives up type-safety */
g_list_free (slist); /* corruption: sizeof (GSList) != ←
sizeof (GList) */
```

The special value all can be used to turn on all options. The special value help can be used to print all available options.

G_RANDOM_VERSION
If this environment variable is set to ‘2.0’, the outdated pseudo-random number seeding and generation algorithms from GLib-2.0 are used instead of the new better ones. Use the GLib-2.0 algorithms only if you have sequences of numbers generated with Glib-2.0 that you need to reproduce exactly.

LIBCHARSET_ALIAS_DIR
Allows to specify a nonstandard location for the charset.aliases file that is used by the character set conversion routines. The default location is the libdir specified at compilation time.

Locale
A number of interfaces in GLib depend on the current locale in which an application is running. Therefore, most GLib-using applications should call setlocale (LC_ALL, "") to set up the current locale.

On Windows, in a C program there are several locale concepts that not necessarily are synchronized. On one hand, there is the system default ANSI code-page, which determines what encoding is used for file names handled by the C library’s functions and the Win32 API. (We are talking about the “narrow” functions here that take character pointers, not the “wide” ones.)

On the other hand, there is the C library’s current locale. The character set (code-page) used by that is not necessarily the same as the system default ANSI code-page. Strings in this character set are returned by functions like strftime().
Traps and traces

Some code portions contain trap variables that can be set during debugging time if GLib has been configured with `--enable-debug=yes`. Such traps lead to immediate code halts to examine the current program state and backtrace.

Currently, the following trap variables exist:

```c
static volatile gulong g_trap_free_size;
static volatile gulong g_trap_realloc_size;
static volatile gulong g_trap_malloc_size;
```

If set to a size > 0, `g_free()`, `g_realloc()` and `g_malloc()` will be intercepted if the size matches the size of the corresponding memory block. This will only work with `g_mem_set_vtable (glib_mem_profiler_table)` upon startup though, because memory profiling is required to match on the memory block sizes.

Note that many modern debuggers support conditional breakpoints, which achieve pretty much the same. E.g. in gdb, you can do

```gdb
break g_malloc
condition 1 n_bytes == 20
```

to break only on `g_malloc()` calls where the size of the allocated memory block is 20.

Memory statistics

`g_mem_profile()` will output a summary `g_malloc()` memory usage, if memory profiling has been enabled by calling `g_mem_set_vtable (glib_mem_profiler_table)` upon startup.

If GLib has been configured with `--enable-debug=yes`, then `g_slice_debug_tree_statistics()` can be called in a debugger to output details about the memory usage of the slice allocator.

1.5 Changes to GLib

Name

Changes to GLib – Incompatible changes made between succeeding versions of GLib

Incompatible changes from 2.0 to 2.2

- GLib changed the seeding algorithm for the pseudo-random number generator Mersenne Twister, as used by GRand and GRandom. This was necessary, because some seeds would yield very bad pseudo-random streams. Also the pseudo-random integers generated by `g_rand*_int_range()` will have a slightly better equal distribution with the new version of GLib.

  Further information can be found at the website of the Mersenne Twister random number generator at [http://www.math.keio.ac.jp/~matumoto/emt.html](http://www.math.keio.ac.jp/~matumoto/emt.html).

  The original seeding and generation algorithms, as found in GLib 2.0.x, can be used instead of the new ones by setting the environment variable `G_RANDOM_VERSION` to the value of '2.0'. Use the GLib-2.0 algorithms only if you have sequences of numbers generated with Glib-2.0 that you need to reproduce exactly.

Incompatible changes from 1.2 to 2.0

The GNOME 2.0 porting guide on [http://developer.gnome.org](http://developer.gnome.org) has some more detailed discussion of porting from 1.2 to 2.0. See the section on GLib.

- The event loop functionality GMain has extensively been revised to support multiple separate main loops in separate threads. All sources (timeouts, idle functions, etc.) are associated with a GMainContext.

  Compatibility functions exist so that most application code dealing with the main loop will continue to work. However, code that creates new custom types of sources will require modification.

  The main changes here are:
Sources are now exposed as GSource *, rather than simply as numeric ids.

New types of sources are created by structure “derivation” from GSource, so the source_data parameter to the GSource virtual functions has been replaced with a GSource *.

Sources are first created, then later added to a specific GMainContext.

Dispatching has been modified so both the callback and data are passed in to the dispatch() virtual function.

To go along with this change, the vtable for GIOChannel has changed and add_watch() has been replaced by create_watch().

- g_list_foreach() and g_slist_foreach() have been changed so they are now safe against removal of the current item, not the next item.

It’s not recommended to mutate the list in the callback to these functions in any case.

- GDate now works in UTF-8, not in the current locale. If you want to use it with the encoding of the locale, you need to convert strings using g_locale_to_utf8() first.

- g_strsplit() has been fixed to:
  - include trailing empty tokens, rather than stripping them
  - split into a maximum of max_tokens tokens, rather than max_tokens + 1

Code depending on either of these bugs will need to be fixed.

- Deprecated functions that got removed: g_set_error_handler(), g_set_warning_handler(), g_set_message_handler(), use g_log_set_handler() instead.

### 1.6 Regular expression syntax

#### Name

Regular expression syntax – Syntax and semantics of the regular expressions supported by GRegex

#### GRegex regular expression details

A regular expression is a pattern that is matched against a string from left to right. Most characters stand for themselves in a pattern, and match the corresponding characters in the string. As a trivial example, the pattern

```
The quick brown fox
```

matches a portion of a string that is identical to itself. When caseless matching is specified (the G_REGEX_CASELESS flag), letters are matched independently of case.

The power of regular expressions comes from the ability to include alternatives and repetitions in the pattern. These are encoded in the pattern by the use of metacharacters, which do not stand for themselves but instead are interpreted in some special way.

There are two different sets of metacharacters: those that are recognized anywhere in the pattern except within square brackets, and those that are recognized in square brackets. Outside square brackets, the metacharacters are as follows:

Part of a pattern that is in square brackets is called a "character class". In a character class the only metacharacters are:

#### Backslash

The backslash character has several uses. Firstly, if it is followed by a non-alphanumeric character, it takes away any special meaning that character may have. This use of backslash as an escape character applies both inside and outside character classes.

For example, if you want to match a * character, you write \* in the pattern. This escaping action applies whether or not the following character would otherwise be interpreted as a metacharacter, so
CHAPTER 1. GLIB OVERVIEW

1.6. REGULAR EXPRESSION SYNTAX

Table 1.1 Metacharacters outside square brackets

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\</td>
<td>general escape character with several uses</td>
</tr>
<tr>
<td>*</td>
<td>assert start of string (or line, in multiline mode)</td>
</tr>
<tr>
<td>+</td>
<td>assert end of string (or line, in multiline mode)</td>
</tr>
<tr>
<td>-</td>
<td>match any character except newline (by default)</td>
</tr>
<tr>
<td>[</td>
<td>start character class definition</td>
</tr>
<tr>
<td>\</td>
<td>start of alternative branch</td>
</tr>
<tr>
<td>(</td>
<td>start subpattern</td>
</tr>
<tr>
<td>)</td>
<td>end subpattern</td>
</tr>
<tr>
<td>?</td>
<td>extends the meaning of (, or 0/1 quantifier, or quantifier minimizer</td>
</tr>
<tr>
<td>*</td>
<td>0 or more quantifier</td>
</tr>
<tr>
<td>+</td>
<td>1 or more quantifier, also &quot;possessive quantifier&quot;</td>
</tr>
<tr>
<td>{</td>
<td>start min/max quantifier</td>
</tr>
</tbody>
</table>

Table 1.2 Metacharacters inside square brackets

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\</td>
<td>general escape character</td>
</tr>
<tr>
<td>-</td>
<td>negate the class, but only if the first character</td>
</tr>
<tr>
<td>[</td>
<td>indicates character range</td>
</tr>
<tr>
<td>\</td>
<td>POSIX character class (only if followed by POSIX syntax)</td>
</tr>
<tr>
<td>]</td>
<td>terminates the character class</td>
</tr>
</tbody>
</table>

it is always safe to precede a non-alphanumeric with backslash to specify that it stands for itself. In particular, if you want to match a backslash, you write \\\. If a pattern is compiled with the G_REGEX_EXTENDED option, whitespace in the pattern (other than in a character class) and characters between a # outside a character class and the next newline are ignored. An escaping backslash can be used to include a whitespace or # character as part of the pattern. If you want to remove the special meaning from a sequence of characters, you can do so by putting them between \Q and \E. The \Q...\E sequence is recognized both inside and outside character classes.

Non-printing characters

A second use of backslash provides a way of encoding non-printing characters in patterns in a visible manner. There is no restriction on the appearance of non-printing characters, apart from the binary zero that terminates a pattern, but when a pattern is being prepared by text editing, it is usually easier to use one of the following escape sequences than the binary character it represents:

Table 1.3 Non-printing characters

<table>
<thead>
<tr>
<th>Escape</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\a</td>
<td>alarm, that is, the BEL character (hex 07)</td>
</tr>
<tr>
<td>\cx</td>
<td>&quot;control-x&quot;, where x is any character</td>
</tr>
<tr>
<td>\e</td>
<td>escape (hex 1B)</td>
</tr>
<tr>
<td>\f</td>
<td>formfeed (hex 0C)</td>
</tr>
<tr>
<td>\n</td>
<td>newline (hex 0A)</td>
</tr>
<tr>
<td>\r</td>
<td>carriage return (hex 0D)</td>
</tr>
<tr>
<td>\t</td>
<td>tab (hex 09)</td>
</tr>
<tr>
<td>\ddd</td>
<td>character with octal code ddd, or backreference</td>
</tr>
<tr>
<td>\xhh</td>
<td>character with hex code hh</td>
</tr>
<tr>
<td>\x{hhh..}</td>
<td>character with hex code hhh..</td>
</tr>
</tbody>
</table>

The precise effect of \cx is as follows: if x is a lower case letter, it is converted to upper case. Then
bit 6 of the character (hex 40) is inverted. Thus \cz becomes hex 1A, but \c{l} becomes hex 3B, while \c{; becomes hex 7B.

After \x, from zero to two hexadecimal digits are read (letters can be in upper or lower case). Any number of hexadecimal digits may appear between \x{ and }, but the value of the character code must be less than 2**31 (that is, the maximum hexadecimal value is 7FFFFFFF). If characters other than hexadecimal digits appear between \x{ and }, or if there is no terminating }, this form of escape is not recognized. Instead, the initial \x will be interpreted as a basic hexadecimal escape, with no following digits, giving a character whose value is zero.

Characters whose value is less than 256 can be defined by either of the two syntaxes for \x. There is no difference in the way they are handled. For example, \x{dc} is exactly the same as \x\{dc\}.

After \0 up to two further octal digits are read. If there are fewer than two digits, just those that are present are used. Thus the sequence \0\x\07 specifies two binary zeros followed by a BEL character (code value 7). Make sure you supply two digits after the initial zero if the pattern character that follows is itself an octal digit.

The handling of a backslash followed by a digit other than 0 is complicated. Outside a character class, GRegex reads it and any following digits as a decimal number. If the number is less than 10, or if there have been at least that many previous capturing left parentheses in the expression, the entire sequence is taken as a back reference. A description of how this works is given later, following the discussion of parenthesized subpatterns.

Inside a character class, or if the decimal number is greater than 9 and there have not been that many capturing subpatterns, GRegex re-reads up to three octal digits following the backslash, and uses them to generate a data character. Any subsequent digits stand for themselves. For example:

<table>
<thead>
<tr>
<th>Escape</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\040</td>
<td>is another way of writing a space</td>
</tr>
<tr>
<td>\40</td>
<td>is the same, provided there are fewer than 40 previous capturing subpatterns</td>
</tr>
<tr>
<td>\7</td>
<td>is always a back reference</td>
</tr>
<tr>
<td>\11</td>
<td>might be a back reference, or another way of writing a tab</td>
</tr>
<tr>
<td>\011</td>
<td>is always a tab</td>
</tr>
<tr>
<td>\0113</td>
<td>is a tab followed by the character &quot;3&quot;</td>
</tr>
<tr>
<td>\113</td>
<td>might be a back reference, otherwise the character with octal code 113</td>
</tr>
<tr>
<td>\377</td>
<td>might be a back reference, otherwise the byte consisting entirely of 1 bits</td>
</tr>
<tr>
<td>\81</td>
<td>is either a back reference, or a binary zero followed by the two characters &quot;8&quot; and &quot;1&quot;</td>
</tr>
</tbody>
</table>

Note that octal values of 100 or greater must not be introduced by a leading zero, because no more than three octal digits are ever read.

All the sequences that define a single character can be used both inside and outside character classes. In addition, inside a character class, the sequence \b is interpreted as the backspace character (hex 08), and the sequences \R and \X are interpreted as the characters "R" and "X", respectively. Outside a character class, these sequences have different meanings (see below).

**Absolute and relative back references**

The sequence \g followed by a positive or negative number, optionally enclosed in braces, is an absolute or relative back reference. Back references are discussed later, following the discussion of parenthesized subpatterns.

**Generic character types**

Another use of backslash is for specifying generic character types. The following are always recognized:

Each pair of escape sequences partitions the complete set of characters into two disjoint sets. Any given character matches one, and only one, of each pair.
Table 1.5 Generic characters

<table>
<thead>
<tr>
<th>Escape</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\d</td>
<td>any decimal digit</td>
</tr>
<tr>
<td>\D</td>
<td>any character that is not a decimal digit</td>
</tr>
<tr>
<td>\s</td>
<td>any whitespace character</td>
</tr>
<tr>
<td>\S</td>
<td>any character that is not a whitespace character</td>
</tr>
<tr>
<td>\w</td>
<td>any &quot;word&quot; character</td>
</tr>
<tr>
<td>\W</td>
<td>any &quot;non-word&quot; character</td>
</tr>
</tbody>
</table>

These character type sequences can appear both inside and outside character classes. They each match one character of the appropriate type. If the current matching point is at the end of the passed string, all of them fail, since there is no character to match.

For compatibility with Perl, \s does not match the VT character (code 11). This makes it different from the the POSIX "space" class. The \s characters are HT (9), LF (10), FF (12), CR (13), and space (32).

A "word" character is an underscore or any character less than 256 that is a letter or digit. Characters with values greater than 128 never match \d, \s, or \w, and always match \D, \S, and \W.

Newline sequences

Outside a character class, the escape sequence \R matches any Unicode newline sequence. This particular group matches either the two-character sequence CR followed by LF, or one of the single characters LF (linefeed, U+000A), VT (vertical tab, U+000B), FF (formfeed, U+000C), CR (carriage return, U+000D), NEL (next line, U+0085), LS (line separator, U+2028), or PS (paragraph separator, U+2029). The two-character sequence is treated as a single unit that cannot be split. Inside a character class, \R matches the letter "R".

Unicode character properties

To support generic character types there are three additional escape sequences, they are:

Table 1.6 Generic character types

<table>
<thead>
<tr>
<th>Escape</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\p{xx}</td>
<td>a character with the xx property</td>
</tr>
<tr>
<td>\P{xx}</td>
<td>a character without the xx property</td>
</tr>
<tr>
<td>\X</td>
<td>an extended Unicode sequence</td>
</tr>
</tbody>
</table>

The property names represented by xx above are limited to the Unicode script names, the general category properties, and "Any", which matches any character (including newline). Other properties such as "InMusicalSymbols" are not currently supported. Note that \P{Any} does not match any characters, so always causes a match failure.

Sets of Unicode characters are defined as belonging to certain scripts. A character from one of these sets can be matched using a script name. For example, \p{Greek} or \P{Han}.

Those that are not part of an identified script are lumped together as 'Common'. The current list of scripts is:

- Arabic
- Armenian
- Balinese
- Bengali
- Bopomofo
- Braille
- Buginese
• Buhid
• Canadian_Aboriginal
• Cherokee
• Common
• Coptic
• Cuneiform
• Cypriot
• Cyrillic
• Deseret
• Devanagari
• Ethiopic
• Georgian
• Glagolitic
• Gothic
• Greek
• Gujarati
• Gurmukhi
• Han
• Hangul
• Hanunoo
• Hebrew
• Hiragana
• Inherited
• Kannada
• Katakana
• Kharoshthi
• Khmer
• Lao
• Latin
• Limbu
• Linear_B
• Malayalam
• Mongolian
• Myanmar
• New_Tai_Lue
• Nko
• Ogham
• Old_Italic
• Old_Persian
• Oriya
• Osmany
• Phags_Pa
• Phoenician
• Runic
• Shavian
• Sinhala
• Syloti_Nagri
• Syriac
• Tagalog
• Tagbanwa
• Tai_Le
• Tamil
• Telugu
• Thaana
• Thai
• Tibetan
• Tifinagh
• Ugaritic
• Yi

Each character has exactly one general category property, specified by a two-letter abbreviation. For compatibility with Perl, negation can be specified by including a circumflex between the opening brace and the property name. For example, \p{Lu} is the same as \P{Lu}.

If only one letter is specified with \p or \P, it includes all the general category properties that start with that letter. In this case, in the absence of negation, the curly brackets in the escape sequence are optional; these two examples have the same effect:

\p{L}
\pL

The following general category property codes are supported:
The special property L& is also supported: it matches a character that has the Lu, Ll, or Lt property, in other words, a letter that is not classified as a modifier or "other".

The long synonyms for these properties that Perl supports (such as \ep{Letter}) are not supported by GRegex, nor is it permitted to prefix any of these properties with "Is".

No character that is in the Unicode table has the Cn (unassigned) property. Instead, this property is assumed for any code point that is not in the Unicode table.

Specifying caseless matching does not affect these escape sequences. For example, \p{Lu} always matches only upper case letters.

The \X escape matches any number of Unicode characters that form an extended Unicode sequence. \X is equivalent to
### Table 1.7 Property codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Other</td>
</tr>
<tr>
<td>Cc</td>
<td>Control</td>
</tr>
<tr>
<td>Cf</td>
<td>Format</td>
</tr>
<tr>
<td>Cn</td>
<td>Unassigned</td>
</tr>
<tr>
<td>Co</td>
<td>Private use</td>
</tr>
<tr>
<td>Cs</td>
<td>Surrogate</td>
</tr>
<tr>
<td>L</td>
<td>Letter</td>
</tr>
<tr>
<td>Li</td>
<td>Lower case letter</td>
</tr>
<tr>
<td>Lm</td>
<td>Modifier letter</td>
</tr>
<tr>
<td>Lo</td>
<td>Other letter</td>
</tr>
<tr>
<td>Lt</td>
<td>Title case letter</td>
</tr>
<tr>
<td>Lu</td>
<td>Upper case letter</td>
</tr>
<tr>
<td>M</td>
<td>Mark</td>
</tr>
<tr>
<td>Mc</td>
<td>Spacing mark</td>
</tr>
<tr>
<td>Me</td>
<td>Enclosing mark</td>
</tr>
<tr>
<td>Mn</td>
<td>Non-spacing mark</td>
</tr>
<tr>
<td>N</td>
<td>Number</td>
</tr>
<tr>
<td>Nd</td>
<td>Decimal number</td>
</tr>
<tr>
<td>Ni</td>
<td>Letter number</td>
</tr>
<tr>
<td>No</td>
<td>Other number</td>
</tr>
<tr>
<td>P</td>
<td>Punctuation</td>
</tr>
<tr>
<td>Pc</td>
<td>Connector punctuation</td>
</tr>
<tr>
<td>Pd</td>
<td>Dash punctuation</td>
</tr>
<tr>
<td>Pe</td>
<td>Close punctuation</td>
</tr>
<tr>
<td>Pf</td>
<td>Final punctuation</td>
</tr>
<tr>
<td>Pi</td>
<td>Initial punctuation</td>
</tr>
<tr>
<td>Po</td>
<td>Other punctuation</td>
</tr>
<tr>
<td>Ps</td>
<td>Open punctuation</td>
</tr>
<tr>
<td>S</td>
<td>Symbol</td>
</tr>
<tr>
<td>Sc</td>
<td>Currency symbol</td>
</tr>
<tr>
<td>Sk</td>
<td>Modifier symbol</td>
</tr>
<tr>
<td>Sm</td>
<td>Mathematical symbol</td>
</tr>
<tr>
<td>So</td>
<td>Other symbol</td>
</tr>
<tr>
<td>Z</td>
<td>Separator</td>
</tr>
<tr>
<td>Zl</td>
<td>Line separator</td>
</tr>
<tr>
<td>Zp</td>
<td>Paragraph separator</td>
</tr>
<tr>
<td>Zs</td>
<td>Space separator</td>
</tr>
</tbody>
</table>
That is, it matches a character without the "mark" property, followed by zero or more characters with the "mark" property, and treats the sequence as an atomic group (see below). Characters with the "mark" property are typically accents that affect the preceding character.

Matching characters by Unicode property is not fast, because GRegex has to search a structure that contains data for over fifteen thousand characters. That is why the traditional escape sequences such as \d and \w do not use Unicode properties.

Simple assertions

The final use of backslash is for certain simple assertions. An assertion specifies a condition that has to be met at a particular point in a match, without consuming any characters from the string. The use of subpatterns for more complicated assertions is described below. The backslashed assertions are:

Table 1.8 Simple assertions

<table>
<thead>
<tr>
<th>Escape</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\b</td>
<td>matches at a word boundary</td>
</tr>
<tr>
<td>\B</td>
<td>matches when not at a word boundary</td>
</tr>
<tr>
<td>\A</td>
<td>matches at the start of the string</td>
</tr>
<tr>
<td>\Z</td>
<td>matches at the end of the string or before a newline at the end of the string</td>
</tr>
<tr>
<td>\z</td>
<td>matches only at the end of the string</td>
</tr>
<tr>
<td>\G</td>
<td>matches at first matching position in the string</td>
</tr>
</tbody>
</table>

These assertions may not appear in character classes (but note that \b has a different meaning, namely the backspace character, inside a character class).

A word boundary is a position in the string where the current character and the previous character do not both match \w or \W (i.e. one matches \w and the other matches \W), or the start or end of the string if the first or last character matches \w, respectively.

The \A, \Z, and \z assertions differ from the traditional circumflex and dollar (described in the next section) in that they only ever match at the very start and end of the string, whatever options are set. Thus, they are independent of multiline mode. These three assertions are not affected by the G_-REGEX_MATCH_NOTBOL or G_REGEX_MATCH_NOTEOL options, which affect only the behaviour of the circumflex and dollar metacharacters. However, if the start_position argument of a matching function is non-zero, indicating that matching is to start at a point other than the beginning of the string, \A can never match. The difference between \Z and \z is that \Z matches before a newline at the end of the string as well at the very end, whereas \z matches only at the end.

The \G assertion is true only when the current matching position is at the start point of the match, as specified by the start_position argument to the matching functions. It differs from \A when the value of startoffset is non-zero.

Note, however, that the interpretation of \G, as the start of the current match, is subtly different from Perl’s, which defines it as the end of the previous match. In Perl, these can be different when the previously matched string was empty.

If all the alternatives of a pattern begin with \G, the expression is anchored to the starting match position, and the "anchored" flag is set in the compiled regular expression.

Circumflex and dollar

Outside a character class, in the default matching mode, the circumflex character is an assertion that is true only if the current matching point is at the start of the string. If the start_position argument to the matching functions is non-zero, circumflex can never match if the G_REGEX_MULTILINE option is unset. Inside a character class, circumflex has an entirely different meaning (see below).

Circumflex need not be the first character of the pattern if a number of alternatives are involved, but it should be the first thing in each alternative in which it appears if the pattern is ever to match that branch. If all possible alternatives start with a circumflex, that is, if the pattern is constrained to match only at the start of the string, it is said to be an “anchored” pattern. (There are also other constructs that can cause a pattern to be anchored.)
A dollar character is an assertion that is true only if the current matching point is at the end of the string, or immediately before a newline at the end of the string (by default). Dollar need not be the last character of the pattern if a number of alternatives are involved, but it should be the last item in any branch in which it appears. Dollar has no special meaning in a character class.

The meaning of dollar can be changed so that it matches only at the very end of the string, by setting the G_REXX_DOLLAR_ENDONLY option at compile time. This does not affect the $Z assertion.

The meanings of the circumflex and dollar characters are changed if the G_REXX_MULTILINE option is set. When this is the case, a circumflex matches immediately after internal newlines as well as at the start of the string. It does not match after a newline that ends the string. A dollar matches before any newlines in the string, as well as at the very end, when G_REXX_MULTILINE is set. When newline is specified as the two-character sequence CRLF, isolated CR and LF characters do not indicate newlines.

For example, the pattern /\abc$/ matches the string "def
abc" (where \n represents a newline) in multiline mode, but not otherwise. Consequently, patterns that are anchored in single line mode because all branches start with \A are not anchored in multiline mode, and a match for circumflex is possible when the start_position argument of a matching function is non-zero. The G_REXX_DOLLAR_ENDONLY option is ignored if G_REXX_MULTILINE is set.

Note that the sequences \A, $Z, and \z can be used to match the start and end of the string in both modes, and if all branches of a pattern start with \A it is always anchored, whether or not G_REXX_MULTILINE is set.

**Full stop (period, dot)**

Outside a character class, a dot in the pattern matches any one character in the string, including a non-printing character, but not (by default) newline. In UTF-8 a character might be more than one byte long.

When a line ending is defined as a single character, dot never matches that character; when the two-character sequence CRLF is used, dot does not match CR if it is immediately followed by LF, but otherwise it matches all characters (including isolated CRs and LFs). When any Unicode line endings are being recognized, dot does not match CR or LF or any of the other line ending characters.

If the G_REXX_DOTALL flag is set, dots match newlines as well. The handling of dot is entirely independent of the handling of circumflex and dollar, the only relationship being that they both involve newline characters. Dot has no special meaning in a character class.

The behaviour of dot with regard to newlines can be changed. If the G_REXX_DOTALL option is set, a dot matches any one character, without exception. If newline is defined as the two-character sequence CRLF, it takes two dots to match it.

The handling of dot is entirely independent of the handling of circumflex and dollar, the only relationship being that they both involve newlines. Dot has no special meaning in a character class.

**Matching a single byte**

Outside a character class, the escape sequence \C matches any one byte, both in and out of UTF-8 mode. Unlike a dot, it always matches any line ending characters. The feature is provided in Perl in order to match individual bytes in UTF-8 mode. Because it breaks up UTF-8 characters into individual bytes, what remains in the string may be a malformed UTF-8 string. For this reason, the \C escape sequence is best avoided.

GRegex does not allow \C to appear in lookbehind assertions (described below), because in UTF-8 mode this would make it impossible to calculate the length of the lookbehind.

**Square brackets and character classes**

An opening square bracket introduces a character class, terminated by a closing square bracket. A closing square bracket on its own is not special. If a closing square bracket is required as a member of the class, it should be the first data character in the class (after an initial circumflex, if present) or escaped with a backslash.

A character class matches a single character in the string. A matched character must be in the set of characters defined by the class, unless the first character in the class definition is a circumflex, in which case the string character must not be in the set defined by the class. If a circumflex is actually required as a member of the class, ensure it is not the first character, or escape it with a backslash.
For example, the character class [aeiou] matches any lower case vowel, while [ˆaeiou] matches any character that is not a lower case vowel. Note that a circumflex is just a convenient notation for specifying the characters that are in the class by enumerating those that are not. A class that starts with a circumflex is not an assertion: it still consumes a character from the string, and therefore it fails if the current pointer is at the end of the string.

In UTF-8 mode, characters with values greater than 255 can be included in a class as a literal string of bytes, or by using the \x{ escaping mechanism.

When caseless matching is set, any letters in a class represent both their upper case and lower case versions, so for example, a caseless [aeiou] matches "A" as well as "a", and a caseless [ˆaeiou] does not match "A", whereas a caseful version would.

Characters that might indicate line breaks are never treated in any special way when matching character classes, whatever line-ending sequence is in use, and whatever setting of the G_REGEX_DOTALL and G_REGEX_MULTILINE options is used. A class such as [ˆa] always matches one of these characters.

The minus (hyphen) character can be used to specify a range of characters in a character class. For example, [d-m] matches any letter between d and m, inclusive. If a minus character is required in a class, it must be escaped with a backslash or appear in a position where it cannot be interpreted as indicating a range, typically as the first or last character in the class.

It is not possible to have the literal character "]" as the end character of a range. A pattern such as [W-46] is interpreted as a class of two characters ("W" and ",") followed by a literal string ",46", so it would match "W46" or ",46". However, if the "]" is escaped with a backslash it is interpreted as the end of range, so [W-\46] is interpreted as a class containing a range followed by two other characters. The octal or hexadecimal representation of "]" can also be used to end a range.

Ranges operate in the collating sequence of character values. They can also be used for characters specified numerically, for example \000-\037. In UTF-8 mode, ranges can include characters whose values are greater than 255, for example \x{100}-\x{2ff}.

The character types \d, \D, \p, \P, \s, \S, \w, and \W may also appear in a character class, and add the characters that they match to the class. For example, \dABCDEF matches any hexadecimal digit. A circumflex can conveniently be used with the upper case character types to specify a more restricted set of characters than the matching lower case type. For example, the class [\W_] matches any letter or digit, but not underscore.

The only metacharacters that are recognized in character classes are backslash, hyphen (only where it can be interpreted as specifying a range), circumflex (only at the start), opening square bracket (only when it can be interpreted as introducing a POSIX class name - see the next section), and the terminating closing square bracket. However, escaping other non-alphanumeric characters does no harm.

**Posix character classes**

GRegex supports the POSIX notation for character classes. This uses names enclosed by [: and :) within the enclosing square brackets. For example,

```
[01[:alpha:]]
```

matches "0", "1", any alphabetic character, or "". The supported class names are

- The "space" characters are HT (9), LF (10), VT (11), FF (12), CR (13), and space (32). Notice that this list includes the VT character (code 11). This makes "space" different to \s, which does not include VT (for Perl compatibility).
- The name "word" is a Perl extension, and "blank" is a GNU extension. Another Perl extension is negation, which is indicated by a \ character after the colon. For example,

```
[12[:^digit:]]
```

matches "1", "2", or any non-digit. GRegex also recognize the POSIX syntax [.ch] and [=ch=] where "ch" is a "collating element", but these are not supported, and an error is given if they are encountered.

In UTF-8 mode, characters with values greater than 128 do not match any of the POSIX character classes.

**Vertical bar**

Vertical bar characters are used to separate alternative patterns. For example, the pattern

```
gilbert|sullivan
```
matches either "gilbert" or "sullivan". Any number of alternatives may appear, and an empty alternative is permitted (matching the empty string). The matching process tries each alternative in turn, from left to right, and the first one that succeeds is used. If the alternatives are within a subpattern (defined below), "succeeds" means matching the rest of the main pattern as well as the alternative in the subpattern.

**Internal option setting**

The settings of the G_REGEXP_CASELESS, G_REGEXP_MULTILINE, G_REGEXP_MULTILINE, and G_REGEXP_EXTENDED options can be changed from within the pattern by a sequence of Perl-style option letters enclosed between "(?" and ")". The option letters are

<table>
<thead>
<tr>
<th>Option</th>
<th>Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>G_REGEXP_CASELESS</td>
</tr>
<tr>
<td>m</td>
<td>G_REGEXP_MULTILINE</td>
</tr>
<tr>
<td>s</td>
<td>G_REGEXP_DOTALL</td>
</tr>
<tr>
<td>x</td>
<td>G_REGEXP_EXTENDED</td>
</tr>
</tbody>
</table>

For example, (?im) sets caseless, multiline matching. It is also possible to unset these options by preceding the letter with a hyphen, and a combined setting and unset setting such as (?im-sx), which sets G_REGEXP_CASELESS and G_REGEXP_MULTILINE while unsetting G_REGEXP_DOTALL and G_REGEXP_EXTENDED, is also permitted. If a letter appears both before and after the hyphen, the option is unset.

When an option change occurs at top level (that is, not inside subpattern parentheses), the change applies to the remainder of the pattern that follows.

An option change within a subpattern (see below for a description of subpatterns) affects only that part of the current pattern that follows it, so

```
(a(?i)b)c
```

matches abc and aBc and no other strings (assuming G_REGEXP_CASELESS is not used). By this means, options can be made to have different settings in different parts of the pattern. Any changes made in one alternative do carry on into subsequent branches within the same subpattern. For example,

```
(a(?i)b)c
```

matches "ab", "aB", "c", and "C", even though when matching "C" the first branch is abandoned before the option setting. This is because the effects of option settings happen at compile time. There would be some very weird behaviour otherwise.

The options G_REGEXP_UNGREEDY and G_REGEXP_EXTRA and G_REGEXP_DUPNAMES can be changed in the same way as the Perl-compatible options by using the characters U, X and J respectively.
Subpatterns

Subpatterns are delimited by parentheses (round brackets), which can be nested. Turning part of a pattern into a subpattern does two things:

- It localizes a set of alternatives. For example, the pattern `cat(aract|erpillar|)` matches one of the words "cat", "catact", or "caterpillar". Without the parentheses, it would match "catact", "erpillar" or an empty string.

- It sets up the subpattern as a capturing subpattern. This means that, when the whole pattern matches, that portion of the string that matched the subpattern can be obtained using `g_regex_fetch()`. Opening parentheses are counted from left to right (starting from 1, as subpattern 0 is the whole matched string) to obtain numbers for the capturing subpatterns.

For example, if the string "the red king" is matched against the pattern

```
the ((red|white) (king|queen))
```

the captured substrings are "red king", "red", and "king", and are numbered 1, 2, and 3, respectively. The fact that plain parentheses fulfil two functions is not always helpful. There are often times when a grouping subpattern is required without a capturing requirement. If an opening parenthesis is followed by a question mark and a colon, the subpattern does not do any capturing, and is not counted when computing the number of any subsequent capturing subpatterns. For example, if the string "the white queen" is matched against the pattern

```
the ((?:red|white) (king|queen))
```

the captured substrings are "white queen" and "queen", and are numbered 1 and 2. The maximum number of capturing subpatterns is 65535.

As a convenient shorthand, if any option settings are required at the start of a non-capturing subpattern, the option letters may appear between the "?" and the ":". Thus the two patterns

```
(?i:saturday|sunday)
(?i:(?i)saturday|sunday)
```

match exactly the same set of strings. Because alternative branches are tried from left to right, and options are not reset until the end of the subpattern is reached, an option setting in one branch does affect subsequent branches, so the above patterns match "SUNDAY" as well as "Saturday".

Named subpatterns

Identifying capturing parentheses by number is simple, but it can be very hard to keep track of the numbers in complicated regular expressions. Furthermore, if an expression is modified, the numbers may change. To help with this difficulty, GRegex supports the naming of subpatterns. A subpattern can be named in one of three ways: `(?<name>...)` or `(?'name'...)` as in Perl, or `(?P<name>...)` as in Python. References to capturing parentheses from other parts of the pattern, such as backreferences, recursion, and conditions, can be made by name as well as by number.

Names consist of up to 32 alphanumeric characters and underscores. Named capturing parentheses are still allocated numbers as well as names, exactly as if the names were not present. By default, a name must be unique within a pattern, but it is possible to relax this constraint by setting the `G_REG_EX_DUPNAMES` option at compile time. This can be useful for patterns where only one instance of the named parentheses can match. Suppose you want to match the name of a weekday, either as a 3-letter abbreviation or as the full name, and in both cases you want to extract the abbreviation. This pattern (ignoring the line breaks) does the job:

```
(?<DN>Mon|Fri|Sun)(?:day)?|
(?<DN>Tue)(?:sday)?|
(?<DN>Wed)(?:nesday)?|
(?<DN>Thu)(?:rsday)?|
(?<DN>Sat)(?:urday)?
```

There are five capturing substrings, but only one is ever set after a match. The function for extracting the data by name returns the substring for the first (and in this example, the only) subpattern of that
name that matched. This saves searching to find which numbered subpattern it was. If you make a reference to a non-unique named subpattern from elsewhere in the pattern, the one that corresponds to the lowest number is used.

Repetition

Repetition is specified by quantifiers, which can follow any of the following items:

- a literal data character
- the dot metacharacter
- the \C escape sequence
- the \X escape sequence (in UTF-8 mode)
- the \R escape sequence
- an escape such as \d that matches a single character
- a character class
- a back reference (see next section)
- a parenthesized subpattern (unless it is an assertion)

The general repetition quantifier specifies a minimum and maximum number of permitted matches, by giving the two numbers in curly brackets (braces), separated by a comma. The numbers must be less than 65536, and the first must be less than or equal to the second. For example:

\(z\{2,4\}\)

matches "zz", "zzz", or "zzzz". A closing brace on its own is not a special character. If the second number is omitted, but the comma is present, there is no upper limit; if the second number and the comma are both omitted, the quantifier specifies an exact number of required matches. Thus

\([aeiou]\{3,\}\)

matches at least 3 successive vowels, but may match many more, while

\(\d\{8\}\)

matches exactly 8 digits. An opening curly bracket that appears in a position where a quantifier is not allowed, or one that does not match the syntax of a quantifier, is taken as a literal character. For example, {.6} is not a quantifier, but a literal string of four characters.

In UTF-8 mode, quantifiers apply to UTF-8 characters rather than to individual bytes. Thus, for example, \x{100}{2} matches two UTF-8 characters, each of which is represented by a two-byte sequence. Similarly, \X{3} matches three Unicode extended sequences, each of which may be several bytes long (and they may be of different lengths).

The quantifier \(0\) is permitted, causing the expression to behave as if the previous item and the quantifier were not present.

For convenience, the three most common quantifiers have single-character abbreviations:

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>is equivalent to [0,]</td>
</tr>
<tr>
<td>+</td>
<td>is equivalent to [1,]</td>
</tr>
<tr>
<td>?</td>
<td>is equivalent to [0,1]</td>
</tr>
</tbody>
</table>

It is possible to construct infinite loops by following a subpattern that can match no characters with a quantifier that has no upper limit, for example:

\(\{a?\}^*\)
Because there are cases where this can be useful, such patterns are accepted, but if any repetition of the subpattern does in fact match no characters, the loop is forcibly broken.

By default, the quantifiers are 'greedy', that is, they match as much as possible (up to the maximum number of permitted times), without causing the rest of the pattern to fail. The classic example of where this gives problems is in trying to match comments in C programs. These appear between /* and */ and within the comment, individual * and / characters may appear. An attempt to match C comments by applying the pattern

```
/
*.*
*/
```

to the string

```
/* first comment */ not comment /* second comment */
```

fails, because it matches the entire string owing to the greediness of the . * item.

However, if a quantifier is followed by a question mark, it ceases to be greedy, and instead matches the minimum number of times possible, so the pattern

```
/\*.*?\*/
```

does the right thing with the C comments. The meaning of the various quantifiers is not otherwise changed, just the preferred number of matches. Do not confuse this use of question mark with its use as a quantifier in its own right. Because it has two uses, it can sometimes appear doubled, as in

```
\d??\d
```

which matches one digit by preference, but can match two if that is the only way the rest of the pattern matches.

If the G_REGEX_UNGREEDY flag is set, the quantifiers are not greedy by default, but individual ones can be made greedy by following them with a question mark. In other words, it inverts the default behaviour.

When a parenthesized subpattern is quantified with a minimum repeat count that is greater than 1 or with a limited maximum, more memory is required for the compiled pattern, in proportion to the size of the minimum or maximum.

If a pattern starts with .* or .{0,} and the G_REGEX_DOTALL flag is set, thus allowing the dot to match newlines, the pattern is implicitly anchored, because whatever follows will be tried against every character position in the string, so there is no point in retrying the overall match at any position after the first. GRegex normally treats such a pattern as though it were preceded by \A.

In cases where it is known that the string contains no newlines, it is worth setting G_REGEX_DOTALL in order to obtain this optimization, or alternatively using ^ to indicate anchoring explicitly.

However, there is one situation where the optimization cannot be used. When .* is inside capturing parentheses that are the subject of a backreference elsewhere in the pattern, a match at the start may fail where a later one succeeds. Consider, for example:

```
(.*)abc\1
```

If the string is "xyz123abc123" the match point is the fourth character. For this reason, such a pattern is not implicitly anchored.

When a capturing subpattern is repeated, the value captured is the substring that matched the final iteration. For example, after

```
(tweedle[duke]{3}\s*)+
```

has matched "tweedledum tweedledee" the value of the captured substring is "tweedledee". However, if there are nested capturing subpatterns, the corresponding captured values may have been set in previous iterations. For example, after

```
/(a|b)+/
```

matches "aba" the value of the second captured substring is "b".
Atomic grouping and possessive quantifiers

With both maximizing ("greedy") and minimizing ("ungreedy" or "lazy") repetition, failure of what follows normally causes the repeated item to be re-evaluated to see if a different number of repeats allows the rest of the pattern to match. Sometimes it is useful to prevent this, either to change the nature of the match, or to cause it fail earlier than it otherwise might, when the author of the pattern knows there is no point in carrying on.

Consider, for example, the pattern \d+foo when applied to the string

123456bar

After matching all 6 digits and then failing to match "foo", the normal action of the matcher is to try again with only 5 digits matching the \d+ item, and then with 4, and so on, before ultimately failing.

"Atomic grouping" (a term taken from Jeffrey Friedl’s book) provides the means for specifying that once a subpattern has matched, it is not to be re-evaluated in this way.

If we use atomic grouping for the previous example, the matcher give up immediately on failing to match "foo" the first time. The notation is a kind of special parenthesis, starting with (?) as in this example:

(?>\d+)foo

This kind of parenthesis "locks up" the part of the pattern it contains once it has matched, and a failure further into the pattern is prevented from backtracking into it. Backtracking past it to previous items, however, works as normal.

An alternative description is that a subpattern of this type matches the string of characters that an identical standalone pattern would match, if anchored at the current point in the string.

Atomic grouping subpatterns are not capturing subpatterns. Simple cases such as the above example can be thought of as a maximizing repeat that must swallow everything it can. So, while both \d+ and \d+? are prepared to adjust the number of digits they match in order to make the rest of the pattern match, (?>\d+) can only match an entire sequence of digits.

Atomic groups in general can of course contain arbitrarily complicated subpatterns, and can be nested. However, when the subpattern for an atomic group is just a single repeated item, as in the example above, a simpler notation, called a "possessive quantifier" can be used. This consists of an additional + character following a quantifier. Using this notation, the previous example can be rewritten as

\d++foo

Possessive quantifiers are always greedy; the setting of the G_REGEX_UNGREEDY option is ignored. They are a convenient notation for the simpler forms of atomic group. However, there is no difference in the meaning of a possessive quantifier and the equivalent atomic group, though there may be a performance difference; possessive quantifiers should be slightly faster.

The possessive quantifier syntax is an extension to the Perl syntax. It was invented by Jeffrey Friedl in the first edition of his book and then implemented by Mike McCloskey in Sun’s Java package. It ultimately found its way into Perl at release 5.10.

GRegex has an optimization that automatically "possessifies" certain simple pattern constructs. For example, the sequence A+B is treated as A++B because there is no point in backtracking into a sequence of A’s when B must follow.

When a pattern contains an unlimited repeat inside a subpattern that can itself be repeated an unlimited number of times, the use of an atomic group is the only way to avoid some failing matches taking a very long time indeed. The pattern

(\D+|<\d+>)*[!?] 

matches an unlimited number of substrings that either consist of non-digit, or digits enclosed in <>, followed by either ! or ?. When it matches, it runs quickly. However, if it is applied to

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

it takes a long time before reporting failure. This is because the string can be divided between the internal \D+ repeat and the external * repeat in a large number of ways, and all have to be tried. (The example uses [!?] rather than a single character at the end, because GRegex has an optimization that allows for fast failure when a single character is used. It remember the last single character that is
required for a match, and fail early if it is not present in the string.) If the pattern is changed so that it uses an atomic group, like this:

```regex
((?>\D+)|<\d+>)*[!~]
```

sequences of non-digits cannot be broken, and failure happens quickly.

### Back references

Outside a character class, a backslash followed by a digit greater than 0 (and possibly further digits) is a back reference to a capturing subpattern earlier (that is, to its left) in the pattern, provided there have been that many previous capturing left parentheses.

However, if the decimal number following the backslash is less than 10, it is always taken as a back reference, and causes an error only if there are not that many capturing left parentheses in the entire pattern. In other words, the parentheses that are referenced need not be to the left of the reference for numbers less than 10. A "forward back reference" of this type can make sense when a repetition is involved and the subpattern to the right has participated in an earlier iteration.

It is not possible to have a numerical "forward back reference" to subpattern whose number is 10 or more using this syntax because a sequence such as \\e50 is interpreted as a character defined in octal. See the subsection entitled "Non-printing characters" above for further details of the handling of digits following a backslash. There is no such problem when named parentheses are used. A back reference to any subpattern is possible using named parentheses (see below).

Another way of avoiding the ambiguity inherent in the use of digits following a backslash is to use the \g escape sequence (introduced in Perl 5.10.) This escape must be followed by a positive or a negative number, optionally enclosed in braces.

A positive number specifies an absolute reference without the ambiguity that is present in the older syntax. It is also useful when literal digits follow the reference. A negative number is a relative reference. Consider "(abc\d\fghi)\g{-1}\", the sequence \g{-1} is a reference to the most recently started capturing subpattern before \g, that is, is it equivalent to \2. Similarly, \g{-2} would be equivalent to \1. The use of relative references can be helpful in long patterns, and also in patterns that are created by joining together fragments that contain references within themselves.

A back reference matches whatever actually matched the capturing subpattern in the current string, rather than anything matching the subpattern itself (see "Subpatterns as subroutines" below for a way of doing that). So the pattern

```
(sens|respons)e and \libility
```

matches "sense and sensibility" and "response and responsibility", but not "sense and responsibility". If caseful matching is in force at the time of the back reference, the case of letters is relevant. For example,

```
((?!\1)rah)\s+\1
```

matches "rah rah" and "RAH RAH", but not "RAH rah", even though the original capturing subpattern is matched caselessly.

Back references to named subpatterns use the Perl syntax \\k<name> or \\k'name' or the Python syntax (?P=name). We could rewrite the above example in either of the following ways:

```
(?<pl>(?!\1)rah)\s+\k<pl>
(?P<pl>(?!\1)rah)\s+(?P=pl)
```

A subpattern that is referenced by name may appear in the pattern before or after the reference. There may be more than one back reference to the same subpattern. If a subpattern has not actually been used in a particular match, any back references to it always fail. For example, the pattern

```
(a|bc) \2
```

always fails if it starts to match "a" rather than "bc". Because there may be many capturing parentheses in a pattern, all digits following the backslash are taken as part of a potential back reference number. If the pattern continues with a digit character, some delimiter must be used to terminate the back reference. If the G_REGEX_EXTENDED flag is set, this can be whitespace. Otherwise an empty comment (see "Comments" below) can be used.

A back reference that occurs inside the parentheses to which it refers fails when the subpattern is first used, so, for example, (a\1) never matches. However, such references can be useful inside repeated subpatterns. For example, the pattern
matches any number of "a"s and also "aba", "ababaa" etc. At each iteration of the subpattern, the back reference matches the character string corresponding to the previous iteration. In order for this to work, the pattern must be such that the first iteration does not need to match the back reference. This can be done using alternation, as in the example above, or by a quantifier with a minimum of zero.

**Assertions**

An assertion is a test on the characters following or preceding the current matching point that does not actually consume any characters. The simple assertions coded as \b, \B, \A, \G, \Z, \z, ˆ and $ are described above.

More complicated assertions are coded as subpatterns. There are two kinds: those that look ahead of the current position in the string, and those that look behind it. An assertion subpattern is matched in the normal way, except that it does not cause the current matching position to be changed.

Assertion subpatterns are not capturing subpatterns, and may not be repeated, because it makes no sense to assert the same thing several times. If any kind of assertion contains capturing subpatterns within it, these are counted for the purposes of numbering the capturing subpatterns in the whole pattern. However, substring capturing is carried out only for positive assertions, because it does not make sense for negative assertions.

**Lookahead assertions**

Lookahead assertions start with (?! for positive assertions and (?! for negative assertions. For example,

\w+(?=;)  
matches a word followed by a semicolon, but does not include the semicolon in the match, and

foo(?!bar)  
matches any occurrence of "foo" that is not followed by "bar". Note that the apparently similar pattern

(?!foo)bar  
does not find an occurrence of "bar" that is preceded by something other than "foo"; it finds any occurrence of "bar" whatsoever, because the assertion (?!foo) is always true when the next three characters are "bar". A lookbehind assertion is needed to achieve the other effect.

If you want to force a matching failure at some point in a pattern, the most convenient way to do it is with (?! because an empty string always matches, so an assertion that requires there not to be an empty string must always fail.

**Lookbehind assertions**

Lookbehind assertions start with (?<= for positive assertions and (?<! for negative assertions. For example,

(?<=\w+{>=;})  

is permitted, but

(?<!\w+{>=;})  
causes an error at compile time. Branches that match different length strings are permitted only at the top level of a lookbehind assertion. An assertion such as

(?<=ab{c|de})
is not permitted, because its single top-level branch can match two different lengths, but it is acceptable if rewritten to use two top-level branches:

\[(?<=abc|abde)\]

The implementation of lookbehind assertions is, for each alternative, to temporarily move the current position back by the fixed length and then try to match. If there are insufficient characters before the current position, the assertion fails.

GRegex does not allow the \C escape (which matches a single byte in UTF-8 mode) to appear in lookbehind assertions, because it makes it impossible to calculate the length of the lookbehind. The \X and \R escapes, which can match different numbers of bytes, are also not permitted.

Possessive quantifiers can be used in conjunction with lookbehind assertions to specify efficient matching at the end of the subject string. Consider a simple pattern such as

\[abcd\]

when applied to a long string that does not match. Because matching proceeds from left to right, GRegex will look for each "a" in the string and then see if what follows matches the rest of the pattern. If the pattern is specified as

\[.+.abcd\]

the initial .+ matches the entire string at first, but when this fails (because there is no following "a"), it backtracks to match all but the last character, then all but the last two characters, and so on. Once again the search for "a" covers the entire string, from right to left, so we are no better off. However, if the pattern is written as

\[.+.+(?<=abcd)\]

there can be no backtracking for the .+ item; it can match only the entire string. The subsequent lookbehind assertion does a single test on the last four characters. If it fails, the match fails immediately. For long strings, this approach makes a significant difference to the processing time.

Using multiple assertions

Several assertions (of any sort) may occur in succession. For example,

\[(?<=\d{3}) (?<!999)foo\]

matches "foo" preceded by three digits that are not "999". Notice that each of the assertions is applied independently at the same point in the string. First there is a check that the previous three characters are all digits, and then there is a check that the same three characters are not "999". This pattern does not match "foo" preceded by six characters, the first of which are digits and the last three of which are not "999". For example, it doesn’t match "123abcfoo". A pattern to do that is

\[(?<=\d{3}...)(?<!999)foo\]

This time the first assertion looks at the preceding six characters, checking that the first three are digits, and then the second assertion checks that the preceding three characters are not "999".

Assertions can be nested in any combination. For example,

\[(?<=(?<!foo)bar)baz\]

matches an occurrence of "baz" that is preceded by "bar" which in turn is not preceded by "foo", while

\[(?<=\d{3}(?!999)...)?(?!999)foo\]

is another pattern that matches "foo" preceded by three digits and any three characters that are not "999".

Conditional subpatterns

It is possible to cause the matching process to obey a subpattern conditionally or to choose between two alternative subpatterns, depending on the result of an assertion, or whether a previous capturing subpattern matched or not. The two possible forms of conditional subpattern are
If the condition is satisfied, the yes-pattern is used; otherwise the no-pattern (if present) is used. If there are more than two alternatives in the subpattern, a compile-time error occurs.

There are four kinds of condition: references to subpatterns, references to recursion, a pseudo-condition called DEFINE, and assertions.

Checking for a used subpattern by number

If the text between the parentheses consists of a sequence of digits, the condition is true if the capturing subpattern of that number has previously matched.

Consider the following pattern, which contains non-significant white space to make it more readable (assume the G_REGEX_EXTENDED) and to divide it into three parts for ease of discussion:

```
(\(\)?[^\(\)]+ (?(1) \))
```

The first part matches an optional opening parenthesis, and if that character is present, sets it as the first captured substring. The second part matches one or more characters that are not parentheses. The third part is a conditional subpattern that tests whether the first set of parentheses matched or not. If they did, that is, if string started with an opening parenthesis, the condition is true, and so the yes-pattern is executed and a closing parenthesis is required. Otherwise, since no-pattern is not present, the subpattern matches nothing. In other words, this pattern matches a sequence of non-parentheses, optionally enclosed in parentheses.

Checking for a used subpattern by name

Perl uses the syntax (?<name>...) or (?('name')...) to test for a used subpattern by name, the Python syntax (?(name)...) is also recognized. However, there is a possible ambiguity with this syntax, because subpattern names may consist entirely of digits. GRegex looks first for a named subpattern; if it cannot find one and the name consists entirely of digits, GRegex looks for a subpattern of that number, which must be greater than zero. Using subpattern names that consist entirely of digits is not recommended.

Rewriting the above example to use a named subpattern gives this:

```
(?<OPEN>(\(\)?[^\(\)]+ (?(<OPEN>) \)))
```

Checking for pattern recursion

If the condition is the string (R), and there is no subpattern with the name R, the condition is true if a recursive call to the whole pattern or any subpattern has been made. If digits or a name preceded by ampersand follow the letter R, for example:

```
(?(R3)...)
(?(R&name)...)
```

the condition is true if the most recent recursion is into the subpattern whose number or name is given. This condition does not check the entire recursion stack.

At "top level", all these recursion test conditions are false. Recursive patterns are described below.

Defining subpatterns for use by reference only

If the condition is the string (DEFINE), and there is no subpattern with the name DEFINE, the condition is always false. In this case, there may be only one alternative in the subpattern. It is always skipped if control reaches this point in the pattern; the idea of DEFINE is that it can be used to define "subroutines" that can be referenced from elsewhere. (The use of "subroutines" is described below.) For example, a pattern to match an IPv4 address could be written like this (ignore whitespace and line breaks):

```
(?(DEFINE) (?<byte> 2[0-4]\d | 25[0-5] | 1\d\d | [1-9]\d) )
\b (?&byte) (\.(?&byte)){3} \b
```
The first part of the pattern is a DEFINE group inside which a another group named "byte" is defined. This matches an individual component of an IPv4 address (a number less than 256). When matching takes place, this part of the pattern is skipped because DEFINE acts like a false condition.

The rest of the pattern uses references to the named group to match the four dot-separated components of an IPv4 address, insisting on a word boundary at each end.

Assertion conditions
If the condition is not in any of the above formats, it must be an assertion. This may be a positive or negative lookahead or lookbehind assertion. Consider this pattern, again containing non-significant white space, and with the two alternatives on the second line:

```
(?(?=.*[^a-z]*[a-z])
 \d{2}-[^a-z]{3}-\d{2} | \d{2}-\d{2}-\d{2} )
```

The condition is a positive lookahead assertion that matches an optional sequence of non-letters followed by a letter. In other words, it tests for the presence of at least one letter in the string. If a letter is found, the string is matched against the first alternative; otherwise it is matched against the second. This pattern matches strings in one of the two forms dd-aaa-dd or dd-dd-dd, where aaa are letters and dd are digits.

Comments
The sequence (?)# marks the start of a comment that continues up to the next closing parenthesis. Nested parentheses are not permitted. The characters that make up a comment play no part in the pattern matching at all.

If the G_REGEX_EXTENDED option is set, an unescaped # character outside a character class introduces a comment that continues to immediately after the next newline in the pattern.

Recursive patterns
Consider the problem of matching a string in parentheses, allowing for unlimited nested parentheses. Without the use of recursion, the best that can be done is to use a pattern that matches up to some fixed depth of nesting. It is not possible to handle an arbitrary nesting depth.

For some time, Perl has provided a facility that allows regular expressions to recurse (amongst other things). It does this by interpolating Perl code in the expression at run time, and the code can refer to the expression itself. A Perl pattern using code interpolation to solve the parentheses problem can be created like this:

```
$re = qr{\( (?(R) | (?p{$re}) )* \)}x;
```

The (?p{...}) item interpolates Perl code at run time, and in this case refers recursively to the pattern in which it appears.

Obviously, GRegex cannot support the interpolation of Perl code. Instead, it supports special syntax for recursion of the entire pattern, and also for individual subpattern recursion. This kind of recursion was introduced into Perl at release 5.10.

A special item that consists of (?) followed by a number greater than zero and a closing parenthesis is a recursive call of the subpattern of the given number, provided that it occurs inside that subpattern. (If not, it is a "subroutine" call, which is described in the next section.) The special item (?)R or (?)0 is a recursive call of the entire regular expression.

In GRegex (like Python, but unlike Perl), a recursive subpattern call is always treated as an atomic group. That is, once it has matched some of the subject string, it is never re-entered, even if it contains untried alternatives and there is a subsequent matching failure.

This pattern solves the nested parentheses problem (assume the G_REGEX_EXTENDED option is set so that white space is ignored):

```
\( ( (?[^)]|)+ | (?R) )* \)
```

First it matches an opening parenthesis. Then it matches any number of substrings which can either be a sequence of non-parentheses, or a recursive match of the pattern itself (that is, a correctly parenthesized substring). Finally there is a closing parenthesis.
If this were part of a larger pattern, you would not want to recurse the entire pattern, so instead you could use this:

```text
( \( \( (?>[^()]+) | (?1) \)* \) )
```

We have put the pattern into parentheses, and caused the recursion to refer to them instead of the whole pattern. In a larger pattern, keeping track of parenthesis numbers can be tricky. It may be more convenient to use named parentheses instead. The Perl syntax for this is (?&name); GRegex also supports the(?P>name) syntax. We could rewrite the above example as follows:

```text
(?<pn> \( \( (?>[^()]+) | (?&pn) \)* \) )
```

If there is more than one subpattern with the same name, the earliest one is used. This particular example pattern contains nested unlimited repeats, and so the use of atomic grouping for matching strings of non-parentheses is important when applying the pattern to strings that do not match. For example, when this pattern is applied to

```text
(aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa()
```

it yields "no match" quickly. However, if atomic grouping is not used, the match runs for a very long time indeed because there are so many different ways the + and * repeats can carve up the string, and all have to be tested before failure can be reported.

At the end of a match, the values set for any capturing subpatterns are those from the outermost level of the recursion at which the subpattern value is set. If the pattern above is matched against

```text
(ab(cd)ef)
```

the value for the capturing parentheses is "ef", which is the last value taken on at the top level. If additional parentheses are added, giving

```text
( ( (?>[^()]+) | (?R) )* )
```

the string they capture is "ab(cd)ef", the contents of the top level parentheses.

Do not confuse the (?R) item with the condition (R), which tests for recursion. Consider this pattern, which matches text in angle brackets, allowing for arbitrary nesting. Only digits are allowed in nested brackets (that is, when recursing), whereas any characters are permitted at the outer level.

```text
< (?: (?(R) \d++ | \[^<>]*+) | (?R)) * >
```

In this pattern, (?(R) is the start of a conditional subpattern, with two different alternatives for the recursive and non-recursive cases. The (?R) item is the actual recursive call.

### Subpatterns as subroutines

If the syntax for a recursive subpattern reference (either by number or by name) is used outside the parentheses to which it refers, it operates like a subroutine in a programming language. The "called" subpattern may be defined before or after the reference. An earlier example pointed out that the pattern

```text
{sens|respons)e and \libility
```

matches "sense and sensibility" and "response and responsibility", but not "sense and responsibility". If instead the pattern

```text
{sens|respons)e and (?l)ibility
```

is used, it does match "sense and responsibility" as well as the other two strings. Another example is given in the discussion of DEFINE above.

Like recursive subpatterns, a "subroutine" call is always treated as an atomic group. That is, once it has matched some of the string, it is never re-entered, even if it contains untried alternatives and there is a subsequent matching failure.

When a subpattern is used as a subroutine, processing options such as case-independence are fixed when the subpattern is defined. They cannot be changed for different calls. For example, consider this pattern:
1.7 Mailing lists and bug reports

Name
Mailing lists and bug reports – Getting help with GLib

Filing a bug report or feature request
If you encounter a bug, misfeature, or missing feature in GLib, please file a bug report on http://bugzilla.gnome.org. We’d also appreciate reports of incomplete or misleading information in the GLib documentation; file those against the "docs" component of the "glib" product in Bugzilla.

Don’t hesitate to file a bug report, even if you think we may know about it already, or aren’t sure of the details. Just give us as much information as you have, and if it’s already fixed or has already been discussed, we’ll add a note to that effect in the report.

The bug tracker should definitely be used for feature requests, it’s not only for bugs. We track all GLib development in Bugzilla, so it’s the way to be sure the GLib developers won’t forget about an issue.
Submitting Patches

If you develop a bugfix or enhancement for GLib, please file that in Bugzilla as well. Bugzilla allows you to attach files; please attach a patch generated by the `diff` utility, using the `-u` option to make the patch more readable. All patches must be offered under the terms of the GNU LGPL license, so be sure you are authorized to give us the patch under those terms.

If you want to discuss your patch before or after developing it, mail `gtk-devel-list@gnome.org`. But be sure to file the Bugzilla report as well; if the patch is only on the list and not in Bugzilla, it’s likely to slip through the cracks.

Mailing lists

There are several mailing lists dedicated to GTK+ and related libraries. Discussion of GLib generally takes place on these lists. You can subscribe or view the archives of these lists on http://mail.gnome.org.

- **gtk-list@gnome.org**  gtk-list covers general GTK+ (and GLib) topics; questions about using GLib in programs, GLib from a user standpoint, announcements of GLib-related projects would all be on-topic. The bulk of the traffic consists of GTK+ programming questions.

- **gtk-devel-list@gnome.org**  gtk-devel-list is for discussion of work on GTK+ (and GLib) itself, it is not for asking questions about how to use GTK+ (or GLib) in applications. gtk-devel-list is appropriate for discussion of patches, bugs, proposed features, and so on.

- **gtk-doc-list@gnome.org**  gtk-doc-list is for discussion of the gtk-doc documentation system (used to document GTK+ and GLib), and for work on the GTK+ (and GLib) documentation.
Chapter 2

GLib Fundamentals

2.1 Version Information

Name

Version Information – Variables and functions to check the GLib version

Synopsis

```
#include <glib.h>

extern const guint glib_major_version;
extern const guint glib_minor_version;
extern const guint glib_micro_version;
extern const guint glib_binary_age;
extern const guint glib_interface_age;
const gchar * glib_check_version (guint required_major,
                                    guint required_minor,
                                    guint required_micro);
```

#define GLIB_MAJOR_VERSION
#define GLIB_MINOR_VERSION
#define GLIB_MICRO_VERSION
#define GLIB_CHECK_VERSION (major,minor,micro)

Description

GLib provides version information, primarily useful in configure checks for builds that have a configure script. Applications will not typically use the features described here.

Details

glib_major_version

```
extern const guint glib_major_version;
```

The major version number of the GLib library. (e.g. in GLib version 1.2.5 this is 1.)

This variable is in the library, so represents the GLib library you have linked against. Contrast with the GLIB_MAJOR_VERSION macro, which represents the major version of the GLib headers you have included.

glib_minor_version

33
extern const guint glib_minor_version;

The minor version number of the GLib library. (e.g. in GLib version 1.2.5 this is 2.)
This variable is in the library, so represents the GLib library you have linked against. Contrast with the GLIB_MINOR_VERSION macro, which represents the minor version of the GLib headers you have included.

glib_micro_version

extern const guint glib_micro_version;

The micro version number of the GLib library. (e.g. in GLib version 1.2.5 this is 5.)
This variable is in the library, so represents the GLib library you have linked against. Contrast with the GLIB_MICRO_VERSION macro, which represents the micro version of the GLib headers you have included.

glib_binary_age

extern const guint glib_binary_age;

This is the binary age passed to libtool. If libtool means nothing to you, don’t worry about it. ;-)

glib_interface_age

extern const guint glib_interface_age;

This is the interface age passed to libtool. If libtool means nothing to you, don’t worry about it. ;-) 

glib_check_version ()

const gchar * glib_check_version (guint required_major,
guint required_minor,
guint required_micro);

Checks that the GLib library in use is compatible with the given version. Generally you would pass in the constants GLIB_MAJOR_VERSION, GLIB_MINOR_VERSION, GLIB_MICRO_VERSION as the three arguments to this function; that produces a check that the library in use is compatible with the version of GLib the application or module was compiled against.

Compatibility is defined by two things: first the version of the running library is newer than the version required_major.required_minor.required_micro. Second the running library must be binary compatible with the version required_major.required_minor.required_micro (same major version.)

required_major: the required major version.
required_minor: the required minor version.
required_micro: the required micro version.

Returns: NULL if the GLib library is compatible with the given version, or a string describing the version mismatch. The returned string is owned by GLib and must not be modified or freed.

Since 2.6

GLIB_MAJOR_VERSION

#define GLIB_MAJOR_VERSION 2

The major version number of the GLib library. Like glib_major_version, but from the headers used at application compile time, rather than from the library linked against at application run time.
GLIB_MINOR_VERSION

#define GLIB_MINOR_VERSION 21

The minor version number of the GLib library. Like gtk_minor_version, but from the headers used at application compile time, rather than from the library linked against at application run time.

GLIB_MICRO_VERSION

#define GLIB_MICRO_VERSION 1

The micro version number of the GLib library. Like gtk_micro_version, but from the headers used at application compile time, rather than from the library linked against at application run time.

GLIB_CHECK_VERSION()

#define GLIB_CHECK_VERSION(major,minor,micro)

Checks the version of the GLib library. Returns TRUE if the version of the GLib header files is the same as or newer than the passed-in version.

Example 2.1 Checking the version of the GLib library

if (!GLIB_CHECK_VERSION (1, 2, 0))
g_error ("GLib version 1.2.0 or above is needed");

major: the major version number.
minor: the minor version number.
micro: the micro version number.

2.2 Basic Types

Name
Basic Types – standard GLib types, defined for ease-of-use and portability

Synopsis

#include <glib.h>

typedef gboolean;
typedef gpointer;
typedef gconstpointer;
typedef gchar;
typedef guchar;
typedef gint;
typedef guint;
typedef gshort;
typedef gushort;
typedef glong;
typedef gulong;
typedef gint8;
typedef guint8;
CHAPTER 2. GLIB FUNDAMENTALS  2.2. BASIC TYPES

typedef gint16;
typedef guint16;
typedef gint32;
typedef guint32;
#define G_HAVE_GINT64
typedef gint64;
typedef guint64;
#define G_GINT64_CONSTANT (val)
#define G_GUINT64_CONSTANT (val)
typedef gfloat;
typedef gdouble;
typedef gsize;
typedef gssize;
typedef goffset;
#define G_GOFFSET_CONSTANT (val)

Description

GLib defines a number of commonly used types, which can be divided into 4 groups:

- New types which are not part of standard C - gboolean, gsize, gssize.
- Integer types which are guaranteed to be the same size across all platforms - gint8, guint8, gint16,
guint16, gint32, guint32, gint64, guint64.
- Types which are easier to use than their standard C counterparts - gpointer, gconstpointer, guchar,
guint, gushort, gulong.
- Types which correspond exactly to standard C types, but are included for completeness - gchar,
gint, gshort, glong, gfloat, gdouble.

Details

gboolean

typedef gint gboolean;

A standard boolean type. Variables of this type should only contain the value TRUE or FALSE.

gpointer

typedef void* gpointer;

An untyped pointer. gpointer looks better and is easier to use than void*.

gconstpointer

typedef const void* gconstpointer;

An untyped pointer to constant data. The data pointed to should not be changed.
This is typically used in function prototypes to indicate that the data pointed to will not be altered
by the function.

gchar

typedef char gchar;

Corresponds to the standard C char type.
CHAPTER 2. GLIB FUNDAMENTALS

2.2. BASIC TYPES

guchar

typedef unsigned char guchar;

Corresponds to the standard C unsigned char type.

gint

typedef int gint;

Corresponds to the standard C int type. Values of this type can range from G_MININT to G_MAXINT.

guint

typedef unsigned int guint;

Corresponds to the standard C unsigned int type. Values of this type can range from 0 to G_MAXUINT.

gshort

typedef short gshort;

Corresponds to the standard C short type. Values of this type can range from G_MINSHORT to G_MAXSHORT.

gushort

typedef unsigned short gushort;

Corresponds to the standard C unsigned short type. Values of this type can range from 0 to G_MAXUSHORT.

glong

typedef long glong;

Corresponds to the standard C long type. Values of this type can range from G_MINLONG to G_MAXLONG.

gulong

typedef unsigned long gulong;

Corresponds to the standard C unsigned long type. Values of this type can range from 0 to G_MAXULONGLONG.

gint8

typedef signed char gint8;

A signed integer guaranteed to be 8 bits on all platforms. Values of this type can range from -128 to 127.

guint8

typedef unsigned char guint8;

An unsigned integer guaranteed to be 8 bits on all platforms. Values of this type can range from 0 to 255.
gint16

typedef signed short gint16;

A signed integer guaranteed to be 16 bits on all platforms. Values of this type can range from -32,768 to 32,767.
To print or scan values of this type, use G_GINT16_MODIFIER and/or G_GINT16_FORMAT.

 guint16

typedef unsigned short guint16;

An unsigned integer guaranteed to be 16 bits on all platforms. Values of this type can range from 0 to 65,535.
To print or scan values of this type, use G_GINT16_MODIFIER and/or G_GUINT16_FORMAT.

 gint32

typedef signed int gint32;

A signed integer guaranteed to be 32 bits on all platforms. Values of this type can range from -2,147,483,648 to 2,147,483,647.
To print or scan values of this type, use G_GINT32_MODIFIER and/or G_GINT32_FORMAT.

 guint32

typedef unsigned int guint32;

An unsigned integer guaranteed to be 32 bits on all platforms. Values of this type can range from 0 to 4,294,967,295.
To print or scan values of this type, use G_GINT32_MODIFIER and/or G_GUINT32_FORMAT.

G_HAVE_GINT64

#define G_HAVE_GINT64 1 /* deprecated, always true */

WARNING

G_HAVE_GINT64 is deprecated and should not be used in newly-written code. GLib requires 64-bit integer support since version 2.0, therefore G_HAVE_GINT64 is always defined.

This macro is defined if 64-bit signed and unsigned integers are available on the platform.

gint64

_GNUC_EXTENSION typedef signed long long gint64;

A signed integer guaranteed to be 64 bits on all platforms. Values of this type can range from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
To print or scan values of this type, use G_GINT64_MODIFIER and/or G_GINT64_FORMAT.
CHAPTER 2. GLIB FUNDAMENTALS

2.2. BASIC TYPES

guint64

An unsigned integer guaranteed to be 64 bits on all platforms. Values of this type can range from 0 to 18,446,744,073,709,551,615.
To print or scan values of this type, use \texttt{G\_GINT64\_MODIFIER} and/or \texttt{G\_GUINT64\_FORMAT}.

\texttt{G\_GINT64\_CONSTANT()}

\begin{verbatim}
#define G_GINT64_CONSTANT(val) (G\_GNUC\_EXTENSION (val##LL))
\end{verbatim}

This macro is used to insert 64-bit integer literals into the source code.
val: a literal integer value, e.g. 0x1d636b02300a7aa7.

\texttt{G\_GUINT64\_CONSTANT()}

\begin{verbatim}
#define G_GUINT64_CONSTANT(val) (G\_GNUC\_EXTENSION (val##ULL))
\end{verbatim}

This macro is used to insert 64-bit unsigned integer literals into the source code.
val: a literal integer value, e.g. 0x1d636b02300a7aa7U.
Since 2.10

gfloat

typedef float gfloat;

Corresponds to the standard C float type. Values of this type can range from \texttt{-G\_MAXFLOAT} to \texttt{G\_MAXFLOAT}.
gdouble

typedef double gdouble;

Corresponds to the standard C double type. Values of this type can range from \texttt{-G\_MAXDOUBLE} to \texttt{G\_MAXDOUBLE}.
gsize

typedef unsigned int gsize;

An unsigned integer type of the result of the sizeof operator, corresponding to the \texttt{size\_t} type defined in C99. This type is wide enough to hold the numeric value of a pointer, so it is usually 32bit wide on a 32bit platform and 64bit wide on a 64bit platform.
To print or scan values of this type, use \texttt{G\_GSIZE\_MODIFIER} and/or \texttt{G\_GSIZE\_FORMAT}.
gssize

typedef signed int gssize;

A signed variant of gsize, corresponding to the \texttt{ssize\_t} defined on most platforms.
To print or scan values of this type, use \texttt{G\_GSIZE\_MODIFIER} and/or \texttt{G\_GSIZE\_FORMAT}.
goffset

typedef gint64 goffset;

A signed integer type that is used for file offsets, corresponding to the C99 type \texttt{off64\_t}.
Since: 2.14
CHAPTER 2. GLIB FUNDAMENTALS 2.3. LIMITS OF BASIC TYPES

G_GOFFSET_CONSTANT()

#define G_GOFFSET_CONSTANT(val) G_GINT64_CONSTANT(val)

This macro is used to insert goffset 64-bit integer literals into the source code. See also G_GINT64_CONSTANT.

val: a literal integer value, e.g. 0x1d636b02300a7aa7. Since: 2.20

2.3 Limits of Basic Types

Name
Limits of Basic Types – portable method of determining the limits of the standard types

Synopsis

#include <glib.h>
#define G_MININT
#define G_MAXINT
#define G_MAXUINT
#define G_MINSHORT
#define G_MAXSHORT
#define G_MAXUSHORT
#define G_MINLONG
#define G_MAXLONG
#define G_MAXULONG
#define G_MININT8
#define G_MAXINT8
#define G_MAXUINT8
#define G_MININT16
#define G_MAXINT16
#define G_MAXUINT16
#define G_MININT32
#define G_MAXINT32
#define G_MAXUINT32
#define G_MININT64
#define G_MAXINT64
#define G_MAXUINT64
#define G_MAXSIZE
#define G_MINSSIZE
#define G_MAXSSIZE
#define G_MINOFFSET
#define G_MAXOFFSET
#define G_MINFLOAT
#define G_MAXFLOAT
#define G_MINDOUBLE
#define G_MAXDOUBLE

40
CHAPTER 2. GLIB FUNDAMENTALS

2.3. LIMITS OF BASIC TYPES

Description
These macros provide a portable method to determine the limits of some of the standard integer and floating point types.

Details

G_MININT

#define G_MININT INT_MIN

The minimum value which can be held in a gint.

G_MAXINT

#define G_MAXINT INT_MAX

The maximum value which can be held in a gint.

G_MAXUINT

#define G_MAXUINT UINT_MAX

The maximum value which can be held in a guint.

G_MINSHORT

#define G_MINSHORT SHRT_MIN

The minimum value which can be held in a gshort.

G_MAXSHORT

#define G_MAXSHORT SHRT_MAX

The maximum value which can be held in a gshort.

G_MAXUSHORT

#define G_MAXUSHORT USHRT_MAX

The maximum value which can be held in a gushort.

G_MINLONG

#define G_MINLONG LONG_MIN

The minimum value which can be held in a glong.

G_MAXLONG

#define G_MAXLONG LONG_MAX

The maximum value which can be held in a glong.

G_MAXULONG

#define G_MAXULONG ULONG_MAX

The maximum value which can be held in a gulong.
CHAPTER 2. GLIB FUNDAMENTALS

2.3. LIMITS OF BASIC TYPES

G_MININT8

#define G_MININT8 ((gint8) 0x80)

The minimum value which can be held in a gint8.
Since 2.4

G_MAXINT8

#define G_MAXINT8 ((gint8) 0x7f)

The maximum value which can be held in a gint8.
Since 2.4

G_MAXUINT8

#define G_MAXUINT8 ((guint8) 0xff)

The maximum value which can be held in a guint8.
Since 2.4

G_MININT16

#define G_MININT16 ((gint16) 0x8000)

The minimum value which can be held in a gint16.
Since 2.4

G_MAXINT16

#define G_MAXINT16 ((gint16) 0x7fff)

The maximum value which can be held in a gint16.
Since 2.4

G_MAXUINT16

#define G_MAXUINT16 ((guint16) 0xffff)

The maximum value which can be held in a guint16.
Since 2.4

G_MININT32

#define G_MININT32 ((gint32) 0x80000000)

The minimum value which can be held in a gint32.
Since 2.4

G_MAXINT32

#define G_MAXINT32 ((gint32) 0x7fffffff)

The maximum value which can be held in a gint32.
Since 2.4

G_MAXUINT32

#define G_MAXUINT32 ((guint32) 0xffffffff)

The maximum value which can be held in a guint32.
Since 2.4
G_MAXUINT32

#define G_MAXUINT32 ((guint32) 0xffffffff)

The maximum value which can be held in a guint32.
Since 2.4

G_MININT64

#define G_MININT64 ((gint64) G_GINT64_CONSTANT(0x8000000000000000))

The minimum value which can be held in a gint64.

G_MAXINT64

#define G_MAXINT64 G_GINT64_CONSTANT(0x7fffffffffffffff)

The maximum value which can be held in a gint64.

G_MAXUINT64

#define G_MAXUINT64 G_GINT64_CONSTANT(0xffffffffffffffffU)

The maximum value which can be held in a guint64.

G_MAXSIZE

#define G_MAXSIZE G_MAXUINT

The maximum value which can be held in a gsize.
Since 2.4

G_MINSSIZE

#define G_MINSSIZE G_MININT

The minimum value which can be held in a gssize.
Since 2.14

G_MAXSSIZE

#define G_MAXSSIZE G_MAXINT

The maximum value which can be held in a gssize.
Since 2.14

G_MINOFFSET

#define G_MINOFFSET G_MININT64

The minimum value which can be held in a goffset.

G_MAXOFFSET

#define G_MAXOFFSET G_MAXINT64

The maximum value which can be held in a goffset.
2.4 Standard Macros

Name
Standard Macros – commonly-used macros.

Synopsis

```
#include <glib.h>

#define G_OS_WIN32
#define G_OS_BEOS
#define G_OS_UNIX

#define G_DIR_SEPARATOR
#define G_DIR_SEPARATOR_S
#define G_IS_DIR_SEPARATOR (c)
#define G_SEARCHPATH_SEPARATOR
#define G_SEARCHPATH_SEPARATOR_S

#define TRUE
#define FALSE

#define NULL

#define MIN (a, b)  
#define MAX (a, b)  
#define ABS (a)     
#define CLAMP (x, low, high)
```

44
#define G_STRUCT_MEMBER (member_type, struct_p, struct_offset)
#define G_STRUCT_MEMBER_P (struct_p, struct_offset)
#define G_STRUCT_OFFSET (struct_type, member)
#define G_MEM_ALIGN
#define G_CONST_RETURN

**Description**

These macros provide a few commonly-used features.

**Details**

**G_OS_WIN32**

#define G_OS_WIN32

This macro is defined only on Windows. So you can bracket Windows-specific code in "#ifdef G_OS_WIN32".

**G_OS_BEOS**

#define G_OS_BEOS

This macro is defined only on BeOS. So you can bracket BeOS-specific code in "#ifdef G_OS_BEOS".

**G_OS_UNIX**

#define G_OS_UNIX

This macro is defined only on UNIX. So you can bracket UNIX-specific code in "#ifdef G_OS_UNIX".

**G_DIR_SEPARATOR**

#define G_DIR_SEPARATOR

The directory separator character. This is ‘/’ on UNIX machines and ‘\’ under Windows.

**G_DIR_SEPARATOR_S**

#define G_DIR_SEPARATOR_S

The directory separator as a string. This is “/” on UNIX machines and “\” under Windows.

**G_IS_DIR_SEPARATOR()**

#define G_IS_DIR_SEPARATOR(c)

Checks whether a character is a directory separator. It returns TRUE for ‘/’ on UNIX machines and for ‘\’ or ‘/’ under Windows.

c: a character

Since 2.6

**G_SEARCHPATH_SEPARATOR**

#define G_SEARCHPATH_SEPARATOR

The search path separator character. This is ‘’ on UNIX machines and ‘,’ under Windows.
CHAPTER 2. GLIB FUNDAMENTALS  2.4. STANDARD MACROS

G_SEARCHPATH_SEPARATOR_S

#define G_SEARCHPATH_SEPARATOR_S

The search path separator as a string. This is ":" on UNIX machines and ";" under Windows.

TRUE

#define TRUE (!FALSE)

Defines the TRUE value for the gboolean type.

FALSE

#define FALSE (0)

Defines the FALSE value for the gboolean type.

NULL

#define NULL

Defines the standard NULL pointer.

MIN()

#define MIN(a, b) (((a) < (b)) ? (a) : (b))

Calculates the minimum of a and b.

a: a numeric value.
b: a numeric value.

Returns: the minimum of a and b.

MAX()

#define MAX(a, b) (((a) > (b)) ? (a) : (b))

Calculates the maximum of a and b.

a: a numeric value.
b: a numeric value.

Returns: the maximum of a and b.

ABS()

#define ABS(a) (((a) < 0) ? -(a) : (a))

Calculates the absolute value of a. The absolute value is simply the number with any negative sign taken away.

For example,

- ABS(-10) is 10.
- ABS(10) is also 10.

a: a numeric value.

Returns: the absolute value of a.
CHAPTER 2. GLIB FUNDAMENTALS

2.4. STANDARD MACROS

CLAMP()

```c
#define CLAMP(x, low, high) (((x) > (high)) ? (high) : (((x) < (low)) ? (low) : (x)))
```

Ensures that \(x\) is between the limits set by \(low\) and \(high\). If \(low\) is greater than \(high\) the result is undefined.

For example,

- \(\text{CLAMP}(5, 10, 15)\) is 10.
- \(\text{CLAMP}(15, 5, 10)\) is 10.
- \(\text{CLAMP}(20, 15, 25)\) is 20.

\(x\): the value to clamp.

\(low\): the minimum value allowed.

\(high\): the maximum value allowed.

Returns: the value of \(x\) clamped to the range between \(low\) and \(high\).

G_STRUCT_MEMBER()

```c
#define G_STRUCT_MEMBER(member_type, struct_p, struct_offset)
```

Returns a member of a structure at a given offset, using the given type.

\(member\_type\): the type of the struct field.

\(struct\_p\): a pointer to a struct.

\(struct\_offset\): the offset of the field from the start of the struct, in bytes.

Returns: the struct member.

G_STRUCT_MEMBER_P()

```c
#define G_STRUCT_MEMBER_P(struct_p, struct_offset)
```

Returns an untyped pointer to a given offset of a struct.

\(struct\_p\): a pointer to a struct.

\(struct\_offset\): the offset from the start of the struct, in bytes.

Returns: an untyped pointer to \(struct\_p\) plus \(struct\_offset\) bytes.

G_STRUCT_OFFSET()

```c
#define G_STRUCT_OFFSET(struct_type, member)
```

Returns the offset, in bytes, of a member of a struct.

\(struct\_type\): a structure type, e.g. GtkWidget.

\(member\): a field in the structure, e.g. window.

Returns: the offset of \(member\) from the start of \(struct\_type\).

G_MEM_ALIGN

```c
#define G_MEM_ALIGN
```

Indicates the number of bytes to which memory will be aligned on the current platform.
CHAPTER 2. GLIB FUNDAMENTALS

2.5 Type Conversion Macros

Name
Type Conversion Macros – portably storing integers in pointer variables

Synopsis

#include <glib.h>

#define GINT_TO_POINTER (i)
#define GPOINTER_TO_INT (p)
#define GUINT_TO_POINTER (u)
#define GPOINTER_TO_UINT (p)
#define GSIZE_TO_POINTER (s)
#define GPOINTER_TO_SIZE (p)

Description

Many times GLib, GTK+, and other libraries allow you to pass "user data" to a callback, in the form of a void pointer. From time to time you want to pass an integer instead of a pointer. You could allocate an integer, with something like:

```c
int *ip = g_new (int, 1);
*ip = 42;
```

But this is inconvenient, and it’s annoying to have to free the memory at some later time.

Pointers are always at least 32 bits in size (on all platforms GLib intends to support). Thus you can store at least 32-bit integer values in a pointer value. Naively, you might try this, but it’s incorrect:

```c
gpointer p;
int i;
p = (void*) 42;
i = (int) p;
```

Again, that example was not correct, don’t copy it. The problem is that on some systems you need to do this:

```c
gpointer p;
int i;
p = (void*) (long) 42;
i = (int) (long) p;
```

So GPOINTER_TO_INT(), GINT_TO_POINTER(), etc. do the right thing on the current platform.
CHAPTER 2. GLIB FUNDAMENTALS 2.5. TYPE CONVERSION MACROS

**WARNING**

YOU MAY NOT STORE POINTERS IN INTEGERS. THIS IS NOT PORTABLE IN ANY WAY SHAPE OR FORM. These macros ONLY allow storing integers in pointers, and only preserve 32 bits of the integer; values outside the range of a 32-bit integer will be mangled.

---

**Details**

**GINT_TO_POINTER()**

```c
#define GINT_TO_POINTER(i) ((gpointer) (i))
```

Stuffs an integer into a pointer type.

Remember, YOU MAY NOT STORE POINTERS IN INTEGERS. THIS IS NOT PORTABLE IN ANY WAY SHAPE OR FORM. These macros ONLY allow storing integers in pointers, and only preserve 32 bits of the integer; values outside the range of a 32-bit integer will be mangled.

*i*: integer to stuff into a pointer.

**GPOINTER_TO_INT()**

```c
#define GPOINTER_TO_INT(p) ((gint) (p))
```

Extracts an integer from a pointer. The integer must have been stored in the pointer with **GINT_TO_POINTER()**.

Remember, YOU MAY NOT STORE POINTERS IN INTEGERS. THIS IS NOT PORTABLE IN ANY WAY SHAPE OR FORM. These macros ONLY allow storing integers in pointers, and only preserve 32 bits of the integer; values outside the range of a 32-bit integer will be mangled.

*p*: pointer containing an integer.

**GUINT_TO_POINTER()**

```c
#define GUINT_TO_POINTER(u) ((gpointer) (u))
```

Stuffs an unsigned integer into a pointer type.

*u*: unsigned integer to stuff into the pointer.

**GPOINTER_TO_UINT()**

```c
#define GPOINTER_TO_UINT(p) ((guint) (p))
```

Extracts an unsigned integer from a pointer. The integer must have been stored in the pointer with **GUINT_TO_POINTER()**.

*p*: pointer to extract an unsigned integer from.

**GSIZE_TO_POINTER()**

```c
#define GSIZE_TO_POINTER(s) ((gpointer) (gsize) (s))
```

Stuffs a **gsize** into a pointer type.

*s*: **gsize** to stuff into the pointer.
2.6 Byte Order Macros

Name
Byte Order Macros – a portable way to convert between different byte orders

Synopsis

```
#include <glib.h>

#define G_BYTE_ORDER     (val)
#define G_LITTLE_ENDIAN  (val)
#define G_BIG_ENDIAN     (val)
#define G_PDP_ENDIAN     (val)

#define g_htonl (val)     (val)
#define g_htons (val)     (val)
#define g_ntohl (val)     (val)
#define g_ntohs (val)     (val)

#define GINT_FROM_BE (val)     (val)
#define GINT_FROM_LE (val)     (val)
#define GINT_TO_BE  (val)     (val)
#define GINT_TO_LE  (val)     (val)

#define GUINT_FROM_BE (val)     (val)
#define GUINT_FROM_LE (val)     (val)
#define GUINT_TO_BE  (val)     (val)
#define GUINT_TO_LE  (val)     (val)

#define GLONG_FROM_BE (val)     (val)
#define GLONG_FROM_LE (val)     (val)
#define GLONG_TO_BE  (val)     (val)
#define GLONG_TO_LE  (val)     (val)

#define GULONG_FROM_BE (val)     (val)
#define GULONG_FROM_LE (val)     (val)
#define GULONG_TO_BE  (val)     (val)
#define GULONG_TO_LE  (val)     (val)

#define GINT16_FROM_BE (val)     (val)
#define GINT16_FROM_LE (val)     (val)
#define GINT16_TO_BE  (val)     (val)
#define GINT16_TO_LE  (val)     (val)

#define GUINT16_FROM_BE (val)     (val)
#define GUINT16_FROM_LE (val)     (val)
#define GUINT16_TO_BE  (val)     (val)
#define GUINT16_TO_LE  (val)     (val)
```
CHAPTER 2. GLIB FUNDAMENTALS

2.6. BYTE ORDER MACROS

#define GINT32_FROM_BE (val)
#define GINT32_FROM_LE (val)
#define GINT32_TO_BE (val)
#define GINT32_TO_LE (val)
#define GUINT32_FROM_BE (val)
#define GUINT32_FROM_LE (val)
#define GUINT32_TO_BE (val)
#define GUINT32_TO_LE (val)
#define GINT64_FROM_BE (val)
#define GINT64_FROM_LE (val)
#define GINT64_TO_BE (val)
#define GINT64_TO_LE (val)
#define GUINT64_FROM_BE (val)
#define GUINT64_FROM_LE (val)
#define GUINT64_TO_BE (val)
#define GUINT64_TO_LE (val)
#define GUINT16_SWAP_BE_PDP (val)
#define GUINT16_SWAP_LE_BE (val)
#define GUINT16_SWAP_LE_PDP (val)
#define GUINT32_SWAP_BE_PDP (val)
#define GUINT32_SWAP_LE_BE (val)
#define GUINT32_SWAP_LE_PDP (val)
#define GUINT64_SWAP_LE_BE (val)

Description

These macros provide a portable way to determine the host byte order and to convert values between different byte orders.

The byte order is the order in which bytes are stored to create larger data types such as the gint and glong values. The host byte order is the byte order used on the current machine.

Some processors store the most significant bytes (i.e. the bytes that hold the largest part of the value) first. These are known as big-endian processors.

Other processors (notably the x86 family) store the most significant byte last. These are known as little-endian processors.

Finally, to complicate matters, some other processors store the bytes in a rather curious order known as PDP-endian. For a 4-byte word, the 3rd most significant byte is stored first, then the 4th, then the 1st and finally the 2nd.

Obviously there is a problem when these different processors communicate with each other, for example over networks or by using binary file formats. This is where these macros come in. They are typically used to convert values into a byte order which has been agreed on for use when communicating between different processors. The Internet uses what is known as ‘network byte order’ as the standard byte order (which is in fact the big-endian byte order).

Note that the byte order conversion macros may evaluate their arguments multiple times, thus you should not use them with arguments which have side-effects.

Details

G_BYTE_ORDER

#define G_BYTE_ORDER G_LITTLE_ENDIAN

The host byte order. This can be either G_LITTLE_ENDIAN or G_BIG_ENDIAN (support for G_PDP_ENDIAN may be added in future.)
### G_LITTLE_ENDIAN

```c
#define G_LITTLE_ENDIAN 1234
```

Specifies one of the possible types of byte order. See G_BYTE_ORDER.

### G_BIG_ENDIAN

```c
#define G_BIG_ENDIAN 4321
```

Specifies one of the possible types of byte order. See G_BYTE_ORDER.

### G_PDP_ENDIAN

```c
#define G_PDP_ENDIAN 3412 /* unused, need specific PDP check */
```

Specifies one of the possible types of byte order (currently unused). See G_BYTE_ORDER.

### g_htonl()

```c
#define g_htonl(val)
```

Converts a 32-bit integer value from host to network byte order.

**val**: a 32-bit integer value in host byte order.

**Returns**: @val converted to network byte order.

### g_ntohl()

```c
#define g_ntohl(val)
```

Converts a 32-bit integer value from network to host byte order.

**val**: a 32-bit integer value in network byte order.

**Returns**: @val converted to host byte order.

### g_htons()

```c
#define g_htons(val)
```

Converts a 16-bit integer value from host to network byte order.

**val**: a 16-bit integer value in host byte order.

**Returns**: @val converted to network byte order.

### g_ntohs()

```c
#define g_ntohs(val)
```

Converts a 16-bit integer value from network to host byte order.

**val**: a 16-bit integer value in network byte order.

**Returns**: @val converted to host byte order.
CHAPTER 2. GLIB FUNDAMENTALS 2.6. BYTE ORDER MACROS

GINT_FROM_BE()

#define GINT_FROM_BE(val) (GINT_TO_BE (val))

Converts a gint value from big-endian to host byte order.

val : a gint value in big-endian byte order.

Retruns : @val converted to host byte order.

GINT_FROM_LE()

#define GINT_FROM_LE(val) (GINT_TO_LE (val))

Converts a gint value from little-endian to host byte order.

val : a gint value in little-endian byte order.

Retruns : @val converted to host byte order.

GINT_TO_BE()

#define GINT_TO_BE(val) ((gint) GINT32_TO_BE (val))

Converts a gint value from host byte order to big-endian.

val : a gint value in host byte order.

Retruns : @val converted to big-endian byte order.

GINT_TO_LE()

#define GINT_TO_LE(val) ((gint) GINT32_TO_LE (val))

Converts a gint value from host byte order to little-endian.

val : a gint value in host byte order.

Retruns : @val converted to little-endian byte order.

GUINT_FROM_BE()

#define GUINT_FROM_BE(val) (GUINT_TO_BE (val))

Converts a guint value from big-endian to host byte order.

val : a guint value in big-endian byte order.

Retruns : @val converted to host byte order.

GUINT_FROM_LE()

#define GUINT_FROM_LE(val) (GUINT_TO_LE (val))

Converts a guint value from little-endian to host byte order.

val : a guint value in little-endian byte order.

Retruns : @val converted to host byte order.
CHAPTER 2. GLIB FUNDAMENTALS 2.6. BYTE ORDER MACROS

GUINT_TO_BE()

#define GUINT_TO_BE(val) ((guint) GUINT32_TO_BE(val))

Converts a guint value from host byte order to big-endian.

val: a guint value in host byte order.

Returns: @val converted to big-endian byte order.

GUINT_TO_LE()

#define GUINT_TO_LE(val) ((guint) GUINT32_TO_LE(val))

Converts a guint value from host byte order to little-endian.

val: a guint value in host byte order.

Returns: @val converted to little-endian byte order.

GLONG_FROM_BE()

#define GLONG_FROM_BE(val) (GLONG_TO_BE(val))

Converts a glong value from big-endian to the host byte order.

val: a glong value in big-endian byte order.

Returns: @val converted to host byte order.

GLONG_FROM_LE()

#define GLONG_FROM_LE(val) (GLONG_TO_LE(val))

Converts a glong value from little-endian to host byte order.

val: a glong value in little-endian byte order.

Returns: @val converted to host byte order.

GLONG_TO_BE()

#define GLONG_TO_BE(val) ((glong) GINT32_TO_BE(val))

Converts a glong value from host byte order to big-endian.

val: a glong value in host byte order.

Returns: @val converted to big-endian byte order.

GLONG_TO_LE()

#define GLONG_TO_LE(val) ((glong) GINT32_TO_LE(val))

Converts a glong value from host byte order to little-endian.

val: a glong value in host byte order.

Returns: @val converted to little-endian.
CHAPTER 2. GLIB FUNDAMENTALS

2.6. BYTE ORDER MACROS

GULONG_FROM_BE()

#define GULONG_FROM_BE(val) (GULONG_TO_BE (val))

Converts a gulong value from big-endian to host byte order.

val: a gulong value in big-endian byte order.

Returns: @val converted to host byte order.

GULONG_FROM_LE()

#define GULONG_FROM_LE(val) (GULONG_TO_LE (val))

Converts a gulong value from little-endian to host byte order.

val: a gulong value in little-endian byte order.

Returns: @val converted to host byte order.

GULONG_TO_BE()

#define GULONG_TO_BE(val) ((gulong) GUINT32_TO_BE (val))

Converts a gulong value from host byte order to big-endian.

val: a gulong value in host byte order.

Returns: @val converted to big-endian.

GULONG_TO_LE()

#define GULONG_TO_LE(val) ((gulong) GUINT32_TO_LE (val))

Converts a gulong value from host byte order to little-endian.

val: a gulong value in host byte order.

Returns: @val converted to little-endian.

GIN16_FROM_BE()

#define GINT16_FROM_BE(val) (GINT16_TO_BE (val))

Converts a gint16 value from big-endian to host byte order.

val: a gint16 value in big-endian byte order.

Returns: @val converted to host byte order.

GIN16_FROM_LE()

#define GINT16_FROM_LE(val) (GINT16_TO_LE (val))

Converts a gint16 value from little-endian to host byte order.

val: a gint16 value in little-endian byte order.

Returns: @val converted to host byte order.
CHAPTER 2. GLIB FUNDAMENTALS  

2.6. BYTE ORDER MACROS

GINT16_TO_BE()

#define GINT16_TO_BE(val) ((gint16) GUINT16_SWAP_LE_BE (val))

Converts a gint16 value from host byte order to big-endian.

val: a gint16 value in host byte order.

Returns: @val converted to big-endian.

GINT16_TO_LE()

#define GINT16_TO_LE(val) ((gint16) (val))

Converts a gint16 value from host byte order to little-endian.

val: a gint16 value in host byte order.

Returns: @val converted to little-endian.

GUINT16_FROM_BE()

#define GUINT16_FROM_BE(val) (GUINT16_TO_BE (val))

Converts a guint16 value from big-endian to host byte order.

val: a guint16 value in big-endian byte order.

Returns: @val converted to host byte order.

GUINT16_FROM_LE()

#define GUINT16_FROM_LE(val) (GUINT16_TO_LE (val))

Converts a guint16 value from little-endian to host byte order.

val: a guint16 value in little-endian byte order.

Returns: @val converted to host byte order.

GUINT16_TO_BE()

#define GUINT16_TO_BE(val) (GUINT16_SWAP_LE_BE (val))

Converts a guint16 value from host byte order to big-endian.

val: a guint16 value in host byte order.

Returns: @val converted to big-endian.

GUINT16_TO_LE()

#define GUINT16_TO_LE(val) ((guint16) (val))

Converts a guint16 value from host byte order to little-endian.

val: a guint16 value in host byte order.

Returns: @val converted to little-endian.
CHAPTER 2. GLIB FUNDAMENTALS 2.6. BYTE ORDER MACROS

GINT32_FROM_BE()

#define GINT32_FROM_BE(val) (GINT32_TO_BE (val))

Converts a gint32 value from big-endian to host byte order.

val: a gint32 value in big-endian byte order.

Returns: @val converted to host byte order.

GINT32_FROM_LE()

#define GINT32_FROM_LE(val) (GINT32_TO_LE (val))

Converts a gint32 value from little-endian to host byte order.

val: a gint32 value in little-endian byte order.

Returns: @val converted to host byte order.

GINT32_TO_BE()

#define GINT32_TO_BE(val) ((gint32) GUINT32_SWAP_LE_BE (val))

Converts a gint32 value from host byte order to big-endian.

val: a gint32 value in host byte order.

Returns: @val converted to big-endian.

GINT32_TO_LE()

#define GINT32_TO_LE(val) ((gint32) (val))

Converts a gint32 value from host byte order to little-endian.

val: a gint32 value in host byte order.

Returns: @val converted to little-endian.

GUINT32_FROM_BE()

#define GUINT32_FROM_BE(val) (GUINT32_TO_BE (val))

Converts a guint32 value from big-endian to host byte order.

val: a guint32 value in big-endian byte order.

Returns: @val converted to host byte order.

GUINT32_FROM_LE()

#define GUINT32_FROM_LE(val) (GUINT32_TO_LE (val))

Converts a guint32 value from little-endian to host byte order.

val: a guint32 value in little-endian byte order.

Returns: @val converted to host byte order.
CHAPTER 2. GLIB FUNDAMENTALS 2.6. BYTE ORDER MACROS

GUINT32_TO_BE()

#define GUINT32_TO_BE(val) (GUINT32_SWAP_LE_BE (val))

Converts a guint32 value from host byte order to big-endian.

val: a guint32 value in host byte order.

Returns: @val converted to big-endian.

GUINT32_TO_LE()

#define GUINT32_TO_LE(val) ((guint32) (val))

Converts a guint32 value from host byte order to little-endian.

val: a guint32 value in host byte order.

Returns: @val converted to little-endian.

GINT64_FROM_BE()

#define GINT64_FROM_BE(val) (GINT64_TO_BE (val))

Converts a gint64 value from big-endian to host byte order.

val: a gint64 value in big-endian byte order.

Returns: @val converted to host byte order.

GINT64_FROM_LE()

#define GINT64_FROM_LE(val) (GINT64_TO_LE (val))

Converts a gint64 value from little-endian to host byte order.

val: a gint64 value in little-endian byte order.

Returns: @val converted to host byte order.

GINT64_TO_BE()

#define GINT64_TO_BE(val) ((gint64) GUINT64_SWAP_LE_BE (val))

Converts a gint64 value from host byte order to big-endian.

val: a gint64 value in host byte order.

Returns: @val converted to big-endian.

GINT64_TO_LE()

#define GINT64_TO_LE(val) ((gint64) (val))

Converts a gint64 value from host byte order to little-endian.

val: a gint64 value in host byte order.

Returns: @val converted to little-endian.
CHAPTER 2. GLIB FUNDAMENTALS  2.6. BYTE ORDER MACROS

GUINT64_FROM_BE()

#define GUINT64_FROM_BE(val) (GUINT64_TO_BE (val))

Converts a guint64 value from big-endian to host byte order.

_val: a guint64 value in big-endian byte order.

_Returns_: @val converted to host byte order.

GUINT64_FROM_LE()

#define GUINT64_FROM_LE(val) (GUINT64_TO_LE (val))

Converts a guint64 value from little-endian to host byte order.

_val: a guint64 value in little-endian byte order.

_Returns_: @val converted to host byte order.

GUINT64_TO_BE()

#define GUINT64_TO_BE(val) (GUINT64_SWAP_LE_BE (val))

Converts a guint64 value from host byte order to big-endian.

_val: a guint64 value in host byte order.

_Returns_: @val converted to big-endian.

GUINT64_TO_LE()

#define GUINT64_TO_LE(val) ((guint64) (val))

Converts a guint64 value from host byte order to little-endian.

_val: a guint64 value in host byte order.

_Returns_: @val converted to little-endian.

GUINT16_SWAP_BE_PDP()

#define GUINT16_SWAP_BE_PDP(val) (GUINT16_SWAP_LE_BE (val))

Converts a guint16 value between big-endian and pdp-endian byte order. The conversion is symmetric so it can be used both ways.

_val: a guint16 value in big-endian or pdp-endian byte order.

_Returns_: @val converted to the opposite byte order.

GUINT16_SWAP_LE_BE()

#define GUINT16_SWAP_LE_BE(val)

Converts a guint16 value between little-endian and big-endian byte order. The conversion is symmetric so it can be used both ways.

_val: a guint16 value in little-endian or big-endian byte order.

_Returns_: @val converted to the opposite byte order.
2.7 Numerical Definitions

Name

Numerical Definitions – mathematical constants, and floating point decomposition
Synopsis

```c
#include <glib.h>

#define G_IEEE754_FLOAT_BIAS
#define G_IEEE754_DOUBLE_BIAS
union GFloatIEEE754;
union GDoubleIEEE754;

#define G_E
#define G_LN2
#define G_LN10
#define G_PI
#define G_PI_2
#define G_PI_4
#define G_SQRT2
#define G_LOG_2_BASE_10
```

Description

GLib offers mathematical constants such as `G_PI` for the value of pi; many platforms have these in the C library, but some don’t, the GLib versions always exist.

The `GFloatIEEE754` and `GDoubleIEEE754` unions are used to access the sign, mantissa and exponent of IEEE floats and doubles. These unions are defined as appropriate for a given platform. IEEE floats and doubles are supported (used for storage) by at least Intel, PPC and Sparc, for reference: [http://cch.loria.fr/documentation/IEEE754/numerical_comp_guide/ncg_math.doc.html](http://cch.loria.fr/documentation/IEEE754/numerical_comp_guide/ncg_math.doc.html)

Details

**G_IEEE754_FLOAT_BIAS**

```c
#define G_IEEE754_FLOAT_BIAS (127)
```


**G_IEEE754_DOUBLE_BIAS**

```c
#define G_IEEE754_DOUBLE_BIAS (1023)
```


union GFloatIEEE754

```c
union GFloatIEEE754
{
    gchar v_float;
    struct {
        guint mantissa : 23;
        guint biased_exponent : 8;
        guint sign : 1;
    } mpn;
};
```

The `GFloatIEEE754` and `GDoubleIEEE754` unions are used to access the sign, mantissa and exponent of IEEE floats and doubles. These unions are defined as appropriate for a given platform. IEEE floats and doubles are supported (used for storage) by at least Intel, PPC and Sparc, for reference: [http://cch.loria.fr/documentation/IEEE754/numerical_comp_guide/ncg_math.doc.html](http://cch.loria.fr/documentation/IEEE754/numerical_comp_guide/ncg_math.doc.html)
union GDoubleIEEE754

union GDoubleIEEE754
{
    gdouble v_double;
    struct {
        guint mantissa_low : 32;
        guint mantissa_high : 20;
        guint biased_exponent : 11;
        guint sign : 1;
    } mpn;
};

The GFloatIEEE754 and GDoubleIEEE754 unions are used to access the sign, mantissa and exponent of IEEE floats and doubles. These unions are defined as appropriate for a given platform. IEEE floats and doubles are supported (used for storage) by at least Intel, PPC and Sparc, for reference: http://cch.loria.fr/documentation/IEEE754/numerical_comp_guide/ncg_math.doc.html

G_E

#define G_E 2.7182818284590452353602874713526624977572747093700

The base of natural logarithms.

G_LN2

#define G_LN2 0.69314718055994530941723212145817656807550013436026

The natural logarithm of 2.

G_LN10

#define G_LN10 2.3025850929940456843642076011014886288

The natural logarithm of 10.

G_PI

#define G_PI 3.1415926535897932384626433832795028841971693993751

The value of pi (ratio of circle’s circumference to its diameter).

G_PI_2

#define G_PI_2 1.5707963267948966192313216916397514420985846996876

Pi divided by 2.

G_PI_4

#define G_PI_4 0.78539816339744830961566084581987572104929234984378

Pi divided by 4.

G_SQRT2

#define G_SQRT2 1.4142135623730950488016887242096980785696718753769

The square root of two.
CHAPTER 2. GLIB FUNDAMENTALS

2.8. MISCELLANEOUS MACROS

G_LOG_2_BASE_10

#define G_LOG_2_BASE_10 (0.30102999566398119521)

Used for fooling around with float formats, see http://cch.loria.fr/documentation/IEEE754/numerical_comp_guide/ncg_math.doc.html

See Also


2.8 Miscellaneous Macros

Name

Miscellaneous Macros – specialized macros which are not used often

Synopsis

#include <glib.h>

#define G_INLINE_FUNC

#define G_STMT_START

#define G_STMT_END

#define G_BEGIN_DECLS

#define G_END_DECLS

#define G_N_ELEMENTS (arr)

#define G_VA_COPY (ap1,ap2)

#define G_STRINGIFY (macro_or_string)

#define G_PASTE (identifier1,identifier2)

#define G_PASTE_ARGS (identifier1,identifier2)

#define G_STATIC_ASSERT (expr)

#define G_GNUC_EXTENSION

#define G_GNUC_CONST

#define G_GNUC_PURE

#define G_GNUC_MALLOC

#define G_GNUC_ALLOC_SIZE (x)

#define G_GNUC_ALLOC_SIZE2 (x,y)

#define G_GNUC_DEPRECATED

#define G_GNUC_NORETURN

#define G_GNUC_UNUSED

#define G_GNUC_PRINTF ( format_idx, arg_idx )

#define G_GNUC_SCANF ( format_idx, arg_idx )

#define G_GNUC_FORMAT ( arg_idx )

#define G_GNUC_NULL_TERMINATED

#define G_GNUC_WARN_UNUSED_RESULT

#define G_GNUC_FUNCTION

#define G_GNUC_PRETTY_FUNCTION

#define G_GNUC_NO_INSTRUMENT

#define G_HAVE_GNUC_VISIBILITY

#define G_GNUC_INTERNAL
#define G_GNUC_MAY_ALIAS

if G_LIKELY ()
#define G_UNLIKELY (expr)

#define G_STRLOC
#define G_STRFUNC

#define G_GINT16_MODIFIER
#define G_GINT16_FORMAT
#define G_GUINT16_MODIFIER
#define G_GUINT16_FORMAT
#define G_GINT32_MODIFIER
#define G_GINT32_FORMAT
#define G_GUINT32_FORMAT
#define G_GINT64_MODIFIER
#define G_GINT64_FORMAT
#define G_GUINT64_FORMAT
#define G_GSIZE_MODIFIER
#define G_GSIZE_FORMAT
#define G_GSSIZE_FORMAT
#define G_GOFFSET_MODIFIER
#define G_GOFFSET_FORMAT

Description

These macros provide more specialized features which are not needed so often by application programmers.

Details

G_INLINE_FUNC

#define G_INLINE_FUNC

This macro is used to export function prototypes so they can be linked with an external version when no inlining is performed. The file which implements the functions should define G_IMPLEMENTED_INLINES before including the headers which contain G_INLINE_FUNC declarations. Since inlining is very compiler-dependent using these macros correctly is very difficult. Their use is strongly discouraged.

This macro is often mistaken for a replacement for the inline keyword; inline is already declared in a portable manner in the glib headers and can be used normally.

G_STMT_START

#define G_STMT_START do

Used within multi-statement macros so that they can be used in places where only one statement is expected by the compiler.

G_STMT_END

#define G_STMT_END while (0)

Used within multi-statement macros so that they can be used in places where only one statement is expected by the compiler.
G_BEGIN_DECLS

#define G_BEGIN_DECLS

Used (along with G_END_DECLS) to bracket header files. If the compiler in use is a C++ compiler, adds extern "C" around the header.

G_END_DECLS

#define G_END_DECLS

Used (along with G_BEGIN_DECLS) to bracket header files. If the compiler in use is a C++ compiler, adds extern "C" around the header.

G_N_ELEMENTS()

#define G_N_ELEMENTS(arr) (sizeof (arr) / sizeof ((arr)[0]))

Determines the number of elements in an array. The array must be declared so the compiler knows its size at compile-time; this macro will not work on an array allocated on the heap, only static arrays or arrays on the stack.

arr: the array

G_VA_COPY()

#define G_VA_COPY(ap1,ap2)

Portable way to copy va_list variables.

In order to use this function, you must include string.h yourself, because this macro may use memmove() and GLib does not include string.h for you.

ap1: the va_list variable to place a copy of ap2 in.
ap2: a va_list.

G_STRINGIFY()

#define G_STRINGIFY(macro_or_string) G_STRINGIFY_ARG (macro_or_string)

Accepts a macro or a string and converts it into a string after preprocessor argument expansion.

macro_or_string: a macro or a string.

G_PASTE()

#define G_PASTE(identifier1,identifier2) G_PASTE_ARGS (identifier1, identifier2)

Yields a new preprocessor pasted identifier ‘identifier1identifier2’ from its expanded arguments ‘identifier1’ and ‘identifier2’.

identifier1: an identifier

identifier2: an identifier

Since 2.20
G_PASTE_ARGS()

#define G_PASTE_ARGS(identifier1, identifier2) identifier1 ## identifier2

identifier1:
identifier2:

G_STATIC_ASSERT()

#define G_STATIC_ASSERT(expr) typedef struct { char Compile_Time_Assertion[(expr) ←
? 1 : -1]; } G_PASTE(_GStaticAssert_, __LINE__)  #include <string.h>

The G_STATIC_ASSERT macro lets the programmer check a condition at compile time, the condition needs to be compile time computable. The macro can be used in any place where a typedef is valid. The macro should only be used once per source code line.

expr: a constant expression.

Since 2.20

G_GNUC_EXTENSION

#define G_GNUC_EXTENSION

Expands to __extension__ when gcc is used as the compiler. This simply tells gcc not to warn about the following non-standard code when compiling with the -pedantic option.

G_GNUC_CONST

#define G_GNUC_CONST

Expands to the GNU C const function attribute if the compiler is gcc. Declaring a function as const enables better optimization of calls to the function. A const function doesn’t examine any values except its parameters, and has no effects except its return value. See the GNU C documentation for details.

NOTE

A function that has pointer arguments and examines the data pointed to must not be declared const. Likewise, a function that calls a non-const function usually must not be const. It doesn’t make sense for a const function to return void.

G_GNUC_PURE

#define G_GNUC_PURE

Expands to the GNU C pure function attribute if the compiler is gcc. Declaring a function as pure enables better optimization of calls to the function. A pure function has no effects except its return value and the return value depends only on the parameters and/or global variables. See the GNU C documentation for details.
G_GNUC_MALLOC

#define G_GNUC_MALLOC

Expands to the GNU C malloc function attribute if the compiler is gcc. Declaring a function as malloc enables better optimization of the function. A function can have the malloc attribute if it returns a pointer which is guaranteed to not alias with any other pointer when the function returns (in practice, this means newly allocated memory). See the GNU C documentation for details.

Since 2.6

G_GNUC_ALLOC_SIZE()

#define G_GNUC_ALLOC_SIZE(x)

Expands to the GNU C alloc_size function attribute if the compiler is a new enough gcc. This attribute tells the compiler that the function returns a pointer to memory of a size that is specified by the $x$th function parameter. See the GNU C documentation for details.

$x$: the index of the argument specifying the allocation size

Since 2.18

G_GNUC_ALLOC_SIZE2()

#define G_GNUC_ALLOC_SIZE2(x,y)

Expands to the GNU C alloc_size function attribute if the compiler is a new enough gcc. This attribute tells the compiler that the function returns a pointer to memory of a size that is specified by the product of two function parameters. See the GNU C documentation for details.

$x$: the index of the argument specifying one factor of the allocation size

$y$: the index of the argument specifying the second factor of the allocation size

Since 2.18

G_GNUC_DEPRECATED

#define G_GNUC_DEPRECATED

Expands to the GNU C deprecated attribute if the compiler is gcc. It can be used to mark typedefs, variables and functions as deprecated. When called with the -Wdeprecated option, the compiler will generate warnings when deprecated interfaces are used. See the GNU C documentation for details.

Since 2.2

G_GNUC_NORETURN

#define G_GNUC_NORETURN

Expands to the GNU C noreturn function attribute if the compiler is gcc. It is used for declaring functions which never return. It enables optimization of the function, and avoids possible compiler warnings. See the GNU C documentation for details.

G_GNUC_UNUSED

#define G_GNUC_UNUSED

Expands to the GNU C unused function attribute if the compiler is gcc. It is used for declaring functions which may never be used. It avoids possible compiler warnings. See the GNU C documentation for details.
CHAPTER 2. GLIB FUNDAMENTALS  2.8. MISCELLANEOUS MACROS

G_GNUC_PRINTF()

#define G_GNUC_PRINTF( format_idx, arg_idx )

Expands to the GNU C format function attribute if the compiler is gcc. This is used for declaring functions which take a variable number of arguments, with the same syntax as printf(). It allows the compiler to type-check the arguments passed to the function. See the GNU C documentation for details.

gint g_snprintf (gchar *string,
    gulong n,
    gchar const *format,
    ...) G_GNUC_PRINTF (3, 4);

format_idx: the index of the argument corresponding to the format string. (The arguments are numbered from 1).

arg_idx: the index of the first of the format arguments.

G_GNUC_SCANF()

#define G_GNUC_SCANF( format_idx, arg_idx )

Expands to the GNU C format function attribute if the compiler is gcc. This is used for declaring functions which take a variable number of arguments, with the same syntax as scanf(). It allows the compiler to type-check the arguments passed to the function. See the GNU C documentation for details.

format_idx: the index of the argument corresponding to the format string. (The arguments are numbered from 1).

arg_idx: the index of the first of the format arguments.

G_GNUC_FORMAT()

#define G_GNUC_FORMAT( arg_idx )

Expands to the GNU C format_arg function attribute if the compiler is gcc. This function attribute specifies that a function takes a format string for a printf(), scanf(), strftime() or strftime() style function and modifies it, so that the result can be passed to a printf(), scanf(), strftime() or strftime() style function (with the remaining arguments to the format function the same as they would have been for the unmodified string). See the GNU C documentation for details.

gchar *g_dgettext (gchar *domain_name, gchar *msgid) G_GNUC_FORMAT (2);

arg_idx: the index of the argument.

G_GNUC_NULL_TERMINATED

#define G_GNUC_NULL_TERMINATED

Expands to the GNU C sentinel function attribute if the compiler is gcc, or "" if it isn’t. This function attribute only applies to variadic functions and instructs the compiler to check that the argument list is terminated with an explicit NULL. See the GNU C documentation for details.

Since: 2.8

G_GNUC_WARN_UNUSED_RESULT

#define G_GNUC_WARN_UNUSED_RESULT

Expands to the GNU C warn_unused_result function attribute if the compiler is gcc, or "" if it isn’t. This function attribute makes the compiler emit a warning if the result of a function call is ignored. See the GNU C documentation for details.

Since: 2.10
### G_GNUC_FUNCTION

```c
#define G_GNUC_FUNCTION
```

**Warning**

G_GNUC_FUNCTION is deprecated and should not be used in newly-written code. 2.16

Expands to "" on all modern compilers, and to __FUNCTION__ on gcc version 2.x. Don't use it.

### G_GNUC_PRETTY_FUNCTION

```c
#define G_GNUC_PRETTY_FUNCTION
```

**Warning**

G_GNUC_PRETTY_FUNCTION is deprecated and should not be used in newly-written code. 2.16

Expands to "" on all modern compilers, and to __PRETTY_FUNCTION__ on gcc version 2.x. Don’t use it.

### G_GNUC_NO_INSTRUMENT

```c
#define G_GNUC_NO_INSTRUMENT
```

Expands to the GNU C no_instrument_function function attribute if the compiler is gcc. Functions with this attribute will not be instrumented for profiling, when the compiler is called with the -finstrument-functions option. See the GNU C documentation for details.

### G_HAVE_GNUC_VISIBILITY

```c
#define G_HAVE_GNUC_VISIBILITY 1
```

### G_GNUC_INTERNAL

```c
#define G_GNUC_INTERNAL
```

This attribute can be used for marking library functions as being used internally to the library only, which may allow the compiler to handle function calls more efficiently. Note that static functions do not need to be marked as internal in this way. See the GNU C documentation for details.

When using a compiler that supports the GNU C hidden visibility attribute, this macro expands to

```c
__attribute__((visibility("hidden")))
```

When using the Sun Studio compiler, it expands to

```c
__hidden
```

Note that for portability, the attribute should be placed before the function declaration. While GCC allows the macro after the declaration, Sun Studio does not.
CHAPTER 2. GLIB FUNDAMENTALS  2.8. MISCELLANEOUS MACROS

G_GNUC_INTERNAL

void _g_log_fallback_handler (const gchar *log_domain,
   GLogLevelFlags log_level,
   const gchar *message,
   gpointer unused_data);

Since: 2.6

G_GNUC_MAY_ALIAS

#define G_GNUC_MAY_ALIAS

Expands to the GNU C may_alias type attribute if the compiler is gcc. Types with this attribute will not be subjected to type-based alias analysis, but are assumed to alias with any other type, just like char. See the GNU C documentation for details.

Since: 2.14

G_LIKELY()

if G_LIKELY ();

Hints the compiler that the expression is likely to evaluate to a true value. The compiler may use this information for optimizations.

if (G_LIKELY (random () != 1))
g_print("not one");

Returns: the value of expr

Since 2.2

G_UNLIKELY()

#define G_UNLIKELY(expr)

Hints the compiler that the expression is unlikely to evaluate to a true value. The compiler may use this information for optimizations.

if (G_UNLIKELY (random () == 1))
g_print("a random one");

expr: the expression

Returns: the value of expr

Since 2.2

G_STRLOC

#define G_STRLOC

Expands to a string identifying the current code position.

G_STRFUNC

#define G_STRFUNC

Expands to a string identifying the current function.

Since 2.4
### G_GINT16_MODIFIER

```c
#define G_GINT16_MODIFIER "h"
```

The platform dependent length modifier for conversion specifiers for scanning and printing values of type `gint16` or `guint16`. It is a string literal, but doesn’t include the percent-sign, such that you can add precision and length modifiers between percent-sign and conversion specifier and append a conversion specifier.

The following example prints "0x7b":

```c
gint16 value = 123;
g_print("%" G_GINT16_MODIFIER "x", value);
```

Since 2.4

### G_GINT16_FORMAT

```c
#define G_GINT16_FORMAT "hi"
```

This is the platform dependent conversion specifier for scanning and printing values of type `gint16`. It is a string literal, but doesn’t include the percent-sign, such that you can add precision and length modifiers between percent-sign and conversion specifier.

```c
gint16 in;
gint32 out;
sscanf("42", "%" G_GINT16_FORMAT, &in)
out = in * 1000;
g_print("%" G_GINT32_FORMAT, out);
```

### G_GUINT16_FORMAT

```c
#define G_GUINT16_FORMAT "hu"
```

This is the platform dependent conversion specifier for scanning and printing values of type `guint16`. See also `G_GINT16_FORMAT`.

### G_GINT32_MODIFIER

```c
#define G_GINT32_MODIFIER ""
```

The platform dependent length modifier for conversion specifiers for scanning and printing values of type `gint32` or `guint32`. It is a string literal, See also `G_GINT16_MODIFIER`.

Since 2.4

### G_GINT32_FORMAT

```c
#define G_GINT32_FORMAT "i"
```

This is the platform dependent conversion specifier for scanning and printing values of type `gint32`. See also `G_GINT16_FORMAT`.

### G_GUINT32_FORMAT

```c
#define G_GUINT32_FORMAT "u"
```

This is the platform dependent conversion specifier for scanning and printing values of type `guint32`. See also `G_GINT16_FORMAT`. 

71
G_GINT64_MODIFIER

#define G_GINT64_MODIFIER "ll"

The platform dependent length modifier for conversion specifiers for scanning and printing values of type gint64 or guint64. It is a string literal.

NOTE

Some platforms do not support printing 64 bit integers, even though the types are supported. On such platforms G_GINT64_MODIFIER is not defined.

Since 2.4

G_GINT64_FORMAT

#define G_GINT64_FORMAT "lll"

This is the platform dependent conversion specifier for scanning and printing values of type gint64. See also G_GINT16_FORMAT.

NOTE

Some platforms do not support scanning and printing 64 bit integers, even though the types are supported. On such platforms G_GINT64_FORMAT is not defined. Note that scanf() may not support 64 bit integers, even if G_GINT64_FORMAT is defined. Due to its weak error handling, scanf() is not recommended for parsing anyway; consider using g_ascii_strtoull() instead.

G_GUINT64_FORMAT

#define G_GUINT64_FORMAT "llu"

This is the platform dependent conversion specifier for scanning and printing values of type guint64. See also G_GINT16_FORMAT.

NOTE

Some platforms do not support scanning and printing 64 bit integers, even though the types are supported. On such platforms G_GUINT64_FORMAT is not defined. Note that scanf() may not support 64 bit integers, even if G_GINT64_FORMAT is defined. Due to its weak error handling, scanf() is not recommended for parsing anyway; consider using g_strtoull() instead.

G_GSIZE_MODIFIER

#define G_GSIZE_MODIFIER ""

The platform dependent length modifier for conversion specifiers for scanning and printing values of type gsize or gssize. It is a string literal.

Since 2.6
CHAPTER 2. GLIB FUNDAMENTALS

2.9 Atomic Operations

Name
Atomic Operations – basic atomic integer and pointer operations

Synopsis

#include <glib.h>

gint g_atomic_int_get ();
void g_atomic_int_set ();
void g_atomic_int_add ();
gint g_atomic_int_exchange_and_add ();
gboolean g_atomic_int_compare_and_exchange ();
gpointer g_atomic_pointer_get ();
void g_atomic_pointer_set ();
gboolean g_atomic_pointer_compare_and_exchange ();
void g_atomic_int_inc (gint *atomic);
gboolean g_atomic_int_dec_and_test (gint *atomic);
CHAPTER 2. GLIB FUNDAMENTALS 2.9. ATOMIC OPERATIONS

Description
The following functions can be used to atomically access integers and pointers. They are implemented as inline assembler function on most platforms and use slower fall-backs otherwise. Using them can sometimes save you from using a performance-expensive GMutex to protect the integer or pointer.

The most important usage is reference counting. Using g_atomic_int_inc() and g_atomic_int_dec_and_test() makes reference counting a very fast operation.

NOTE
You must not directly read integers or pointers concurrently accessed by multiple threads, but use the atomic accessor functions instead. That is, always use g_atomic_int_get() and g_atomic_pointer_get() for read outs. They provide the neccessary synchronization mechanisms like memory barriers to access memory locations concurrently.

NOTE
If you are using those functions for anything apart from simple reference counting, you should really be aware of the implications of doing that. There are literally thousands of ways to shoot yourself in the foot. So if in doubt, use a GMutex. If you don't know, what memory barriers are, do not use anything but g_atomic_int_inc() and g_atomic_int_dec_and_test().

NOTE
It is not safe to set an integer or pointer just by assigning to it, when it is concurrently accessed by other threads with the following functions. Use g_atomic_int_compare_and_exchange() or g_atomic_pointer_compare_and_exchange() respectively.

Details

**g_atomic_int_get()**

```c
gint g_atomic_int_get ();
```

Reads the value of the integer pointed to by *atomic*. Also acts as a memory barrier.

**Returns**: the value of *atomic*

Since 2.4

**g_atomic_int_set()**

```c
void g_atomic_int_set ();
```

Sets the value of the integer pointed to by *atomic*. Also acts as a memory barrier.

Since 2.10

**g_atomic_int_add()**

```c
void g_atomic_int_add ();
```
Atomically adds `val` to the integer pointed to by `atomic`. Also acts as a memory barrier.
Since 2.4

`g_atomic_int_exchange_and_add ()`

```c
gint g_atomic_int_exchange_and_add ()
```

Atomically adds `val` to the integer pointed to by `atomic`. It returns the value of *`atomic` just before the addition took place. Also acts as a memory barrier.

**Returns**: the value of *`atomic` before the addition.
Since 2.4

`g_atomic_int_compare_and_exchange ()`

```c
gboolean g_atomic_int_compare_and_exchange ()
```

Compares `oldval` with the integer pointed to by `atomic` and if they are equal, atomically exchanges *`atomic` with `newval`. Also acts as a memory barrier.

**Returns**: %TRUE, if *`atomic` was equal `oldval`. FALSE otherwise.
Since 2.4

`g_atomic_pointer_get ()`

```c
gpointer g_atomic_pointer_get ()
```

Reads the value of the pointer pointed to by `atomic`. Also acts as a memory barrier.

**Returns**: the value to add to *`atomic`.
Since 2.4

`g_atomic_pointer_set ()`

```c
void g_atomic_pointer_set ()
```

Sets the value of the pointer pointed to by `atomic`. Also acts as a memory barrier.
Since 2.10

`g_atomic_pointer_compare_and_exchange ()`

```c
gboolean g_atomic_pointer_compare_and_exchange ()
```

Compares `oldval` with the pointer pointed to by `atomic` and if they are equal, atomically exchanges *`atomic` with `newval`. Also acts as a memory barrier.

**Returns**: %TRUE, if *`atomic` was equal `oldval`. FALSE otherwise.
Since 2.4

`g_atomic_int_inc ()`

```c
void g_atomic_int_inc (gint *atomic);
```

Atomically increments the integer pointed to by `atomic` by 1.

*atomic*: a pointer to an integer.
Since 2.4
g_atomic_int_dec_and_test()

gboolean g_atomic_int_dec_and_test (gint *atomic);

Atomically decrements the integer pointed to by *atomic by 1.

*atomic: a pointer to an integer.

Returns: %TRUE, if the integer pointed to by *atomic is 0 after decrementing it.

Since 2.4

See Also

GMutex  GLib mutual exclusions.
Chapter 3

GLib Core Application Support

3.1 The Main Event Loop

Name
The Main Event Loop – manages all available sources of events

Synopsis

```c
#include <glib.h>

GMainLoop;
GMainLoop * g_main_loop_new (GMainContext *context, gboolean is_running);
GMainLoop * g_main_loop_ref (GMainLoop *loop);
void g_main_loop_unref (GMainLoop *loop);
void g_main_loop_run (GMainLoop *loop);
void g_main_loop_quit (GMainLoop *loop);
gboolean g_main_loop_is_running (GMainLoop *loop);
GMainContext * g_main_loop_get_context (GMainLoop *loop);
#define g_main_new (is_running)
#define g_main_destroy (loop)
#define g_main_run (loop)
#define g_main_quit (loop)
#define g_main_is_running (loop)

#define G_PRIORITY_HIGH
#define G_PRIORITY_DEFAULT
#define G_PRIORITY_HIGH_IDLE
#define G_PRIORITY_DEFAULT_IDLE
#define G_PRIORITY_LOW

GMainContext;
GMainContext * g_main_context_new (void);
GMainContext * g_main_context_ref (GMainContext *context);
void g_main_context_unref (GMainContext *context);
GMainContext * g_main_context_default (void);
gboolean g_main_context_iteration (GMainContext *context, gboolean may_block);
#define g_main_iteration (may_block)
#define g_main_pending ()
GSource * g_main_context_find_source_by_id (GMainContext *context, guint source_id);
```

77
### 3.1. THE MAIN EVENT LOOP

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GSource * g_main_context_find_source_by_user_data</td>
<td>(GMainContext *context, gpointer user_data);</td>
</tr>
<tr>
<td>GSource * g_main_context_find_source_by_funcs_user_data</td>
<td>(GMainContext *context, GSourceFuncs *funcs, gpointer user_data);</td>
</tr>
<tr>
<td>void g_main_context_wakeup</td>
<td>(GMainContext *context);</td>
</tr>
<tr>
<td>gboolean g_main_context_acquire</td>
<td>(GMainContext *context);</td>
</tr>
<tr>
<td>void g_main_context_release</td>
<td>(GMainContext *context);</td>
</tr>
<tr>
<td>gboolean g_main_context_is_owner</td>
<td>(GMainContext *context);</td>
</tr>
<tr>
<td>gboolean g_main_context_wait</td>
<td>(GMainContext *context, GCond *cond, GMutex *mutex);</td>
</tr>
<tr>
<td>gboolean g_main_context_prepare</td>
<td>(GMainContext *context, gint *priority);</td>
</tr>
<tr>
<td>gint g_main_context_query</td>
<td>(GMainContext *context, gint max_priority, gint *timeout_, GPollFD *fds, gint n_fds);</td>
</tr>
<tr>
<td>gint g_main_context_check</td>
<td>(GMainContext *context, gint max_priority, GPollFD *fds, gint n_fds);</td>
</tr>
<tr>
<td>void g_main_context_dispatch</td>
<td>(GMainContext *context);</td>
</tr>
<tr>
<td>void g_main_context_set_poll_func</td>
<td>(GMainContext *context, GPollFunc func);</td>
</tr>
<tr>
<td>GPollFunc g_main_context_get_poll_func</td>
<td>(GMainContext *context);</td>
</tr>
<tr>
<td>gint (*GPollFunc)</td>
<td>(GPollFD *ufds, guint nfds, gint timeout_);</td>
</tr>
<tr>
<td>void g_main_context_add_poll</td>
<td>(GMainContext *context, GPollFD *fd, gint priority);</td>
</tr>
<tr>
<td>void g_main_context_remove_poll</td>
<td>(GMainContext *context, GPollFD *fd);</td>
</tr>
<tr>
<td>gint g_main_depth</td>
<td>(void);</td>
</tr>
<tr>
<td>GSource * g_main_current_source</td>
<td>(void);</td>
</tr>
<tr>
<td>#define g_main_set_poll_func</td>
<td>(func)</td>
</tr>
<tr>
<td>GSource * g_timeout_source_new</td>
<td>(guint interval);</td>
</tr>
<tr>
<td>GSource * g_timeout_source_new_seconds</td>
<td>(guint interval);</td>
</tr>
<tr>
<td>guint g_timeout_add</td>
<td>(guint interval, GSourceFunc function, gpointer data);</td>
</tr>
<tr>
<td>guint g_timeout_add_full</td>
<td>(guint priority, guint interval, GSourceFunc function, gpointer data, GDestroyNotify notify);</td>
</tr>
<tr>
<td>guint g_timeout_add_seconds</td>
<td>(guint priority, guint interval, GSourceFunc function, gpointer data, GDestroyNotify notify);</td>
</tr>
<tr>
<td>guint g_timeout_add_seconds_full</td>
<td>(guint priority, guint interval, GSourceFunc function, gpointer data, GDestroyNotify notify);</td>
</tr>
</tbody>
</table>
3.1. THE MAIN EVENT LOOP

GSource * g_idle_source_new (void);

gboolean g_idle_remove_by_data (gpointer data);

typedef GPid;

void (*GChildWatchFunc) (GPid pid, gint status, gpointer data);

GSource * g_child_watch_source_new (GPid pid);

void (*GSourceDummyMarshal) (void);

GSourceFuncs;

GSource * g_source_new (GSourceFuncs *source_funcs, guint struct_size);

GSource * g_source_ref (GSource *source);

void g_source_unref (GSource *source);

void g_source_set_funcs (GSource *source, GSourceFuncs *funcs);

gboolean g_source_is_destroyed (GSource *source);

GMainContext * g_source_get_context (GSource *source);

gboolean (*GSourceFunc) (gpointer data);

gboolean g_source_set_callback_indirect (GSource *source, gpointer callback_data, GSourceCallbackFuncs *callback_funcs);

GSource * g_source_set_callback (GSource *source, GSourceFunc func, gpointer data, GDestroyNotify notify);

gboolean g_source_get_can_recurse (GSource *source);

guint g_source_get_id (GSource *source);

void g_source_set_can_recurse (GSource *source, gboolean can_recurse);

gboolean g_source_get_can_recurse (GSource *source);

gboolean g_source_set_callback_indirect (GSource *source, gpointer callback_data, GSourceCallbackFuncs *callback_funcs);

GSource * g_source_set_callback (GSource *source, GSourceFunc func, gpointer data, GDestroyNotify notify);

gboolean g_source_get_can_recurse (GSource *source);

gboolean g_source_set_callback_indirect (GSource *source, gpointer callback_data, GSourceCallbackFuncs *callback_funcs);

GSource * g_source_set_callback (GSource *source, GSourceFunc func, gpointer data, GDestroyNotify notify);

gboolean g_source_get_can_recurse (GSource *source);

gboolean g_source_set_callback_indirect (GSource *source, gpointer callback_data, GSourceCallbackFuncs *callback_funcs);

GSource * g_source_set_callback (GSource *source, GSourceFunc func, gpointer data, GDestroyNotify notify);

gboolean g_source_get_can_recurse (GSource *source);

gboolean g_source_set_callback_indirect (GSource *source, gpointer callback_data, GSourceCallbackFuncs *callback_funcs);

GSource * g_source_set_callback (GSource *source, GSourceFunc func, gpointer data, GDestroyNotify notify);

gboolean g_source_get_can_recurse (GSource *source);

gboolean g_source_set_callback_indirect (GSource *source, gpointer callback_data, GSourceCallbackFuncs *callback_funcs);
Description

The main event loop manages all the available sources of events for GLib and GTK+ applications. These events can come from any number of different types of sources such as file descriptors (plain files, pipes or sockets) and timeouts. New types of event sources can also be added using `g_source_attach()`.

To allow multiple independent sets of sources to be handled in different threads, each source is associated with a `GMainContext`. A `GMainContext` can only be running in a single thread, but sources can be added to it and removed from it from other threads.

Each event source is assigned a priority. The default priority, `G_PRIORITY_DEFAULT`, is 0. Values less than 0 denote higher priorities. Values greater than 0 denote lower priorities. Events from high priority sources are always processed before events from lower priority sources.

Idle functions can also be added, and assigned a priority. These will be run whenever no events with a higher priority are ready to be processed.

The `GMainLoop` data type represents a main event loop. A `GMainLoop` is created with `g_main_loop_new()`. After adding the initial event sources, `g_main_loop_run()` is called. This continuously checks for new events from each of the event sources and dispatches them. Finally, the processing of an event from one of the sources leads to a call to `g_main_loop_quit()` to exit the main loop, and `g_main_loop_run()` returns.

It is possible to create new instances of `GMainLoop` recursively. This is often used in GTK+ applications when showing modal dialog boxes. Note that event sources are associated with a particular `GMainContext`, and will be checked and dispatched for all main loops associated with that `GMainContext`.

GTK+ contains wrappers of some of these functions, e.g. `gtk_main()`, `gtk_main_quit()` and `gtk_events_pending()`.

Creating new sources types

One of the unusual features of the GTK+ main loop functionality is that new types of event source can be created and used in addition to the built-in type of event source. A new event source type is used for handling GDK events. A new source type is created by deriving from the `GSource` structure. The derived type of source is represented by a structure that has the `GSource` structure as a first element, and other elements specific to the new source type. To create an instance of the new source type, call `g_source_new()` passing in the size of the derived structure and a table of functions. These `GSourceFuncs` determine the behavior of the new source types.

New source types basically interact with the main context in two ways. Their prepare function in `GSourceFuncs` can set a timeout to determine the maximum amount of time that the main loop will sleep before checking the source again. In addition, or as well, the source can add file descriptors to the set that the main context checks using `g_source_add_poll()`.

Customizing the main loop iteration

Single iterations of a `GMainContext` can be run with `g_main_context_iteration()`. In some cases, more detailed control of exactly how the details of the main loop work is desired, for instance, when integrating the `GMainLoop` with an external main loop. In such cases, you can call the component functions of `g_main_context_iteration()` directly. These functions are `g_main_context_prepare()`, `g_main_context_query()`, `g_main_context_check()` and `g_main_context_dispatch()`.

The operation of these functions can best be seen in terms of a state diagram, as shown in Figure 3.1.
CHAPTER 3. GLIB CORE APPLICATION… 3.1. THE MAIN EVENT LOOP

Figure 3.1 States of a Main Context

Details

GMainLoop

typedef struct _GMainLoop GMainLoop;

The GMainLoop struct is an opaque data type representing the main event loop of a GLib or GTK+ application.

g_main_loop_new()

GMainLoop * g_main_loop_new (GMainContext *context, gboolean is_running);

Creates a new GMainLoop structure.

context: a GMainContext (if NULL, the default context will be used).

is_running: set to TRUE to indicate that the loop is running. This is not very important since calling g_main_loop_run() will set this to TRUE anyway.

Returns: a new GMainLoop.

g_main_loop_ref()

GMainLoop * g_main_loop_ref (GMainLoop *loop);

Increases the reference count on a GMainLoop object by one.

loop: a GMainLoop

Returns: loop

g_main_loop_unref()

void g_main_loop_unref (GMainLoop *loop);

Decreases the reference count on a GMainLoop object by one. If the result is zero, free the loop and free all associated memory.

loop: a GMainLoop

g_main_loop_run()

void g_main_loop_run (GMainLoop *loop);

Runs a main loop until g_main_loop_quit() is called on the loop. If this is called for the thread of the loop’s GMainContext, it will process events from the loop, otherwise it will simply wait.

loop: a GMainLoop

g_main_loop_quit()

void g_main_loop_quit (GMainLoop *loop);

Stops a GMainLoop from running. Any calls to g_main_loop_run() for the loop will return.

Note that sources that have already been dispatched when g_main_loop_quit() is called will still be executed.

loop: a GMainLoop
3.1. THE MAIN EVENT LOOP

**g_main_loop_is_running()**

```c
gboolean g_main_loop_is_running (GMainLoop *loop);
```

Checks to see if the main loop is currently being run via `g_main_loop_run()`.

*loop*: a `GMainLoop`.

*Returns*: TRUE if the mainloop is currently being run.

**g_main_loop_get_context()**

```c
GMainContext * g_main_loop_get_context (GMainLoop *loop);
```

Returns the `GMainContext` of `loop`.

*loop*: a `GMainLoop`.

*Returns*: the `GMainContext` of `loop`.

**g_main_new()**

```c
#define g_main_new(is_running)
```

**WARNING**

`g_main_new` has been deprecated since version 2.2 and should not be used in newly-written code. Use `g_main_loop_new()` instead.

Creates a new `GMainLoop` for the default main loop.

*is_running*: set to TRUE to indicate that the loop is running. This is not very important since calling `g_main_run()` will set this to TRUE anyway.

*Returns*: a new `GMainLoop`.

**g_main_destroy()**

```c
#define g_main_destroy(loop)
```

**WARNING**

`g_main_destroy` has been deprecated since version 2.2 and should not be used in newly-written code. Use `g_main_loop_unref()` instead.

Frees the memory allocated for the `GMainLoop`.

*loop*: a `GMainLoop`. 
g_main_run()

```c
#define g_main_run(loop)
```

**WARNING**

`g_main_run` has been deprecated since version 2.2 and should not be used in newly-written code. Use `g_main_loop_run()` instead.

Runs a main loop until it stops running.

**loop**: a `GMainLoop`.

---

g_main_quit()

```c
#define g_main_quit(loop)
```

**WARNING**

`g_main_quit` has been deprecated since version 2.2 and should not be used in newly-written code. Use `g_main_loop_quit()` instead.

Stops the `GMainLoop`. If `g_main_run()` was called to run the `GMainLoop`, it will now return.

**loop**: a `GMainLoop`.

---

g_main_is_running()

```c
#define g_main_is_running(loop)
```

**WARNING**

`g_main_is_running` has been deprecated since version 2.2 and should not be used in newly-written code. Use `g_main_loop_is_running()` instead.

Checks if the main loop is running.

**loop**: a `GMainLoop`.

**Returns**: %TRUE if the main loop is running.

**G_PRIORITY_HIGH**

```c
#define G_PRIORITY_HIGH -100
```

Use this for high priority event sources. It is not used within GLib or GTK+.
3.1. THE MAIN EVENT LOOP

G_PRIORITY_DEFAULT

#define G_PRIORITY_DEFAULT 0

Use this for default priority event sources. In GLib this priority is used when adding timeout functions with g_timeout_add(). In GDK this priority is used for events from the X server.

G_PRIORITY_HIGH_IDLE

#define G_PRIORITY_HIGH_IDLE 100

Use this for high priority idle functions. GTK+ uses G_PRIORITY_HIGH_IDLE + 10 for resizing operations, and G_PRIORITY_HIGH_IDLE + 20 for redrawing operations. (This is done to ensure that any pending resizes are processed before any pending redraws, so that widgets are not redrawn twice unnecessarily.)

G_PRIORITY_DEFAULT_IDLE

#define G_PRIORITY_DEFAULT_IDLE 200

Use this for default priority idle functions. In GLib this priority is used when adding idle functions with g_idle_add().

G_PRIORITY_LOW

#define G_PRIORITY_LOW 300

Use this for very low priority background tasks. It is not used within GLib or GTK+.

GMainContext

typedef struct _GMainContext GMainContext;

The GMainContext struct is an opaque data type representing a set of sources to be handled in a main loop.

g_main_context_new ()

GMainContext * g_main_context_new (void);

Creates a new GMainContext structure.

Returns: the new GMainContext

g_main_context_ref ()

GMainContext * g_main_context_ref (GMainContext *context);

Increases the reference count on a GMainContext object by one.

context: a GMainContext

Returns: the context that was passed in (since 2.6)

g_main_context_unref ()

void g_main_context_unref (GMainContext *context);

Decreases the reference count on a GMainContext object by one. If the result is zero, free the context and free all associated memory.

context: a GMainContext
3.1. THE MAIN EVENT LOOP

`g_main_context_default()`

```c
GMainContext * g_main_context_default (void);
```

Returns the default main context. This is the main context used for main loop functions when a main loop is not explicitly specified.

**Returns**: the default main context.

`g_main_context_iteration()`

```c
gboolean g_main_context_iteration (GMainContext *context, gboolean may_block);
```

Runs a single iteration for the given main loop. This involves checking to see if any event sources are ready to be processed, then if no events sources are ready and `may_block` is TRUE, waiting for a source to become ready, then dispatching the highest priority events sources that are ready. Otherwise, if `may_block` is FALSE sources are not waited to become ready, only those highest priority events sources will be dispatched (if any), that are ready at this given moment without further waiting.

Note that even when `may_block` is TRUE, it is still possible for `g_main_context_iteration()` to return FALSE, since the the wait may be interrupted for other reasons than an event source becoming ready.

**context**: a GMainContext (if NULL, the default context will be used)

**may_block**: whether the call may block.

**Returns**: TRUE if events were dispatched.

```
g_main_iteration()
```

```c
#define g_main_iteration(may_block)
```

**WARNING**

`g_main_iteration` has been deprecated since version 2.2 and should not be used in newly-written code. Use `g_main_context_iteration()` instead.

Runs a single iteration for the default GMainContext.

**may_block**: set to TRUE if it should block (i.e. wait) until an event source becomes ready. It will return after an event source has been processed. If set to FALSE it will return immediately if no event source is ready to be processed.

**Returns**: %TRUE if more events are pending.

`g_main_context_pending()`

```c
gboolean g_main_context_pending (GMainContext *context);
```

Checks if any sources have pending events for the given context.

**context**: a GMainContext (if NULL, the default context will be used)

**Returns**: TRUE if events are pending.
### 3.1. The Main Event Loop

**g_main_pending()**

```
#define g_main_pending()
```

**WARNING**

`g_main_pending` has been deprecated since version 2.2 and should not be used in newly-written code. Use `g_main_context_pending()` instead.

Checks if any events are pending for the default `GMainContext` (i.e. ready to be processed).

**Returns**: %TRUE if any events are pending.

**g_main_context_find_source_by_id()**

```
GSource * g_main_context_find_source_by_id (GMainContext *context, guint source_id);
```

Finds a `GSource` given a pair of context and ID.

**context**: a `GMainContext` (if NULL, the default context will be used)

**source_id**: the source ID, as returned by `g_source_get_id()`.

**Returns**: the `GSource` if found, otherwise, NULL

**g_main_context_find_source_by_user_data()**

```
GSource * g_main_context_find_source_by_user_data (GMainContext *context, gpointer user_data);
```

Finds a source with the given user data for the callback. If multiple sources exist with the same user data, the first one found will be returned.

**context**: a `GMainContext`

**user_data**: the user data for the callback.

**Returns**: the source, if one was found, otherwise NULL

**g_main_context_find_source_by_funcs_user_data()**

```
GSource * g_main_context_find_source_by_funcs_user_data (GMainContext *context, GSourceFuncs *funcs, gpointer user_data);
```

Finds a source with the given source functions and user data. If multiple sources exist with the same source function and user data, the first one found will be returned.

**context**: a `GMainContext` (if NULL, the default context will be used).

**funcs**: the `source_funcs` passed to `g_source_new()`.

**user_data**: the user data from the callback.

**Returns**: the source, if one was found, otherwise NULL
CHAPTER 3. GLIB CORE APPLICATION...

3.1. THE MAIN EVENT LOOP

`g_main_context_wakeup()`

```c
void g_main_context_wakeup (GMainContext *context);
```

If `context` is currently waiting in a `poll()`, interrupt the `poll()`, and continue the iteration process.

`context`: a `GMainContext`

`g_main_context_acquire()`

```c
gboolean g_main_context_acquire (GMainContext *context);
```

Tries to become the owner of the specified context. If some other thread is the owner of the context, returns `FALSE` immediately. Ownership is properly recursive: the owner can require ownership again and will release ownership when `g_main_context_release()` is called as many times as `g_main_context_acquire()`. You must be the owner of a context before you can call `g_main_context_prepare()`, `g_main_context_query()`, `g_main_context_check()`, `g_main_context_dispatch()`.

`context`: a `GMainContext`

`Returns`: `TRUE` if the operation succeeded, and this thread is now the owner of `context`.

`g_main_context_release()`

```c
void g_main_context_release (GMainContext *context);
```

Releases ownership of a context previously acquired by this thread with `g_main_context_acquire()`. If the context was acquired multiple times, the ownership will be released only when `g_main_context_release()` is called as many times as it was acquired.

`context`: a `GMainContext`

`g_main_context_is_owner()`

```c
gboolean g_main_context_is_owner (GMainContext *context);
```

Determines whether this thread holds the (recursive) ownership of this `GMainContext`. This is useful to know before waiting on another thread that may be blocking to get ownership of `context`.

`context`: a `GMainContext`

`Returns`: `TRUE` if current thread is owner of `context`.

Since 2.10

`g_main_context_wait()`

```c
gboolean g_main_context_wait (GMainContext *context, GCond *cond, GMutex *mutex);
```

Tries to become the owner of the specified context, as with `g_main_context_acquire()`. But if another thread is the owner, atomically drop `mutex` and wait on `cond` until that owner releases ownership or until `cond` is signaled, then try again (once) to become the owner.

`context`: a `GMainContext`

`cond`: a condition variable

`mutex`: a mutex, currently held

`Returns`: `TRUE` if the operation succeeded, and this thread is now the owner of `context`. 
## 3.1. The Main Event Loop

### `g_main_context_prepare()`

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| gboolean | `g_main_context_prepare` | (GMainContext *context, gint *priority); Prepares to poll sources within a main loop. The resulting information for polling is determined by calling `g_main_context_query()`.

- **context**: a GMainContext
- **priority**: location to store priority of highest priority source already ready.

**Returns**: TRUE if some source is ready to be dispatched prior to polling.

### `g_main_context_query()`

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| gint     | `g_main_context_query` | (GMainContext *context, gint max_priority, gint *timeout_, GPollFD *fds, gint n_fds); Determines information necessary to poll this main loop.

- **context**: a GMainContext
- **max_priority**: maximum priority source to check
- **timeout_**: location to store timeout to be used in polling
- **fds**: location to store GPollFD records that need to be polled.
- **n_fds**: length of **fds**.

**Returns**: the number of records actually stored in **fds**, or, if more than **n_fds** records need to be stored, the number of records that need to be stored.

### `g_main_context_check()`

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| gint     | `g_main_context_check` | (GMainContext *context, gint max_priority, GPollFD *fds, gint n_fds); Passes the results of polling back to the main loop.

- **context**: a GMainContext
- **max_priority**: the maximum numerical priority of sources to check
- **fds**: array of GPollFD’s that was passed to the last call to `g_main_context_query()`
- **n_fds**: return value of `g_main_context_query()`

**Returns**: TRUE if some sources are ready to be dispatched.

### `g_main_context_dispatch()`

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| void     | `g_main_context_dispatch` | (GMainContext *context); Dispatches all pending sources.

- **context**: a GMainContext
3.1. THE MAIN EVENT LOOP

void g_main_context_set_poll_func (GMainContext *context, GPollFunc func);

Sets the function to use to handle polling of file descriptors. It will be used instead of the poll() system call (or GLib’s replacement function, which is used where poll() isn’t available).

This function could possibly be used to integrate the GLib event loop with an external event loop.

*context*: a GMainContext

*func*: the function to call to poll all file descriptors

---

GPollFunc g_main_context_get_poll_func (GMainContext *context);

Gets the poll function set by g_main_context_set_poll_func().

*context*: a GMainContext

Returns: the poll function

---

gint (*GPollFunc) (GPollFD *ufds, guint nfsd, gint timeout_);

Specifies the type of function passed to g_main_context_set_poll_func(). The semantics of the function should match those of the poll() system call.

*ufds*: an array of GPollFD elements.

*nfsd*: the number of elements in *ufds*.

*timeout_*: the maximum time to wait for an event of the file descriptors. A negative value indicates an infinite timeout.

Returns: the number of GPollFD elements which have events or errors reported, or -1 if an error occurred.

---

void g_main_context_add_poll (GMainContext *context, GPollFD *fd, gint priority);

Adds a file descriptor to the set of file descriptors polled for this context. This will very seldomly be used directly. Instead a typical event source will use g_source_add_poll() instead.

*context*: a GMainContext (or NULL for the default context)

*fd*: a GPollFD structure holding information about a file descriptor to watch.

*priority*: the priority for this file descriptor which should be the same as the priority used for g_source_attach() to ensure that the file descriptor is polled whenever the results may be needed.
CHAPTER 3. GLIB CORE APPLICATION...  

3.1. THE MAIN EVENT LOOP

```c
void g_main_context_remove_poll (GMainContext *context,
     GPollFD *fd);
```

Removes file descriptor from the set of file descriptors to be polled for a particular context.

**context**: a GMainContext  

**fd**: a GPollFD descriptor previously added with g_main_context_add_poll()

```c
gint g_main_depth (void);
```

Returns the depth of the stack of calls to g_main_context_dispatch() on any GMainContext in the current thread. That is, when called from the toplevel, it gives 0. When called from within a callback from g_main_context_iteration() (or g_main_loop_run(), etc.) it returns 1. When called from within a callback to a recursive call to g_main_context_iterate(), it returns 2. And so forth.

This function is useful in a situation like the following: Imagine an extremely simple "garbage collected" system.

```c
static GList *free_list;

gpointer
allocate_memory (gsize size)
{
    gpointer result = g_malloc (size);
    free_list = g_list_prepend (free_list, result);
    return result;
}

void
free_allocated_memory (void)
{
    GList *l;
    for (l = free_list; l; l = l->next);
        g_free (l->data);
    g_list_free (free_list);
    free_list = NULL;
}

[...]
while (TRUE);
{
    g_main_context_iteration (NULL, TRUE);
    free_allocated_memory();
}
```

This works from an application, however, if you want to do the same thing from a library, it gets more difficult, since you no longer control the main loop. You might think you can simply use an idle function to make the call to free_allocated_memory(), but that doesn’t work, since the idle function could be called from a recursive callback. This can be fixed by using g_main_depth()

```c

gpointer
allocate_memory (gsize size)
{
    FreeListBlock *block = g_new (FreeListBlock, 1);
    block->mem = g_malloc (size);
    block->depth = g_main_depth ();
    free_list = g_list_prepend (free_list, block);
    return block->mem;
}
```
void free_allocated_memory (void)
{
    GList *l;

    int depth = g_main_depth ();
    for (l = free_list; l; )
    {
        GList *next = l->next;
        FreeListBlock *block = l->data;
        if (block->depth > depth)
        {
            g_free (block->mem);
            g_free (block);
            free_list = g_list_delete_link (free_list, l);
        }
        l = next;
    }
}

There is a temptation to use `g_main_depth()` to solve problems with reentrancy. For instance, while waiting for data to be received from the network in response to a menu item, the menu item might be selected again. It might seem that one could make the menu item’s callback return immediately and do nothing if `g_main_depth()` returns a value greater than 1. However, this should be avoided since the user then sees selecting the menu item do nothing. Furthermore, you’ll find yourself adding these checks all over your code, since there are doubtless many, many things that the user could do. Instead, you can use the following techniques:

1. Use `gtk_widget_set_sensitive()` or modal dialogs to prevent the user from interacting with elements while the main loop is recursing.

2. Avoid main loop recursion in situations where you can’t handle arbitrary callbacks. Instead, structure your code so that you simply return to the main loop and then get called again when there is more work to do.

**Returns**: The main loop recursion level in the current thread

```c
GSource * g_main_current_source (void);
```

Returns the currently firing source for this thread.

**Returns**: The currently firing source or NULL.

Since 2.12

```c
#define g_main_set_poll_func(func)
```

**WARNING**

`g_main_set_poll_func` has been deprecated since version 2.2 and should not be used in newly-written code. Use `g_main_context_set_poll_func()` instead.

Sets the function to use for the handle polling of file descriptors for the default main context.

**func**: the function to call to poll all file descriptors.
CHAPTER 3. GLIB CORE APPLICATION...

3.1. THE MAIN EVENT LOOP

**g_timeout_source_new()**

```c
GSource * g_timeout_source_new (guint interval);
```

Creates a new timeout source.

The source will not initially be associated with any `GMainContext` and must be added to one with `g_source_attach()` before it will be executed.

- **interval**: the timeout interval in milliseconds.

**Returns**: the newly-created timeout source

**g_timeout_source_new_seconds()**

```c
GSource * g_timeout_source_new_seconds (guint interval);
```

Creates a new timeout source.

The source will not initially be associated with any `GMainContext` and must be added to one with `g_source_attach()` before it will be executed.

The scheduling granularity/accuracy of this timeout source will be in seconds.

- **interval**: the timeout interval in seconds

**Returns**: the newly-created timeout source

Since 2.14

**g_timeout_add()**

```c
guint g_timeout_add (guint interval,
                     GSourceFunc function,
                     gpointer data);
```

Sets a function to be called at regular intervals, with the default priority, `G_PRIORITY_DEFAULT`. The function is called repeatedly until it returns `FALSE`, at which point the timeout is automatically destroyed and the function will not be called again. The first call to the function will be at the end of the first `interval`.

Note that timeout functions may be delayed, due to the processing of other event sources. Thus they should not be relied on for precise timing. After each call to the timeout function, the time of the next timeout is recalculated based on the current time and the given interval (it does not try to ‘catch up’ time lost in delays).

If you want to have a timer in the "seconds" range and do not care about the exact time of the first call of the timer, use the `g_timeout_add_seconds()` function; this function allows for more optimizations and more efficient system power usage.

This internally creates a main loop source using `g_timeout_source_new()` and attaches it to the main loop context using `g_source_attach()`. You can do these steps manually if you need greater control.

- **interval**: the time between calls to the function, in milliseconds (1/1000ths of a second)
- **function**: function to call
- **data**: data to pass to `function`

**Returns**: the ID (greater than 0) of the event source.

**g_timeout_add_full()**

```c
guint g_timeout_add_full (gint priority,
                          guint interval,
                          GSourceFunc function,
                          gpointer data,
                          GDestroyNotify notify);
```


CHAPTER 3. GLIB CORE APPLICATION

3.1. THE MAIN EVENT LOOP

Sets a function to be called at regular intervals, with the given priority. The function is called repeatedly until it returns FALSE, at which point the timeout is automatically destroyed and the function will not be called again. The notify function is called when the timeout is destroyed. The first call to the function will be at the end of the first interval.

Note that timeout functions may be delayed, due to the processing of other event sources. Thus they should not be relied on for precise timing. After each call to the timeout function, the time of the next timeout is recalculated based on the current time and the given interval (it does not try to ‘catch up’ time lost in delays).

This internally creates a main loop source using g_timeout_source_new() and attaches it to the main loop context using g_source_attach(). You can do these steps manually if you need greater control.

priority: the priority of the timeout source. Typically this will be in the range between G_PRIORITY_DEFAULT and G_PRIORITY_HIGH.

interval: the time between calls to the function, in milliseconds (1/1000ths of a second)

function: function to call

data: data to pass to function

notify: function to call when the timeout is removed, or NULL

Returns: the ID (greater than 0) of the event source.

g_timeout_add_seconds()

```c
uint g_timeout_add_seconds (guint interval,
GSourceFunc function,
gpointer data);
```

Sets a function to be called at regular intervals with the default priority, G_PRIORITY_DEFAULT. The function is called repeatedly until it returns FALSE, at which point the timeout is automatically destroyed and the function will not be called again.

This internally creates a main loop source using g_timeout_source_new_seconds() and attaches it to the main loop context using g_source_attach(). You can do these steps manually if you need greater control. Also see g_timeout_add_seconds_full().

interval: the time between calls to the function, in seconds

function: function to call

data: data to pass to function

Returns: the ID (greater than 0) of the event source.

Since 2.14

g_timeout_add_seconds_full()

```c
uint g_timeout_add_seconds_full (gint priority,
 guint interval,
GSourceFunc function,
gpointer data,
GDestroyNotify notify);
```

Sets a function to be called at regular intervals, with priority. The function is called repeatedly until it returns FALSE, at which point the timeout is automatically destroyed and the function will not be called again.

Unlike g_timeout_add(), this function operates at whole second granularity. The initial starting point of the timer is determined by the implementation and the implementation is expected to group multiple timers together so that they fire all at the same time. To allow this grouping, the interval to the first timer is rounded and can deviate up to one second from the specified interval. Subsequent timer iterations will generally run at the specified interval.
Note that timeout functions may be delayed, due to the processing of other event sources. Thus they should not be relied on for precise timing. After each call to the timeout function, the time of the next timeout is recalculated based on the current time and the given interval.

If you want timing more precise than whole seconds, use g_timeout_add() instead.

The grouping of timers to fire at the same time results in a more power and CPU efficient behavior so if your timer is in multiples of seconds and you don’t require the first timer exactly one second from now, the use of g_timeout_add_seconds() is preferred over g_timeout_add().

This internally creates a main loop source using g_timeout_source_new_seconds() and attaches it to the main loop context using g_source_attach(). You can do these steps manually if you need greater control.

priority: the priority of the timeout source. Typically this will be in the range between G_PRIORITY_DEFAULT and G_PRIORITY_HIGH.

interval: the time between calls to the function, in seconds

function: function to call

data: data to pass to function

notify: function to call when the timeout is removed, or NULL

Returns: the ID (greater than 0) of the event source.

Since 2.14

**g_idle_source_new()**

GSource * g_idle_source_new (void);

Creates a new idle source.

The source will not initially be associated with any GMainContext and must be added to one with g_source_attach() before it will be executed. Note that the default priority for idle sources is G_PRIORITY_DEFAULT_IDLE, as compared to other sources which have a default priority of G_PRIORITY_DEFAULT.

Returns: the newly-created idle source

**g_idle_add()**

 guint g_idle_add (GSourceFunc function, gpointer data);

Adds a function to be called whenever there are no higher priority events pending to the default main loop. The function is given the default idle priority, G_PRIORITY_DEFAULT_IDLE. If the function returns FALSE it is automatically removed from the list of event sources and will not be called again.

This internally creates a main loop source using g_idle_source_new() and attaches it to the main loop context using g_source_attach(). You can do these steps manually if you need greater control.

function: function to call

data: data to pass to function.

Returns: the ID (greater than 0) of the event source.

**g_idle_add_full()**

 guint g_idle_add_full (gint priority, GSourceFunc function, gpointer data, GDestroyNotify notify);

---

94
CHAPTER 3. GLIB CORE APPLICATION

3.1. THE MAIN EVENT LOOP

Adds a function to be called whenever there are no higher priority events pending. If the function returns FALSE it is automatically removed from the list of event sources and will not be called again.

This internally creates a main loop source using g_idle_source_new() and attaches it to the main loop context using g_source_attach(). You can do these steps manually if you need greater control.

**priority**: the priority of the idle source. Typically this will be in the range between G_PRIORITY_DEFAULT_IDLE and G_PRIORITY_HIGH_IDLE.

**function**: function to call

**data**: data to pass to function

**notify**: function to call when the idle is removed, or NULL

**Returns**: the ID (greater than 0) of the event source.

---

**g_idle_remove_by_data()**

 gboolean g_idle_remove_by_data (gpointer data);

Removes the idle function with the given data.

**data**: the data for the idle source's callback.

**Returns**: TRUE if an idle source was found and removed.

---

**GPid**

 typedef int GPid;

A type which is used to hold a process identification. On Unix, processes are identified by a process id (an integer), while Windows uses process handles (which are pointers).

---

**GChildWatchFunc()**

 void ( *GChildWatchFunc) (GPid pid, gint status, gpointer data);

The type of functions to be called when a child exists.

**pid**: the process id of the child process

**status**: Status information about the child process, see waitpid(2) for more information about this field

**data**: user data passed to g_child_watch_add()

---

**g_child_watch_source_new()**

 GSource * g_child_watch_source_new (GPid pid);

Creates a new child_watch source.

The source will not initially be associated with any GMainContext and must be added to one with g_source_attach() before it will be executed.

Note that child watch sources can only be used in conjunction with g_spawn... when the G_SPAWN_DO_NOT_REAP_CHILD flag is used.

Note that on platforms where GPid must be explicitly closed (see g_spawn_close_pid()) pid must not be closed while the source is still active. Typically, you will want to call g_spawn_close_pid() in the callback function for the source.

Note further that using g_child_watch_source_new() is not compatible with calling waitpid(-1) in the application. Calling waitpid() for individual pids will still work fine.
**chapter 3. Glib core application…**

## 3.1. The main event loop

- **pid**: process to watch. On POSIX the pid of a child process. On Windows a handle for a process (which doesn’t have to be a child).

**Returns**: the newly-created child watch source

Since 2.4

```c
void g_child_watch_add (GChildWatchFunc function, gpointer data);
```

Sets a function to be called when the child indicated by `pid` exits, at a default priority, G_PRIORITY_DEFAULT.

If you obtain `pid` from `g_spawn_async()` or `g_spawn_async_with_pipes()` you will need to pass `G_SPAWN_DO_NOT_REAP_CHILD` as flag to the spawn function for the child watching to work.

Note that on platforms where GPid must be explicitly closed (see `g_spawn_close_pid()`) `pid` must not be closed while the source is still active. Typically, you will want to call `g_spawn_close_pid()` in the callback function for the source.

GLib supports only a single callback per process id.

This internally creates a main loop source using `g_child_watch_source_new()` and attaches it to the main loop context using `g_source_attach()`. You can do these steps manually if you need greater control.

- **pid**: process id to watch. On POSIX the pid of a child process. On Windows a handle for a process (which doesn’t have to be a child).

**function**: function to call

**data**: data to pass to `function`

**Returns**: the ID (greater than 0) of the event source.

Since 2.4

```c
void g_child_watch_add_full (gint priority, GChildWatchFunc function, gpointer data, GDestroyNotify notify);
```

Sets a function to be called when the child indicated by `pid` exits, at the priority `priority`.

If you obtain `pid` from `g_spawn_async()` or `g_spawn_async_with_pipes()` you will need to pass `G_SPAWN_DO_NOT_REAP_CHILD` as flag to the spawn function for the child watching to work.

Note that on platforms where GPid must be explicitly closed (see `g_spawn_close_pid()`) `pid` must not be closed while the source is still active. Typically, you will want to call `g_spawn_close_pid()` in the callback function for the source.

GLib supports only a single callback per process id.

This internally creates a main loop source using `g_child_watch_source_new()` and attaches it to the main loop context using `g_source_attach()`. You can do these steps manually if you need greater control.

- **priority**: the priority of the idle source. Typically this will be in the range between `G_PRIORITY_DEFAULT_IDLE` and `G_PRIORITY_HIGH_IDLE`.

- **pid**: process to watch. On POSIX the pid of a child process. On Windows a handle for a process (which doesn’t have to be a child).

**function**: function to call

**data**: data to pass to `function`
**notify**: function to call when the idle is removed, or **NULL**

**Returns**: the ID (greater than 0) of the event source.

Since 2.4

**GPollFD**

```c
typedef struct {
#if defined (G_OS_WIN32) && GLIB_SIZEOF_VOID_P == 8
    gint64 fd;
#else
    gint fd;
#endif
    gushort events;
    gushort revents;
} GPollFD;
```

- **fd**: the file descriptor to poll (or a HANDLE on Win32 platforms).
- **events**: a bitwise combination of flags from **GIOCondition**, specifying which events should be polled for. Typically for reading from a file descriptor you would use `G_IO_IN | G_IO_HUP | G_IO_ERR`, and for writing you would use `G_IO_OUT | G_IO_ERR`.
- **revents**: a bitwise combination of flags from **GIOCondition**, returned from the `poll()` function to indicate which events occurred.

**g_poll()**

```c
gint g_poll (GPollFD *fds, guint nfds, gint timeout);
```

Polls `fds`, as with the `poll()` system call, but portably. (On systems that don’t have `poll()`, it is emulated using `select()`.) This is used internally by **GMainContext**, but it can be called directly if you need to block until a file descriptor is ready, but don’t want to run the full main loop.

Each element of `fds` is a **GPollFD** describing a single file descriptor to poll. The `fd` field indicates the file descriptor, and the `events` field indicates the events to poll for. On return, the `revents` fields will be filled with the events that actually occurred.

On POSIX systems, the file descriptors in `fds` can be any sort of file descriptor, but the situation is much more complicated on Windows. If you need to use `g_poll()` in code that has to run on Windows, the easiest solution is to construct all of your **GPollFDs** with `g_io_channel_win32_make_pollfd()`.

- **fds**: file descriptors to poll
- **nfds**: the number of file descriptors in `fds`
- **timeout**: amount of time to wait, in milliseconds, or -1 to wait forever

**Returns**: the number of entries in `fds` whose `revents` fields were filled in, or 0 if the operation timed out, or -1 on error or if the call was interrupted.

Since 2.20

**GSource**

```c
typedef struct {
} GSource;
```

The **GSource** struct is an opaque data type representing an event source.
GSourceDummyMarshal()

```c
void (*GSourceDummyMarshal)(void);
```

This is just a placeholder for GClosureMarshal, which cannot be used here for dependency reasons.

GSourceFuncs

```c
typedef struct {
    gboolean (*prepare)(GSource *source, gint *timeout_);
    gboolean (*check)(GSource *source);
    gboolean (*dispatch)(GSource *source, GSourceFunc callback, gpointer user_data);
    void (*finalize)(GSource *source); /* Can be NULL */
    /* For use by g_source_set_closure */
    GSourceFunc closure_callback;
    GSourceDummyMarshal closure_marshal; /* Really is of type GClosureMarshal */
} GSourceFuncs;
```

The GSourceFuncs struct contains a table of functions used to handle event sources in a generic manner.

For idle sources, the prepare and check functions always return TRUE to indicate that the source is always ready to be processed. The prepare function also returns a timeout value of 0 to ensure that the poll() call doesn’t block (since that would be time wasted which could have been spent running the idle function).

For timeout sources, the prepare and check functions both return TRUE if the timeout interval has expired. The prepare function also returns a timeout value to ensure that the poll() call doesn’t block too long and miss the next timeout.

For file descriptor sources, the prepare function typically returns FALSE, since it must wait until poll() has been called before it knows whether any events need to be processed. It sets the returned timeout to -1 to indicate that it doesn’t mind how long the poll() call blocks. In the check function, it tests the results of the poll() call to see if the required condition has been met, and returns TRUE if so.

prepare() Called before all the file descriptors are polled. If the source can determine that it is ready here (without waiting for the results of the poll() call) it should return TRUE. It can also return a timeout_value which should be the maximum timeout (in milliseconds) which should be passed to the poll() call. The actual timeout used will be -1 if all sources returned -1, or it will be the minimum of all the timeout_values returned which were >= 0.

check() Called after all the file descriptors are polled. The source should return TRUE if it is ready to be dispatched. Note that some time may have passed since the previous prepare function was called, so the source should be checked again here.

dispatch() Called to dispatch the event source, after it has returned TRUE in either its prepare or its check function. The dispatch function is passed in a callback function and data. The callback function may be NULL if the source was never connected to a callback using g_source_set_callback(). The dispatch function should call the callback function with user_data and whatever additional parameters are needed for this type of event source.

finalize() Called when the source is finalized.

GSourceFunc closure_callback;

GSourceDummyMarshal closure_marshal;
GSourceCallbackFuncs

```c
typedef struct {
    void (*ref) (gpointer cb_data);
    void (*unref) (gpointer cb_data);
    void (*get) (gpointer cb_data,
                 GSource *source,
                 GSourceFunc *func,
                 gpointer *data);
} GSourceCallbackFuncs;
```

The GSourceCallbackFuncs struct contains functions for managing callback objects.

- **ref** 0 Called when a reference is added to the callback object.
- **unref** 0 Called when a reference to the callback object is dropped.
- **get** 0 Called to extract the callback function and data from the callback object.

### g_source_new()

```c
GSource * g_source_new (GSourceFuncs *source_funcs,
                         guint struct_size);
```

Creates a new GSource structure. The size is specified to allow creating structures derived from GSource that contain additional data. The size passed in must be at least `sizeof (GSource)`.

- **source_funcs**: structure containing functions that implement the sources behavior.
- **struct_size**: size of the GSource structure to create.

Returns: the newly-created GSource.

### g_source_ref()

```c
GSource * g_source_ref (GSource *source);
```

Increases the reference count on a source by one.

- **source**: a GSource

Returns: source

### g_source_unref()

```c
void g_source_unref (GSource *source);
```

Decreases the reference count of a source by one. If the resulting reference count is zero the source and associated memory will be destroyed.

- **source**: a GSource

### g_source_set_funcs()

```c
void g_source_set_funcs (GSource *source,
                         GSourceFuncs *funcs);
```

Sets the source functions (can be used to override default implementations) of an unattached source.

- **source**: a GSource
- **funcs**: the new GSourceFuncs

Since 2.12
### 3.1. THE MAIN EVENT LOOP

#### g_source_attach()

```c
 guint g_source_attach (GSource *source, GMainContext *context);
```

**Description:** Adds a GSource to a context so that it will be executed within that context. Remove it by calling `g_source_destroy()`.

**Arguments:**
- `source`: a GSource
- `context`: a GMainContext (if NULL, the default context will be used)

**Returns:** the ID (greater than 0) for the source within the GMainContext.

#### g_source_destroy()

```c
 void g_source_destroy (GSource *source);
```

**Description:** Removes a source from its GMainContext, if any, and mark it as destroyed. The source cannot be subsequently added to another context.

**Arguments:**
- `source`: a GSource

#### g_source_is_destroyed()

```c
 gboolean g_source_is_destroyed (GSource *source);
```

**Description:** Returns whether `source` has been destroyed.

This is important when you operate upon your objects from within idle handlers, but may have freed the object before the dispatch of your idle handler.

```c
static gboolean
idle_callback (gpointer data)
{
    SomeWidget *self = data;

    GDK_THREADS_ENTER ();
    /* do stuff with self */
    self->idle_id = 0;
    GDK_THREADS_LEAVE ();

    return FALSE;
}
```

```c
static void
some_widget_do_stuff_later (SomeWidget *self)
{
    self->idle_id = g_idle_add (idle_callback, self);
}
```

```c
static void
some_widget_finalize (GObject *object)
{
    SomeWidget *self = SOME_WIDGET (object);

    if (self->idle_id)
        g_source_remove (self->idle_id);

    G_OBJECT_CLASS (parent_class)->finalize (object);
}
```

This will fail in a multi-threaded application if the widget is destroyed before the idle handler fires due to the use after free in the callback. A solution, to this particular problem, is to check to if the source has already been destroy within the callback.
static gboolean
idle_callback (gpointer data)
{
    SomeWidget *self = data;

    GDK_THREADS_ENTER ();
    if (!g_source_is_destroyed (g_main_current_source ()))
    {
        /* do stuff with self */
    }
    GDK_THREADS_LEAVE ();
    return FALSE;
}

source: a GSource

Returns: TRUE if the source has been destroyed
Since 2.12

g_source_set_priority ()

void g_source_set_priority (GSource *source, gint priority);

Sets the priority of a source. While the main loop is being run, a source will be dispatched if it is ready to be dispatched and no sources at a higher (numerically smaller) priority are ready to be dispatched.

source: a GSource

priority: the new priority.

g_source_get_priority ()

gint g_source_get_priority (GSource *source);

Gets the priority of a source.

source: a GSource

Returns: the priority of the source

g_source_set_can_recurse ()

void g_source_set_can_recurse (GSource *source, gboolean can_recurse);

Sets whether a source can be called recursively. If can_recurse is TRUE, then while the source is being dispatched then this source will be processed normally. Otherwise, all processing of this source is blocked until the dispatch function returns.

source: a GSource

can_recurse: whether recursion is allowed for this source

g_source_get_can_recurse ()

gboolean g_source_get_can_recurse (GSource *source);

Checks whether a source is allowed to be called recursively. see g_source_set_can_recurse().

source: a GSource

Returns: whether recursion is allowed.
g_source_get_id ()

```
quint g_source_get_id (GSource *source);
```

Returns the numeric ID for a particular source. The ID of a source is a positive integer which is unique within a particular main loop context. The reverse mapping from ID to source is done by `g_main_context_find_source_by_id()`.

**source**: a GSource

**Returns**: the ID (greater than 0) for the source

---

g_source_get_context ()

```
GMainContext * g_source_get_context (GSource *source);
```

Gets the GMainContext with which the source is associated. Calling this function on a destroyed source is an error.

**source**: a GSource

**Returns**: the GMainContext with which the source is associated, or NULL if the context has not yet been added to a source.

---

g_source_set_callback ()

```
void g_source_set_callback (GSource *source,
                          GSourceFunc func,
                          gpointer data,
                          GDestroyNotify notify);
```

Sets the callback function for a source. The callback for a source is called from the source’s dispatch function.

The exact type of `func` depends on the type of source; i.e. you should not count on `func` being called with `data` as its first parameter.

Typically, you won’t use this function. Instead use functions specific to the type of source you are using.

**source**: the source

**func**: a callback function

**data**: the data to pass to callback function

**notify**: a function to call when `data` is no longer in use, or NULL.

---

GSourceFunc ()

```
gboolean (*GSourceFunc) (gpointer data);
```

Specifies the type of function passed to `g_timeout_add()`, `g_timeout_add_full()`, `g_idle_add()`, and `g_idle_add_full()`.

**data**: data passed to the function, set when the source was created with one of the above functions.

**Returns**: it should return FALSE if the source should be removed.
### g_source_set_callback_indirect()

```c
void g_source_set_callback_indirect (GSource *source,
    gpointer callback_data,
    GSourceCallbackFuncs *callback_funcs);
```

Sets the callback function storing the data as a refcounted callback "object". This is used internally. Note that calling `g_source_set_callback_indirect()` assumes an initial reference count on `callback_data`, and thus `callback_funcs->unref` will eventually be called once more than `callback_funcs->ref`.

- **source**: the source
- **callback_data**: pointer to callback data "object"
- **callback_funcs**: functions for reference counting `callback_data` and getting the callback and data

### g_source_add_poll()

```c
void g_source_add_poll (GSource *source,
    GPollFD *fd);
```

Adds a file descriptor to the set of file descriptors polled for this source. This is usually combined with `g_source_new()` to add an event source. The event source's check function will typically test the `revents` field in the `GPollFD` struct and return `TRUE` if events need to be processed.

- **source**: a GSource
- **fd**: a `GPollFD` structure holding information about a file descriptor to watch.

### g_source_remove_poll()

```c
void g_source_remove_poll (GSource *source,
    GPollFD *fd);
```

Removes a file descriptor from the set of file descriptors polled for this source.

- **source**: a GSource
- **fd**: a `GPollFD` structure previously passed to `g_source_add_poll()`.

### g_source_get_current_time()

```c
void g_source_get_current_time (GSource *source,
    GTimeVal *timeval);
```

Gets the "current time" to be used when checking this source. The advantage of calling this function over calling `g_get_current_time()` directly is that when checking multiple sources, GLib can cache a single value instead of having to repeatedly get the system time.

- **source**: a GSource
- **timeval**: `GTimeVal` structure in which to store current time.

### g_source_remove()

```c
gboolean g_source_remove (guint tag);
```

Removes the source with the given id from the default main context. The id of a GSource is given by `g_source_get_id()`, or will be returned by the functions `g_source_attach()`, `g_idle_add()`, `g_idle_add_full()`, `g_timeout_add()`, `g_timeout_add_full()`, `g_child_watch_add()`, `g_child_watch_add_full()`, `g_io_add_watch()`, and `g_io_add_watch_full()`.

See also `g_source_destroy()`. You must use `g_source_destroy()` for sources added to a non-default main context.
**CHAPTER 3. GLIB CORE APPLICATION**

### 3.2 Threads

**Name**

Threads – thread abstraction; including threads, different mutexes, conditions and thread private data

**Synopsis**

```c
#include <glib.h>

#define G_THREADS_ENABLED
#define G_THREADS_IMPL_POSIX
#define G_THREADS_IMPL_NONE
#define G_THREAD_ERROR
enum GThreadError;

GThreadFunctions;

void g_thread_init (GThreadFunctions *vtable);
gboolean g_thread_supported ();
gboolean g_thread_get_initialized (void);

GThreadFunc (*GThreadFunc) (gpointer data);
enum GThreadPriority;
GThread;

GThread * g_thread_create (GThreadFunc func, gpointer data, gboolean joinable, GError **error);
gThread* g_thread_create_full (GThreadFunc func,
```
3.2. THREADS

gpointer data,
gulong stack_size,
gboolean joinable,
gboolean bound,
GThreadPriority priority,
GError **error);

(void);

GThread* g_thread_self (void);

Gpointer g_thread_join (GThread *thread);

(void);

void g_thread_set_priority (GThread *thread,
GThreadPriority priority);

(void);

void g_thread_yield (void);

void g_thread_exit (gpointer retval);

void g_thread_foreach (GFunc thread_func,
gpointer user_data);

GMutex;

GMutex * g_mutex_new ();

GMutex * g_mutex_lock (GMutex *mutex);

gboolean g_mutex_trylock (GMutex *mutex);

void g_mutex_unlock (GMutex *mutex);

void g_mutex_free (GMutex *mutex);

#define G_STATIC_MUTEX_INIT

void g_static_mutex_init (GStaticMutex *mutex);

void g_static_mutex_lock (GStaticMutex *mutex);

gboolean g_static_mutex_trylock (GStaticMutex *mutex);

void g_static_mutex_unlock (GStaticMutex *mutex);

GMutex * g_static_mutex_get_mutex (GStaticMutex *mutex);

void g_static_mutex_free (GStaticMutex *mutex);

#define G_LOCK_DEFINE (name)

#define G_LOCK_DEFINE_STATIC (name)

#define G_LOCK_EXTERN (name)

#define G_LOCK (name)

#define G_TRYLOCK (name)

#define G_UNLOCK (name)

GStaticRecMutex;

#define G_STATIC_REC_MUTEX_INIT

void g_static_rec_mutex_init (GStaticRecMutex *mutex);

void g_static_rec_mutex_lock (GStaticRecMutex *mutex);

gboolean g_static_rec_mutex_trylock (GStaticRecMutex *mutex);

void g_static_rec_mutex_unlock (GStaticRecMutex *mutex);

void g_static_rec_mutex_lock_full (GStaticRecMutex *mutex,
guint depth);

guint g_static_rec_mutex_unlock_full (GStaticRecMutex *mutex);

void g_static_rec_mutex_free (GStaticRecMutex *mutex);

GStaticRWLock;

#define G_STATIC_RW_LOCK_INIT

void g_static_rw_lock_init (GStaticRWLock *lock);

void g_static_rw_lock_reader_lock (GStaticRWLock *lock);

gboolean g_static_rw_lock_reader_trylock (GStaticRWLock *lock);

void g_static_rw_lock_reader_unlock (GStaticRWLock *lock);

gboolean g_static_rw_lock_writer_lock (GStaticRWLock *lock);

void g_static_rw_lock_writer_trylock (GStaticRWLock *lock);

void g_static_rw_lock_writer_unlock (GStaticRWLock *lock);

void g_static_rw_lock_free (GStaticRWLock *lock);
CHAPTER 3. GLIB CORE APPLICATION...

3.2. THREADS

GCond;

GCond* g_cond_new ();
void g_cond_signal (GCond *cond);
void g_cond_broadcast (GCond *cond);
void g_cond_wait (GCond *cond, GMutex *mutex);
gboolean g_cond_timed_wait (GCond *cond, GMutex *mutex, GTimeVal *abs_time);
void g_cond_free (GCond *cond);

GPrivate;

GPrivate* g_private_new (GDestroyNotify destructor);
gpointer g_private_get (GPrivate *private_key);
void g_private_set (GPrivate *private_key, gpointer data);

GStaticPrivate;

#define G_STATIC_PRIVATE_INIT
void g_static_private_init (GStaticPrivate *private_key);
gpointer g_static_private_get (GStaticPrivate *private_key);
void g_static_private_set (GStaticPrivate *private_key, gpointer data,
GDestroyNotify notify);
void g_static_private_free (GStaticPrivate *private_key);

GOnce;

enum GOnceStatus;
#define G_ONCE_INIT
#define g_once (once, func, arg)
void g_once_init_enter (volatile gsize *value_location);
void g_once_init_leave (volatile gsize *value_location, gsize initialization_value);

Description

Threads act almost like processes, but unlike processes all threads of one process share the same memory. This is good, as it provides easy communication between the involved threads via this shared memory, and it is bad, because strange things (so called "Heisenbugs") might happen if the program is not carefully designed. In particular, due to the concurrent nature of threads, no assumptions on the order of execution of code running in different threads can be made, unless order is explicitly forced by the programmer through synchronization primitives.

The aim of the thread related functions in GLib is to provide a portable means for writing multi-threaded software. There are primitives for mutexes to protect the access to portions of memory (GMutex, GStaticMutex, G_LOCK_DEFINE, GStaticRecMutex and GStaticRWLock). There are primitives for condition variables to allow synchronization of threads (GCond). There are primitives for thread-private data - data that every thread has a private instance of (GPrivate, GStaticPrivate). Last but definitely not least there are primitives to portably create and manage threads (GThread).

You must call g_thread_init() before executing any other GLib functions (except g_mem_set_vtable()) in a GLib program if g_thread_init() will be called at all. This is a requirement even if no threads are in fact ever created by the process. It is enough that g_thread_init() is called. If other GLib functions have been called before that, the behaviour of the program is undefined. An exception is g_mem_set_vtable() which may be called before g_thread_init(). Failing this requirement can lead to hangs or crashes, apparently more easily on Windows than on Linux, for example. Please note that if you call functions in some GLib-using library, in particular those above the GTK+ stack, that library might well call g_thread_init() itself, or call some other library that calls g_thread_init(). Thus, if you use some GLib-based library that is above the GTK+ stack, it is safest to call g_thread_init() in your application’s main() before calling any GLib functions or functions in GLib-using libraries. After calling g_thread_init(), GLib is completely

106
thread safe (all global data is automatically locked), but individual data structure instances are not automatically locked for performance reasons. So, for example you must coordinate accesses to the same GHashTable from multiple threads. The two notable exceptions from this rule are GMainLoop and GAsyncQueue, which are threadsafe and needs no further application-level locking to be accessed from multiple threads.

To help debugging problems in multithreaded applications, GLib supports error-checking mutexes that will give you helpful error messages on common problems. To use error-checking mutexes, define the symbol G_ERRORCHECK_MUTEXES when compiling the application.

Details

**G_THREADS_ENABLED**

```
#define G_THREADS_ENABLED
```

This macro is defined if GLib was compiled with thread support. This does not necessarily mean that there is a thread implementation available, but it does mean that the infrastructure is in place and that once you provide a thread implementation to g_thread_init(), GLib will be multi-thread safe. If G_THREADS_ENABLED is not defined, then Glib is not, and cannot be, multi-thread safe.

**G_THREADS_IMPL_POSIX**

```
#define G_THREADS_IMPL_POSIX
```

This macro is defined if POSIX style threads are used.

**G_THREADS_IMPL_NONE**

```
#define G_THREADS_IMPL_NONE
```

This macro is defined if no thread implementation is used. You can, however, provide one to g_thread_init() to make GLib multi-thread safe.

**G_THREAD_ERROR**

```
#define G_THREAD_ERROR g_thread_error_quark ()
```

The error domain of the GLib thread subsystem.

**enum GThreadError**

```
typedef enum
{
  G_THREAD_ERROR_AGAIN /* Resource temporarily unavailable */
} GThreadError;
```

Possible errors of thread related functions.

**G_THREAD_ERROR_AGAIN** a thread couldn’t be created due to resource shortage. Try again later.

**GThreadFunctions**

```
typedef struct {
  GMutex* (*mutex_new) (void);
  void (*mutex_lock) (GMutex *mutex);
  gboolean (*mutex_trylock) (GMutex *mutex);
  void (*mutex_unlock) (GMutex *mutex);
  void (*mutex_free) (GMutex *mutex);
  GCond* (*cond_new) (void);
  void (*cond_signal) (GCond *cond);
} GThreadFunctions;
```
This function table is used by `g_thread_init()` to initialize the thread system. The functions in the table are directly used by their `g_*` prepended counterparts (described in this document). For example, if you call `g_mutex_new()` then `mutex_new()` from the table provided to `g_thread_init()` will be called.

**NOTE**

Do not use this struct unless you know what you are doing.

### g_thread_init()

```c
void g_thread_init (GThreadFunctions *vtable);
```

If you use GLib from more than one thread, you must initialize the thread system by calling `g_thread_init()`. Most of the time you will only have to call `g_thread_init (NULL)`.

**NOTE**

Do not call `g_thread_init()` with a non-NULL parameter unless you really know what you are doing.
3.2. THREADS

**NOTE**

* g_thread_init() must not be called directly or indirectly as a callback from GLib. Also no mutexes may be currently locked while calling g_thread_init().

**NOTE**

* g_thread_init() changes the way in which GTimer measures elapsed time. As a consequence, timers that are running while g_thread_init() is called may report unreliable times.

  g_thread_init() might only be called once. On the second call it will abort with an error. If you want to make sure that the thread system is initialized, you can do this:

```c
if (!g_thread_supported ()) g_thread_init (NULL);
```

After that line, either the thread system is initialized or, if no thread system is available in GLib (i.e. either G_THREADS_ENABLED is not defined or G_THREADS_IMPL_NONE is defined), the program will abort.

If no thread system is available and vtable is NULL or if not all elements of vtable are non-NULL, then g_thread_init() will abort.

**NOTE**

To use g_thread_init() in your program, you have to link with the libraries that the command `pkg-config --libs gthread-2.0` outputs. This is not the case for all the other thread related functions of GLib. Those can be used without having to link with the thread libraries.

**vtable:** a function table of type GThreadFunctions, that provides the entry points to the thread system to be used.

**g_thread_supported ()**

```c
gboolean g_thread_supported (void);
```

This function returns TRUE if the thread system is initialized, and FALSE if it is not.

**NOTE**

This function is actually a macro. Apart from taking the address of it you can however use it as if it was a function.

**Returns :** %TRUE, if the thread system is initialized.

**g_thread_get_initialized ()**

```c
gboolean g_thread_get_initialized (void);
```

Indicates if g_thread_init() has been called.
CHAPTER 3. GLIB CORE APPLICATION...

3.2. THREADS

Returns: TRUE if threads have been initialized.

Since 2.20

GThreadFunc ()

gpointer (*GThreadFunc) (gpointer data);

Specifies the type of the func functions passed to g_thread_create() or g_thread_create_full().

data: data passed to the thread.

Returns: the return value of the thread, which will be returned by g_thread_join().

enum GThreadPriority
typedef enum
{
    G_THREAD_PRIORITY_LOW,
    G_THREAD_PRIORITY_NORMAL,
    G_THREAD_PRIORITY_HIGH,
    G_THREAD_PRIORITY_URGENT
} GThreadPriority;

Specifies the priority of a thread.

NOTE
It is not guaranteed that threads with different priorities really behave accordingly. On
some systems (e.g. Linux) there are no thread priorities. On other systems (e.g. Solaris)
there doesn’t seem to be different scheduling for different priorities. All in all try to avoid
being dependent on priorities.

G_THREAD_PRIORITY_LOW a priority lower than normal
G_THREAD_PRIORITY_NORMAL the default priority
G_THREAD_PRIORITY_HIGH a priority higher than normal
G_THREAD_PRIORITY_URGENT the highest priority

GThread
typedef struct {
} GThread;

The GThread struct represents a running thread. It has three public read-only members, but the
underlying struct is bigger, so you must not copy this struct.

NOTE
Resources for a joinable thread are not fully released until g_thread_join() is called for that
thread.
**g_thread_create()**

```c
GThread * g_thread_create (GThreadFunc func,
                          gpointer data,
                          gboolean joinable,
                          GError **error);
```

This function creates a new thread with the default priority. If `joinable` is `TRUE`, you can wait for this thread's termination calling `g_thread_join()`. Otherwise, the thread will just disappear when it terminates.

The new thread executes the function `func` with the argument `data`. If the thread was created successfully, it is returned.

`error` can be `NULL` to ignore errors, or non-`NULL` to report errors. The error is set, if and only if the function returns `NULL`.

**func**: a function to execute in the new thread.

**data**: an argument to supply to the new thread.

**joinable**: should this thread be joinable?

**error**: return location for error.

**Returns**: the new `GThread` on success.

**g_thread_create_full()**

```c
GThread* g_thread_create_full (GThreadFunc func,
                               gpointer data,
                               gulong stack_size,
                               gboolean joinable,
                               gboolean bound,
                               GThreadPriority priority,
                               GError **error);
```

This function creates a new thread with the priority `priority`. If the underlying thread implementation supports it, the thread gets a stack size of `stack_size` or the default value for the current platform, if `stack_size` is `0`.

If `joinable` is `TRUE`, you can wait for this thread's termination calling `g_thread_join()`. Otherwise, the thread will just disappear when it terminates. If `bound` is `TRUE`, this thread will be scheduled in the system scope, otherwise the implementation is free to do scheduling in the process scope. The first variant is more expensive resource-wise, but generally faster. On some systems (e.g. Linux) all threads are bound.

The new thread executes the function `func` with the argument `data`. If the thread was created successfully, it is returned.

`error` can be `NULL` to ignore errors, or non-`NULL` to report errors. The error is set, if and only if the function returns `NULL`.

**Note**

It is not guaranteed that threads with different priorities really behave accordingly. On some systems (e.g. Linux) there are no thread priorities. On other systems (e.g. Solaris) there doesn’t seem to be different scheduling for different priorities. All in all try to avoid being dependent on priorities. Use `G_THREAD_PRIORITY_NORMAL` here as a default.
3.2. THREADS

**NOTE**

Only use `g_thread_create_full()` if you really can’t use `g_thread_create()` instead. `g_thread_create()` does not take `stack_size`, `bound`, and `priority` as arguments, as they should only be used in cases in which it is unavoidable.

`func`: a function to execute in the new thread.

`data`: an argument to supply to the new thread.

`stack_size`: a stack size for the new thread.

`joinable`: should this thread be joinable?

`bound`: should this thread be bound to a system thread?

`priority`: a priority for the thread.

`error`: return location for error.

**Returns**: the new GThread on success.

### g_thread_self()

```c
GThread* g_thread_self (void);
```

This function returns the GThread corresponding to the calling thread.

**Returns**: the current thread.

### g_thread_join()

```c
gpointer g_thread_join (GThread *thread);
```

Waits until `thread` finishes, i.e. the function `func`, as given to `g_thread_create()`, returns or `g_thread_exit()` is called by `thread`. All resources of `thread` including the GThread struct are released. `thread` must have been created with `joinable=TRUE` in `g_thread_create()`. The value returned by `func` or given to `g_thread_exit()` by `thread` is returned by this function.

`thread`: a GThread to be waited for.

**Returns**: the return value of the thread.

### g_thread_set_priority()

```c
void g_thread_set_priority (GThread *thread,
GThreadPriority priority);
```

Changes the priority of `thread` to `priority`.

**NOTE**

It is not guaranteed that threads with different priorities really behave accordingly. On some systems (e.g. Linux) there are no thread priorities. On other systems (e.g. Solaris) there doesn’t seem to be different scheduling for different priorities. All in all try to avoid being dependent on priorities.
thread: a GThread.

priority: a new priority for thread.

g_thread_yield()

```c
void g_thread_yield();
```

Gives way to other threads waiting to be scheduled.
This function is often used as a method to make busy wait less evil. But in most cases you will encounter, there are better methods to do that. So in general you shouldn’t use this function.

g_thread_exit()

```c
void g_thread_exit (gpointer retval);
```

Exits the current thread. If another thread is waiting for that thread using g_thread_join() and the current thread is joinable, the waiting thread will be woken up and get `retval` as the return value of g_thread_join(). If the current thread is not joinable, `retval` is ignored. Calling

```
g_thread_exit (retval);
```

is equivalent to calling

```
return retval;
```

in the function `func`, as given to g_thread_create().

**Note**

Never call `g_thread_exit()` from within a thread of a GThreadPool, as that will mess up the bookkeeping and lead to funny and unwanted results.

`retval`: the return value of this thread.

g_thread_foreach()

```c
void g_thread_foreach (GFunc thread_func, 
          gpointer user_data);
```

Call `thread_func` on all existing GThread structures. Note that threads may decide to exit while `thread_func` is running, so without intimate knowledge about the lifetime of foreign threads, `thread_func` shouldn’t access the GThread* pointer passed in as first argument. However, `thread_func` will not be called for threads which are known to have exited already.

Due to thread lifetime checks, this function has an execution complexity which is quadratic in the number of existing threads.

`thread_func`: function to call for all GThread structures

`user_data`: second argument to `thread_func`

Since 2.10
GMutex

typedef struct _GMutex GMutex;

The GMutex struct is an opaque data structure to represent a mutex (mutual exclusion). It can be used to protect data against shared access. Take for example the following function:

Example 3.1 A function which will not work in a threaded environment

```c
int give_me_next_number ()
{
    static int current_number = 0;
    /* now do a very complicated calculation to calculate the new number,
    this might for example be a random number generator */
    current_number = calc_next_number (current_number);
    return current_number;
}
```

It is easy to see that this won’t work in a multi-threaded application. There current_number must be protected against shared access. A first naive implementation would be:

Example 3.2 The wrong way to write a thread-safe function

```c
int give_me_next_number ()
{
    static int current_number = 0;
    int ret_val;
    static GMutex * mutex = NULL;
    if (!mutex)
        mutex = g_mutex_new ();
    g_mutex_lock (mutex);
    ret_val = current_number = calc_next_number (current_number);
    g_mutex_unlock (mutex);
    return ret_val;
}
```

This looks like it would work, but there is a race condition while constructing the mutex and this code cannot work reliable. Please do not use such constructs in your own programs! One working solution is:

Example 3.3 A correct thread-safe function

```c
static GMutex *give_me_next_number_mutex = NULL;
/* this function must be called before any call to give_me_next_number ()
   it must be called exactly once. */
void init_give_me_next_number ()
{
    g_assert (give_me_next_number_mutex == NULL);
    give_me_next_number_mutex = g_mutex_new ();
}
int give_me_next_number ()
{
    static int current_number = 0;
    int ret_val;
    g_mutex_lock (give_me_next_number_mutex);
    ret_val = current_number = calc_next_number (current_number);
    g_mutex_unlock (give_me_next_number_mutex);
    return ret_val;
}
```
GStaticMutex provides a simpler and safer way of doing this.
If you want to use a mutex, and your code should also work without calling g_thread_init() first,
then you can not use a GMutex, as g_mutex_new() requires that the thread system be initialized. Use a
GStaticMutex instead.
A GMutex should only be accessed via the following functions.

NOTE

All of the g_mutex_* functions are actually macros. Apart from taking their addresses,
you can however use them as if they were functions.

### g_mutex_new()

GMutex * g_mutex_new ()

Creates a new GMutex.

NOTE

This function will abort if g_thread_init() has not been called yet.

Returns: a new GMutex.

### g_mutex_lock()

void g_mutex_lock (GMutex *mutex);

Locks mutex. If mutex is already locked by another thread, the current thread will block until mutex
is unlocked by the other thread.
This function can be used even if g_thread_init() has not yet been called, and, in that case, will do
nothing.

NOTE

GMutex is neither guaranteed to be recursive nor to be non-recursive, i.e. a thread could
deadlock while calling g_mutex_lock(), if it already has locked mutex. Use GStaticRec-
Mutex, if you need recursive mutexes.

mutex: a GMutex.

### g_mutex_trylock()

gboolean g_mutex_trylock (GMutex *mutex);

Tries to lock mutex. If mutex is already locked by another thread, it immediately returns FALSE.
Otherwise it locks mutex and returns TRUE.
This function can be used even if g_thread_init() has not yet been called, and, in that case, will
immediately return TRUE.
3.2. THREADS

**NOTE**

GMutex is neither guaranteed to be recursive nor to be non-recursive, i.e. the return value of `g_mutex_trylock()` could be both FALSE or TRUE, if the current thread already has locked mutex. Use GStaticRecMutex, if you need recursive mutexes.

**mutex**: a GMutex.

**Returns**: %TRUE, if mutex could be locked.

---

**g_mutex_unlock()**

```c
void g_mutex_unlock (GMutex *mutex);
```

Unlocks mutex. If another thread is blocked in a `g_mutex_lock()` call for mutex, it will be woken and can lock mutex itself.

This function can be used even if `g_thread_init()` has not yet been called, and, in that case, will do nothing.

**mutex**: a GMutex.

---

**g_mutex_free()**

```c
void g_mutex_free (GMutex *mutex);
```

Destroys mutex.

**mutex**: a GMutex.

---

**GStaticMutex**

```c
typedef struct _GStaticMutex GStaticMutex;
```

A GStaticMutex works like a GMutex, but it has one significant advantage. It doesn’t need to be created at run-time like a GMutex, but can be defined at compile-time. Here is a shorter, easier and safer version of our `give_me_next_number()` example:

---

**Example 3.4 Using GStaticMutex to simplify thread-safe programming**

```c
int give_me_next_number ()
{
    static int current_number = 0;
    int ret_val;
    static GStaticMutex mutex = G_STATIC_MUTEX_INIT;
    g_static_mutex_lock (&mutex);
    ret_val = current_number = calc_next_number (current_number);
    g_static_mutex_unlock (&mutex);
    return ret_val;
}
```

---

Sometimes you would like to dynamically create a mutex. If you don’t want to require prior calling to `g_thread_init()`, because your code should also be usable in non-threaded programs, you are not able to use `g_mutex_new()` and thus GMutex, as that requires a prior call to `g_thread_init()`. In these cases you can also use a GStaticMutex. It must be initialized with `g_static_mutex_init()` before using it and freed with with `g_static_mutex_free()` when not needed anymore to free up any allocated resources.
Even though GStaticMutex is not opaque, it should only be used with the following functions, as it is defined differently on different platforms.

All of the g_static_mutex_* functions apart from g_static_mutex_get_mutex can also be used even if g_thread_init() has not yet been called. Then they do nothing, apart from g_static_mutex_trylock, which does nothing but returning TRUE.

NOTE

All of the g_static_mutex_* functions are actually macros. Apart from taking their addresses, you can however use them as if they were functions.

G_STATIC_MUTEX_INIT

#define G_STATIC_MUTEX_INIT

A GStaticMutex must be initialized with this macro, before it can be used. This macro can be used to initialize a variable, but it cannot be assigned to a variable. In that case you have to use g_static_mutex_init().

GStaticMutex my_mutex = G_STATIC_MUTEX_INIT;

g_static_mutex_init ()

void g_static_mutex_init (GStaticMutex *mutex);

Initializes mutex. Alternatively you can initialize it with G_STATIC_MUTEX_INIT.

mutex: a GStaticMutex to be initialized.

g_static_mutex_lock ()

void g_static_mutex_lock (GStaticMutex *mutex);

Works like g_mutex_lock(), but for a GStaticMutex.

mutex: a GStaticMutex.

g_static_mutex_trylock ()

gboolean g_static_mutex_trylock (GStaticMutex *mutex);

Works like g_mutex_trylock(), but for a GStaticMutex.

mutex: a GStaticMutex.

Returns: %TRUE, if the GStaticMutex could be locked.

g_static_mutex_unlock ()

void g_static_mutex_unlock (GStaticMutex *mutex);

Works like g_mutex_unlock(), but for a GStaticMutex.

mutex: a GStaticMutex.
3.2. THREADS

**g_static_mutex_get_mutex ()**

GMutex * g_static_mutex_get_mutex (GStaticMutex *mutex);

For some operations (like `g_cond_wait()`) you must have a GMutex instead of a GStaticMutex. This function will return the corresponding GMutex for `mutex`.

*mutex*: a GStaticMutex.

*Returns*: the GMutex corresponding to `mutex`.

**g_static_mutex_free ()**

void g_static_mutex_free (GStaticMutex *mutex);

Releases all resources allocated to `mutex`.

You don’t have to call this functions for a GStaticMutex with an unbounded lifetime, i.e. objects declared ‘static’, but if you have a GStaticMutex as a member of a structure and the structure is freed, you should also free the GStaticMutex.

*mutex*: a GStaticMutex to be freed.

**G_LOCK_DEFINE()**

#define G_LOCK_DEFINE(name)

The G_LOCK_* macros provide a convenient interface to GStaticMutex with the advantage that they will expand to nothing in programs compiled against a thread-disabled GLib, saving code and memory there. G_LOCK_DEFINE defines a lock. It can appear anywhere variable definitions may appear in programs, i.e. in the first block of a function or outside of functions. The name parameter will be mangled to get the name of the GStaticMutex. This means that you can use names of existing variables as the parameter - e.g. the name of the variable you intent to protect with the lock. Look at our give_me_next_number() example using the G_LOCK_* macros:

**Example 3.5 Using the G_LOCK_* convenience macros**

```c
G_LOCK_DEFINE (current_number);
int give_me_next_number ()
{
    static int current_number = 0;
    int ret_val;
    G_LOCK (current_number);
    ret_val = current_number = calc_next_number (current_number);
    G_UNLOCK (current_number);
    return ret_val;
}
```

*name*: the name of the lock.

**G_LOCK_DEFINE_STATIC()**

#define G_LOCK_DEFINE_STATIC(name)

This works like G_LOCK_DEFINE, but it creates a static object.

*name*: the name of the lock.
### G_LOCK_EXTERN()

```c
#define G_LOCK_EXTERN(name)
```

This declares a lock, that is defined with G_LOCK_DEFINE in another module.

**name**: the name of the lock.

### G_LOCK()

```c
#define G_LOCK(name)
```

Works like g_mutex_lock(), but for a lock defined with G_LOCK_DEFINE.

**name**: the name of the lock.

### G_TRYLOCK()

```c
#define G_TRYLOCK(name)
```

Works like g_mutex_trylock(), but for a lock defined with G_LOCK_DEFINE.

**name**: the name of the lock.

**Returns**: %TRUE, if the lock could be locked.

### G_UNLOCK()

```c
#define G_UNLOCK(name)
```

Works like g_mutex_unlock(), but for a lock defined with G_LOCK_DEFINE.

**name**: the name of the lock.

### G_StaticRecMutex

```c
typedef struct {
    
} GStaticRecMutex;
```

A GStaticRecMutex works like a GStaticMutex, but it can be locked multiple times by one thread. If you enter it n times, you have to unlock it n times again to let other threads lock it. An exception is the function g_static_rec_mutex_unlock_full(): that allows you to unlock a GStaticRecMutex completely returning the depth, (i.e. the number of times this mutex was locked). The depth can later be used to restore the state of the GStaticRecMutex by calling g_static_rec_mutex_lock_full().

Even though GStaticRecMutex is not opaque, it should only be used with the following functions.

All of the g_static_rec_mutex_* functions can be used even if g_thread_init() has not been called. Then they do nothing, apart from g_static_rec_mutex_trylock, which does nothing but returning TRUE.

### G_STATIC_REC_MUTEX_INIT

```c
#define G_STATIC_REC_MUTEX_INIT { G_STATIC_MUTEX_INIT }
```

A GStaticRecMutex must be initialized with this macro before it can be used. This macro can used be to initialize a variable, but it cannot be assigned to a variable. In that case you have to use g_static_rec_mutex_init().

```c
GStaticRecMutex my_mutex = G_STATIC_REC_MUTEX_INIT;
```
CHAPTER 3. GLIB CORE APPLICATION

3.2. THREADS

**g_static_rec_mutex_init ()**

```c
void g_static_rec_mutex_init (GStaticRecMutex *mutex);
```

A GStaticRecMutex must be initialized with this function before it can be used. Alternatively you can initialize it with `G_STATIC_REC_MUTEX_INIT`.

**mutex**: a GStaticRecMutex to be initialized.

**g_static_rec_mutex_lock ()**

```c
void g_static_rec_mutex_lock (GStaticRecMutex *mutex);
```

Locks `mutex`. If `mutex` is already locked by another thread, the current thread will block until `mutex` is unlocked by the other thread. If `mutex` is already locked by the calling thread, this functions increases the depth of `mutex` and returns immediately.

**mutex**: a GStaticRecMutex to lock.

**g_static_rec_mutex_trylock ()**

```c
gboolean g_static_rec_mutex_trylock (GStaticRecMutex *mutex);
```

Tries to lock `mutex`. If `mutex` is already locked by another thread, it immediately returns `FALSE`. Otherwise it locks `mutex` and returns `TRUE`. If `mutex` is already locked by the calling thread, this functions increases the depth of `mutex` and immediately returns `TRUE`.

**mutex**: a GStaticRecMutex to lock.

**Returns**: %TRUE, if `mutex` could be locked.

**g_static_rec_mutex_unlock ()**

```c
void g_static_rec_mutex_unlock (GStaticRecMutex *mutex);
```

Unlocks `mutex`. Another thread will be allowed to lock `mutex` only when it has been unlocked as many times as it had been locked before. If `mutex` is completely unlocked and another thread is blocked in a `g_static_rec_mutex_lock()` call for `mutex`, it will be woken and can lock `mutex` itself.

**mutex**: a GStaticRecMutex to unlock.

**g_static_rec_mutex_lock_full ()**

```c
void g_static_rec_mutex_lock_full (GStaticRecMutex *mutex, guint depth);
```

Works like calling `g_static_rec_mutex_lock()` for `mutex` depth times.

**mutex**: a GStaticRecMutex to lock.

**depth**: number of times this mutex has to be unlocked to be completely unlocked.

**g_static_rec_mutex_unlock_full ()**

```c
guint g_static_rec_mutex_unlock_full (GStaticRecMutex *mutex);
```

Completely unlocks `mutex`. If another thread is blocked in a `g_static_rec_mutex_lock()` call for `mutex`, it will be woken and can lock `mutex` itself. This function returns the number of times that `mutex` has been locked by the current thread. To restore the state before the call to `g_static_rec_mutex_unlock_full()` you can call `g_static_rec_mutex_lock_full()` with the depth returned by this function.

**mutex**: a GStaticRecMutex to completely unlock.

**Returns**: number of times `mutex` has been locked by the current thread.
g_static_rec_mutex_free

```c
void g_static_rec_mutex_free (GStaticRecMutex *mutex);
```

Releases all resources allocated to a GStaticRecMutex.

You don’t have to call this function for a GStaticRecMutex with an unbounded lifetime, i.e. objects
declared ‘static’, but if you have a GStaticRecMutex as a member of a structure and the structure is freed,
you should also free the GStaticRecMutex.

**mutex**: a GStaticRecMutex to be freed.

GStaticRWLock

```c
typedef struct {
} GStaticRWLock;
```

The GStaticRWLock struct represents a read-write lock. A read-write lock can be used for protecting
data that some portions of code only read from, while others also write. In such situations it is desirable
that several readers can read at once, whereas of course only one writer may write at a time. Take a look
at the following example:

**Example 3.6 An array with access functions**

```c
GStaticRWLock rwlock = G_STATIC_RW_LOCK_INIT;
GPtrArray *array;

gpointer my_array_get (guint index)
{
    gpointer retval = NULL;
    if (!array)
       return NULL;
    g_static_rwlock_reader_lock (&rwlock);
    if (index < array->len)
       retval = g_ptr_array_index (array, index);
    g_static_rwlock_reader_unlock (&rwlock);
    return retval;
}

void my_array_set (guint index, gpointer data)
{
    g_static_rwlock_writer_lock (&rwlock);
    if (!array)
       array = g_ptr_array_new ();
    if (index >= array->len)
       g_ptr_array_set_size (array, index+1);
    g_ptr_array_index (array, index) = data;
    g_static_rwlock_writer_unlock (&rwlock);
}
```

This example shows an array which can be accessed by many readers (the *my_array_get()* function)
simultaneously, whereas the writers (the *my_array_set()* function) will only be allowed once
at a time and only if no readers currently access the array. This is because of the potentially dangerous
resizing of the array. Using these functions is fully multi-thread safe now.

Most of the time, writers should have precedence over readers. That means, for this implementation,
that as soon as a writer wants to lock the data, no other reader is allowed to lock the data, whereas, of
course, the readers that already have locked the data are allowed to finish their operation. As soon as
the last reader unlocks the data, the writer will lock it.

Even though GStaticRWLock is not opaque, it should only be used with the following functions.

All of the *g_static_rwlock_* functions can be used even if *g_thread_init()* has not been called.
Then they do nothing, apart from *g_static_rwlock_*_trylock*, which does nothing but returning
TRUE.
CHAPTER 3. GLIB CORE APPLICATION

3.2. THREADS

NOTE

A read-write lock has a higher overhead than a mutex. For example, both g_static_rwlock_reader_lock() and g_static_rwlock_reader_unlock() have to lock and unlock a GStaticMutex, so it takes at least twice the time to lock and unlock a GStaticRWLock that it does to lock and unlock a GStaticMutex. So only data structures that are accessed by multiple readers, and which keep the lock for a considerable time justify a GStaticRWLock. The above example most probably would fare better with a GStaticMutex.

G_STATIC_RW_LOCK_INIT

#define G_STATIC_RW_LOCK_INIT { G_STATIC_MUTEX_INIT, NULL, NULL, 0, FALSE, 0, 0 }

A GStaticRWLock must be initialized with this macro before it can be used. This macro can be used to initialize a variable, but it cannot be assigned to a variable. In that case you have to use g_static_rwlock_init().

GStaticRWLock my_lock = G_STATIC_RW_LOCK_INIT;

g_static_rwlock_init()

void g_static_rwlock_init (GStaticRWLock *lock);

A GStaticRWLock must be initialized with this function before it can be used. Alternatively you can initialize it with G_STATIC_RW_LOCK_INIT.

lock: a GStaticRWLock to be initialized.

g_static_rwlock_reader_lock()

void g_static_rwlock_reader_lock (GStaticRWLock *lock);

Locks lock for reading. There may be unlimited concurrent locks for reading of a GStaticRWLock at the same time. If lock is already locked for writing by another thread or if another thread is already waiting to lock lock for writing, this function will block until lock is unlocked by the other writing thread and no other writing threads want to lock lock. This lock has to be unlocked by g_static_rwlock_reader_unlock().

GStaticRWLock is not recursive. It might seem to be possible to recursively lock for reading, but that can result in a deadlock, due to writer preference.

lock: a GStaticRWLock to lock for reading.

g_static_rwlock_reader_trylock()

gboolean g_static_rwlock_reader_trylock (GStaticRWLock *lock);

Tries to lock lock for reading. If lock is already locked for writing by another thread or if another thread is already waiting to lock lock for writing, immediately returns FALSE. Otherwise locks lock for reading and returns TRUE. This lock has to be unlocked by g_static_rwlock_reader_unlock().

lock: a GStaticRWLock to lock for reading.

Returns: %TRUE, if lock could be locked for reading.
CHAPTER 3. GLIB CORE APPLICATION...

3.2. THREADS

**g_static_rw_lock_reader_unlock ()**

```c
void g_static_rw_lock_reader_unlock (GStaticRWLock *lock);
```

Unlocks `lock`. If a thread waits to lock `lock` for writing and all locks for reading have been unlocked, the waiting thread is woken up and can lock `lock` for writing.

**lock**: a `GStaticRWLock` to unlock after reading.

**g_static_rw_lock_writer_lock ()**

```c
void g_static_rw_lock_writer_lock (GStaticRWLock *lock);
```

Locks `lock` for writing. If `lock` is already locked for writing or reading by other threads, this function will block until `lock` is completely unlocked and then lock `lock` for writing. While this function waits to lock `lock`, no other thread can lock `lock` for reading. When `lock` is locked for writing, no other thread can lock `lock` (neither for reading nor writing). This lock has to be unlocked by `g_static_rw_lock_writer_unlock()`.

**lock**: a `GStaticRWLock` to lock for writing.

**g_static_rw_lock_writer_trylock ()**

```c
gboolean g_static_rw_lock_writer_trylock (GStaticRWLock *lock);
```

Tries to lock `lock` for writing. If `lock` is already locked (for either reading or writing) by another thread, it immediately returns FALSE. Otherwise it locks `lock` for writing and returns TRUE. This lock has to be unlocked by `g_static_rw_lock_writer_unlock()`.

**lock**: a `GStaticRWLock` to lock for writing.

**Returns**: %TRUE, if `lock` could be locked for writing.

**g_static_rw_lock_writer_unlock ()**

```c
void g_static_rw_lock_writer_unlock (GStaticRWLock *lock);
```

Unlocks `lock`. If a thread is waiting to lock `lock` for writing and all locks for reading have been unlocked, the waiting thread is woken up and can lock `lock` for writing. If no thread is waiting to lock `lock` for writing, and some thread or threads are waiting to lock `lock` for reading, the waiting threads are woken up and can lock `lock` for reading.

**lock**: a `GStaticRWLock` to unlock after writing.

**g_static_rw_lock_free ()**

```c
void g_static_rw_lock_free (GStaticRWLock *lock);
```

Releases all resources allocated to `lock`. You don’t have to call this functions for a `GStaticRWLock` with an unbounded lifetime, i.e. objects declared ‘static’, but if you have a `GStaticRWLock` as a member of a structure, and the structure is freed, you should also free the `GStaticRWLock`.

**lock**: a `GStaticRWLock` to be freed.
GCond

typedef struct _GCond GCond;

The GCond struct is an opaque data structure that represents a condition. Threads can block on a GCond if they find a certain condition to be false. If other threads change the state of this condition they signal the GCond, and that causes the waiting threads to be woken up.

Example 3.7 Using GCond to block a thread until a condition is satisfied

GCond* data_cond = NULL; /* Must be initialized somewhere */
GMutex* data_mutex = NULL; /* Must be initialized somewhere */
gpointer current_data = NULL;
void push_data (gpointer data)
{
    g_mutex_lock (data_mutex);
    current_data = data;
    g_cond_signal (data_cond);
    g_mutex_unlock (data_mutex);
}
gpointer pop_data ()
{
    gpointer data;
    g_mutex_lock (data_mutex);
    while (!current_data)
        g_cond_wait (data_cond, data_mutex);
    data = current_data;
    current_data = NULL;
    g_mutex_unlock (data_mutex);
    return data;
}

Whenever a thread calls pop_data() now, it will wait until current_data is non-NULL, i.e. until some other thread has called push_data().

**Note**

It is important to use the g_cond_wait() and g_cond_timed_wait() functions only inside a loop which checks for the condition to be true. It is not guaranteed that the waiting thread will find the condition fulfilled after it wakes up, even if the signaling thread left the condition in that state: another thread may have altered the condition before the waiting thread got the chance to be woken up, even if the condition itself is protected by a GMutex, like above.

A GCond should only be accessed via the following functions.

**Note**

All of the g_cond_* functions are actually macros. Apart from taking their addresses, you can however use them as if they were functions.

g_cond_new ()

GCond* g_cond_new ();

124
CHAPTER 3. GLIB CORE APPLICATION...

3.2. THREADS

Creates a new GCond. This function will abort, if g_thread_init() has not been called yet.

Returns: a new GCond.

**g_cond_signal ()**

```c
void g_cond_signal (GCond *cond);
```

If threads are waiting for `cond`, exactly one of them is woken up. It is good practice to hold the same lock as the waiting thread while calling this function, though not required.

This function can be used even if g_thread_init() has not yet been called, and, in that case, will do nothing.

**cond**: a GCond.

**g_cond_broadcast ()**

```c
void g_cond_broadcast (GCond *cond);
```

If threads are waiting for `cond`, all of them are woken up. It is good practice to lock the same mutex as the waiting threads, while calling this function, though not required.

This function can be used even if g_thread_init() has not yet been called, and, in that case, will do nothing.

**cond**: a GCond.

**g_cond_wait ()**

```c
void g_cond_wait (GCond *cond, GMutex *mutex);
```

Waits until this thread is woken up on `cond`. The mutex is unlocked before falling asleep and locked again before resuming.

This function can be used even if g_thread_init() has not yet been called, and, in that case, will immediately return.

**cond**: a GCond.

**mutex**: a GMutex, that is currently locked.

**g_cond_timed_wait ()**

```c
gboolean g_cond_timed_wait (GCond *cond, GMutex *mutex, GTimeVal *abs_time);
```

Waits until this thread is woken up on `cond`, but not longer than until the time specified by `abs_time`. The mutex is unlocked before falling asleep and locked again before resuming.

If `abs_time` is NULL, g_cond_timed_wait() acts like g_cond_wait().

This function can be used even if g_thread_init() has not yet been called, and, in that case, will immediately return TRUE.

To easily calculate `abs_time` a combination of g_get_current_time() and g_time_val_add() can be used.

**cond**: a GCond.

**mutex**: a GMutex that is currently locked.

**abs_time**: a GTimeVal, determining the final time.

Returns: %TRUE if `cond` was signalled, or FALSE on timeout.
**g_cond_free()**

```c
void g_cond_free (GCond *cond);
```

Destroys the GCond.

*cond*: a GCond.

**GPrivate**

```c
typedef struct _GPrivate GPrivate;
```

The `GPrivate` struct is an opaque data structure to represent a thread private data key. Threads can thereby obtain and set a pointer which is private to the current thread. Take our `give_me_next_number()` example from above. Suppose we don’t want `current_number` to be shared between the threads, but instead to be private to each thread. This can be done as follows:

**Example 3.8 Using GPrivate for per-thread data**

```c
GPrivate* current_number_key = NULL; /* Must be initialized somewhere */
    /* with g_private_new (g_free); */
int give_me_next_number ()
{   int *current_number = g_private_get (current_number_key);
    if (!current_number)
    {        current_number = g_new (int, 1);
        *current_number = 0;
        g_private_set (current_number_key, current_number);
    }
    *current_number = calc_next_number (*current_number);
    return *current_number;
}
```

Here the pointer belonging to the key `current_number_key` is read. If it is NULL, it has not been set yet. Then get memory for an integer value, assign this memory to the pointer and write the pointer back. Now we have an integer value that is private to the current thread.

The `GPrivate` struct should only be accessed via the following functions.

---

**NOTE**

All of the `g_private_*` functions are actually macros. Apart from taking their addresses, you can however use them as if they were functions.

---

**g_private_new()**

```c
GPrivate* g_private_new (GDestroyNotify destructor);
```

Creates a new GPrivate. If *destructor* is non-NULL, it is a pointer to a destructor function. Whenever a thread ends and the corresponding pointer keyed to this instance of GPrivate is non-NULL, the destructor is called with this pointer as the argument.
CHAPTER 3. GLIB CORE APPLICATION…

3.2. THREADS

**NOTE**

destructor is used quite differently from notify in g_static_private_set().

**NOTE**

A GPrivate cannot be freed. Reuse it instead, if you can, to avoid shortage, or use GStaticPrivate.

**NOTE**

This function will abort if g_thread_init() has not been called yet.

destructor: a function to destroy the data keyed to GPrivate when a thread ends.

*Returns*: a new GPrivate.

g_private_get()

```c
gpointer g_private_get (GPrivate *private_key);
```

Returns the pointer keyed to private_key for the current thread. If g_private_set() hasn’t been called for the current private_key and thread yet, this pointer will be NULL.

This function can be used even if g_thread_init() has not yet been called, and, in that case, will return the value of private_key casted to gpointer. Note however, that private data set before g_thread_init() will not be retained after the call. Instead, NULL will be returned in all threads directly after g_thread_init(), regardless of any g_private_set() calls issued before threading system initialization.

*private_key*: a GPrivate.

*Returns*: the corresponding pointer.

g_private_set()

```c
void g_private_set (GPrivate *private_key, gpointer data);
```

Sets the pointer keyed to private_key for the current thread.

This function can be used even if g_thread_init() has not yet been called, and, in that case, will set private_key to data casted to GPrivate*. See g_private_get() for resulting caveats.

*private_key*: a GPrivate.

*data*: the new pointer.

GStaticPrivate

```c
typedef struct {
} GStaticPrivate;
```
A GStaticPrivate works almost like a GPrivate, but it has one significant advantage. It doesn’t need to be created at run-time like a GPrivate, but can be defined at compile-time. This is similar to the difference between GMutex and GStaticMutex. Now look at our give_me_next_number() example with

Example 3.9 Using GStaticPrivate for per-thread data

```c
int give_me_next_number ()
{
    static GStaticPrivate current_number_key = G_STATIC_PRIVATE_INIT;
    int *current_number = g_static_private_get (&current_number_key);
    if (!current_number)
    {
        current_number = g_new (int,1);
        *current_number = 0;
        g_static_private_set (&current_number_key, current_number, g_free);
    }
    *current_number = calc_next_number (*current_number);
    return *current_number;
}
```

G_STATIC_PRIVATE_INIT

```c
#define G_STATIC_PRIVATE_INIT
```

Every GStaticPrivate must be initialized with this macro, before it can be used.

```c
GStaticPrivate my_private = G_STATIC_PRIVATE_INIT;
```

g_static_private_init()

```c
void g_static_private_init (GStaticPrivate *
private_key);
```

Initializes private_key. Alternatively you can initialize it with G_STATIC_PRIVATE_INIT.

**private_key**: a GStaticPrivate to be initialized.

g_static_private_get()

```c
gpointer g_static_private_get (GStaticPrivate *
private_key);
```

Works like g_private_get() only for a GStaticPrivate.
This function works even if g_thread_init() has not yet been called.

**private_key**: a GStaticPrivate.

**Returns**: the corresponding pointer.

g_static_private_set()

```c
void g_static_private_set (GStaticPrivate *
private_key,
                gpointer data,
                GDestroyNotify notify);
```
Sets the pointer keyed to `private_key` for the current thread and the function `notify` to be called with that pointer (NULL or non-NULL), whenever the pointer is set again or whenever the current thread ends.

This function works even if `g_thread_init()` has not yet been called. If `g_thread_init()` is called later, the `data` keyed to `private_key` will be inherited only by the main thread, i.e. the one that called `g_thread_init()`.

**NOTE**

`notify` is used quite differently from `destructor` in `g_private_new()`.

`private_key`: a GStaticPrivate.

`data`: the new pointer.

`notify`: a function to be called with the pointer whenever the current thread ends or sets this pointer again.

```c
void g_static_private_free (GStaticPrivate * private_key);
```

Releases all resources allocated to `private_key`.

You don’t have to call this functions for a GStaticPrivate with an unbounded lifetime, i.e. objects declared ‘static’, but if you have a GStaticPrivate as a member of a structure and the structure is freed, you should also free the GStaticPrivate.

`private_key`: a GStaticPrivate to be freed.

**GOnce**

```c
typedef struct {
    volatile GOnceStatus status;
    volatile gpointer retval;
} GOnce;
```

A GOnce struct controls a one-time initialization function. Any one-time initialization function must have its own unique GOnce struct.

`volatile GOnceStatus status;` the status of the GOnce

`volatile gpointer retval;` the value returned by the call to the function, if `status` is `G_ONCE_STATUS_READY`

Since 2.4

```c
eum GOnceStatus
{
    G_ONCE_STATUS_NOTCALLED,
    G_ONCE_STATUS_PROGRESS,
    G_ONCE_STATUS_READY
} GOnceStatus;
```

The possible statuses of a one-time initialization function controlled by a GOnce struct.

`G_ONCE_STATUS_NOTCALLED` the function has not been called yet.
**G_ONCE_STATUS_PROGRESS** the function call is currently in progress.

**G_ONCE_STATUS_READY** the function has been called.

Since 2.4

### G_ONCE_INIT

```c
#define G_ONCE_INIT { G_ONCE_STATUS_NOTCALLED, NULL }
```

A **GOnce** must be initialized with this macro before it can be used.

```c
GOnce my_once = G_ONCE_INIT;
```

Since 2.4

### g_once()

```c
#define g_once(once, func, arg)
```

The first call to this routine by a process with a given **GOnce** struct calls `func` with the given argument. Thereafter, subsequent calls to `g_once()` with the same **GOnce** struct do not call `func` again, but return the stored result of the first call. On return from `g_once()`, the status of `once` will be **G_ONCE_STATUS_READY**.

For example, a mutex or a thread-specific data key must be created exactly once. In a threaded environment, calling `g_once()` ensures that the initialization is serialized across multiple threads.

**NOTE**

Calling `g_once()` recursively on the same **GOnce** struct in `func` will lead to a deadlock.

```c
static gpointer
get_debug_flags ()
{
    static GOnce my_once = G_ONCE_INIT;
    g_once (&my_once, parse_debug_flags, NULL);
    return my_once.retval;
}
```

**once**: a **GOnce** structure

**func**: the **GThreadFunc** function associated to `once`. This function is called only once, regardless of the number of times it and its associated **GOnce** struct are passed to `g_once()`.

**arg**: data to be passed to `func`

Since 2.4

### g_once_init_enter()

```c
gboolean g_once_init_enter (volatile gsize *value_location);
```

Function to be called when starting a critical initialization section. The argument `value_location` must point to a static 0-initialized variable that will be set to a value other than 0 at the end of the initialization section. In combination with `g_once_init_leave()` and the unique address `value_location`, it can be ensured that an initialization section will be executed only once during a program’s life time, and that concurrent threads are blocked until initialization completed. To be used in constructs like this:
static gsize initialization_value = 0;
if (g_once_init_enter (&initialization_value)) /* section start */
{
    gsize setup_value = 42; /* initialization code here */
    g_once_init_leave (&initialization_value, setup_value); /* section end */
}
/* use initialization_value here */

value_location: location of a static initializable variable containing 0.

Returns: %TRUE if the initialization section should be entered, FALSE and blocks otherwise

Since 2.14

g_once_init_leave()

void g_once_init_leave (volatile gsize *value_location,
                        gsize *initialization_value);

Counterpart to g_once_init_enter(). Expects a location of a static 0-initialized initialization variable,
and an initialization value other than 0. Sets the variable to the initialization value, and releases concurrent
threads blocking in g_once_init_enter() on this initialization variable.

value_location: location of a static initializable variable containing 0.
initialization_value: new non-0 value for *value_location.

Since 2.14

See Also

GThreadPool Thread pools.
GAsyncQueue Send asynchronous messages between threads.

3.3 Thread Pools

Name
Thread Pools – pools of threads to execute work concurrently

Synopsis

#include <glib.h>

GThreadPool; GThreadPool* g_thread_pool_new (GFunc func,
(gpointer user_data,
gint max_threads,
gboolean exclusive,
GError **error);

void g_thread_pool_push (GThreadPool *pool,
gpointer data,
GError **error);

void g_thread_pool_set_max_threads (GThreadPool *pool,
CHAPTER 3. GLIB CORE APPLICATION

3.3. THREAD POOLS

```c
int g_thread_pool_get_max_threads,
GError **error);
uint g_thread_pool_get_num_threads
(GThreadPool *pool);
uint g_thread_pool_unprocessed
(GThreadPool *pool);
void g_thread_pool_free
(GThreadPool *pool,
gboolean immediate,
gboolean wait_);
void g_thread_pool_set_max_unused_threads
(gint max_threads);
int g_thread_pool_get_max_unused_threads
(void);
uint g_thread_pool_get_num_unused_threads
(void);
void g_thread_pool_stop_unused_threads
(void);
void g_thread_pool_set_sort_function
(GThreadPool *pool,
GCompareDataFunc func,
 gpointer user_data);
```  
```c
void g_thread_pool_set_max_idle_time
(guint interval);
uint g_thread_pool_get_max_idle_time
(void);
```  
Description

Sometimes you wish to asynchronously fork out the execution of work and continue working in your
own thread. If that will happen often, the overhead of starting and destroying a thread each time might
be too high. In such cases reusing already started threads seems like a good idea. And it indeed is, but
implementing this can be tedious and error-prone.

Therefore GLib provides thread pools for your convenience. An added advantage is, that the threads
can be shared between the different subsystems of your program, when they are using GLib.

To create a new thread pool, you use g_thread_pool_new(). It is destroyed by g_thread_pool_free().
If you want to execute a certain task within a thread pool, you call g_thread_pool_push().
To get the current number of running threads you call g_thread_pool_get_num_threads(). To get the
number of still unprocessed tasks you call g_thread_pool_unprocessed(). To control the maximal num-
ber of threads for a thread pool, you use g_thread_pool_get_max_threads() and g_thread_pool_set_max_threads().

Finally you can control the number of unused threads, that are kept alive by GLib for future use. The
current number can be fetched with g_thread_pool_get_num_unused_threads(). The maximal number
can be controlled by g_thread_pool_get_max_unused_threads() and g_thread_pool_set_max_unused_threads().
All currently unused threads can be stopped by calling g_thread_pool_stop_unused_threads().

Details

GThreadPool

```c
typedef struct {
    GFunc func;
    gpointer user_data;
    gboolean exclusive;
} GThreadPool;
```  
The **GThreadPool** struct represents a thread pool. It has three public read-only members, but the
underlying struct is bigger, so you must not copy this struct.

**GFunc func**; the function to execute in the threads of this pool

**gpointer user_data**; the user data for the threads of this pool

**gboolean exclusive**; are all threads exclusive to this pool
### 3.3. THREAD POOLS

**g_thread_pool_new()**

```c
GThreadPool* g_thread_pool_new (GFunc func,
                                 gpointer user_data,
                                 gint max_threads,
                                 gboolean exclusive,
                                 GError **error);
```

This function creates a new thread pool. Whenever you call `g_thread_pool_push()`, either a new thread is created or an unused one is reused. At most `max_threads` threads are running concurrently for this thread pool. `max_threads = -1` allows unlimited threads to be created for this thread pool. The newly created or reused thread now executes the function `func` with the two arguments. The first one is the parameter to `g_thread_pool_push()` and the second one is `user_data`.

The parameter `exclusive` determines, whether the thread pool owns all threads exclusive or whether the threads are shared globally. If `exclusive` is `TRUE`, `max_threads` threads are started immediately and they will run exclusively for this thread pool until it is destroyed by `g_thread_pool_free()`. If `exclusive` is `FALSE`, threads are created, when needed and shared between all non-exclusive thread pools. This implies that `max_threads` may not be `-1` for exclusive thread pools.

`error` can be `NULL` to ignore errors, or non-`NULL` to report errors. An error can only occur when `exclusive` is set to `TRUE` and not all `max_threads` threads could be created.

**func**: a function to execute in the threads of the new thread pool

**user_data**: user data that is handed over to `func` every time it is called

**max_threads**: the maximal number of threads to execute concurrently in the new thread pool, `-1` means no limit

**exclusive**: should this thread pool be exclusive?

**error**: return location for error

Returns: the new `GThreadPool`

**g_thread_pool_push()**

```c
void g_thread_pool_push (GThreadPool *pool,
                         gpointer data,
                         GError **error);
```

Inserts `data` into the list of tasks to be executed by `pool`. When the number of currently running threads is lower than the maximal allowed number of threads, a new thread is started (or reused) with the properties given to `g_thread_pool_new()`. Otherwise `data` stays in the queue until a thread in this pool finishes its previous task and processes `data`.

`error` can be `NULL` to ignore errors, or non-`NULL` to report errors. An error can only occur when a new thread couldn’t be created. In that case `data` is simply appended to the queue of work to do.

**pool**: a `GThreadPool`

**data**: a new task for `pool`

**error**: return location for error

**g_thread_pool_set_max_threads()**

```c
void g_thread_pool_set_max_threads (GThreadPool *pool,
                                   gint max_threads,
                                   GError **error);
```
Sets the maximal allowed number of threads for `pool`. A value of -1 means, that the maximal number of threads is unlimited.

Setting `max_threads` to 0 means stopping all work for `pool`. It is effectively frozen until `max_threads` is set to a non-zero value again.

A thread is never terminated while calling `func`, as supplied by `g_thread_pool_new()`. Instead the maximal number of threads only has effect for the allocation of new threads in `g_thread_pool_push()`. A new thread is allocated, whenever the number of currently running threads in `pool` is smaller than the maximal number.

`error` can be `NULL` to ignore errors, or non-`NULL` to report errors. An error can only occur when a new thread couldn’t be created.

`pool`: a `GThreadPool`

`max_threads`: a new maximal number of threads for `pool`

`error`: return location for error

```c
void g_thread_pool_free (GThreadPool *pool,
                         gboolean immediate,
                         gboolean wait_);
```

Frees all resources allocated for `pool`. If `immediate` is `TRUE`, no new task is processed for `pool`. Otherwise `pool` is not freed before the last task is processed. Note however, that no thread of this pool is interrupted, while processing a task. Instead at least all still running threads can finish their tasks before the `pool` is freed.

If `wait_` is `TRUE`, the functions does not return before all tasks to be processed (dependent on `immediate`, whether all or only the currently running) are ready. Otherwise the function returns immediately.

After calling this function `pool` must not be used anymore.

`pool`: a `GThreadPool`

`immediate`: should `pool` shut down immediately?

`wait_`: should the function wait for all tasks to be finished?
### 3.3. Thread Pools

#### g_thread_pool_set_max_unused_threads ()

```c
void g_thread_pool_set_max_unused_threads (gint max_threads);
```

Sets the maximal number of unused threads to `max_threads`. If `max_threads` is -1, no limit is imposed on the number of unused threads.

**max_threads**: maximal number of unused threads

#### g_thread_pool_get_max_unused_threads ()

```c
gint g_thread_pool_get_max_unused_threads (void);
```

Returns the maximal allowed number of unused threads.

**Returns**: the maximal number of unused threads

#### g_thread_pool_get_num_unused_threads ()

```c
guint g_thread_pool_get_num_unused_threads (void);
```

Returns the number of currently unused threads.

**Returns**: the number of currently unused threads

#### g_thread_pool_stop_unused_threads ()

```c
void g_thread_pool_stop_unused_threads (void);
```

Stops all currently unused threads. This does not change the maximal number of unused threads. This function can be used to regularly stop all unused threads e.g. from `g_timeout_add()`.

#### g_thread_pool_set_sort_function ()

```c
void g_thread_pool_set_sort_function (GThreadPool *pool,
                                      GCompareDataFunc func,
                                      gpointer user_data);
```

Sets the function used to sort the list of tasks. This allows the tasks to be processed by a priority determined by `func`, and not just in the order in which they were added to the pool.

Note, if the maximum number of threads is more than 1, the order that threads are executed cannot be guaranteed 100%. Threads are scheduled by the operating system and are executed at random. It cannot be assumed that threads are executed in the order they are created.

**pool**: a `GThreadPool`

**func**: the `GCompareDataFunc` used to sort the list of tasks. This function is passed two tasks. It should return 0 if the order in which they are handled does not matter, a negative value if the first task should be processed before the second or a positive value if the second task should be processed first.

**user_data**: user data passed to `func`.

Since 2.10
3.4. Asynchronous Queues

Name
Asynchronous Queues – asynchronous communication between threads

Synopsis

```c
#include <glib.h>

GAsyncQueue;
GAsyncQueue* g_async_queue_new (void);
GAsyncQueue* g_async_queue_new_full (GDestroyNotify item_free_func);
GAsyncQueue* g_async_queue_unref (GAsyncQueue *queue);
void g_async_queue_push (GAsyncQueue *queue, gpointer data);
void g_async_queue_push_sorted (GAsyncQueue *queue, gpointer data,
                                GCompareDataFunc func, gpointer user_data);
gpointer g_async_queue_pop (GAsyncQueue *queue);
gpointer g_async_queue_try_pop (GAsyncQueue *queue);
gpointer g_async_queue_timed_pop (GAsyncQueue *queue,
                                  GTimeVal *end_time);
gint g_async_queue_length (GAsyncQueue *queue);
void g_async_queue_sort (GAsyncQueue *queue,
                         GCompareDataFunc func, gpointer user_data);
```

### g_thread_pool_set_max_idle_time ()

```c
void g_thread_pool_set_max_idle_time (guint interval);
```

This function will set the maximum `interval` that a thread waiting in the pool for new tasks can be idle for before being stopped. This function is similar to calling `g_thread_pool_stop_unused_threads()` on a regular timeout, except, this is done on a per thread basis.

By setting `interval` to 0, idle threads will not be stopped. This function makes use of `g_async_queue_timed_pop()` using `interval`.

`interval`: the maximum `interval` (1/1000ths of a second) a thread can be idle.

Since 2.10

### g_thread_pool_get_max_idle_time ()

```c
guint g_thread_pool_get_max_idle_time (void);
```

This function will return the maximum `interval` that a thread will wait in the thread pool for new tasks before being stopped.

If this function returns 0, threads waiting in the thread pool for new work are not stopped.

**Returns**: the maximum `interval` to wait for new tasks in the thread pool before stopping the thread (1/1000ths of a second).

Since 2.10

See Also

GThread GLib thread system.
Description

Often you need to communicate between different threads. In general it's safer not to do this by shared memory, but by explicit message passing. These messages only make sense asynchronously for multi-threaded applications though, as a synchronous operation could as well be done in the same thread.

Asynchronous queues are an exception from most other GLib data structures, as they can be used simultaneously from multiple threads without explicit locking and they bring their own builtin reference counting. This is because the nature of an asynchronous queue is that it will always be used by at least 2 concurrent threads.

For using an asynchronous queue you first have to create one with `g_async_queue_new()`. A newly-created queue will get the reference count 1. Whenever another thread is creating a new reference of (that is, pointer to) the queue, it has to increase the reference count (using `g_async_queue_ref()`). Also, before removing this reference, the reference count has to be decreased (using `g_async_queue_unref()`). After that the queue might no longer exist so you must not access it after that point.

A thread, which wants to send a message to that queue simply calls `g_async_queue_push()` to push the message to the queue.

A thread, which is expecting messages from an asynchronous queue simply calls `g_async_queue_pop()` for that queue. If no message is available in the queue at that point, the thread is now put to sleep until a message arrives. The message will be removed from the queue and returned. The functions `g_async_queue_try_pop()` and `g_async_queue_timed_pop()` can be used to only check for the presence of messages or to only wait a certain time for messages respectively.

For almost every function there exist two variants, one that locks the queue and one that doesn’t. That way you can hold the queue lock (acquire it with `g_async_queue_lock()` and release it with `g_async_queue_unlock()`) over multiple queue accessing instructions. This can be necessary to ensure the integrity of the queue, but should only be used when really necessary, as it can make your life harder if used unwisely. Normally you should only use the locking function variants (those without the suffix _unlocked)

Details

GAsyncQueue

typedef struct _GAsyncQueue GAsyncQueue;

The GAsyncQueue struct is an opaque data structure, which represents an asynchronous queue. It should only be accessed through the g_async_queue_* functions.
3.4. ASYNCHRONOUS QUEUES

**g_async_queue_new ()**

```c
GAsyncQueue* g_async_queue_new (void);
```

Creates a new asynchronous queue with the initial reference count of 1.

*Returns*: the new GAsyncQueue.

**g_async_queue_new_full ()**

```c
GAsyncQueue* g_async_queue_new_full (GDestroyNotify item_free_func);
```

Creates a new asynchronous queue with an initial reference count of 1 and sets up a destroy notify function that is used to free any remaining queue items when the queue is destroyed after the final unref.

*item_free_func*: function to free queue elements

*Returns*: the new GAsyncQueue.

Since 2.16

**g_async_queue_ref ()**

```c
GAsyncQueue* g_async_queue_ref (GAsyncQueue *queue);
```

Increases the reference count of the asynchronous queue by 1. You do not need to hold the lock to call this function.

*queue*: a GAsyncQueue.

*Returns*: the queue that was passed in (since 2.6)

**g_async_queue_unref ()**

```c
void g_async_queue_unref (GAsyncQueue *queue);
```

Decreases the reference count of the asynchronous queue by 1. If the reference count went to 0, the queue will be destroyed and the memory allocated will be freed. So you are not allowed to use the queue afterwards, as it might have disappeared. You do not need to hold the lock to call this function.

*queue*: a GAsyncQueue.

**g_async_queue_push ()**

```c
void g_async_queue_push (GAsyncQueue *queue, gpointer data);
```

Pushes the data into the queue. *data must not be NULL.*

*queue*: a GAsyncQueue.

*data*: data to push into the queue.
CHAPTER 3. GLIB CORE APPLICATION

3.4. ASYNCHRONOUS QUEUES

**g_async_queue_push_sorted ()**

```c
void g_async_queue_push_sorted (GAsyncQueue *queue,
                               gpointer data,
                               GCompareDataFunc func,
                               gpointer user_data);
```

Inserts data into queue using func to determine the new position. This function requires that the queue is sorted before pushing on new elements. This function will lock queue before it sorts the queue and unlock it when it is finished. For an example of func see g_async_queue_sort().

- **queue**: a GAsyncQueue
- **data**: the data to push into the queue
- **func**: the GCompareDataFunc is used to sort queue. This function is passed two elements of the queue. The function should return 0 if they are equal, a negative value if the first element should be higher in the queue or a positive value if the first element should be lower in the queue than the second element.
- **user_data**: user data passed to func.

Since 2.10

**g_async_queue_pop ()**

```c
gpointer g_async_queue_pop (GAsyncQueue *queue);
```

Pops data from the queue. This function blocks until data become available.

- **queue**: a GAsyncQueue.

**Returns**: data from the queue.

**g_async_queue_try_pop ()**

```c
gpointer g_async_queue_try_pop (GAsyncQueue *queue);
```

Tries to pop data from the queue. If no data is available, NULL is returned.

- **queue**: a GAsyncQueue.

**Returns**: data from the queue or NULL, when no data is available immediately.

**g_async_queue_timed_pop ()**

```c
gpointer g_async_queue_timed_pop (GAsyncQueue *queue,
                                  GTimeVal *end_time);
```

Pops data from the queue. If no data is received before end_time, NULL is returned. To easily calculate end_time a combination of g_get_current_time() and g_time_val_add() can be used.

- **queue**: a GAsyncQueue.
- **end_time**: a GTimeVal, determining the final time.

**Returns**: data from the queue or NULL, when no data is received before end_time.
3.4. ASYNCHRONOUS QUEUES

**g_async_queue_length ()**

```c
int g_async_queue_length (GAsyncQueue *queue);
```

Returns the length of the queue, negative values mean waiting threads, positive values mean available entries in the queue. Actually this function returns the number of data items in the queue minus the number of waiting threads. Thus a return value of 0 could mean 'n' entries in the queue and 'n' threads waiting. That can happen due to locking of the queue or due to scheduling.

**queue**: a GAsyncQueue.

**Returns**: the length of the queue.

**g_async_queue_sort ()**

```c
void g_async_queue_sort (GAsyncQueue *queue, GCompareDataFunc func, gpointer user_data);
```

Sorts queue using func. This function will lock queue before it sorts the queue and unlock it when it is finished.

If you were sorting a list of priority numbers to make sure the lowest priority would be at the top of the queue, you could use:

```c
gint32 id1;
gint32 id2;

id1 = GPOINTER_TO_INT (element1);
id2 = GPOINTER_TO_INT (element2);

return (id1 > id2 ? +1 : id1 == id2 ? 0 : -1);
```

**queue**: a GAsyncQueue

**func**: the GCompareDataFunc is used to sort queue. This function is passed two elements of the queue. The function should return 0 if they are equal, a negative value if the first element should be higher in the queue or a positive value if the first element should be lower in the queue than the second element.

**user_data**: user data passed to func

Since 2.10

**g_async_queue_lock ()**

```c
void g_async_queue_lock (GAsyncQueue *queue);
```

Acquires the queue’s lock. After that you can only call the g_async_queue_*_unlocked() function variants on that queue. Otherwise it will deadlock.

**queue**: a GAsyncQueue.

**g_async_queue_unlock ()**

```c
void g_async_queue_unlock (GAsyncQueue *queue);
```

Releases the queue’s lock.

**queue**: a GAsyncQueue.
### 3.4. Asynchronous Queues

#### g_async_queue_ref_unlocked()

```c
void g_async_queue_ref_unlocked (GAsyncQueue *queue);
```

**Warning**

`g_async_queue_ref_unlocked` is deprecated and should not be used in newly-written code.

Increases the reference count of the asynchronous `queue` by 1.

**Deprecated:** Since 2.8, reference counting is done atomically so `g_async_queue_ref()` can be used regardless of the `queue`'s lock.

**Arguments:**

- `queue`: a `GAsyncQueue`.

#### g_async_queue_unref_and_unlock()

```c
void g_async_queue_unref_and_unlock (GAsyncQueue *queue);
```

**Warning**

`g_async_queue_unref_and_unlock` is deprecated and should not be used in newly-written code.

Decreases the reference count of the asynchronous `queue` by 1 and releases the lock. This function must be called while holding the `queue`'s lock. If the reference count went to 0, the `queue` will be destroyed and the memory allocated will be freed.

**Deprecated:** Since 2.8, reference counting is done atomically so `g_async_queue_unref()` can be used regardless of the `queue`'s lock.

**Arguments:**

- `queue`: a `GAsyncQueue`.

#### g_async_queue_push_unlocked()

```c
void g_async_queue_push_unlocked (GAsyncQueue *queue, gpointer data);
```

Pushes the `data` into the `queue`. `data` must not be NULL. This function must be called while holding the `queue`'s lock.

**Arguments:**

- `queue`: a `GAsyncQueue`.
- `data`: data to push into the `queue`.

#### g_async_queue_push_sorted_unlocked()

```c
void g_async_queue_push_sorted_unlocked (GAsyncQueue *queue, gpointer data, GCompareDataFunc func, gpointer user_data);
```
CHAPTER 3. GLIB CORE APPLICATION...

3.4. ASYNCHRONOUS QUEUES

Inserts data into queue using func to determine the new position. This function requires that the queue is sorted before pushing on new elements. This function is called while holding the queue's lock. For an example of func see g_async_queue_sort().

queue: a GAsyncQueue

data: the data to push into the queue

func: the GCompareDataFunc is used to sort queue. This function is passed two elements of the queue. The function should return 0 if they are equal, a negative value if the first element should be higher in the queue or a positive value if the first element should be lower in the queue than the second element.

user_data: user data passed to func.

Since 2.10

g_async_queue_pop_unlocked ()

qpointer g_async_queue_pop_unlocked (GAsyncQueue *queue);

Pops data from the queue. This function blocks until data become available. This function must be called while holding the queue’s lock.

queue: a GAsyncQueue.

Returns: data from the queue.

g_async_queue_try_pop_unlocked ()

qpointer g_async_queue_try_pop_unlocked (GAsyncQueue *queue);

Tries to pop data from the queue. If no data is available, NULL is returned. This function must be called while holding the queue’s lock.

queue: a GAsyncQueue.

Returns: data from the queue or NULL, when no data is available immediately.

g_async_queue_timed_pop_unlocked ()

qpointer g_async_queue_timed_pop_unlocked (GAsyncQueue *queue, GTimeVal *end_time);

Pops data from the queue. If no data is received before end_time, NULL is returned. This function must be called while holding the queue’s lock.

To easily calculate end_time a combination of g_get_current_time() and g_time_val_add() can be used.

queue: a GAsyncQueue.

dtime: a GTimeVal, determining the final time.

Returns: data from the queue or NULL, when no data is received before end_time.
3.5 Dynamic Loading of Modules

Name

Dynamic Loading of Modules – portable method for dynamically loading 'plug-ins'

Synopsis

```c
#include <gmodule.h>

gboolean g_module_supported (void);
gchar* g_module_build_path (const gchar *directory, const gchar *module_name);
GModule* g_module_open (const gchar *file_name, GModuleFlags flags);
enum GModuleFlags;
gboolean g_module_symbol (GModule *module, const gchar *symbol_name, gpointer *symbol);
const gchar* g_module_name (GModule *module);
void g_module_make_resident (GModule *module);
gboolean g_module_close (GModule *module);
const gchar* g_module_error (void);
const gchar* (*GModuleCheckInit) (GModule *module);
```

**g_async_queue_length_unlocked()**

```c
#include <gasyncqueue.h>

Gint g_async_queue_length_unlocked (GAsyncQueue *queue);
```

Returns the length of the queue, negative values mean waiting threads, positive values mean available entries in the queue. Actually this function returns the number of data items in the queue minus the number of waiting threads. Thus a return value of 0 could mean 'n' entries in the queue and 'n' thread waiting. That can happen due to locking of the queue or due to scheduling. This function must be called while holding the queue's lock.

`queue`: a GAsyncQueue.

**Returns**: the length of the queue.

**g_async_queue_sort_unlocked()**

```c
#include <gasyncqueue.h>

void g_async_queue_sort_unlocked (GAsyncQueue *queue, GCompareDataFunc func, gpointer user_data);
```

Sorts queue using `func`. This function is called while holding the queue's lock.

`queue`: a GAsyncQueue

`func`: the GCompareDataFunc is used to sort `queue`. This function is passed two elements of the queue. The function should return 0 if they are equal, a negative value if the first element should be higher in the queue or a positive value if the first element should be lower in the queue than the second element.

`user_data`: user data passed to `func`

Since 2.10
CHAPTER 3. GLIB CORE APPLICATION… 3.5. DYNAMIC LOADING OF MODULES

void (*GModuleUnload)(GModule *module);
#define G_MODULE_SUFFIX
#define G_MODULE_EXPORT
#define G_MODULE_IMPORT

Description

These functions provide a portable way to dynamically load object files (commonly known as ‘plugins’). The current implementation supports all systems that provide an implementation of dlopen() (e.g. Linux/Sun), as well as HP-UX via its shl_load() mechanism, and Windows platforms via DLLs.

A program which wants to use these functions must be linked to the libraries output by the command pkg-config --libs gmodule-2.0.

To use them you must first determine whether dynamic loading is supported on the platform by calling g_module_supported(). If it is, you can open a module with g_module_open(), find the module’s symbols (e.g. function names) with g_module_symbol(), and later close the module with g_module_close().

GModule_name() will return the file name of a currently opened module.

If any of the above functions fail, the error status can be found with g_module_error().

The GModule implementation features reference counting for opened modules, and supports hook functions within a module which are called when the module is loaded and unloaded (see GModuleCheckInit and GModuleUnload).

If your module introduces static data to common subsystems in the running program, e.g. through calling g_quark_from_static_string ("my-module-stuff"), it must ensure that it is never unloaded, by calling g_module_make_resident().

Example 3.10 Calling a function defined in a GModule

/* the function signature for 'say_hello' */
typedef void (*SayHelloFunc)(const char *message);
gboolean just_say_hello(const char *filename, GError **error){
  SayHelloFunc say_hello;
  GModule *module;
  module = g_module_open(filename, G_MODULE_BIND_LAZY);
  if(!module){
    g_set_error(error, FOO_ERROR, FOO_ERROR_BLAH, 
                "%s", g_module_error());
    return FALSE;
  }
  if(!g_module_symbol(module, "say_hello", (gpointer *)&say_hello)) {
    g_set_error(error, SAY_ERROR, SAY_ERROR_OPEN, 
                "%s: %s", filename, g_module_error());
    if(!g_module_close(module))
      g_warning("%s: %s", filename, g_module_error());
    return FALSE;
  }
  if(say_hello == NULL){
    g_set_error(error, SAY_ERROR, SAY_ERROR_OPEN, "symbol say_hello is NULL");
    if(!g_module_close(module))
      g_warning("%s: %s", filename, g_module_error());
    return FALSE;
  }
  /* call our function in the module */
  say_hello("Hello world!");
  if(!g_module_close(module))
    g_warning("%s: %s", filename, g_module_error());
  return TRUE;
}
Details

GModule

typedef struct _GModule GModule;

The GModule struct is an opaque data structure to represent a Dynamically-Loaded Module. It should only be accessed via the following functions.

**g_module_supported ()**

gboolean g_module_supported (void);

Checks if modules are supported on the current platform.

*Returns*: %TRUE if modules are supported.

**g_module_build_path ()**

gchar* g_module_build_path (const gchar *directory,
                          const gchar *module_name);

A portable way to build the filename of a module. The platform-specific prefix and suffix are added to the filename, if needed, and the result is added to the directory, using the correct separator character.

The directory should specify the directory where the module can be found. It can be NULL or an empty string to indicate that the module is in a standard platform-specific directory, though this is not recommended since the wrong module may be found.

For example, calling g_module_build_path() on a Linux system with a directory of /lib and a module_name of "mylibrary" will return /lib/libmylibrary.so. On a Windows system, using \Windows as the directory it will return \Windows\mylibrary.dll.

*directory*: the directory where the module is. This can be NULL or the empty string to indicate that the standard platform-specific directories will be used, though that is not recommended.

*module_name*: the name of the module.

*Returns*: the complete path of the module, including the standard library prefix and suffix. This should be freed when no longer needed.

**g_module_open ()**

GModule* g_module_open (const gchar *file_name,
                          GModuleFlags flags);

Opens a module. If the module has already been opened, its reference count is incremented.

First of all g_module_open() tries to open file_name as a module. If that fails and file_name has the ".la"-suffix (and is a libtool archive) it tries to open the corresponding module. If that fails and it doesn’t have the proper module suffix for the platform (G_MODULE_SUFFIX), this suffix will be appended and the corresponding module will be opened. If that fails and file_name doesn’t have the ".la"-suffix, this suffix is appended and g_module_open() tries to open the corresponding module. If eventually that fails as well, NULL is returned.

*file_name*: the name of the file containing the module, or NULL to obtain a GModule representing the main program itself.

*flags*: the flags used for opening the module. This can be the logical OR of any of the GModuleFlags.

*Returns*: a GModule on success, or NULL on failure.
enum GModuleFlags
{
    G_MODULE_BIND_LAZY = 1 << 0,
    G_MODULE_BIND_LOCAL = 1 << 1,
    G_MODULE_BIND_MASK = 0x03
} GModuleFlags;

Flags passed to `g_module_open()`. Note that these flags are not supported on all platforms.

**G_MODULE_BIND_LAZY** specifies that symbols are only resolved when needed. The default action is to bind all symbols when the module is loaded.

**G_MODULE_BIND_LOCAL** specifies that symbols in the module should not be added to the global name space. The default action on most platforms is to place symbols in the module in the global name space, which may cause conflicts with existing symbols.

**G_MODULE_BIND_MASK** mask for all flags.

`g_module_symbol()`

```c
gboolean g_module_symbol (GModule *module, const gchar *symbol_name, gpointer *symbol);
```

Gets a symbol pointer from a module, such as one exported by `G_MODULE_EXPORT`. Note that a valid symbol can be `NULL`.

- **module**: a `GModule`.
- **symbol_name**: the name of the symbol to find.
- **symbol**: returns the pointer to the symbol value.

**Returns**: `%TRUE` on success.

`g_module_name()`

```c
const gchar* g_module_name (GModule *module);
```

Gets the filename from a `GModule`.

- **module**: a `GModule`.

**Returns**: the filename of the module, or "main" if the module is the main program itself.

`g_module_make_resident()`

```c
void g_module_make_resident (GModule *module);
```

Ensures that a module will never be unloaded. Any future `g_module_close()` calls on the module will be ignored.

- **module**: a `GModule` to make permanently resident.

`g_module_close()`

```c
gboolean g_module_close (GModule *module);
```

Closes a module.

- **module**: a `GModule` to close.

**Returns**: `%TRUE` on close.
g_module_error()

```c
const gchar* g_module_error (void);```

Gets a string describing the last module error.

**Returns**: a string describing the last module error.

GModuleCheckInit()

```c
const gchar* (*GModuleCheckInit) (GModule *module);
```

Specifies the type of the module initialization function. If a module contains a function named `g_module_check_init()` it is called automatically when the module is loaded. It is passed the GModule structure and should return NULL on success or a string describing the initialization error.

**module**: the GModule corresponding to the module which has just been loaded.

**Returns**: %NULL on success, or a string describing the initialization error.

GModuleUnload()

```c
void (*GModuleUnload) (GModule *module);
```

Specifies the type of the module function called when it is unloaded. If a module contains a function named `g_module_unload()` it is called automatically when the module is unloaded. It is passed the GModule structure.

**module**: the GModule about to be unloaded.

G_MODULE_SUFFIX

```c
#define G_MODULE_SUFFIX "so"
```

Expands to the proper shared library suffix for the current platform without the leading dot. For the most Unices and Linux this is "so", for some HP-UX versions this is "sl" and for Windows this is "dll".

G_MODULE_EXPORT

```c
#define G_MODULE_EXPORT
```

Used to declare functions exported by modules. This is a no-op on Linux and Unices, but when compiling for Windows, it marks a symbol to be exported from the library or executable being built.

G_MODULE_IMPORT

```c
#define G_MODULE_IMPORT extern
```

Used to declare functions imported from modules.

### 3.6 Memory Allocation

**Name**

Memory Allocation – general memory-handling
### Synopsis

```c
#include <glib.h>

#define g_new (struct_type, n_structs)
#define g_new0 (struct_type, n_structs)
#define g_renew (struct_type, mem, n_structs)
#define g_try_new (struct_type, n_structs)
#define g_try_new0 (struct_type, n_structs)
#define g_try_renew (struct_type, mem, n_structs)

gpointer g_malloc (gsize n_bytes);
gpointer g_malloc0 (gsize n_bytes);
gpointer g_realloc (gpointer mem, gsize n_bytes);
gpointer g_try_malloc (gsize n_bytes);
gpointer g_try_malloc0 (gsize n_bytes);
gpointer g_try_realloc (gpointer mem, gsize n_bytes);

void g_free (gpointer mem);
extern gboolean g_mem_gc_friendly;

#define galloca (size)
#define g_newa (struct_type, n_structs)
#define g_memmove (dest,src,len)
gpointer g_memdup (gconstpointer mem, guint byte_size);

GMemVTable;
void g_mem_set_vtable (GMemVTable *vtable);
gboolean g_mem_is_system_malloc (void);
extern GMemVTable *glib_mem_profiler_table;
void g_mem_profile (void);
```

### Description

These functions provide support for allocating and freeing memory.

**Note**

If any call to allocate memory fails, the application is terminated. This also means that there is no need to check if the call succeeded.

**Note**

It's important to match `g_malloc()` with `g_free()`, plain `malloc()` with `free()`, and (if you’re using C++) `new` with `delete` and `new[]` with `delete[]`. Otherwise bad things can happen, since these allocators may use different memory pools (and new/delete call constructors and destructors). See also `g_mem_set_vtable()`.
Details

**g_new()**

```c
#define g_new(struct_type, n_structs)
```

Allocates \(n\_\text{structs}\) elements of type `struct_type`. The returned pointer is cast to a pointer to the given type. If \(n\_\text{structs}\) is 0 it returns `NULL`. Since the returned pointer is already casted to the right type, it is normally unnecessary to cast it explicitly, and doing so might hide memory allocation errors.

**struct_type**: the type of the elements to allocate

**n_structs**: the number of elements to allocate

**Returns**: a pointer to the allocated memory, cast to a pointer to `struct_type`

**g_new0()**

```c
#define g_new0(struct_type, n_structs)
```

Allocates \(n\_\text{structs}\) elements of type `struct_type`, initialized to 0's. The returned pointer is cast to a pointer to the given type. If \(n\_\text{structs}\) is 0 it returns `NULL`. Since the returned pointer is already casted to the right type, it is normally unnecessary to cast it explicitly, and doing so might hide memory allocation errors.

**struct_type**: the type of the elements to allocate.

**n_structs**: the number of elements to allocate.

**Returns**: a pointer to the allocated memory, cast to a pointer to `struct_type`.

**g_renew()**

```c
#define g_renew(struct_type, mem, n_structs)
```

Reallocates the memory pointed to by `mem`, so that it now has space for \(n\_\text{structs}\) elements of type `struct_type`. It returns the new address of the memory, which may have been moved.

**struct_type**: the type of the elements to allocate

**mem**: the currently allocated memory

**n_structs**: the number of elements to allocate

**Returns**: a pointer to the new allocated memory, cast to a pointer to `struct_type`

**g_try_new()**

```c
#define g_try_new(struct_type, n_structs)
```

Attempts to allocate \(n\_\text{structs}\) elements of type `struct_type`, and returns `NULL` on failure. Contrast with `g_new()`, which aborts the program on failure. The returned pointer is cast to a pointer to the given type. If \(n\_\text{structs}\) is 0 it returns `NULL`.

**struct_type**: the type of the elements to allocate

**n_structs**: the number of elements to allocate

**Returns**: a pointer to the allocated memory, cast to a pointer to `struct_type`

Since 2.8
3.6. MEMORY ALLOCATION

**g_try_new0()**

```c
#define g_try_new0(struct_type, n_structs)
```

Attempts to allocate `n_structs` elements of type `struct_type`, initialized to 0's, and returns NULL on failure. Contrast with `g_new0()`, which aborts the program on failure. The returned pointer is cast to a pointer to the given type. The function returns NULL when `n_structs` is 0.

**struct_type**: the type of the elements to allocate  

**n_structs**: the number of elements to allocate  

**Returns**: a pointer to the allocated memory, cast to a pointer to `struct_type`  

Since 2.8

**g_try_renew()**

```c
#define g_try_renew(struct_type, mem, n_structs)
```

Attempts to reallocate the memory pointed to by `mem`, so that it now has space for `n_structs` elements of type `struct_type`, and returns NULL on failure. Contrast with `g_renew()`, which aborts the program on failure. It returns the new address of the memory, which may have been moved.

**struct_type**: the type of the elements to allocate  

**mem**: the currently allocated memory  

**n_structs**: the number of elements to allocate  

**Returns**: a pointer to the new allocated memory, cast to a pointer to `struct_type`  

Since 2.8

**g_malloc()**

```c
gpointer g_malloc (gsize n_bytes);
```

Allocates `n_bytes` bytes of memory. If `n_bytes` is 0 it returns NULL.

**n_bytes**: the number of bytes to allocate  

**Returns**: a pointer to the allocated memory

**g_malloc0()**

```c
gpointer g_malloc0 (gsize n_bytes);
```

Allocates `n_bytes` bytes of memory, initialized to 0's. If `n_bytes` is 0 it returns NULL.

**n_bytes**: the number of bytes to allocate  

**Returns**: a pointer to the allocated memory

**g_realloc()**

```c
gpointer g_realloc (gpointer mem, gsize n_bytes);
```

Reallocates the memory pointed to by `mem`, so that it now has space for `n_bytes` bytes of memory. It returns the new address of the memory, which may have been moved. `mem` may be NULL, in which case it's considered to have zero-length. `n_bytes` may be 0, in which case NULL will be returned and `mem` will be freed unless it is NULL.

**mem**: the memory to reallocate  

**n_bytes**: new size of the memory in bytes  

**Returns**: the new address of the allocated memory
g_try_malloc()

gpointer g_try_malloc (gsize n_bytes);

Attempts to allocate n_bytes, and returns NULL on failure. Contrast with g_malloc(), which aborts the program on failure.

n_bytes: number of bytes to allocate.

Returns: the allocated memory, or NULL.

g_try_malloc0()

gpointer g_try_malloc0 (gsize n_bytes);

Attempts to allocate n_bytes, initialized to 0's, and returns NULL on failure. Contrast with g_malloc0(), which aborts the program on failure.

n_bytes: number of bytes to allocate

Returns: the allocated memory, or NULL.

Since 2.8

g_try_realloc()

gpointer g_try_realloc (gpointer mem, gsize n_bytes);

Attempts to realloc mem to a new size, n_bytes, and returns NULL on failure. Contrast with g_realloc(), which aborts the program on failure. If mem is NULL, behaves the same as g_try_malloc().

mem: previously-allocated memory, or NULL.

n_bytes: number of bytes to allocate.

Returns: the allocated memory, or NULL.

g_free()

void g_free (gpointer mem);

Frees the memory pointed to by mem. If mem is NULL it simply returns.

mem: the memory to free

g_mem_gc_friendly

extern gboolean g_mem_gc_friendly;

This variable is TRUE if the G_DEBUG environment variable includes the key gc-friendly.

galloca()

#define g_alloca(size)

Allocates size bytes on the stack; these bytes will be freed when the current stack frame is cleaned up. This macro essentially just wraps the alloca() function present on most UNIX variants. Thus it provides the same advantages and pitfalls as alloca():

+ alloca() is very fast, as on most systems it’s implemented by just adjusting the stack pointer register.
+ It doesn’t cause any memory fragmentation, within its scope, separate alloc() blocks just build up and are released together at function end.

- Allocation sizes have to fit into the current stack frame. For instance in a threaded environment on Linux, the per-thread stack size is limited to 2 Megabytes, so be sparse with alloc() uses.

- Allocation failure due to insufficient stack space is not indicated with a NULL return like e.g. with malloc(). Instead, most systems probably handle it the same way as out of stack space situations from infinite function recursion, i.e. with a segmentation fault.

- Special care has to be taken when mixing alloc() with GNU C variable sized arrays. Stack space allocated with alloc() in the same scope as a variable sized array will be freed together with the variable sized array upon exit of that scope, and not upon exit of the enclosing function scope.

size: number of bytes to allocate.

Returns: space for size bytes, allocated on the stack

g_newa()

#define g_newa(struct_type, n_structs)

Wraps g_alloca() in a more typesafe manner.

struct_type: Type of memory chunks to be allocated

n_structs: Number of chunks to be allocated

Returns: Pointer to stack space for n_structs chunks of type struct_type

g_memmove()

#define g_memmove(dest, src, len)

Copies a block of memory len bytes long, from src to dest. The source and destination areas may overlap.

In order to use this function, you must include string.h yourself, because this macro will typically simply resolve to memmove() and GLib does not include string.h for you.

dest: the destination address to copy the bytes to.

src: the source address to copy the bytes from.

len: the number of bytes to copy.

g_memdup()

gpointer g_memdup (gconstpointer mem, guint byte_size);

Allocates byte_size bytes of memory, and copies byte_size bytes into it from mem. If mem is NULL it returns NULL.

mem: the memory to copy.

byte_size: the number of bytes to copy.

Returns: a pointer to the newly-allocated copy of the memory, or NULL if mem is NULL.
GMemVTable

typedef struct {
    gpointer (*malloc) (gsize n_bytes);
    gpointer (*realloc) (gpointer mem,
        gsize n_bytes);
    void (*free) (gpointer mem);
    /* optional; set to NULL if not used */
    gpointer (*calloc) (gsize n_blocks,
        gsize n_block_bytes);
    gpointer (*try_malloc) (gsize n_bytes);
    gpointer (*try_realloc) (gpointer mem,
        gsize n_bytes);
} GMemVTable;

A set of functions used to perform memory allocation. The same GMemVTable must be used for all
allocations in the same program; a call to g_mem_set_vtable(), if it exists, should be prior to any use of
GLib.

malloc() function to use for allocating memory.

realloc() function to use for reallocating memory.

free() function to use to free memory.

calloc() function to use for allocating zero-filled memory.

try_malloc() function to use for allocating memory without a default error handler.

try_realloc() function to use for reallocating memory without a default error handler.

g_mem_set_vtable()

void g_mem_set_vtable (GMemVTable *vtable);

Sets the GMemVTable to use for memory allocation. You can use this to provide custom memory
allocation routines. This function must be called before using any other GLib functions. The vtable only
needs to provide malloc(), realloc(), and free() functions; GLib can provide default implementations of
the others. The malloc() and realloc() implementations should return NULL on failure, GLib will handle
error-checking for you. vtable is copied, so need not persist after this function has been called.

vtable: table of memory allocation routines.

g_mem_is_system_malloc()

gboolean g_mem_is_system_malloc (void);

Checks whether the allocator used by g_malloc() is the system’s malloc implementation. If it re-
turns TRUE memory allocated with malloc() can be used interchangeable with memory allocated using
g_malloc(). This function is useful for avoiding an extra copy of allocated memory returned by a non-
GLib-based API.

A different allocator can be set using g_mem_set_vtable().

Returns: if TRUE, malloc() and g_malloc() can be mixed.

glib_mem_profiler_table

extern GMemVTable *glib_mem_profiler_table;

A GMemVTable containing profiling variants of the memory allocation functions. Use them together
with g_mem_profile() in order to get information about the memory allocation pattern of your program.
### 3.7 IO Channels

#### Name

IO Channels – portable support for using files, pipes and sockets

#### Synopsis

```c
#include <glib.h>

GIOChannel;

GIOChannel* g_io_channel_unix_new (int fd);
gint g_io_channel_unix_get_fd (GIOChannel *channel);
GIOChannel* g_io_channel_win32_new_fd (gint fd);
GIOChannel* g_io_channel_win32_new_socket (gint socket);
GIOChannel* g_io_channel_win32_new_messages (gsize hwnd);

void g_io_channel_init (GIOChannel *channel);

GIOChannel* g_io_channel_new_file (const gchar *filename,
const gchar *mode, GError **error);

GIOStatus g_io_channel_read_chars (GIOChannel *channel,
gchar *buf,
gsize count,
gsize *bytes_read,
GError **error);

GIOStatus g_io_channel_read_unichar (GIOChannel *channel,
guchar *thechar,
GError **error);

GIOStatus g_io_channel_read_line (GIOChannel *channel,
gchar **str_return,
gsize *length,
gsize *terminator_pos,
GError **error);

GIOStatus g_io_channel_read_line_string (GIOChannel *channel,
GString *buffer,
gsize *terminator_pos,
GError **error);

GIOStatus g_io_channel_read_to_end (GIOChannel *channel,
gchar **str_return,
gsize *length,
GError **error);

GIOStatus g_io_channel_write_chars (GIOChannel *channel,
const gchar *buf,
```

#### g_mem_profile

```c
void g_mem_profile (void);
```

Outputs a summary of memory usage.

It outputs the frequency of allocations of different sizes, the total number of bytes which have been allocated, the total number of bytes which have been freed, and the difference between the previous two values, i.e. the number of bytes still in use.

Note that this function will not output anything unless you have previously installed the `glib_mem_profiler_table` with `g_mem_set_vtable()`.
## 3.7. IO CHANNELS

- **GIOStatus**
  - `g_io_channel_write_unichar`
  - `GIOStatus g_io_channel_flush`
  - `GIOStatus g_io_channel_seek_position`
  - `GIOStatus g_io_channel_shutdown`

- **enum**
  - `GSeekType;`

- **GIOStatus**
  - `g_io_channel_ref`
  - `g_io_channel_unref`

- **GSnore**
  - `g_io_create_watch`

- **guint**
  - `g_io_add_watch`
  - `g_io_add_watch_full`

- **enum**
  - `GIOCondition;`

- **gboolean**
  - `(*GIOFunc)`

- **gsize**
  - `g_io_channel_get_buffer_size`
  - `g_io_channel_set_buffer_size`

- **GIOCondition**
  - `g_io_channel_get_buffer_condition`

- **GIOFlags**
  - `g_io_channel_get_flags`
  - `g_io_channel_set_flags`

- **enum**
  - `GIOFlags;`

- **const gchar**
  - `g_io_channel_get_line_term`
  - `g_io_channel_set_line_term`

- **gboolean**
  - `g_io_channel_get_buffered`

```c
GIOStatus g_io_channel_write_unichar (GIOChannel *channel, gunichar thechar, GError **error);
GIOStatus g_io_channel_flush (GIOChannel *channel, GError **error);
GIOStatus g_io_channel_seek_position (GIOChannel *channel, gint64 offset, GSeekType type, GError **error);
enum GSeekType;
GIOStatus g_io_channel_shutdown (GIOChannel *channel, gboolean flush, GError **error);
enum GIOStatus;
enum GIOChannelError;
#define G_IO_CHANNEL_ERROR
GIOChannelError g_io_channel_error_from_errno (gint en);
GIOChannel * g_io_channel_ref (GIOChannel *channel);
void g_io_channel_unref (GIOChannel *channel);
GSnore * g_io_create_watch (GIOChannel *channel, GIOCondition condition);
guint g_io_add_watch (GIOChannel *channel, GIOCondition condition, GIOFunc func, gpointer user_data);
guint g_io_add_watch_full (GIOChannel *channel, gint priority, GIOCondition condition, GIOFunc func, gpointer user_data, GDestroyNotify notify);
enum GIOCondition;
gboolean (*GIOFunc) (GIOChannel *source, GIOCondition condition, gpointer data);
enum GIOFuncs;
gsize g_io_channel_get_buffer_size (GIOChannel *channel);
void g_io_channel_set_buffer_size (GIOChannel *channel, gsize size);
GIOCondition g_io_channel_get_buffer_condition (GIOChannel *channel);
GIOFlags g_io_channel_get_flags (GIOChannel *channel);
GIOStatus g_io_channel_set_flags (GIOChannel *channel, GIOFlags flags, GError **error);
enum GIOFlags;
const gchar* g_io_channel_get_line_term (GIOChannel *channel, gint *length);
void g_io_channel_set_line_term (GIOChannel *channel, const gchar *line_term, gint length);
gboolean g_io_channel_get_buffered (GIOChannel *channel);
```
### 3.7. IO CHANNELS

```c
void g_io_channel_set_buffered (GIOChannel *channel, gboolean buffered);
const gchar* g_io_channel_get_encoding (GIOChannel *channel);
GIOStatus g_io_channel_set_encoding (GIOChannel *channel, const gchar *encoding, GError **error);
gboolean g_io_channel_get_close_on_unref (GIOChannel *channel);
void g_io_channel_set_close_on_unref (GIOChannel *channel, gboolean do_close);

GIOError g_io_channel_read (GIOChannel *channel, gchar *buf, gsize count, gsize *bytes_read);
enum GIOError;
GIOError g_io_channel_write (GIOChannel *channel, const gchar *buf, gsize count, gsize *bytes_written);
GIOError g_io_channel_seek (GIOChannel *channel, gint64 offset, GSeekType type);
void g_io_channel_close (GIOChannel *channel);
```

**Description**

The `GIOChannel` data type aims to provide a portable method for using file descriptors, pipes, and sockets, and integrating them into the main event loop. Currently full support is available on UNIX platforms, support for Windows is only partially complete.

To create a new `GIOChannel` on UNIX systems use `g_io_channel_unix_new()`. This works for plain file descriptors, pipes and sockets. Alternatively, a channel can be created for a file in a system independent manner using `g_io_channel_new_file()`.

Once a `GIOChannel` has been created, it can be used in a generic manner with the functions `g_io_channel_read_chars()`, `g_io_channel_write_chars()`, `g_io_channel_seek_position()`, and `g_io_channel_shutdown()`.

To add a `GIOChannel` to the main event loop use `g_io_add_watch()` or `g_io_add_watch_full()`. Here you specify which events you are interested in on the `GIOChannel`, and provide a function to be called whenever these events occur.

`GIOChannel` instances are created with an initial reference count of 1. `g_io_channel_ref()` and `g_io_channel_unref()` can be used to increment or decrement the reference count respectively. When the reference count falls to 0, the `GIOChannel` is freed. (Though it isn't closed automatically, unless it was created using `g_io_channel_new_from_file()`.) Using `g_io_add_watch()` or `g_io_add_watch_full()` increments a channel's reference count.

The new functions `g_io_channel_read_chars()`, `g_io_channel_read_line()`, `g_io_channel_read_line_string()`, `g_io_channel_read_to_end()`, `g_io_channel_write_chars()`, `g_io_channel_seek_position()`, and `g_io_channel_flush()` should not be mixed with the deprecated functions `g_io_channel_read()`, `g_io_channel_write()`, and `g_io_channel_seek()` on the same channel.

**Details**

**GIOChannel**

```c
typedef struct {
} GIOChannel;
```

A data structure representing an IO Channel. The fields should be considered private and should only be accessed with the following functions.
g_io_channel_unix_new ()

`GIOChannel* g_io_channel_unix_new (int fd);`

Creates a new `GIOChannel` given a file descriptor. On UNIX systems this works for plain files, pipes, and sockets.

- The returned `GIOChannel` has a reference count of 1.
- The default encoding for `GIOChannel` is UTF-8. If your application is reading output from a command using via pipe, you may need to set the encoding to the encoding of the current locale (see `g_get_charset()`) with the `g_io_channel_set_encoding()` function.
- If you want to read raw binary data without interpretation, then call the `g_io_channel_set_encoding()` function with `NULL` for the encoding argument.
- This function is available in GLib on Windows, too, but you should avoid using it on Windows. The domain of file descriptors and sockets overlap. There is no way for GLib to know which one you mean in case the argument you pass to this function happens to be both a valid file descriptor and socket. If that happens a warning is issued, and GLib assumes that it is the file descriptor you mean.

```
fd : a file descriptor.
```

Returns: a new `GIOChannel`.

---

**g_io_channel_unix_get_fd ()**

```gint g_io_channel_unix_get_fd (GIOChannel *channel);```

Returns the file descriptor of the `GIOChannel`.

- On Windows this function returns the file descriptor or socket of the `GIOChannel`.

```
channel : a `GIOChannel`, created with `g_io_channel_unix_new()`.
```

Returns: the file descriptor of the `GIOChannel`.

---

**g_io_channel_win32_new_fd ()**

```GIOChannel* g_io_channel_win32_new_fd (gint fd);```

Creates a new `GIOChannel` given a file descriptor on Windows. This works for file descriptors from the C runtime.

- This function works for file descriptors as returned by the `open()`, `creat()`, `pipe()` and `fileno()` calls in the Microsoft C runtime. In order to meaningfully use this function your code should use the same C runtime as GLib uses, which is msvcrtdll. Note that in current Microsoft compilers it is near impossible to convince it to build code that would use msvcrtdll. The last Microsoft compiler version that supported using msvcrtdll as the C runtime was version 6. The GNU compiler and toolchain for Windows, also known as Mingw, fully supports msvcrtdll.
- If you have created a `GIOChannel` for a file descriptor and started watching (polling) it, you shouldn’t call `read()` on the file descriptor. This is because adding polling for a file descriptor is implemented in GLib on Windows by starting a thread that sits blocked in a `read()` from the file descriptor most of the time. All reads from the file descriptor should be done by this internal GLib thread. Your code should call only `g_io_channel_read()`.
- This function is available only in GLib on Windows.

```
fd : a C library file descriptor.
```

Returns: a new `GIOChannel`.

---

**g_io_channel_win32_new_socket ()**

```GIOChannel* g_io_channel_win32_new_socket (gint socket);```

---

157
CHAPTER 3. GLIB CORE APPLICATION... 3.7. IO CHANNELS

Creates a new GIOChannel given a socket on Windows. This function works for sockets created by Winsock. It’s available only in GLib on Windows. Polling a GSource created to watch a channel for a socket puts the socket in non-blocking mode. This is a side-effect of the implementation and unavoidable.

\[ \text{socket}: \text{a Winsock socket} \]

\[ \text{Returns}: \text{a new GIOChannel} \]

\[ \text{g_io_channel_win32_new_messages()} \]

\[
\text{GIOChannel * g_io_channel_win32_new_messages (gsize hwnd);} \]

Creates a new GIOChannel given a window handle on Windows. This function creates a GIOChannel that can be used to poll for Windows messages for the window in question.

\[ \text{hwnd: a window handle.} \]

\[ \text{Returns}: \text{a new GIOChannel.} \]

\[ \text{g_io_channel_init()} \]

\[
\text{void g_io_channel_init (GIOChannel *channel);} \]

Initializes a GIOChannel struct. This is called by each of the above functions when creating a GIOChannel, and so is not often needed by the application programmer (unless you are creating a new type of GIOChannel).

\[ \text{channel: a GIOChannel} \]

\[ \text{g_io_channel_new_file()} \]

\[
\text{GIOChannel* g_io_channel_new_file (const gchar *filename, const gchar *mode, GError **error);} \]

Open a file filename as a GIOChannel using mode mode. This channel will be closed when the last reference to it is dropped, so there is no need to call g_io_channel_close() (though doing so will not cause problems, as long as no attempt is made to access the channel after it is closed).

\[ \text{filename: A string containing the name of a file} \]

\[ \text{mode: One of "r", "w", "a", "rt", "w+", "a+". These have the same meaning as in fopen()} \]

\[ \text{error: A location to return an error of type G_FILE_ERROR} \]

\[ \text{Returns}: \text{A GIOChannel on success, NULL on failure.} \]

\[ \text{g_io_channel_read_chars()} \]

\[
\text{GIOStatus g_io_channel_read_chars (GIOChannel *channel, gchar *buf, gsize count, gsize *bytes_read, GError **error);} \]

Replacement for g_io_channel_read() with the new API.

\[ \text{channel: a GIOChannel} \]

\[ \text{buf: a buffer to read data into} \]

158
3.7. IO CHANNELS

**count**: the size of the buffer. Note that the buffer may not be completely filled even if there is data in the buffer if the remaining data is not a complete character.

**bytes_read**: The number of bytes read. This may be zero even on success if count < 6 and the channel’s encoding is non-NULL. This indicates that the next UTF-8 character is too wide for the buffer.

**error**: a location to return an error of type GConvertError or GIOChannelError.

**Returns**: the status of the operation.

### g_io_channel_read_unichar()

```c
GIOStatus g_io_channel_read_unichar (GIOChannel *channel, gunichar *thechar, GError **error);
```

Reads a Unicode character from `channel`. This function cannot be called on a channel with NULL encoding.

- **channel**: a GIOChannel
- **thechar**: a location to return a character
- **error**: a location to return an error of type GConvertError or GIOChannelError

**Returns**: a GIOStatus

### g_io_channel_read_line()

```c
GIOStatus g_io_channel_read_line (GIOChannel *channel, gchar **str_return, gsize *length, gsize *terminator_pos, GError **error);
```

Reads a line, including the terminating character(s), from a GIOChannel into a newly-allocated string. `str_return` will contain allocated memory if the return is G_IO_STATUS_NORMAL.

- **channel**: a GIOChannel
- **str_return**: The line read from the GIOChannel, including the line terminator. This data should be freed with `g_free()` when no longer needed. This is a null-terminated string. If a `length` of zero is returned, this will be NULL instead.
- **length**: location to store length of the read data, or NULL
- **terminator_pos**: location to store position of line terminator, or NULL
- **error**: A location to return an error of type GConvertError or GIOChannelError

**Returns**: the status of the operation.

### g_io_channel_read_line_string()

```c
GIOStatus g_io_channel_read_line_string (GIOChannel *channel, GString *buffer, gsize *terminator_pos, GError **error);
```

Reads a line from a GIOChannel, using a GString as a buffer.

- **channel**: a GIOChannel
- **buffer**: a GString into which the line will be written. If `buffer` already contains data, the old data will be overwritten.
terminator_pos: location to store position of line terminator, or NULL

error: a location to store an error of type GConvertError or GIOChannelError

Returns: the status of the operation.

**g_io_channel_read_to_end()**

| GIOStatus | g_io_channel_read_to_end | (GIOChannel *channel,
|           |                        | gchar **str_return,
|           |                        | gsize *length,
|           |                        | GError **error); |

Reads all the remaining data from the file.

channel: a GIOChannel

str_return: Location to store a pointer to a string holding the remaining data in the GIOChannel. This data should be freed with g_free() when no longer needed. This data is terminated by an extra null character, but there may be other nulls in the intervening data.

length: location to store length of the data

error: location to return an error of type GConvertError or GIOChannelError

Returns: G_IO_STATUS_NORMAL on success. This function never returns G_IO_STATUS_EOF.

**g_io_channel_write_chars()**

| GIOStatus | g_io_channel_write_chars | (GIOChannel *channel,
|           |                        | const gchar *buf,
|           |                        | gssize count,
|           |                        | gsize *bytes_written,
|           |                        | GError **error); |

Replacement for g_io_channel_write() with the new API.

On seekable channels with encodings other than NULL or UTF-8, generic mixing of reading and writing is not allowed. A call to g_io_channel_write_chars() may only be made on a channel from which data has been read in the cases described in the documentation for g_io_channel_set_encoding().

channel: a GIOChannel

buf: a buffer to write data from

count: the size of the buffer. If -1, the buffer is taken to be a null-terminated string.

bytes_written: The number of bytes written. This can be nonzero even if the return value is not G_IO_STATUS_NORMAL. If the return value is G_IO_STATUS_NORMAL and the channel is blocking, this will always be equal to count if count >= 0.

error: a location to return an error of type GConvertError or GIOChannelError

Returns: the status of the operation.

**g_io_channel_write_unichar()**

| GIOStatus | g_io_channel_write_unichar | (GIOChannel *channel,
|           |                        | gunichar thechar,
|           |                        | GError **error); |

Writes a Unicode character to channel. This function cannot be called on a channel with NULL encoding.

channel: a GIOChannel
thechar: a character

error: location to return an error of type GConvertError or GIOChannelError

Returns: a GIOStatus

g_io_channel_flush()

| GIOStatus | g_io_channel_flush | (GIOChannel *channel, GError **error); |

Flushes the write buffer for the GIOChannel.

channel: a GIOChannel

error: location to store an error of type GIOChannelError

Returns: the status of the operation. One of G_IO_STATUS_NORMAL, G_IO_STATUS_AGAIN, or G_IO_STATUS_ERROR.

g_io_channel_seek_position()

| GIOStatus | g_io_channel_seek_position | (GIOChannel *channel, gint64 offset, GSeekType type, GError **error); |

Replacement for g_io_channel_seek() with the new API.

channel: a GIOChannel

offset: The offset in bytes from the position specified by type

type: a GSeekType. The type G_SEEK_CUR is only allowed in those cases where a call to g_io_channel_set_encoding() is allowed. See the documentation for g_io_channel_set_encoding() for details.

error: A location to return an error of type GIOChannelError

Returns: the status of the operation.

eNum GSeekType

typedef enum
{
    G_SEEK_CUR,
    G_SEEK_SET,
    G_SEEK_END
} GSeekType;

An enumeration specifying the base position for a g_io_channel_seek_position() operation.

G_SEEK_CUR the current position in the file.

G_SEEK_SET the start of the file.

G_SEEK_END the end of the file.
3.7. IO CHANNELS

**g_io_channel_shutdown()**

```c
GIOStatus g_io_channel_shutdown (GIOChannel *channel, gboolean flush, GError **err);
```

Close an IO channel. Any pending data to be written will be flushed if `flush` is **TRUE**. The channel will not be freed until the last reference is dropped using `g_io_channel_unref()`.

- *channel*: a `GIOChannel`
- *flush*: if **TRUE**, flush pending
- *err*: location to store a `GIOChannelError`

**Returns**: the status of the operation.

```c
typedef enum
{
    G_IO_STATUS_ERROR,
    G_IO_STATUS_NORMAL,
    G_IO_STATUS_EOF,
    G_IO_STATUS_AGAIN
} GIOStatus;
```

Status returned by most of the `GIOFuncs` functions.

- **G_IO_STATUS_ERROR**: An error occurred.
- **G_IO_STATUS_NORMAL**: Success.
- **G_IO_STATUS_EOF**: End of file.
- **G_IO_STATUS_AGAIN**: Resource temporarily unavailable.

```c
typedef enum
{
    /* Derived from errno */
    G_IO_CHANNEL_ERROR_FBIG,
    G_IO_CHANNEL_ERROR_INVAL,
    G_IO_CHANNEL_ERROR_IO,
    G_IO_CHANNEL_ERROR_ISDIR,
    G_IO_CHANNEL_ERROR_NOSPC,
    G_IO_CHANNEL_ERROR_NXIO,
    G_IO_CHANNEL_ERROR_OVERFLOW,
    G_IO_CHANNEL_ERROR_PIPE,
    /* Other */
    G_IO_CHANNEL_ERROR_FAILED
} GIOChannelError;
```

Error codes returned by `GIOChannel` operations.

- **G_IO_CHANNEL_ERROR_FBIG**: File too large.
- **G_IO_CHANNEL_ERROR_INVAL**: Invalid argument.
- **G_IO_CHANNEL_ERROR_IO**: IO error.
- **G_IO_CHANNEL_ERROR_ISDIR**: File is a directory.
- **G_IO_CHANNEL_ERROR_NOSPC**: No space left on device.
G_IO_CHANNEL_ERROR_NXIO  No such device or address.

G_IO_CHANNEL_ERROR_OVERFLOW  Value too large for defined datatype.

G_IO_CHANNEL_ERROR_PIPE  Broken pipe.

G_IO_CHANNEL_ERROR_FAILED  Some other error.

G_IO_CHANNEL_ERROR

#define G_IO_CHANNEL_ERROR g_io_channel_error_quark()

Error domain for GIOChannel operations. Errors in this domain will be from the GIOChannelError enumeration. See GError for information on error domains.

g_io_channel_error_from_errno ()

GIOChannelError  g_io_channel_error_from_errno  (gint en);

Converts an errn error number to a GIOChannelError.

en: an errno error number, e.g. EINVAL

Returns: a GIOChannelError error number, e.g. G_IO_CHANNEL_ERROR_INVAL.

g_io_channel_ref ()

GIOChannel  *  g_io_channel_ref  (GIOChannel  *channel);

Increments the reference count of a GIOChannel.

channel: a GIOChannel

Returns: the channel that was passed in (since 2.6)

g_io_channel_unref ()

void  g_io_channel_unref  (GIOChannel  *channel);

Decrementsthe reference count of a GIOChannel.

channel: a GIOChannel

g_io_create_watch ()

GSource  *  g_io_create_watch  (GIOChannel  *channel,
                                      GIOCondition  condition);

Creates a GSource that’s dispatched when condition is met for the given channel. For example, if condition is G_IO_IN, the source will be dispatched when there’s data available for reading.

g_io_add_watch() is a simpler interface to this same functionality, for the case where you want to add the source to the default main loop context at the default priority.

On Windows, polling a GSource created to watch a channel for a socket puts the socket in non-blocking mode. This is a side-effect of the implementation and unavoidable.

channel: a GIOChannel to watch

condition: conditions to watch for

Returns: a new GSource
### 3.7. IO CHANNELS

**g_io_add_watch()**

```c
 guint g_io_add_watch (GIOChannel *channel, GIOCondition condition, GIOFunc func, gpointer user_data);
```

Adds the GIOChannel into the default main loop context with the default priority.

- **channel**: a GIOChannel
- **condition**: the condition to watch for
- **func**: the function to call when the condition is satisfied
- **user_data**: user data to pass to `func`

**Returns**: the event source id

**g_io_add_watch_full()**

```c
 guint g_io_add_watch_full (GIOChannel *channel, gint priority, GIOCondition condition, GIOFunc func, gpointer user_data, GDestroyNotify notify);
```

Adds the GIOChannel into the default main loop context with the given priority.

This internally creates a main loop source using `g_io_create_watch()` and attaches it to the main loop context with `g_source_attach()`. You can do these steps manually if you need greater control.

- **channel**: a GIOChannel
- **priority**: the priority of the GIOChannel source
- **condition**: the condition to watch for
- **func**: the function to call when the condition is satisfied
- **user_data**: user data to pass to `func`
- **notify**: the function to call when the source is removed

**Returns**: the event source id

**enum GIOCondition**

```c
typedef enum {
    G_IO_IN GLIB_SYSDEF_POLLIN,
    G_IO_OUT GLIB_SYSDEF_POLLOUT,
    G_IO_PRI GLIB_SYSDEF_POLLPRI,
    G_IO_ERR GLIB_SYSDEF_POLLPERR,
    G_IO_HUP GLIB_SYSDEF_POLLPOLLHUP,
    G_IO_NVAL GLIB_SYSDEF_POLLPOLLNVAL
} GIOCondition;
```

A bitwise combination representing a condition to watch for on an event source.

- **G_IO_IN** There is data to read.
- **G_IO_OUT** Data can be written (without blocking).
- **G_IO_PRI** There is urgent data to read.
**G_IO_ERR** Error condition.

**G_IO_HUP** Hung up (the connection has been broken, usually for pipes and sockets).

**G_IO_NVAL** Invalid request. The file descriptor is not open.

### GIOFunc()

```c
gboolean (GIOFunc) (GIOChannel *source,
                     GIOCondition condition,
                     gpointer data);
```

Specifies the type of function passed to `g_io_add_watch()` or `g_io_add_watch_full()`, which is called when the requested condition on a `GIOChannel` is satisfied.

- **source**: the `GIOChannel` event source
- **condition**: the condition which has been satisfied
- **data**: user data set in `g_io_add_watch()` or `g_io_add_watch_full()`

**Returns**: the function should return `FALSE` if the event source should be removed

### GIOFuncs

```c
typedef struct {
  GIOStatus (*io_read) (GIOChannel *channel,
                         gchar *buf,
                         gsize count,
                         gsize *bytes_read,
                         GError **err);
  GIOStatus (*io_write) (GIOChannel *channel,
                         const gchar *buf,
                         gsize count,
                         gsize *bytes_written,
                         GError **err);
  GIOStatus (*io_seek) (GIOChannel *channel,
                        gint64 offset,
                        GSeekType type,
                        GError **err);
  GIOStatus (*io_close) (GIOChannel *channel,
                        GError **err);
  GSource* (*io_create_watch) (GIOChannel *channel,
                                GIOCondition condition);
  void (*io_free) (GIOChannel *channel);
  GIOStatus (*io_set_flags) (GIOChannel *channel,
                             GIOFlags flags,
                             GError **err);
  GIOFlags (*io_get_flags) (GIOChannel *channel);
} GIOFuncs;
```

A table of functions used to handle different types of `GIOChannel` in a generic way.

### g_io_channel_get_buffer_size()

```c
FSIZE g_io_channel_get_buffer_size (GIOChannel *channel);
```

Gets the buffer size.

- **channel**: a `GIOChannel`

**Returns**: the size of the buffer.
3.7. IO CHANNELS

**g_io_channel_set_buffer_size ()**

```c
void g_io_channel_set_buffer_size (GIOChannel *channel, gsize size);
```

Sets the buffer size.

*channel*: a GIOChannel

*size*: the size of the buffer, or 0 to let GLib pick a good size

**g_io_channel_get_buffer_condition ()**

```c
GIOCondition g_io_channel_get_buffer_condition (GIOChannel *channel);
```

This function returns a GIOCondition depending on whether there is data to be read/space to write data in the internal buffers in the GIOChannel. Only the flags G_IO_IN and G_IO_OUT may be set.

*channel*: A GIOChannel

**Returns**: A GIOCondition

**g_io_channel_get_flags ()**

```c
GIOFlags g_io_channel_get_flags (GIOChannel *channel);
```

Gets the current flags for a GIOChannel, including read-only flags such as G_IO_FLAG_IS_READABLE.

The values of the flags G_IO_FLAG_IS_READABLE and G_IO_FLAG_IS_WRITEABLE are cached for internal use by the channel when it is created. If they should change at some later point (e.g. partial shutdown of a socket with the UNIX shutdown() function), the user should immediately call g_io_channel_get_flags() to update the internal values of these flags.

*channel*: a GIOChannel

**Returns**: the flags which are set on the channel

**g_io_channel_set_flags ()**

```c
GIOStatus g_io_channel_set_flags (GIOChannel *channel, GIOFlags flags, GError **error);
```

Sets the (writeable) flags in *channel* to (flags & G_IO_CHANNEL_SET_MASK).

*channel*: a GIOChannel

*flags*: the flags to set on the IO channel

*error*: A location to return an error of type GIOChannelError

**Returns**: the status of the operation.

**enum GIOFlags**

```c
typedef enum
{
  G_IO_FLAG_APPEND = 1 << 0,
  G_IO_FLAG_NONBLOCK = 1 << 1,
  G_IO_FLAG_IS_READABLE = 1 << 2, /* Read only flag */
  G_IO_FLAG_IS_WRITEABLE = 1 << 3, /* Read only flag */
  G_IO_FLAG_IS_SEEKABLE = 1 << 4, /* Read only flag */
  G_IO_FLAG_MASK = (1 << 5) - 1,
  G_IO_FLAG_GET_MASK = G_IO_FLAG_MASK,
  G_IO_FLAG_SET_MASK = G_IO_FLAG_APPEND | G_IO_FLAG_NONBLOCK
} GIOFlags;
```
Specifies properties of a GIOChannel. Some of the flags can only be read with `g_io_channel_get_flags()`, but not changed with `g_io_channel_set_flags()`.

`G_IO_FLAG_APPEND` turns on append mode, corresponds to O_APPEND (see the documentation of the UNIX open() syscall).

`G_IO_FLAG_NONBLOCK` turns on nonblocking mode, corresponds to O_NONBLOCK/O_NDELAY (see the documentation of the UNIX open() syscall).

`G_IO_FLAG_IS_READABLE` indicates that the io channel is readable. This flag can not be changed.

`G_IO_FLAG_IS_WRITEABLE` indicates that the io channel is writable. This flag can not be changed.

`G_IO_FLAG_IS_SEEKABLE` indicates that the io channel is seekable, i.e. that `g_io_channel_seek_position()` can be used on it. This flag can not be changed.

`G_IO_FLAG_MASK`

`G_IO_FLAG_GET_MASK`

`G_IO_FLAG_SET_MASK`

`g_io_channel_get_line_term()`

```c
const gchar* g_io_channel_get_line_term (GIOChannel *channel,
                                        gint *length);
```

This returns the string that GIOChannel uses to determine where in the file a line break occurs. A value of NULL indicates autodetection.

`channel`: a GIOChannel

`length`: a location to return the length of the line terminator

**Returns**: The line termination string. This value is owned by GLib and must not be freed.

`g_io_channel_set_line_term()`

```c
void g_io_channel_set_line_term (GIOChannel *channel,
                                 const gchar *line_term,
                                 gint length);
```

This sets the string that GIOChannel uses to determine where in the file a line break occurs.

`channel`: a GIOChannel

`line_term`: The line termination string. Use NULL for autodetect. Autodetection breaks on "\n", "\r\n", "\r", "\0", and the Unicode paragraph separator. Autodetection should not be used for anything other than file-based channels.

`length`: The length of the termination string. If -1 is passed, the string is assumed to be nul-terminated.

This option allows termination strings with embedded nuls.

`g_io_channel_get_buffered()`

```c
gboolean g_io_channel_get_buffered (GIOChannel *channel);
```

Returns whether `channel` is buffered.

`channel`: a GIOChannel

**Returns**: TRUE if the `channel` is buffered.
CHAPTER 3. GLIB CORE APPLICATION…

3.7. IO CHANNELS

`g_io_channel_set_buffered()`

```c
void g_io_channel_set_buffered (GIOChannel *channel, gboolean buffered);
```

The buffering state can only be set if the channel's encoding is `NULL`. For any other encoding, the channel must be buffered.

A buffered channel can only be set unbuffered if the channel's internal buffers have been flushed. Newly created channels or channels which have returned `G_IO_STATUS_EOF` not require such a flush. For write-only channels, a call to `g_io_channel_flush()` is sufficient. For all other channels, the buffers may be flushed by a call to `g_io_channel_seek_position()`. This includes the possibility of seeking with seek type `G_SEEK_CUR` and an offset of zero. Note that this means that socket-based channels cannot be set unbuffered once they have had data read from them.

On unbuffered channels, it is safe to mix read and write calls from the new and old APIs, if this is necessary for maintaining old code.

The default state of the channel is buffered.

- `channel`: a `GIOChannel`
- `buffered`: whether to set the channel buffered or unbuffered

`g_io_channel_get_encoding()`

```c
const gchar* g_io_channel_get_encoding (GIOChannel *channel);
```

Gets the encoding for the input/output of the channel. The internal encoding is always UTF-8. The encoding `NULL` makes the channel safe for binary data.

- `channel`: a `GIOChannel`

Returns: A string containing the encoding, this string is owned by GLib and must not be freed.

`g_io_channel_set_encoding()`

```c
GIOStatus g_io_channel_set_encoding (GIOChannel *channel, const gchar *encoding, GError **error);
```

Sets the encoding for the input/output of the channel. The internal encoding is always UTF-8. The default encoding for the external file is UTF-8.

The encoding `NULL` is safe to use with binary data.

The encoding can only be set if one of the following conditions is true:

- The channel was just created, and has not been written to or read from yet.
- The channel is write-only.
- The channel is a file, and the file pointer was just repositioned by a call to `g_io_channel_seek_position()`. (This flushes all the internal buffers.)
- The current encoding is `NULL` or UTF-8.
- One of the (new API) read functions has just returned `G_IO_STATUS_EOF` (or, in the case of `g_io_channel_read_to_end()`, `G_IO_STATUS_NORMAL`).
- One of the functions `g_io_channel_read_chars()` or `g_io_channel_read_unichar()` has returned `G_IO_STATUS_AGAIN` or `G_IO_STATUS_ERROR`. This may be useful in the case of `G_CONVERT_ERROR_ILLEGAL_SEQUENCE`. Returning one of these statuses from `g_io_channel_read_line()`, `g_io_channel_read_line_string()`, or `g_io_channel_read_to_end()` does not guarantee that the encoding can be changed.

Channels which do not meet one of the above conditions cannot call `g_io_channel_seek_position()` with an offset of `G_SEEK_CUR`, and, if they are "seekable", cannot call `g_io_channel_write_chars()` after calling one of the API "read" functions.
**3.7. IO CHANNELS**

*channel*: a GIOChannel

*encoding*: the encoding type

*error*: location to store an error of type GConvertError

**Returns**: G_IO_STATUS_NORMAL if the encoding was successfully set.

### `g_io_channel_get_close_on_unref ()`

```
gboolean g_io_channel_get_close_on_unref (GIOChannel *channel);
```

Returns whether the file/socket/whatever associated with *channel* will be closed when *channel* receives its final unref and is destroyed. The default value of this is TRUE for channels created by `g_io_channel_new_file()`, and FALSE for all other channels.

*channel*: a GIOChannel.

**Returns**: Whether the channel will be closed on the final unref of the GIOChannel data structure.

### `g_io_channel_set_close_on_unref ()`

```
void g_io_channel_set_close_on_unref (GIOChannel *channel, gboolean do_close);
```

Setting this flag to TRUE for a channel you have already closed can cause problems.

*channel*: a GIOChannel

*do_close*: Whether to close the channel on the final unref of the GIOChannel data structure. The default value of this is TRUE for channels created by `g_io_channel_new_file()`, and FALSE for all other channels.

### `g_io_channel_read ()`

```
GIOError g_io_channel_read (GIOChannel *channel, gchar *buf, gsize count, gsize *bytes_read);
```

**WARNING**

`g_io_channel_read` has been deprecated since version 2.2 and should not be used in newly-written code. Use `g_io_channel_read_chars()` instead.

Reads data from a GIOChannel.

*channel*: a GIOChannel

*buf*: a buffer to read the data into (which should be at least count bytes long)

*count*: the number of bytes to read from the GIOChannel

*bytes_read*: returns the number of bytes actually read

**Returns**: G_IO_ERROR_NONE if the operation was successful.
enum GIOError

typedef enum
{
    G_IO_ERROR_NONE,
    G_IO_ERROR_AGAIN,
    G_IO_ERROR_INVAL,
    G_IO_ERROR_UNKNOWN
} GIOError;

GIOError is only used by the deprecated functions g_io_channel_read(), g_io_channel_write(), and g_io_channel_seek().

- **G_IO_ERROR_NONE** no error
- **G_IO_ERROR_AGAIN** an EAGAIN error occurred
- **G_IO_ERROR_INVAL** an EINVAL error occurred
- **G_IO_ERROR_UNKNOWN** another error occurred

### g_io_channel_write()

`GIOError g_io_channel_write (GIOChannel *channel, const gchar *buf, gsize count, gsize *bytes_written);`

**WARNING**

*g_io_channel_write* has been deprecated since version 2.2 and should not be used in newly-written code. Use *g_io_channel_write_chars()* instead.

Writes data to a GIOChannel.

- **channel**: a GIOChannel
- **buf**: the buffer containing the data to write
- **count**: the number of bytes to write
- **bytes_written**: the number of bytes actually written

**Returns**: **G_IO_ERROR_NONE** if the operation was successful.

### g_io_channel_seek()

`GIOError g_io_channel_seek (GIOChannel *channel, gint64 offset, GSeekType type);`

**WARNING**

*g_io_channel_seek* has been deprecated since version 2.2 and should not be used in newly-written code. Use *g_io_channel_seek_position()* instead.

Sets the current position in the GIOChannel, similar to the standard library function *fseek()*.
3.8. Error Reporting

**Name**

Error Reporting – a system for reporting errors

**Synopsis**

```c
#include <glib.h>

GError;  
GError* g_error_new (GQuark domain, gint code, const gchar *format, ...);  
GError* g_error_new_literal (GQuark domain, gint code, const gchar *message);  
void g_error_free (GError *error);  
GError* g_error_copy (const GError *error);  
gboolean g_error_matches (const GError *error, GQuark domain, gint code);  
void g_set_error (GError **err, GQuark domain, gint code, gchar *message);
```

**channel**: a GIOChannel

**offset**: an offset, in bytes, which is added to the position specified by **type**

**type**: the position in the file, which can be G_SEEK_CUR (the current position), G_SEEK_SET (the start of the file), or G_SEEK_END (the end of the file)

**Returns**: G_IO_ERROR_NONE if the operation was successful.

### g_io_channel_close ()

```c
void g_io_channel_close (GIOChannel *channel);
```

**WARNING**

`g_io_channel_close` has been deprecated since version 2.2 and should not be used in newly-written code. Use `g_io_channel_shutdown()` instead.

Close an IO channel. Any pending data to be written will be flushed, ignoring errors. The channel will not be freed until the last reference is dropped using `g_io_channel_unref()`.

**channel**: A GIOChannel

**See Also**

gtk_input_add_full(), gtk_input_remove(), gdk_input_add(), gdk_input_add_full(), gdk_input_remove()  
Convenience functions for creating GIOChannel instances and adding them to the main event loop.

3.8 Error Reporting
void g_set_error_literal (GError **err, GQuark domain, gint code, const gchar *message);
void g_propagate_error (GError **dest, GError *src);
void g_clear_error (GError **err);
void g_prefix_error (GError **err, const gchar *format, ...);
void g_propagate_prefixed_error (GError **dest, GError *src, const gchar *format, ...);

Description

GLib provides a standard method of reporting errors from a called function to the calling code. (This is the same problem solved by exceptions in other languages.) It’s important to understand that this method is both a data type (the GError object) and a set of rules. If you use GError incorrectly, then your code will not properly interoperate with other code that uses GError, and users of your API will probably get confused.

First and foremost: GError should only be used to report recoverable runtime errors, never to report programming errors. If the programmer has screwed up, then you should use g_warning(), g_return_if_fail(), g_assert(), g_error(), or some similar facility. (Incidentally, remember that the g_error() function should only be used for programming errors, it should not be used to print any error reportable via GError.)

Examples of recoverable runtime errors are "file not found" or "failed to parse input." Examples of programming errors are "NULL passed to strcmp()" or "attempted to free the same pointer twice." These two kinds of errors are fundamentally different: runtime errors should be handled or reported to the user, programming errors should be eliminated by fixing the bug in the program. This is why most functions in GLib and GTK+ do not use the GError facility.

Functions that can fail take a return location for a GError as their last argument. For example:

```c
gboolean g_file_get_contents (const gchar *filename, gchar **contents, gsize *length, GError **error);
```

If you pass a non-NULL value for the error argument, it should point to a location where an error can be placed. For example:

```c
gchar *contents;
GError *err = NULL;
g_file_get_contents ("foo.txt", &contents, NULL, &err);
g_assert ((contents == NULL && err != NULL) || (contents != NULL && err == NULL)) ;
if (err != NULL)
{
    /* Report error to user, and free error */
    g_assert (contents == NULL);
    printf (stderr, "Unable to read file: %s\n", err->message);
    g_error_free (err);
}
else
{
    /* Use file contents */
    g_assert (contents != NULL);
}
```
Chapter 3. GLib Core Application

3.8. Error Reporting

Note that err != NULL in this example is a reliable indicator of whether g_file_get_contents() failed. Additionally, g_file_get_contents() returns a boolean which indicates whether it was successful.

Because g_file_get_contents() returns FALSE on failure, if you are only interested in whether it failed and don’t need to display an error message, you can pass NULL for the error argument:

```c
if (g_file_get_contents ("foo.txt", &contents, NULL, NULL)) /* ignore errors */
    /* no error occurred */
else
    /* error */
```

The GError object contains three fields: domain indicates the module the error-reporting function is located in, code indicates the specific error that occurred, and message is a user-readable error message with as many details as possible. Several functions are provided to deal with an error received from a called function: g_error_matches() returns TRUE if the error matches a given domain and code, g_propagate_error() copies an error into an error location (so the calling function will receive it), and g_clear_error() clears an error location by freeing the error and resetting the location to NULL. To display an error to the user, simply display error->message, perhaps along with additional context known only to the calling function (the file being opened, or whatever -- though in the g_file_get_contents() case, error->message already contains a filename).

When implementing a function that can report errors, the basic tool is g_set_error(). Typically, if a fatal error occurs you want to g_set_error(), then return immediately. g_set_error() does nothing if the error location passed to it is NULL. Here’s an example:

```c
gint
foo_open_file (GError **error)
{
    gint fd;
    fd = open ("file.txt", O_RDONLY);
    if (fd < 0)
        {
            g_set_error (error,
                        FOO_ERROR, /* error domain */
                        FOO_ERROR_BLAH, /* error code */
                        "Failed to open file: %s", /* error message format string */
                        g_strerror (errno));
            return -1;
        }
    else
        return fd;
}
```

Things are somewhat more complicated if you yourself call another function that can report a GError. If the sub-function indicates fatal errors in some way other than reporting a GError, such as by returning TRUE on success, you can simply do the following:

```c
gboolean
my_function_that_can_fail (GError **err)
{
    g_return_val_if_fail (err == NULL || *err == NULL, FALSE);
    if (!sub_function_that_can_fail (err))
        {
            /* assert that error was set by the sub-function */
            g_assert (err == NULL || *err != NULL);
            return FALSE;
        }
    /* otherwise continue, no error occurred */
    g_assert (err == NULL || *err == NULL);
}
```

If the sub-function does not indicate errors other than by reporting a GError, you need to create a temporary GError since the passed-in one may be NULL. g_propagate_error() is intended for use in this case.

```c
gboolean
```
my_function_that_can_fail (GError **err)
{
    GError *tmp_error;
    g_return_val_if_fail (err == NULL || *err == NULL, FALSE);
    tmp_error = NULL;
    sub_function_that_can_fail (&tmp_error);
    if (tmp_error != NULL)
    {
        /* store tmp_error in err, if err != NULL,*
         * otherwise call g_error_free() on tmp_error
         */
        g_propagate_error (err, tmp_error);
        return FALSE;
    }
    /* otherwise continue, no error occurred */
}

Error pileups are always a bug. For example, this code is incorrect:

gboolean
my_function_that_can_fail (GError **err)
{
    GError *tmp_error;
    g_return_val_if_fail (err == NULL || *err == NULL, FALSE);
    tmp_error = NULL;
    sub_function_that_can_fail (&tmp_error);
    other_function_that_can_fail (&tmp_error);
    if (tmp_error != NULL)
    {
        g_propagate_error (err, tmp_error);
        return FALSE;
    }
}

tmp_error should be checked immediately after sub_function_that_can_fail(), and either cleared or propagated upward. The rule is: after each error, you must either handle the error, or return it to the calling function. Note that passing NULL for the error location is the equivalent of handling an error by always doing nothing about it. So the following code is fine, assuming errors in sub_function_that_can_fail() are not fatal to my_function_that_can_fail():
The error codes are in an enumeration called `<Namespace><Module>Error`, for example, `GThreadError` or `GSpawnError`.

Members of the error code enumeration are called `<NAMESPACE>_ERROR_<CODE>`, for example `G_SPAWN_ERROR_FORK` or `G_THREAD_ERROR_AGAIN`.

If there’s a "generic" or "unknown" error code for unrecoverable errors it doesn’t make sense to distinguish with specific codes, it should be called `<NAMESPACE>_ERROR_FAILED`, for example `G_SPAWN_ERROR_FAILED` or `G_THREAD_ERROR_FAILED`.

Summary of rules for use of ""

- Do not report programming errors via `GError`.
- The last argument of a function that returns an error should be a location where a `GError` can be placed (i.e. "`GError**` error"). If `GError` is used with varargs, the `GError**` should be the last argument before the "...".
- The caller may pass `NULL` for the `GError**` if they are not interested in details of the exact error that occurred.
- If `NULL` is passed for the `GError**` argument, then errors should not be returned to the caller, but your function should still abort and return if an error occurs. That is, control flow should not be affected by whether the caller wants to get a `GError`.
- If a `GError` is reported, then your function by definition had a fatal failure and did not complete whatever it was supposed to do. If the failure was not fatal, then you handled it and you should not report it. If it was fatal, then you must report it and discontinue whatever you were doing immediately.
- A `GError*` must be initialized to `NULL` before passing its address to a function that can report errors.
- "Piling up" errors is always a bug. That is, if you assign a new `GError` to a `GError*` that is non-`NULL`, thus overwriting the previous error, it indicates that you should have aborted the operation instead of continuing. If you were able to continue, you should have cleared the previous error with `g_clear_error()`. `g_set_error()` will complain if you pile up errors.
- By convention, if you return a boolean value indicating success then `TRUE` means success and `FALSE` means failure. If `FALSE` is returned, the error `must` be set to a non-`NULL` value.
- A `NULL` return value is also frequently used to mean that an error occurred. You should make clear in your documentation whether `NULL` is a valid return value in non-error cases; if `NULL` is a valid value, then users must check whether an error was returned to see if the function succeeded.
- When implementing a function that can report errors, you may want to add a check at the top of your function that the error return location is either `NULL` or contains a `NULL` error (e.g. `g_return_if_fail (error == NULL || *error == NULL);`).

Details

`GError`

typedef struct {
    GQuark domain;
    gint code;
    gchar *message;
    } GError;

The `GError` structure contains information about an error that has occurred.
3.8. ERROR REPORTING

GQuark domain; error domain, e.g. G_FILE_ERROR.

gint code; error code, e.g. G_FILE_ERROR_NOENT.

 gchar *message; human-readable informative error message.

_g_error_new()

<table>
<thead>
<tr>
<th>GError* g_error_new</th>
</tr>
</thead>
<tbody>
<tr>
<td>(GQuark domain, gint code, gchar *format, ...);</td>
</tr>
</tbody>
</table>

Creates a new GError with the given domain and code, and a message formatted with format.

domain: error domain
code: error code
format: printf()-style format for error message
...: parameters for message format

Returns: a new GError

_g_error_new_literal()

<table>
<thead>
<tr>
<th>GError* g_error_new_literal</th>
</tr>
</thead>
<tbody>
<tr>
<td>(GQuark domain, gint code, gchar *message);</td>
</tr>
</tbody>
</table>

Creates a new GError; unlike _g_error_new(), message is not a printf()-style format string. Use this function if message contains text you don’t have control over, that could include printf() escape sequences.

domain: error domain
code: error code
message: error message

Returns: a new GError

_g_error_free()

<table>
<thead>
<tr>
<th>void g_error_free</th>
</tr>
</thead>
<tbody>
<tr>
<td>(GError *error);</td>
</tr>
</tbody>
</table>

Frees a GError and associated resources.
	error: a GError

_g_error_copy()

<table>
<thead>
<tr>
<th>GError* g_error_copy</th>
</tr>
</thead>
<tbody>
<tr>
<td>(const GError *error);</td>
</tr>
</tbody>
</table>

Makes a copy of error.
	error: a GError

Returns: a new GError
CHAPTER 3. GLIB CORE APPLICATION

3.8. ERROR REPORTING

**g_error_matches ()**

```c
gboolean g_error_matches (const GError *error, GQuark domain, gint code);
```

Returns **TRUE** if `error` matches `domain` and `code`, **FALSE** otherwise.

- **error**: a GError
- **domain**: an error domain
- **code**: an error code
- **Returns**: whether `error` has `domain` and `code`

**g_set_error ()**

```c
void g_set_error (GError **err, GQuark domain, gint code, const gchar *format, ...);
```

Does nothing if `err` is **NULL**; if `err` is non-**NULL**, then *`err` must be **NULL**. A new GError is created and assigned to *`err`.

- **err**: a return location for a GError, or **NULL**
- **domain**: error domain
- **code**: error code
- **format**: printf()-style format
- **...**: args for `format`

**g_set_error_literal ()**

```c
void g_set_error_literal (GError **err, GQuark domain, gint code, const gchar *message);
```

Does nothing if `err` is **NULL**; if `err` is non-**NULL**, then *`err` must be **NULL**. A new GError is created and assigned to *`err`. Unlike `g_set_error()`, `message` is not a printf()-style format string. Use this function if `message` contains text you don’t have control over, that could include printf() escape sequences.

- **err**: a return location for a GError, or **NULL**
- **domain**: error domain
- **code**: error code
- **message**: error message
- Since 2.18

**g_propagate_error ()**

```c
void g_propagate_error (GError **dest, GError *src);
```

If `dest` is **NULL**, free `src`; otherwise, moves `src` into *`dest`. The error variable `dest` points to must be **NULL**.

- **dest**: error return location
- **src**: error to move into the return location
CHAPTER 3. GLIB CORE APPLICATION

3.9. MESSAGE OUTPUT AND DEBUGGING

**g_clear_error ()**

```c
void g_clear_error (GError **err);
```

If *err is **NULL**, does nothing. If *err is non-NULL, calls `g_error_free()` on *err and sets *err to **NULL**.

*err*: a **GError** return location

**g_prefix_error ()**

```c
void g_prefix_error (GError **err, const gchar *format, ...);
```

Formats a string according to *format* and prefix it to an existing error message. If *err is **NULL** (ie: no error variable) then do nothing.

If *err is **NULL** (ie: an error variable is present but there is no error condition) then also do nothing. Whether or not it makes sense to take advantage of this feature is up to you.

*err*: a return location for a **GError**, or **NULL**

*format*: **printf()-style** format string

...*: arguments to *format*

Since 2.16

**g_propagate_prefixed_error ()**

```c
void g_propagate_prefixed_error (GError **dest, GError *src, const gchar *format, ...);
```

If *dest is **NULL**, free *src*; otherwise, moves *src* into *dest. *dest must be **NULL**. After the move, add a prefix as with `g_prefix_error()`.

*dest*: error return location

*src*: error to move into the return location

*format*: **printf()-style** format string

...*: arguments to *format*

Since 2.16

### 3.9 Message Output and Debugging Functions

**Name**

Message Output and Debugging Functions – functions to output messages and help debug applications
### Synopsis

```c
#include <glib.h>

void g_print (const gchar *format, ...);
GPrintFunc g_set_print_handler (GPrintFunc func);
void (*GPrintFunc) (const gchar *string);
void g_printerr (const gchar *format, ...);
GPrintFunc g_set_printerr_handler (GPrintFunc func);
#define g_return_if_fail (expr)
#define g_return_val_if_fail (expr,val)
#define g_return_if_reached ()
#define g_return_val_if_reached (val)
#define g_warn_if_fail (expr)
#define g_warn_if_reached ()

void g_on_error_query (const gchar *prg_name);
void g_on_error_stack_trace (const gchar *prg_name);
#define G_BREAKPOINT ()
```

### Description

These functions provide support for outputting messages.

The `g_return` family of macros (`g_return_if_fail()`, `g_return_val_if_fail()`, `g_return_if_reached()`, `g_return_val_if_reached()`) should only be used for programming errors, a typical use case is checking for invalid parameters at the beginning of a public function. They should not be used if you just mean "if (error) return", they should only be used if you mean "if (bug in program) return". The program behavior is generally considered undefined after one of these checks fails. They are not intended for normal control flow, only to give a perhaps-helpful warning before giving up.

### Details

#### g_print

```c
void g_print (const gchar *format, ...);
```

Outputs a formatted message via the print handler. The default print handler simply outputs the message to stdout.

`g_print()` should not be used from within libraries for debugging messages, since it may be redirected by applications to special purpose message windows or even files. Instead, libraries should use `g_log()`, or the convenience functions `g_message()`, `g_warning()` and `g_error()`.

- **format**: the message format. See the `printf()` documentation.
- **...**: the parameters to insert into the format string.

#### g_set_print_handler

```c
GPrintFunc g_set_print_handler (GPrintFunc func);
```

Sets the print handler. Any messages passed to `g_print()` will be output via the new handler. The default handler simply outputs the message to stdout. By providing your own handler you can redirect the output, to a GTK+ widget or a log file for example.
**func**: the new print handler.

**Returns**: the old print handler.

**GPrintFunc ()**

```c
void (*GPrintFunc)(const gchar *string);
```

Specifies the type of the print handler functions. These are called with the complete formatted string to output.

**string**: the message to be output.

**g_printerr ()**

```c
void g_printerr(const gchar *format, ...);
```

Outputs a formatted message via the error message handler. The default handler simply outputs the message to stderr.

`g_printerr()` should not be used from within libraries. Instead `g_log()` should be used, or the convenience functions `g_message()`, `g_warning()` and `g_error()`.

**format**: the message format. See the `printf()` documentation.

...: the parameters to insert into the format string.

**g_set_printerr_handler ()**

```c
GPrintFunc g_set_printerr_handler(GPrintFunc func);
```

Sets the handler for printing error messages. Any messages passed to `g_printerr()` will be output via the new handler. The default handler simply outputs the message to stderr. By providing your own handler you can redirect the output, to a GTK+ widget or a log file for example.

**func**: the new error message handler.

**Returns**: the old error message handler.

**g_return_if_fail()**

```c
#define g_return_if_fail(expr)
```

Returns from the current function if the expression is not true. If the expression evaluates to `FALSE`, a critical message is logged and the function returns. This can only be used in functions which do not return a value.

**expr**: the expression to check.

**g_return_val_if_fail()**

```c
#define g_return_val_if_fail(expr,val)
```

Returns from the current function, returning the value `val`, if the expression is not true. If the expression evaluates to `FALSE`, a critical message is logged and `val` is returned.

**expr**: the expression to check.

**val**: the value to return from the current function if the expression is not true.
g_return_if_reached()

#define g_return_if_reached()

Logs a critical message and returns from the current function. This can only be used in functions which do not return a value.

g_return_val_if_reached()

#define g_return_val_if_reached(val)

Logs a critical message and returns val.

val: the value to return from the current function.

g_warn_if_fail()

#define g_warn_if_fail(expr)

Logs a warning if the expression is not true.

expr: the expression to check

Since 2.16

g_warn_if_reached()

#define g_warn_if_reached()

Logs a critical warning.

Since 2.16

g_on_error_query()

void g_on_error_query (const gchar *prg_name);

Prompts the user with [E]xit, [H]alt, show [S]tack trace or [P]roceed. This function is intended to be used for debugging use only. The following example shows how it can be used together with the g_log() functions.

#include <glib.h>
static void
log_handler (const gchar *log_domain,
             GLogLevelFlags log_level,
             const gchar *message,
             gpointer user_data)
{
  g_log_default_handler (log_domain, log_level, message, user_data);
  g_on_error_query (MY_PROGRAM_NAME);
}
int main (int argc, char *argv[])
{
  g_log_set_handler (MY_LOG_DOMAIN,
                    G_LOG_LEVEL.WARNING |
                    G_LOG_LEVEL.ERROR |
                    G_LOG_LEVEL_CRITICAL,
                    log_handler,
                    NULL);
  /* ... */
If [E]xit is selected, the application terminates with a call to _exit(0).
If [H]alt is selected, the application enters an infinite loop. The infinite loop can only be stopped by killing the application, or by setting glib_on_error_halt to FALSE (possibly via a debugger).
If [S]tack trace is selected, g_on_error_stack_trace() is called. This invokes gdb, which attaches to the current process and shows a stack trace. The prompt is then shown again.
If [P]roceed is selected, the function returns.
This function may cause different actions on non-UNIX platforms.

**prg_name**: the program name, needed by gdb for the [S]tack trace option. If prg_name is NULL, g_get_prgname() is called to get the program name (which will work correctly if gdk_init() or gtk_init() has been called).

### g_on_error_stack_trace()

```c
void g_on_error_stack_trace (const gchar *prg_name);
```

Invokes gdb, which attaches to the current process and shows a stack trace. Called by g_on_error_query() when the [S]tack trace option is selected.
This function may cause different actions on non-UNIX platforms.

**prg_name**: the program name, needed by gdb for the [S]tack trace option. If prg_name is NULL, g_get_prgname() is called to get the program name (which will work correctly if gdk_init() or gtk_init() has been called).

### G_BREAKPOINT()

```c
#define G_BREAKPOINT()
```

Inserts a breakpoint instruction into the code. On x86 and alpha systems this is implemented as a soft interrupt and on other architectures it raises a SIGTRAP signal.

## 3.10 Message Logging

### Name

Message Logging – versatile support for logging messages with different levels of importance

### Synopsis

```c
#include <glib.h>

#define G_LOG_DOMAIN
#define G_LOG_FATAL_MASK
#define G_LOG_LEVEL_USER_SHIFT
void (*GLogFunc) (const gchar *log_domain,
                   GLogLevelFlags log_level,
                   const gchar *message,
                   gpointer user_data);

enum GLogLevelFlags;

void g_log
    (const gchar *log_domain,
     GLogLevelFlags log_level,
     const gchar *format,
     ...
    );

void g_logv
    (const gchar *log_domain,
     GLogLevelFlags log_level,
     const gchar *format,
     va_list args);
```

182
CHAPTER 3. GLIB CORE APPLICATION . . .

3.10. MESSAGE LOGGING

#define
#define
#define
#define
#define

g_message
g_warning
g_critical
g_error
g_debug

(...)
(...)
(...)
(...)
(...)

guint

g_log_set_handler

void

g_log_remove_handler

GLogLevelFlags
GLogLevelFlags

g_log_set_always_fatal
g_log_set_fatal_mask

void

g_log_default_handler

GLogFunc

g_log_set_default_handler

(const gchar *log_domain,
GLogLevelFlags log_levels,
GLogFunc log_func,
gpointer user_data);
(const gchar *log_domain,
guint handler_id);
(GLogLevelFlags fatal_mask);
(const gchar *log_domain,
GLogLevelFlags fatal_mask);
(const gchar *log_domain,
GLogLevelFlags log_level,
const gchar *message,
gpointer unused_data);
(GLogFunc log_func,
gpointer user_data);

Description
These functions provide support for logging error messages or messages used for debugging.
There are several built-in levels of messages, defined in GLogLevelFlags. These can be extended
with user-defined levels.

Details
G_LOG_DOMAIN
#define G_LOG_DOMAIN

((gchar*) 0)

Defines the log domain. For applications, this is typically left as the default NULL (or "") domain.
Libraries should define this so that any messages which they log can be differentiated from messages
from other libraries and application code. But be careful not to define it in any public header files.
For example, GTK+ uses this in its Makefile.am:
INCLUDES = -DG_LOG_DOMAIN=\"Gtk\"

G_LOG_FATAL_MASK
#define G_LOG_FATAL_MASK

(G_LOG_FLAG_RECURSION | G_LOG_LEVEL_ERROR)

GLib log levels that are considered fatal by default.
G_LOG_LEVEL_USER_SHIFT
#define G_LOG_LEVEL_USER_SHIFT

(8)

Log level shift offset for user defined log levels (0-7 are used by GLib).
GLogFunc ()

183


void (GLogFunc) (const gchar *log_domain, GLogLevelFlags log_level, const gchar *message, gpointer user_data);

Specifies the prototype of log handler functions.

log_domain: the log domain of the message.

log_level: the log level of the message (including the fatal and recursion flags).

message: the message to process.

user_data: user data, set in g_log_set_handler().

e num GLogLevelFlags

type def enum
{
    /* log flags */
    G_LOG_FLAG_RECURSION = 1 << 0,
    G_LOG_FLAG_FATAL = 1 << 1,
    /* GLib log levels */
    G_LOG_LEVEL_ERROR = 1 << 2, /* always fatal */
    G_LOG_LEVEL_CRITICAL = 1 << 3,
    G_LOG_LEVEL_WARNING = 1 << 4,
    G_LOG_LEVEL_MESSAGE = 1 << 5,
    G_LOG_LEVEL_INFO = 1 << 6,
    G_LOG_LEVEL_DEBUG = 1 << 7,
    G_LOG_LEVEL_MASK = ~(G_LOG_FLAG_RECURSION | G_LOG_FLAG_FATAL)
} GLogLevelFlags;

Flags specifying the level of log messages. It is possible to change how GLib treats messages of the various levels using g_log_set_handler() and g_log_set_fatal_mask().

G_LOG_FLAG_RECURSION internal flag

G_LOG_FLAG_FATAL internal flag

G_LOG_LEVEL_ERROR log level for errors, see g_error(). This level is also used for messages produced by g_assert().

G_LOG_LEVEL_CRITICAL log level for critical messages, see g_critical(). This level is also used for messages produced by g_return_if_fail() and g_return_val_if_fail().

G_LOG_LEVEL_WARNING log level for warnings, see g_warning()

G_LOG_LEVEL_MESSAGE log level for messages, see g_message()

G_LOG_LEVEL_INFO log level for informational messages

G_LOG_LEVEL_DEBUG log level for debug messages, see g_debug()

G_LOG_LEVEL_MASK a mask including all log levels.
**g_log()**

```c
void g_log (const gchar *log_domain,
            GLogLevelFlags log_level,
            const gchar *format,
            ...);
```

Logs an error or debugging message. If the log level has been set as fatal, the `abort()` function is called to terminate the program.

- **log_domain**: the log domain, usually `G_LOG_DOMAIN`.
- **log_level**: the log level, either from `GLogLevelFlags` or a user-defined level.
- **format**: the message format. See the `printf()` documentation.
- **...**: the parameters to insert into the format string.

**g_logv()**

```c
void g_logv (const gchar *log_domain,
             GLogLevelFlags log_level,
             const gchar *format,
             va_list args);
```

Logs an error or debugging message. If the log level has been set as fatal, the `abort()` function is called to terminate the program.

- **log_domain**: the log domain.
- **log_level**: the log level.
- **format**: the message format. See the `printf()` documentation.
- **args**: the parameters to insert into the format string.

**g_message()**

```c
#define g_message(...)  
```

A convenience function/macro to log a normal message.

- **...**: format string, followed by parameters to insert into the format string (as with `printf()`)

**g_warning()**

```c
#define g_warning(...)  
```

A convenience function/macro to log a warning message.

You can make warnings fatal at runtime by setting the `G_DEBUG` environment variable (see Running GLib Applications).

- **...**: format string, followed by parameters to insert into the format string (as with `printf()`)

**g_critical()**

```c
#define g_critical(...)  
```

Logs a "critical warning" (`G_LOG_LEVEL_CRITICAL`). It's more or less application-defined what constitutes a critical vs. a regular warning. You could call `g_log_set_always_fatal()` to make critical warnings exit the program, then use `g_critical()` for fatal errors, for example.

You can also make critical warnings fatal at runtime by setting the `G_DEBUG` environment variable (see Running GLib Applications).

- **...**: format string, followed by parameters to insert into the format string (as with `printf()`)
3.10. MESSAGE LOGGING

**g_error()**

```c
#define g_error(...) (void) abort();
```

A convenience function/macro to log an error message. Error messages are always fatal, resulting in a call to `abort()` to terminate the application. This function will result in a core dump; don’t use it for errors you expect. Using this function indicates a bug in your program, i.e. an assertion failure.

...: format string, followed by parameters to insert into the format string (as with `printf()`)

**g_debug()**

```c
#define g_debug(...) (void) printf(...);
```

A convenience function/macro to log a debug message.

...: format string, followed by parameters to insert into the format string (as with `printf()`)

Since 2.6

**g_log_set_handler()**

```c
guint g_log_set_handler (const gchar *log_domain, GLogLevelFlags log_levels, GLogFunc log_func, gpointer user_data);
```

Sets the log handler for a domain and a set of log levels. To handle fatal and recursive messages the `log_levels` parameter must be combined with the `G_LOG_FLAG_FATAL` and `G_LOG_FLAG_RECURSION` bit flags.

Note that since the `G_LOG_LEVEL_ERROR` log level is always fatal, if you want to set a handler for this log level you must combine it with `G_LOG_FLAG_FATAL`.

**Example 3.11** Adding a log handler for all warning messages in the default (application) domain

```c
g_log_set_handler (NULL, G_LOG_LEVEL_WARNING | G_LOG_FLAG_FATAL | G_LOG_FLAG_RECURSION, my_log_handler, NULL);
```

**Example 3.12** Adding a log handler for all critical messages from GTK+

```c
g_log_set_handler ("Gtk", G_LOG_LEVEL_CRITICAL | G_LOG_FLAG_FATAL | G_LOG_FLAG_RECURSION, my_log_handler, NULL);
```

**Example 3.13** Adding a log handler for all messages from GLib

```c
g_log_set_handler ("GLib", G_LOG_LEVEL_MASK | G_LOG_FLAG_FATAL | G_LOG_FLAG_RECURSION, my_log_handler, NULL);
```

- **log_domain**: the log domain, or `NULL` for the default application domain.
- **log_levels**: the log levels to apply the log handler for. To handle fatal and recursive messages as well, combine the log levels with the `G_LOG_FLAG_FATAL` and `G_LOG_FLAG_RECURSION` bit flags.
- **log_func**: the log handler function.
- **user_data**: data passed to the log handler.
- **Returns**: the id of the new handler.
3.10. MESSAGE LOGGING

**g_log_remove_handler ( )**

```c
void g_log_remove_handler (const gchar *log_domain,
                           guint handler_id);
```

Removes the log handler.

*log_domain*: the log domain.

*handler_id*: the id of the handler, which was returned in *g_log_set_handler*( ).

**g_log_set_always_fatal ( )**

```c
GLogLevelFlags g_log_set_always_fatal (GLogLevelFlags ← fatal_mask);
```

Sets the message levels which are always fatal, in any log domain. When a message with any of these levels is logged the program terminates. You can only set the levels defined by GLib to be fatal. **G_LOG_LEVEL_ERROR** is always fatal.

You can also make some message levels fatal at runtime by setting the G_DEBUG environment variable (see Running GLib Applications).

*fatal_mask*: the mask containing bits set for each level of error which is to be fatal.

*Returns*: the old fatal mask.

**g_log_set_fatal_mask ( )**

```c
GLogLevelFlags g_log_set_fatal_mask (const gchar *log_domain,
                                      GLogLevelFlags ← fatal_mask);
```

Sets the log levels which are fatal in the given domain. **G_LOG_LEVEL_ERROR** is always fatal.

*log_domain*: the log domain.

*fatal_mask*: the new fatal mask.

*Returns*: the old fatal mask for the log domain.

**g_log_default_handler ( )**

```c
void g_log_default_handler (const gchar *log_domain,
                           GLogLevelFlags log_level ← ,
                           const gchar *message,
                           gpointer unused_data);
```

The default log handler set up by GLib; *g_log_set_default_handler*( ) allows to install an alternate default log handler. This is used if no log handler has been set for the particular log domain and log level combination. It outputs the message to stderr or stdout and if the log level is fatal it calls abort().

stderr is used for levels **G_LOG_LEVEL_ERROR**, **G_LOG_LEVEL_CRITICAL**, **G_LOG_LEVEL_WARNING** and **G_LOG_LEVEL_MESSAGE**. stdout is used for the rest.

*log_domain*: the log domain of the message.

*log_level*: the level of the message.

*message*: the message.

*unused_data*: data passed from *g_log()* which is unused.
3.10. MESSAGE LOGGING

**g_log_set_default_handler()**

| GLogFunc | g_log_set_default_handler | (GLogFunc log_func, gpointer user_data); |

Installs a default log handler which is used if no log handler has been set for the particular log domain and log level combination. By default, GLib uses `g_log_default_handler()` as default log handler.

**log_func**: the log handler function.

**user_data**: data passed to the log handler.

**Returns**: the previous default log handler

Since 2.6
Chapter 4

GLib Utilities

4.1 String Utility Functions

Name
String Utility Functions – various string-related functions

Synopsis

```c
#include <glib.h>
#include <glib/gprintf.h>

char* g_strdup (const char *str);
char* g_strndup (const char *str,
                 size_t n);
char** g_strdupv (char **str_array);
char* g_strnfill (size_t length,
                  char fill_char);
char * g_stpcpy (char *dest,
                 const char *src);
char * g_strstr_len (const char *haystack,
                     size_t haystack_len,
                     const char *needle);
char * g_strrstr (const char *haystack,
                 const char *needle);
char * g_strrstr_len (const char *haystack,
                     size_t haystack_len,
                     const char *needle);
bool g_str_has_prefix (const char *str,
                       const char *prefix);
bool g_str_has_suffix (const char *str,
                      const char *suffix);
int g_strcmp0 (const char *str1,
              const char *str2);
size_t g_strlcpy (char *dest,
                const char *src,
                size_t dest_size);
size_t g_strlcat (char *dest,
                  const char *src,
                  size_t dest_size);
char* g_strdup_printf (const char *format,
                       ...);
```
4.1. STRING UTILITY FUNCTIONS

gchar* g_strdup_vprintf (const gchar *format, va_list args);
gint g_printf (gchar const *format, ...);
gint g_vprintf (gchar const *format, va_list args);
gint g_fprintf (FILE *file, gchar const *format, ...);
gint g_vfprintf (FILE *file, gchar *string, va_list args);
gint g_sprintf (gchar *string, gchar const *format, ...);
gint g_vsprintf (gchar *string, gchar const *format, va_list args);
gint g_snprintf (gchar *string, gulong n, gchar const *format, ...);
gint g_vsnprintf (gchar *string, gulong n, gchar const *format, va_list args);
gint g_vasprintf (gchar **string, gchar const *format, va_list args);
gsize g_printf_string_upper_bound (const gchar *format, va_list args);
gboolean g_ascii_isalnum (gchar c);
gboolean g_ascii_isalpha (gchar c);
gboolean g_ascii_iscntrl (gchar c);
gboolean g_ascii_isdigit (gchar c);
gboolean g_ascii_isgraph (gchar c);
gboolean g_ascii_islower (gchar c);
gboolean g_ascii_isprint (gchar c);
gboolean g_ascii_ispunct (gchar c);
gboolean g_ascii_isspace (gchar c);
gboolean g_ascii_isupper (gchar c);
gboolean g_ascii_isxdigit (gchar c);
gint g_ascii_digit_value (gchar c);
gint g_ascii_xdigit_value (gchar c);
gint g_ascii_strcasecmp (const gchar *s1, const gchar *s2);
gint g_ascii_strncasecmp (const gchar *s1, const gchar *s2, gsize n);
gchar* g_ascii_strup (const gchar *str, gssize len);
gchar* g_ascii_strdown (const gchar *str, gssize len);
gchar g_ascii_tolower (gchar c);
CHAPTER 4. GLIB UTILITIES
4.1. STRING UTILITY FUNCTIONS

```c
#include <glib.h>

// 4.1. STRING UTILITY FUNCTIONS

// g_ascii_toupper takes a single character and converts it to uppercase.
\verb|gchar g_ascii_toupper (gchar c);|

// g_string_ascii_up takes a GString and converts all characters to uppercase.
\verb|GString* g_string_ascii_up (GString *string);|

// g_string_ascii_down takes a GString and converts all characters to lowercase.
\verb|GString* g_string_ascii_down (GString *string);|

// g_strup takes a string and converts all characters to uppercase.
\verb|gchar* g_strup (gchar *string);|

// g_strdown takes a string and converts all characters to lowercase.
\verb|gchar* g_strdown (gchar *string);|

// g_strcasecmp compares two strings case-insensitively.
\verb|gint g_strcasecmp (const gchar *s1, const gchar *s2);|

// g_strncasecmp compares two strings case-insensitively up to n characters.
\verb|gint g_strncasecmp (const gchar *s1, const gchar *s2, guint n);|

// g_strreverse reverses the order of characters in a string.
\verb|gchar* g_strreverse (gchar *string);|

// g_ascii_strtoll converts a string to a long integer, considering the specified base.
\verb|guint64 g_ascii_strtoll (const gchar *nptr, gchar **endptr, guint base);|

// g_ascii_strtoull converts a string to an unsigned long integer, considering the specified base.
\verb|guint64 g_ascii_strtoull (const gchar *nptr, gchar **endptr, guint base);|

// g_ascii_dtostr_buf_size defines the buffer size for ASCII representation.
\verb|#define G_ASCII_DTOSTR_BUF_SIZE|

// g_ascii_strtod converts a string to a double precision floating point number.
\verb|gdouble g_ascii_strtod (const gchar *nptr, gchar **endptr);|

// g_ascii_dtostr converts a double to a string.
\verb|gchar * g_ascii_dtostr (gchar *buffer, gint buf_len, gdouble d);|

// g_ascii_formatd formats a double according to a format string.
\verb|gchar * g_ascii_formatd (gchar *buffer, gint buf_len, const gchar *format, gdouble d);|

// g_strtod converts a string to a double precision floating point number.
\verb|gdouble g_strtod (const gchar *nptr, gchar **endptr);|

// g_strchug unwinds the hugging of characters in a string.
\verb|gchar* g_strchug (gchar *string);|

// g_strchomp removes trailing whitespace from a string.
\verb|gchar* g_strchomp (gchar *string);|

// g_strstrip strips leading and trailing whitespace from a string.
\verb|#define g_strstrip ( string )|

// g_strdelimit delimits a string.
\verb|gchar* g_strdelimit (gchar *string, const gchar *delimiters, gchar new_delimiter);|

// g_strescape escapes characters in a string.
\verb|#define G_STR_DELIMITERS|

// g_strcompress compresses a string.
\verb|gchar* g_strcompress (const gchar *source);|

// g_strcanon canonicalizes a string.
\verb|gchar* g_strcanon (gchar *string, const gchar *valid_chars, gchar substitutor);|

// g_strsplit splits a string based on a delimiter.
\verb|gchar** g_strsplit (const gchar *string, const gchar *delimiter, gint max_tokens);|

// g_strsplit_set splits a string based on a set of delimiters.
\verb|gchar** g_strsplit_set (const gchar *string, const gchar *delimiters, gint max_tokens);|

// g_strfreev frees an array of strings.
\verb|void g_strfreev (gchar **str_array);|

// g_strconcat concatenates multiple strings.
\verb|gchar* g_strconcat (const gchar *string1, ...);|

// g_strjoin joins multiple strings with a separator.
\verb|gchar* g_strjoin (const gchar *separator, ...);|
```

191
CHAPTER 4. GLIB UTILITIES

4.1. STRING UTILITY FUNCTIONS

...);

(gchar **str_array);

(const gchar *separator,

gchar **str_array);

Description

This section describes a number of utility functions for creating, duplicating, and manipulating strings.

Note that the functions g_printf(), g_fprintf(), g_sprintf(), g_snprintf(), g_vprintf(), g_vfprintf(), g_vsprintf() and g_vsnprintf() are declared in the header gprintf.h which is not included in glib.h (otherwise using glib.h would drag in stdio.h), so you’ll have to explicitly include <glib/gprintf.h> in order to use the GLib printf() functions.

While you may use the printf() functions to format UTF-8 strings, notice that the precision of a %Ns parameter is interpreted as the number of bytes, not characters to print. On top of that, the GNU libc implementation of the printf() functions has the “feature” that it checks that the string given for the %Ns parameter consists of a whole number of characters in the current encoding. So, unless you are sure you are always going to be in an UTF-8 locale or your know your text is restricted to ASCII, avoid using %Ns. If your intention is to format strings for a certain number of columns, then %Ns is not a correct solution anyway, since it fails to take wide characters (see g_unichar_iswide()) into account.

Details

g_strdup ()

Duplicates a string. If str is NULL it returns NULL. The returned string should be freed with g_free() when no longer needed.

str : the string to duplicate

Returns : a newly-allocated copy of str

g_strndup ()

Duplicates the first n bytes of a string, returning a newly-allocated buffer n + 1 bytes long which will always be nul-terminated. If str is less than n bytes long the buffer is padded with nuls. If str is NULL it returns NULL. The returned value should be freed when no longer needed.

NOTE

To copy a number of characters from a UTF-8 encoded string, use g_utf8_strncpy() instead.

str : the string to duplicate

n : the maximum number of bytes to copy from str

Returns : a newly-allocated buffer containing the first n bytes of str, nul-terminated
CHAPTER 4. GLIB UTILITIES

4.1. STRING UTILITY FUNCTIONS

**g_strdupv()**

```c
void g_strdupv (gchar **str_array);
```

Copies NULL-terminated array of strings. The copy is a deep copy; the new array should be freed by first freeing each string, then the array itself. `g_strfreev()` does this for you. If called on a NULL value, `g_strdupv()` simply returns NULL.

*str_array*: NULL-terminated array of strings.

*Returns*: a new NULL-terminated array of strings.

**g_strnfill()**

```c
gchar* g_strnfill (gsize length, gchar fill_char);
```

Creates a new string `length` bytes long filled with `fill_char`. The returned string should be freed when no longer needed.

*length*: the length of the new string

*fill_char*: the byte to fill the string with

*Returns*: a newly-allocated string filled the `fill_char`

**g_stpcpy()**

```c
gchar* g_stpcpy (gchar *dest, const char *src);
```

Copies a null-terminated string into the `dest` buffer, include the trailing null, and return a pointer to the trailing null byte. This is useful for concatenating multiple strings together without having to repeatedly scan for the end.

*dest*: destination buffer.

*src*: source string.

*Returns*: a pointer to trailing null byte.

**g_strstr_len()**

```c
gchar* g_strstr_len (const gchar *haystack, gssize haystack_len, const gchar *needle);
```

Searches the string `haystack` for the first occurrence of the string `needle`, limiting the length of the search to `haystack_len`.

*haystack*: a string.

*haystack_len*: the maximum length of `haystack`. Note that -1 is a valid length, if `haystack` is null-terminated, meaning it will search through the whole string.

*needle*: the string to search for.

*Returns*: a pointer to the found occurrence, or NULL if not found.
CHAPTER 4. GLIB UTILITIES 4.1. STRING UTILITY FUNCTIONS

**g_strrstr ()**

```c
const gchar * g_strrstr (const gchar *haystack, const gchar *needle);
```

Searches the string `haystack` for the last occurrence of the string `needle`.

- **haystack**: a nul-terminated string.
- **needle**: the nul-terminated string to search for.
- **Returns**: a pointer to the found occurrence, or `NULL` if not found.

**g_strrstr_len ()**

```c
const gchar * g_strrstr_len (const gchar *haystack, gssize haystack_len, const gchar *needle);
```

Searches the string `haystack` for the last occurrence of the string `needle`, limiting the length of the search to `haystack_len`.

- **haystack**: a nul-terminated string.
- **haystack_len**: the maximum length of `haystack`.
- **needle**: the nul-terminated string to search for.
- **Returns**: a pointer to the found occurrence, or `NULL` if not found.

**g_str_has_prefix ()**

```c
gboolean g_str_has_prefix (const gchar *str, const gchar *prefix);
```

Looks whether the string `str` begins with `prefix`.

- **str**: a nul-terminated string.
- **prefix**: the nul-terminated prefix to look for.
- **Returns**: `TRUE` if `str` begins with `prefix`, `FALSE` otherwise.

Since 2.2

**g_str_has_suffix ()**

```c
gboolean g_str_has_suffix (const gchar *str, const gchar *suffix);
```

Looks whether the string `str` ends with `suffix`.

- **str**: a nul-terminated string.
- **suffix**: the nul-terminated suffix to look for.
- **Returns**: `TRUE` if `str` end with `suffix`, `FALSE` otherwise.

Since 2.2
CHAPTER 4. GLIB UTILITIES

4.1. STRING UTILITY FUNCTIONS

g_strcmp0 ()

int g_strcmp0 (const char *str1, const char *str2);

Compares \textit{str1} and \textit{str2} like \texttt{strcmp}. Handles \texttt{NULL} gracefully by sorting it before non-\texttt{NULL} strings.

\textit{str1} : a C string or \texttt{NULL}

\textit{str2} : another C string or \texttt{NULL}

\textbf{Returns} : -1, 0 or 1, if \textit{str1} is \textless, \texttt{==} or \textgreater{} than \textit{str2}.

Since 2.16

g_strlcpy ()

\begin{verbatim}
gsize g_strlcpy (gchar *dest, const gchar *src, gsize dest_size);
\end{verbatim}

Portability wrapper that calls \texttt{strlcpy} on systems which have it, and emulates \texttt{strlcpy} otherwise. Copies \textit{src} to \textit{dest}; \textit{dest} is guaranteed to be null-terminated; \textit{src} must be null-terminated; \textit{dest\_size} is the buffer size, not the number of chars to copy.

At most \textit{dest\_size} - 1 characters will be copied. Always null-terminates (unless \textit{dest\_size} == 0). This function does not allocate memory. Unlike \texttt{strncpy}, this function doesn’t pad \textit{dest} (so it’s often faster). It returns the size of the attempted result, \texttt{strlen} (\textit{src}), so if \textit{retval} >= \textit{dest\_size}, truncation occurred.

\begin{itemize}
\item \textit{dest} : destination buffer
\item \textit{src} : source buffer
\item \textit{dest\_size} : length of \textit{dest} in bytes
\end{itemize}

\textbf{Returns} : length of \textit{src}

g_strlcat ()

\begin{verbatim}
gsize g_strlcat (gchar *dest, const gchar *src, gsize dest_size);
\end{verbatim}

Portability wrapper that calls \texttt{strlcat} on systems which have it, and emulates it otherwise. Appends null-terminated \textit{src} string to \textit{dest}, guaranteeing null-termination for \textit{dest}. The total size of \textit{dest} won’t exceed \textit{dest\_size}.

At most \textit{dest\_size} - 1 characters will be copied. Unlike \texttt{strcat}, \textit{dest\_size} is the full size of \textit{dest}, not the space left over. This function does NOT allocate memory. This always \texttt{NULL} terminates (unless \textit{sz} == 0 or there were no \texttt{NULL} characters in the \textit{dest\_size} characters of \textit{dest} to start with).

\begin{itemize}
\item \textit{dest} : destination buffer, already containing one null-terminated string
\item \textit{src} : source buffer
\item \textit{dest\_size} : length of \textit{dest} buffer in bytes (not length of existing string inside \textit{dest})
\end{itemize}
CHAPTER 4. GLIB UTILITIES

4.1. STRING UTILITY FUNCTIONS

Returns: size of attempted result, which is MIN(dest_size, strlen(original dest)) + strlen(src), so if
retval >= dest_size, truncation occurred.

<table>
<thead>
<tr>
<th>NOTE</th>
</tr>
</thead>
</table>
| Caveat: this is supposedly a more secure alternative to strcat() or strncat(), but for
real security g_strconcat() is harder to mess up. |

**g_strdup_printf()**

gchar* g_strdup_printf (const gchar *format, ...);

Similar to the standard C sprintf() function but safer, since it calculates the maximum space required
and allocates memory to hold the result. The returned string should be freed with g_free() when no
longer needed.

*format*: a standard printf() format string, but notice string precision pitfalls

...: the parameters to insert into the format string

Returns: a newly-allocated string holding the result

**g_strdup_vprintf()**

gchar* g_strdup_vprintf (const gchar *format, va_list args);

Similar to the standard C vsprintf() function but safer, since it calculates the maximum space required
and allocates memory to hold the result. The returned string should be freed with g_free() when no
longer needed.

See also g_vasprintf(), which offers the same functionality, but additionally returns the length of the
allocated string.

*format*: a standard printf() format string, but notice string precision pitfalls

*args*: the list of parameters to insert into the format string

Returns: a newly-allocated string holding the result

**g_printf()**

gint g_printf (gchar const *format, ...);

An implementation of the standard printf() function which supports positional parameters, as spec-
ified in the Single Unix Specification.

*format*: a standard printf() format string, but notice string precision pitfalls.

...: the arguments to insert in the output.

Returns: the number of bytes printed.

Since 2.2
CHAPTER 4. GLIB UTILITIES

4.1. STRING UTILITY FUNCTIONS

**g_vprintf ()**

| gint g_vprintf (gchar const *format, va_list args); |

An implementation of the standard `vprintf()` function which supports positional parameters, as specified in the Single Unix Specification.

**format**: a standard `printf()` format string, but notice string precision pitfalls.

**args**: the list of arguments to insert in the output.

**Returns**: the number of bytes printed.

Since 2.2

**g_fprintf ()**

| gint g_fprintf (FILE *file, gchar const *format, ...); |

An implementation of the standard `fprintf()` function which supports positional parameters, as specified in the Single Unix Specification.

**file**: the stream to write to.

**format**: a standard `printf()` format string, but notice string precision pitfalls.

**...**: the arguments to insert in the output.

**Returns**: the number of bytes printed.

Since 2.2

**g_vfprintf ()**

| gint g_vfprintf (FILE *file, gchar const *format, va_list args); |

An implementation of the standard `fprintf()` function which supports positional parameters, as specified in the Single Unix Specification.

**file**: the stream to write to.

**format**: a standard `printf()` format string, but notice string precision pitfalls.

**args**: the list of arguments to insert in the output.

**Returns**: the number of bytes printed.

Since 2.2

**g_sprintf ()**

| gint g_sprintf (gchar *string, gchar const *format, ...); |

An implementation of the standard `sprintf()` function which supports positional parameters, as specified in the Single Unix Specification.

**string**: A pointer to a memory buffer to contain the resulting string. It is up to the caller to ensure that the allocated buffer is large enough to hold the formatted result.
CHAPTER 4. GLIB UTILITIES

4.1. STRING UTILITY FUNCTIONS

**format**: a standard printf() format string, but notice string precision pitfalls.

...: the arguments to insert in the output.

**Returns**: the number of bytes printed.

Since 2.2

**g_vsprintf ()**

| gint         | g_vsprintf            | (gchar *string, gchar const *format, va_list args); |

An implementation of the standard vsprintf() function which supports positional parameters, as specified in the Single Unix Specification.

**string**: the buffer to hold the output.

**format**: a standard printf() format string, but notice string precision pitfalls.

**args**: the list of arguments to insert in the output.

**Returns**: the number of bytes printed.

Since 2.2

**g_snprintf ()**

| gint         | g_snprintf            | (gchar *string, gulong n, gchar const *format, ...); |

A safer form of the standard sprintf() function. The output is guaranteed to not exceed \( n \) characters (including the terminating nul character), so it is easy to ensure that a buffer overflow cannot occur.

See also g_strdup_printf().

In versions of GLib prior to 1.2.3, this function may return -1 if the output was truncated, and the truncated string may not be nul-terminated. In versions prior to 1.3.12, this function returns the length of the output string.

The return value of g_snprintf() conforms to the snprintf() function as standardized in ISO C99. Note that this is different from traditional snprintf(), which returns the length of the output string.

The format string may contain positional parameters, as specified in the Single Unix Specification.

**string**: the buffer to hold the output.

**n**: the maximum number of bytes to produce (including the terminating nul character).

**format**: a standard printf() format string, but notice string precision pitfalls.

...: the arguments to insert in the output.

**Returns**: the number of bytes which would be produced if the buffer was large enough.

**g_vsnprintf ()**

| gint         | g_vsnprintf            | (gchar *string, gulong n, gchar const *format, va_list args); |
A safer form of the standard vsprintf() function. The output is guaranteed to not exceed \( n \) characters (including the terminating null character), so it is easy to ensure that a buffer overflow cannot occur.

See also g_strdup_vprintf().

In versions of GLib prior to 1.2.3, this function may return -1 if the output was truncated, and the truncated string may not be null-terminated. In versions prior to 1.3.12, this function returns the length of the output string.

The return value of g_vsnprintf() conforms to the vsnprintf() function as standardized in ISO C99. Note that this is different from traditional vsnprintf(), which returns the length of the output string.

The format string may contain positional parameters, as specified in the Single Unix Specification.

**string**: the buffer to hold the output.

**n**: the maximum number of bytes to produce (including the terminating null character).

**format**: a standard printf() format string, but notice string precision pitfalls.

**args**: the list of arguments to insert in the output.

**Returns**: the number of bytes which would be produced if the buffer was large enough.

### g_vasprintf()

```c
#include <glib.h>

gint g_vasprintf (gchar **string,
                 const gchar *format,
                 va_list args);
```

An implementation of the GNU vasprintf() function which supports positional parameters, as specified in the Single Unix Specification. This function is similar to g_vsprintf(), except that it allocates a string to hold the output, instead of putting the output in a buffer you allocate in advance.

**string**: the return location for the newly-allocated string.

**format**: a standard printf() format string, but notice string precision pitfalls.

**args**: the list of arguments to insert in the output.

**Returns**: the number of bytes printed.

Since 2.4

### g_printf_string_upper_bound()

```c
#include <glib.h>

gsize g_printf_string_upper_bound (const gchar *format,
                                   va_list args);
```

Calculates the maximum space needed to store the output of the sprintf() function.

**format**: the format string. See the printf() documentation.

**args**: the parameters to be inserted into the format string.

**Returns**: the maximum space needed to store the formatted string.

### g_ascii_isalnum()

```c
#include <glib.h>

gboolean g_ascii_isalnum (gchar c);
```

Determines whether a character is alphanumeric.

Unlike the standard C library isalnum() function, this only recognizes standard ASCII letters and ignores the locale, returning FALSE for all non-ASCII characters. Also unlike the standard library function, this takes a char, not an int, so don’t call it on EOF but no need to cast to gchar before passing a possibly non-ASCII character in.

**c**: any character

**Returns**: %TRUE if \( c \) is an ASCII alphanumeric character
CHAPTER 4. GLIB UTILITIES

4.1. STRING UTILITY FUNCTIONS

`g_ascii_isalpha()`

```c
gboolean g_ascii_isalpha (gchar c);
```

Determines whether a character is alphabetic (i.e. a letter).

Unlike the standard C library `isalpha()` function, this only recognizes standard ASCII letters and ignores the locale, returning `FALSE` for all non-ASCII characters. Also unlike the standard library function, this takes a char, not an int, so don’t call it on `EOF` but no need to cast to `guchar` before passing a possibly non-ASCII character in.

`c`: any character

**Returns**: `%TRUE` if `c` is an ASCII alphabetic character

`g_ascii_iscntrl()`

```c
gboolean g_ascii_iscntrl (gchar c);
```

Determines whether a character is a control character.

Unlike the standard C library `iscntrl()` function, this only recognizes standard ASCII control characters and ignores the locale, returning `FALSE` for all non-ASCII characters. Also unlike the standard library function, this takes a char, not an int, so don’t call it on `EOF` but no need to cast to `guchar` before passing a possibly non-ASCII character in.

`c`: any character

**Returns**: `%TRUE` if `c` is an ASCII control character.

`g_ascii_isdigit()`

```c
gboolean g_ascii_isdigit (gchar c);
```

Determines whether a character is digit (0-9).

Unlike the standard C library `isdigit()` function, this takes a char, not an int, so don’t call it on `EOF` but no need to cast to `guchar` before passing a possibly non-ASCII character in.

`c`: any character

**Returns**: `%TRUE` if `c` is an ASCII digit.

`g_ascii_isgraph()`

```c
gboolean g_ascii_isgraph (gchar c);
```

Determines whether a character is a printing character and not a space.

Unlike the standard C library `isgraph()` function, this only recognizes standard ASCII characters and ignores the locale, returning `FALSE` for all non-ASCII characters. Also unlike the standard library function, this takes a char, not an int, so don’t call it on `EOF` but no need to cast to `guchar` before passing a possibly non-ASCII character in.

`c`: any character

**Returns**: `%TRUE` if `c` is an ASCII printing character other than space.
CHAPTER 4. GLIB UTILITIES 4.1. STRING UTILITY FUNCTIONS

\textbf{g_ascii_islower ()}

\begin{verbatim}
 gboolean g_ascii_islower (gchar c);
\end{verbatim}

Determines whether a character is an ASCII lower case letter. Unlike the standard C library islower() function, this only recognizes standard ASCII letters and ignores the locale, returning \texttt{FALSE} for all non-ASCII characters. Also unlike the standard library function, this takes a char, not an int, so don’t call it on EOF but no need to worry about casting to \texttt{guchar} before passing a possibly non-ASCII character in.

\texttt{c}: any character

\textbf{Returns}: \%TRUE if \texttt{c} is an ASCII lower case letter

\textbf{g_ascii_isprint ()}

\begin{verbatim}
 gboolean g_ascii_isprint (gchar c);
\end{verbatim}

Determines whether a character is a printing character. Unlike the standard C library isprint() function, this only recognizes standard ASCII characters and ignores the locale, returning \texttt{FALSE} for all non-ASCII characters. Also unlike the standard library function, this takes a char, not an int, so don’t call it on EOF but no need to cast to \texttt{guchar} before passing a possibly non-ASCII character in.

\texttt{c}: any character

\textbf{Returns}: \%TRUE if \texttt{c} is an ASCII printing character.

\textbf{g_ascii_ispunct ()}

\begin{verbatim}
 gboolean g_ascii_ispunct (gchar c);
\end{verbatim}

Determines whether a character is a punctuation character. Unlike the standard C library ispunct() function, this only recognizes standard ASCII letters and ignores the locale, returning \texttt{FALSE} for all non-ASCII characters. Also unlike the standard library function, this takes a char, not an int, so don’t call it on EOF but no need to cast to \texttt{guchar} before passing a possibly non-ASCII character in.

\texttt{c}: any character

\textbf{Returns}: \%TRUE if \texttt{c} is an ASCII punctuation character.

\textbf{g_ascii_isspace ()}

\begin{verbatim}
 gboolean g_ascii_isspace (gchar c);
\end{verbatim}

Determines whether a character is a white-space character. Unlike the standard C library isspace() function, this only recognizes standard ASCII white-space and ignores the locale, returning \texttt{FALSE} for all non-ASCII characters. Also unlike the standard library function, this takes a char, not an int, so don’t call it on EOF but no need to cast to \texttt{guchar} before passing a possibly non-ASCII character in.

\texttt{c}: any character

\textbf{Returns}: \%TRUE if \texttt{c} is an ASCII white-space character. 

201
CHAPTER 4. GLIB UTILITIES

4.1. STRING UTILITY FUNCTIONS

g_ascii_isupper ()

gboolean g_ascii_isupper (gchar c);  

Determines whether a character is an ASCII upper case letter. 
Unlike the standard C library isupper() function, this only recognizes standard ASCII letters and ignores the locale, returning FALSE for all non-ASCII characters. Also unlike the standard library function, this takes a char, not an int, so don’t call it on EOF but no need to worry about casting to guchar before passing a possibly non-ASCII character in.

\(c\): any character 

*Returns*: %TRUE if \(c\) is an ASCII upper case letter

g_ascii_isxdigit ()

gboolean g_ascii_isxdigit (gchar c);  

Determines whether a character is a hexadecimal-digit character. 
Unlike the standard C library isxdigit() function, this takes a char, not an int, so don’t call it on EOF but no need to cast to guchar before passing a possibly non-ASCII character in.

\(c\): any character 

*Returns*: %TRUE if \(c\) is an ASCII hexadecimal-digit character.

g_ascii_digit_value ()

gint g_ascii_digit_value (gchar c);  

Determines the numeric value of a character as a decimal digit. Differs from g_unichar_digit_value() because it takes a char, so there’s no worry about sign extension if characters are signed.

\(c\): an ASCII character. 

*Returns*: If \(c\) is a decimal digit (according to g_ascii_isdigit()), its numeric value. Otherwise, -1.

g_ascii_xdigit_value ()

gint g_ascii_xdigit_value (gchar c);  

Determines the numeric value of a character as a hexadecimal digit. Differs from g_unichar_xdigit_value() because it takes a char, so there’s no worry about sign extension if characters are signed.

\(c\): an ASCII character. 

*Returns*: If \(c\) is a hex digit (according to g_ascii_isxdigit()), its numeric value. Otherwise, -1.

g_ascii_strcasecmp ()

gint g_ascii_strcasecmp (const gchar *s1, const gchar *s2);  

Compare two strings, ignoring the case of ASCII characters. 
Unlike the BSD strcasecmp() function, this only recognizes standard ASCII letters and ignores the locale, treating all non-ASCII bytes as if they are not letters. 
This function should be used only on strings that are known to be in encodings where the bytes corresponding to ASCII letters always represent themselves. This includes UTF-8 and the ISO-8859-* charsets, but not for instance double-byte encodings like the Windows Codepage 932, where the trailing bytes of double-byte characters include all ASCII letters. If you compare two CP932 strings using this function, you will get false matches.
**s1**: string to compare with \( s2 \).

**s2**: string to compare with \( s1 \).

**Returns**: 0 if the strings match, a negative value if \( s1 < s2 \), or a positive value if \( s1 > s2 \).

### g_ascii_strncasecmp()

\[
gint \ g_ascii_strncasecmp \ (\text{const} \ \text{gchar} \ *s1, \ \\
 \text{const} \ \text{gchar} \ *s2, \ \\
gsize \ n);
\]

Compare \( s1 \) and \( s2 \), ignoring the case of ASCII characters and any characters after the first \( n \) in each string.

Unlike the BSD `strcasecmp()` function, this only recognizes standard ASCII letters and ignores the locale, treating all non-ASCII characters as if they are not letters.

The same warning as in `g_ascii_strcasecmp()` applies: Use this function only on strings known to be in encodings where bytes corresponding to ASCII letters always represent themselves.

**s1**: string to compare with \( s2 \).

**s2**: string to compare with \( s1 \).

**n**: number of characters to compare.

**Returns**: 0 if the strings match, a negative value if \( s1 < s2 \), or a positive value if \( s1 > s2 \).

### g_ascii_strup()

\[
gchar* \ g_ascii_strup \ (\text{const} \ \text{gchar} \ *str, \ \\
gssize \ len);
\]

Converts all lower case ASCII letters to upper case ASCII letters.

**str**: a string.

**len**: length of \( str \) in bytes, or -1 if \( str \) is nul-terminated.

**Returns**: a newly allocated string, with all the lower case characters in \( str \) converted to upper case, with semantics that exactly match `g_ascii_toupper()`. (Note that this is unlike the old `gstrup()`, which modified the string in place.)

### g_ascii_strdown()

\[
gchar* \ g_ascii_strdown \ (\text{const} \ \text{gchar} \ *str, \ \\
gssize \ len);
\]

Converts all upper case ASCII letters to lower case ASCII letters.

**str**: a string.

**len**: length of \( str \) in bytes, or -1 if \( str \) is nul-terminated.

**Returns**: a newly-allocated string, with all the upper case characters in \( str \) converted to lower case, with semantics that exactly match `g_ascii_tolower()`. (Note that this is unlike the old `gstrdown()`, which modified the string in place.)
CHAPTER 4. GLIB UTILITIES

4.1. STRING UTILITY FUNCTIONS

**g_ascii_tolower()**

```
gchar g_ascii_tolower (gchar c);
```

Convert a character to ASCII lower case.

Unlike the standard C library `tolower()` function, this only recognizes standard ASCII letters and ignores the locale, returning all non-ASCII characters unchanged, even if they are lower case letters in a particular character set. Also unlike the standard library function, this takes and returns a char, not an int, so don’t call it on EOF but no need to worry about casting to `guchar` before passing a possibly non-ASCII character in.

`c`: any character.

**Returns**: the result of converting `c` to lower case. If `c` is not an ASCII upper case letter, `c` is returned unchanged.

**g_ascii_toupper()**

```
gchar g_ascii_toupper (gchar c);
```

Convert a character to ASCII upper case.

Unlike the standard C library `toupper()` function, this only recognizes standard ASCII letters and ignores the locale, returning all non-ASCII characters unchanged, even if they are upper case letters in a particular character set. Also unlike the standard library function, this takes and returns a char, not an int, so don’t call it on EOF but no need to worry about casting to `guchar` before passing a possibly non-ASCII character in.

`c`: any character.

**Returns**: the result of converting `c` to upper case. If `c` is not an ASCII lower case letter, `c` is returned unchanged.

**g_string_ascii_up()**

```
GString* g_string_ascii_up (GString *string);
```

Converts all lower case ASCII letters to upper case ASCII letters.

`string`: a GString

**Returns**: passed-in `string` pointer, with all the lower case characters converted to upper case in place, with semantics that exactly match `g_ascii_toupper()`.

**g_string_ascii_down()**

```
GString* g_string_ascii_down (GString *string);
```

Converts all upper case ASCII letters to lower case ASCII letters.

`string`: a GString

**Returns**: passed-in `string` pointer, with all the upper case characters converted to lower case in place, with semantics that exactly match `g_ascii_tolower()`.
4.1. STRING UTILITY FUNCTIONS

**g_strup ()**

```c
 gchar* g_strup (gchar *string);
```

**Warning**

*g_strup* has been deprecated since version 2.2 and should not be used in newly-written code. This function is totally broken for the reasons discussed in the *g_strncasecmp()* docs - use *g_ascii_strup()* or *g_utf8_strup()* instead.

Converts a string to upper case.

*string*: the string to convert.

*Returns*: the string

**g_strdown ()**

```c
 gchar* g_strdown (gchar *string);
```

**Warning**

*g_strdown* has been deprecated since version 2.2 and should not be used in newly-written code. This function is totally broken for the reasons discussed in the *g_strncasecmp()* docs - use *g_ascii_strdown()* or *g_utf8_strdown()* instead.

Converts a string to lower case.

*string*: the string to convert.

*Returns*: the string

**g_strcasecmp ()**

```c
 gint g_strcasecmp (const gchar *s1, const gchar *s2);
```

**Warning**

*g_strcasecmp* has been deprecated since version 2.2 and should not be used in newly-written code. See *g_strncasecmp()* for a discussion of why this function is deprecated and how to replace it.

A case-insensitive string comparison, corresponding to the standard *strcasecmp()* function on platforms which support it.

*s1*: a string.

*s2*: a string to compare with *s1*.

*Returns*: 0 if the strings match, a negative value if *s1* < *s2*, or a positive value if *s1* > *s2*. 

205
**CHAPTER 4. GLIB UTILITIES**

**4.1. STRING UTILITY FUNCTIONS**

**g_strncasecmp**

```c
qint g_strncasecmp (const gchar *s1,
const gchar *s2,
quint n);
```

**WARNING**

*g_strncasecmp* has been deprecated since version 2.2 and should not be used in newly-written code. The problem with *g_strncasecmp()* is that it does the comparison by calling `toupper()/tolower()`. These functions are locale-specific and operate on single bytes. However, it is impossible to handle things correctly from an I18N standpoint by operating on bytes, since characters may be multibyte. Thus *g_strncasecmp()* is broken if your string is guaranteed to be ASCII, since it's locale-sensitive, and it's broken if your string is localized, since it doesn't work on many encodings at all, including UTF-8, EUC-JP, etc.

There are therefore two replacement functions: *g_ascii_strncasecmp()*, which only works on ASCII and is not locale-sensitive, and *g_utf8_casefold()*, which is good for case-insensitive sorting of UTF-8.

A case-insensitive string comparison, corresponding to the standard *strncasecmp()* function on platforms which support it. It is similar to *g_strcasecmp()* except it only compares the first *n* characters of the strings.

*s1*: a string.

*s2*: a string to compare with *s1*.

*n*: the maximum number of characters to compare.

**Returns**: 0 if the strings match, a negative value if *s1* < *s2*, or a positive value if *s1* > *s2*.

**g_strreverse**

```c
 gchar* g_strreverse (gchar *string);
```

Reverses all of the bytes in a string. For example, *g_strreverse* ("abcdef") will result in "fedcba".

Note that *g_strreverse()* doesn’t work on UTF-8 strings containing multibyte characters. For that purpose, use *g_utf8_strreverse()*.

*string*: the string to reverse

**Returns**: the same pointer passed in as *string*

**g_ascii_strtoll**

```c
qint64 g_ascii_strtoll (const gchar *nptr,
const gchar **endptr,
quint base);
```

Converts a string to a *qint64* value. This function behaves like the standard *strtoll()* function does in the C locale. It does this without actually changing the current locale, since that would not be thread-safe.

This function is typically used when reading configuration files or other non-user input that should be locale independent. To handle input from the user you should normally use the locale-sensitive system *strtoll()* function.

206
CHAPTER 4. GLIB UTILITIES

4.1. STRING UTILITY FUNCTIONS

If the correct value would cause overflow, G_MAXINT64 or G_MININT64 is returned, and ERANGE
is stored in errno. If the base is outside the valid range, zero is returned, and EINVAL is stored in errno.
If the string conversion fails, zero is returned, and endptr returns nptr (if endptr is non-NULL).

nptr: the string to convert to a numeric value.
endptr: if non-NULL, it returns the character after the last character used in the conversion.
base: to be used for the conversion, 2..36 or 0

Returns: the gint64 value or zero on error.

Since 2.12

\textbf{g_ascii_strtoull ()}

\begin{verbatim}
guint64 g_ascii_strtoull (const gchar *nptr, gchar **endptr, guint base);
\end{verbatim}

Converts a string to a guint64 value. This function behaves like the standard strtoull() function does
in the C locale. It does this without actually changing the current locale, since that would not be thread-
safe.

This function is typically used when reading configuration files or other non-user input that should
be locale independent. To handle input from the user you should normally use the locale-sensitive
system strtoull() function.

If the correct value would cause overflow, G_MAXUINT64 is returned, and ERANGE is stored in
errno. If the base is outside the valid range, zero is returned, and EINVAL is stored in errno. If the string
conversion fails, zero is returned, and endptr returns nptr (if endptr is non-NULL).

nptr: the string to convert to a numeric value.
endptr: if non-NULL, it returns the character after the last character used in the conversion.
base: to be used for the conversion, 2..36 or 0

Returns: the guint64 value or zero on error.

Since 2.2

\textbf{G_ASCII_DTOSTR_BUF_SIZE}

\begin{verbatim}
#define G_ASCII_DTOSTR_BUF_SIZE (29 + 10)
\end{verbatim}

A good size for a buffer to be passed into g_ascii_dtostr(). It is guaranteed to be enough for all output
of that function on systems with 64bit IEEE-compatible doubles.

The typical usage would be something like:

\begin{verbatim}
char buf[G_ASCII_DTOSTR_BUF_SIZE];
fprintf (out, "value=%s\n", g_ascii_dtostr (buf, sizeof (buf), value));
\end{verbatim}

\textbf{g_ascii_strtod ()}

\begin{verbatim}
gdouble g_ascii_strtod (const gchar *nptr, gchar **endptr);
\end{verbatim}

Converts a string to a gdouble value.

This function behaves like the standard strtod() function does in the C locale. It does this without
actually changing the current locale, since that would not be thread-safe. A limitation of the implementa-
tion is that this function will still accept localized versions of infinities and NANs.

This function is typically used when reading configuration files or other non-user input that should
be locale independent. To handle input from the user you should normally use the locale-sensitive
system strtod() function.

\textbf{Since 2.12}\n
To convert from a `gdouble` to a string in a locale-insensitive way, use `g_ascii_dtostr()`.

If the correct value would cause overflow, plus or minus `HUGE_VAL` is returned (according to the
sign of the value), and `ERANGE` is stored in `errno`. If the correct value would cause underflow, zero is
returned and `ERANGE` is stored in `errno`.

This function resets `errno` before calling `strtod()` so that you can reliably detect overflow and under-
flow.

`nptr` : the string to convert to a numeric value.

`endptr` : if non-NULL, it returns the character after the last character used in the conversion.

**Returns** : the `gdouble` value.

### `g_ascii_dtostr()`

```c
gchar * g_ascii_dtostr (gchar *buffer, gint buf_len, gdouble d);
```

Converts a `gdouble` to a string, using the ‘.’ as decimal point.

This function generates enough precision that converting the string back using `g_ascii_strtod()` gives
the same machine-number (on machines with IEEE compatible 64bit doubles). It is guaranteed that the
size of the resulting string will never be larger than `G_ASCII_DTOSTR_BUF_SIZE` bytes.

`buffer` : A buffer to place the resulting string in

`buf_len` : The length of the buffer.

`d` : The `gdouble` to convert

**Returns** : The pointer to the buffer with the converted string.

### `g_ascii_formatd()`

```c
gchar * g_ascii_formatd (gchar *buffer, gint buf_len, const gchar *format, gdouble d);
```

Converts a `gdouble` to a string, using the ‘.’ as decimal point. To format the number you pass in a

If you just want to want to serialize the value into a string, use `g_ascii_dtostr()`.

`buffer` : A buffer to place the resulting string in

`buf_len` : The length of the buffer.

`format` : The printf()-style format to use for the code to use for converting.

`d` : The `gdouble` to convert

**Returns** : The pointer to the buffer with the converted string.

### `g_strtod()`

```c
gdouble g_strtod (const gchar *nptr, gchar **endptr);
```

Converts a string to a `gdouble` value. It calls the standard `strtod()` function to handle the conversion,
but if the string is not completely converted it attempts the conversion again with `g_ascii_strtod()`, and
returns the best match.

This function should seldomly be used. The normal situation when reading numbers not for hu-
man consumption is to use `g_ascii_strtod()`. Only when you know that you must expect both locale
formatted and C formatted numbers should you use this. Make sure that you don’t pass strings such
as comma separated lists of values, since the commas may be interpreted as a decimal point in some
locales, causing unexpected results.
**nptr**: the string to convert to a numeric value.

**endptr**: if non-NULL, it returns the character after the last character used in the conversion.

**Returns**: the gdouble value.

**g_strchug**

```c
char* g_strchug (char *string);
```

Removes leading whitespace from a string, by moving the rest of the characters forward. This function doesn’t allocate or reallocate any memory; it modifies *string in place. The pointer to *string is returned to allow the nesting of functions. Also see g_strchomp() and g_strstrip().

**string**: a string to remove the leading whitespace from.

**Returns**: @string.

**g_strchomp**

```c
char* g_strchomp (char *string);
```

Removes trailing whitespace from a string. This function doesn’t allocate or reallocate any memory; it modifies *string in place. The pointer to *string is returned to allow the nesting of functions. Also see g_strchug() and g_strstrip().

**string**: a string to remove the trailing whitespace from.

**Returns**: @string.

**g_strstrip**

```c
#define g_strstrip( string )
```

Removes leading and trailing whitespace from a string. See g_strchomp() and g_strchug().

**string**: a string to remove the leading and trailing whitespace from.

**g_strdelimit**

```c
char* g_strdelimit (char *string, const char *delimiters, char new_delimiter);
```

Converts any delimiter characters in *string to new_delimiter. Any characters in *string which are found in delimiters are changed to the new_delimiter character. Modifies *string in place, and returns *string itself, not a copy. The return value is to allow nesting such as g_ascii_strup (g_strdelimit (str, "abc", '?')).

**string**: the string to convert.

**delimiters**: a string containing the current delimiters, or NULL to use the standard delimiters defined in G_STR_DELIMITERS.

**new_delimiter**: the new delimiter character.

**Returns**: @string.
CHAPTER 4. GLIB UTILITIES 4.1. STRING UTILITY FUNCTIONS

G_STR_DELIMITERS

#define G_STR_DELIMITERS "-_|> <."

The standard delimiters, used in g_strdelimit().

\gchar* g_strescape (const gchar *source, const gchar *exceptions);

Escapes the special characters ‘\b’, ‘\f’, ‘\n’, ‘\r’, ‘\t’, ‘\’ and ‘”’ in the string source by inserting a ‘\’ before them. Additionally all characters in the range 0x01-0x1F (everything below SPACE) and in the range 0x7F-0xFF (all non-ASCII chars) are replaced with a ‘\’ followed by their octal representation. Characters supplied in exceptions are not escaped.

\g_strcompress() does the reverse conversion.

source: a string to escape.

exceptions: a string of characters not to escape in source.

Returns: a newly-allocated copy of source with certain characters escaped. See above.

\gchar* g_strcompress (const gchar *source);

Replaces all escaped characters with their one byte equivalent. It does the reverse conversion of g_strescape().

source: a string to compress.

Returns: a newly-allocated copy of source with all escaped character compressed.

\gchar* g_strcanon (gchar *string, const gchar *valid_chars, gchar substitutor);

For each character in string, if the character is not in valid_chars, replaces the character with substitutor. Modifies string in place, and return string itself, not a copy. The return value is to allow nesting such as g_ascii_strup (g_strcanon (str, "abc", '?')).

string: a nul-terminated array of bytes.

valid_chars: bytes permitted in string.

substitutor: replacement character for disallowed bytes.

Returns: @string.

\gchar** g_strsplit (const gchar *string, const gchar *delimiter, gint max_tokens);

210
CHAPTER 4. GLIB UTILITIES 4.1. STRING UTILITY FUNCTIONS

Splits a string into a maximum of \texttt{max\_tokens} pieces, using the given \texttt{delimiter}. If \texttt{max\_tokens} is reached, the remainder of \texttt{string} is appended to the last token.

As a special case, the result of splitting the empty string \textquote{"\texttt{\textquoteright\textquoteright}\textquoteright\textquoteright} is an empty vector, not a vector containing a single string. The reason for this special case is that being able to represent a empty vector is typically more useful than consistent handling of empty elements. If you do need to represent empty elements, you'll need to check for the empty string before calling \texttt{g\_strsplit()}.

\texttt{string}: a string to split.

\texttt{delimiter}: a string which specifies the places at which to split the string. The delimiter is not included in any of the resulting strings, unless \texttt{max\_tokens} is reached.

\texttt{max\_tokens}: the maximum number of pieces to split \texttt{string} into. If this is less than 1, the string is split completely.

Returns: a newly-allocated NULL-terminated array of strings. Use \texttt{g\_strfreev()} to free it.

\texttt{g\_strsplit\_set()}:

\begin{verbatim}
gchar ** g_strsplit_set (const gchar *string, const gchar *delimiters, gint max_tokens);
\end{verbatim}

Splits \texttt{string} into a number of tokens not containing any of the characters in \texttt{delimiter}. A token is the (possibly empty) longest string that does not contain any of the characters in \texttt{delimiters}. If \texttt{max\_tokens} is reached, the remainder is appended to the last token.

For example the result of \texttt{g\_strsplit\_set(\textquoteright abc\textquoteright :\textquoteright def\textquoteright /\textquoteright ghi\textquoteright ,\textquoteright :\textquoteright /\textquoteright ,-1)} is a NULL-terminated vector containing the three strings \textquote{\texttt{\textquoteright abc\textquoteright }}, \textquote{\texttt{\textquoteright def\textquoteright }}, and \textquote{\texttt{\textquoteright ghi\textquoteright }}.

The result if \texttt{g\_strsplit\_set(\textquoteright :\textquoteright def\textquoteright /\textquoteright ghi\textquoteright :,\textquoteright :\textquoteright /\textquoteright ,-1)} is a NULL-terminated vector containing the four strings \textquote{\textquoteright \textquoteright}, \textquote{\texttt{\textquoteright def\textquoteright }}, \textquote{\texttt{\textquoteright ghi\textquoteright }}, and \textquote{\textquoteright \textquoteright}.

As a special case, the result of splitting the empty string \textquote{\textquoteright \textquoteright} is an empty vector, not a vector containing a single string. The reason for this special case is that being able to represent a empty vector is typically more useful than consistent handling of empty elements. If you do need to represent empty elements, you'll need to check for the empty string before calling \texttt{g\_strsplit\_set()}.

Note that this function works on bytes not characters, so it can’t be used to delimit UTF-8 strings for anything but ASCII characters.

\texttt{string}: The string to be tokenized

\texttt{delimiters}: A nul-terminated string containing bytes that are used to split the string.

\texttt{max\_tokens}: The maximum number of tokens to split \texttt{string} into. If this is less than 1, the string is split completely.

Returns: a newly-allocated NULL-terminated array of strings. Use \texttt{g\_strfreev()} to free it.

Since 2.4

\texttt{g\_strfreev()}:

\begin{verbatim}
void g_strfreev (gchar **str_array);
\end{verbatim}

Frees a NULL-terminated array of strings, and the array itself. If called on a NULL value, \texttt{g\_strfreev()} simply returns.

\texttt{str\_array}: a NULL-terminated array of strings to free.
CHAPTER 4. GLIB UTILITIES

4.1. STRING UTILITY FUNCTIONS

`g_strconcat()`

```c
char* g_strconcat (const gchar *string1, ...
```

Concatenates all of the given strings into one long string. The returned string should be freed with `g_free()` when no longer needed.

**WARNING**

The variable argument list must end with `NULL`. If you forget the `NULL`, `g_strconcat()` will start appending random memory junk to your string.

`string1`: the first string to add, which must not be `NULL`

...: a `NULL`-terminated list of strings to append to the string

**Returns**: a newly-allocated string containing all the string arguments

`g_strjoin()`

```c
char* g_strjoin (const gchar *separator, ...
```

Joins a number of strings together to form one long string, with the optional `separator` inserted between each of them. The returned string should be freed with `g_free()`.

`separator`: a string to insert between each of the strings, or `NULL`

...: a `NULL`-terminated list of strings to join

**Returns**: a newly-allocated string containing all the strings joined together, with `separator` between them

`g_strjoinv()`

```c
char* g_strjoinv (const gchar *separator, gchar **str_array);
```

Joins a number of strings together to form one long string, with the optional `separator` inserted between each of them. The returned string should be freed with `g_free()`.

`separator`: a string to insert between each of the strings, or `NULL`

`str_array`: a `NULL`-terminated array of strings to join

**Returns**: a newly-allocated string containing all of the strings joined together, with `separator` between them

`g_strv_length()`

```c
guint g_strv_length (gchar **str_array);
```

Returns the length of the given `NULL`-terminated string array `str_array`.

`str_array`: a `NULL`-terminated array of strings.

**Returns**: length of `str_array`.

Since 2.6
CHAPTER 4. GLIB UTILITIES 4.2. CHARACTER SET CONVERSION

**g_strerror()**

```c
const gchar* g_strerror (gint errnum);
```

Returns a string corresponding to the given error code, e.g. "no such process". You should use this function in preference to `strerror()`, because it returns a string in UTF-8 encoding, and since not all platforms support the `strerror()` function.

**errnum**: the system error number. See the standard C `errno` documentation

**Returns**: a UTF-8 string describing the error code. If the error code is unknown, it returns "unknown error (<code>)". The string can only be used until the next call to `g_strerror()`

**g_strsignal()**

```c
const gchar* g_strsignal (gint signum);
```

Returns a string describing the given signal, e.g. "Segmentation fault". You should use this function in preference to `strsignal()`, because it returns a string in UTF-8 encoding, and since not all platforms support the `strsignal()` function.

**signum**: the signal number. See the `signal` documentation

**Returns**: a UTF-8 string describing the signal. If the signal is unknown, it returns "unknown signal (<signum>)". The string can only be used until the next call to `g_strsignal()`

### 4.2 Character Set Conversion

**Name**

Character Set Conversion – convert strings between different character sets using `iconv()`

**Synopsis**

```c
#include <glib.h>

gchar* g_convert (const gchar *str, gssize len, const gchar *to_codeset, const gchar *from_codeset, gsize *bytes_read, gsize *bytes_written, GError **error);

gchar* g_convert_with_fallback (const gchar *str, gssize len, const gchar *to_codeset, const gchar *from_codeset, gchar *fallback, gsize *bytes_read, gsize *bytes_written, GError **error);

gchar* g_convert_with_iconv (const gchar *str, gssize len, GIConv converter, gsize *bytes_read, gsize *bytes_written, GError **error);

#define G_CONVERT_ERROR
```
CHAPTER 4. GLIB UTILITIES

4.2. CHARACTER SET CONVERSION

GIConv  g_iconv_open (const gchar *to_codeset,
                      const gchar *from_codeset);

gsize  g_iconv (GIConv converter,
                gchar **inbuf,
                gsize *inbytes_left,
                gchar **outbuf,
                gsize *outbytes_left);

gint  g_iconv_close (GIConv converter);

gchar*  g_locale_to_utf8 (const gchar *opsysstring,
                          gssize len,
                          gsize *bytes_read,
                          gsize *bytes_written,
                          GError **error);

gchar*  g_filename_to_utf8
                          (const gchar *opsysstring,
                          gssize len,
                          gsize *bytes_read,
                          gsize *bytes_written,
                          GError **error);

gchar*  g_filename_from_utf8
                          (const gchar *utf8string,
                          gssize len,
                          gsize *bytes_read,
                          gsize *bytes_written,
                          GError **error);

gchar*  g_filename_from_uri
                          (const gchar *uri,
                          gchar **hostname,
                          GError **error);

gchar*  g_filename_to_uri
                          (const gchar *filename,
                          const gchar *hostname,
                          GError **error);

gboolean  g_get_filename_charsets
                          (G_CONST_RETURN gchar ***charsets);

gchar*  g_filename_display_name
                          (const gchar *filename);

gchar*  g_filename_display_basename
                          (const gchar *filename);

gchar**  g_uri_list_extract_uris
                          (const gchar *uri_list);

gchar*  g_locale_from_utf8
                          (const gchar *utf8string,
                          gssize len,
                          gsize *bytes_read,
                          gsize *bytes_written,
                          GError **error);

eenum  GConvertError;

gboolean  g_get_charset
                          (G_CONST_RETURN char **charset);

Description

File Name Encodings

Historically, Unix has not had a defined encoding for file names: a file name is valid as long as it does not have path separators in it ("/"). However, displaying file names may require conversion: from the character set in which they were created, to the character set in which the application operates. Consider the Spanish file name "Presentación.sxi". If the application which created it uses ISO-8859-1 for its encoding, then the actual file name on disk would look like this:

| Character: | Presentación.sxi |
| Hex code:  | 50 72 65 73 65 6e 74 61 63 69 f3 6e 2e 73 78 69 |

However, if the application use UTF-8, the actual file name on disk would look like this:

| Character: | Presentación.sxi |
| Hex code:  | 50 72 65 73 65 6e 74 61 63 69 c3 b3 6e 2e 73 78 69 |
Glib uses UTF-8 for its strings, and GUI toolkits like GTK+ that use Glib do the same thing. If you get a file name from the file system, for example, from `readdir(3)` or from `g_dir_read_name()`, and you wish to display the file name to the user, you will need to convert it into UTF-8. The opposite case is when the user types the name of a file he wishes to save: the toolkit will give you that string in UTF-8 encoding, and you will need to convert it to the character set used for file names before you can create the file with `open(2)` or `fopen(3)`.

By default, Glib assumes that file names on disk are in UTF-8 encoding. This is a valid assumption for file systems which were created relatively recently: most applications use UTF-8 encoding for their strings, and that is also what they use for the file names they create. However, older file systems may still contain file names created in “older” encodings, such as ISO-8859-1. In this case, for compatibility reasons, you may want to instruct Glib to use that particular encoding for file names rather than UTF-8. You can do this by specifying the encoding for file names in the `G_FILENAME_ENCODING` environment variable. For example, if your installation uses ISO-8859-1 for file names, you can put this in your `~/.profile`:

```bash
export G_FILENAME_ENCODING=ISO-8859-1
```

Glib provides the functions `g_filename_to_utf8()` and `g_filename_from_utf8()` to perform the necessary conversions. These functions convert file names from the encoding specified in `G_FILENAME_ENCODING` to UTF-8 and vice-versa. Figure 4.1 illustrates how these functions are used to convert between UTF-8 and the encoding for file names in the file system.

![Figure 4.1 Conversion between File Name Encodings](file-name-encodings.png)

**Checklist for Application Writers**  
This section is a practical summary of the detailed description above. You can use this as a checklist of things to do to make sure your applications process file name encodings correctly.

1. If you get a file name from the file system from a function such as `readdir(3)` or `gtk_file_chooser_get_filename()`, you do not need to do any conversion to pass that file name to functions like `open(2)`, `rename(2)`, or `fopen(3)` — those are “raw” file names which the file system understands.

2. If you need to display a file name, convert it to UTF-8 first by using `g_filename_to_utf8()`. If conversion fails, display a string like “Unknown file name”. Do not convert this string back into the encoding used for file names if you wish to pass it to the file system; use the original file name instead. For example, the document window of a word processor could display “Unknown file name” in its title bar but still let the user save the file, as it would keep the raw file name internally. This can happen if the user has not set the `G_FILENAME_ENCODING` environment variable even though he has files whose names are not encoded in UTF-8.

3. If your user interface lets the user type a file name for saving or renaming, convert it to the encoding used for file names in the file system by using `g_filename_from_utf8()`. Pass the converted file name to functions like `fopen(3)`. If conversion fails, ask the user to enter a different file name. This can happen if the user types Japanese characters when `G_FILENAME_ENCODING` is set to ISO-8859-1, for example.

**Details**

`g_convert()`

```c
 gchar* g_convert (const gchar *str, gssize len, const gchar *to_codeset, const gchar *from_codeset, gsize *bytes_read, gsize *bytes_written, GError **error);
```
CHAPTER 4. GLIB UTILITIES

4.2. CHARACTER SET CONVERSION

Converts a string from one character set to another. Note that you should use g_iconv() for streaming conversions².

**str**: the string to convert

**len**: the length of the string, or -1 if the string is nul-terminated¹.

**to_codeset**: name of character set into which to convert **str**

**from_codeset**: character set of **str**.

**bytes_read**: location to store the number of bytes in the input string that were successfully converted, or NULL. Even if the conversion was successful, this may be less than **len** if there were partial characters at the end of the input. If the error G_CONVERT_ERROR_ILLEGAL_SEQUENCE occurs, the value stored will the byte offset after the last valid input sequence.

**bytes_written**: the number of bytes stored in the output buffer (not including the terminating nul).

**error**: location to store the error occuring, or NULL to ignore errors. Any of the errors in GConvertError may occur.

**Returns**: If the conversion was successful, a newly allocated nul-terminated string, which must be freed with g_free(). Otherwise NULL and **error** will be set.

**g_convert_with_fallback()**

```c
 gchar* g_convert_with_fallback (const gchar *str, gssize len, const gchar *to_codeset, const gchar *from_codeset, gchar *fallback, gsize *bytes_read, gsize *bytes_written, GError **error);
```

Converts a string from one character set to another, possibly including fallback sequences for characters not representable in the output. Note that it is not guaranteed that the specification for the fallback sequences in **fallback** will be honored. Some systems may do an approximate conversion from **from_codeset** to **to_codeset** in their iconv() functions, in which case GLib will simply return that approximate conversion.

Note that you should use g_iconv() for streaming conversions².

**str**: the string to convert

**len**: the length of the string, or -1 if the string is nul-terminated¹.

**to_codeset**: name of character set into which to convert **str**

**from_codeset**: character set of **str**.

**fallback**: UTF-8 string to use in place of character not present in the target encoding. (The string must be representable in the target encoding). If NULL, characters not in the target encoding will be represented as Unicode escapes \uxxxx or \Uxxyyyy.

**bytes_read**: location to store the number of bytes in the input string that were successfully converted, or NULL. Even if the conversion was successful, this may be less than **len** if there were partial characters at the end of the input.

**bytes_written**: the number of bytes stored in the output buffer (not including the terminating nul).

**error**: location to store the error occuring, or NULL to ignore errors. Any of the errors in GConvertError may occur.

**Returns**: If the conversion was successful, a newly allocated nul-terminated string, which must be freed with g_free(). Otherwise NULL and **error** will be set.

¹ Note that some encodings may allow nul bytes to occur inside strings. In that case, using -1 for the **len** parameter is unsafe.
GIConv

typedef struct _GIConv GIConv;

The GIConv struct wraps an iconv() conversion descriptor. It contains private data and should only be accessed using the following functions.

**g_convert_with_iconv()**

```c
 gchar* g_convert_with_iconv(const gchar *str, gssize len, GIConv converter, gsize *bytes_read, gsize *bytes_written, GError **error);
```

Converts a string from one character set to another. Note that you should use g_iconv() for streaming conversions².

- **str**: the string to convert
- **len**: the length of the string, or -1 if the string is nul-terminated¹.
- **converter**: conversion descriptor from g_iconv_open()
- **bytes_read**: location to store the number of bytes in the input string that were successfully converted, or NULL. Even if the conversion was successful, this may be less than len if there were partial characters at the end of the input. If the error G_CONVERT_ERROR_ILLEGAL_SEQUENCE occurs, the value stored will be the byte offset after the last valid input sequence.
- **bytes_written**: the number of bytes stored in the output buffer (not including the terminating nul).
- **error**: location to store the error occurring, or NULL to ignore errors. Any of the errors in GConvertError may occur.

**Returns**: If the conversion was successful, a newly allocated nul-terminated string, which must be freed with g_free(). Otherwise NULL and error will be set.

**G_CONVERT_ERROR**

```c
#define G_CONVERT_ERROR g_convert_error_quark()
```

Error domain for character set conversions. Errors in this domain will be from the GConvertError enumeration. See GError for information on error domains.

**g_iconv_open()**

```c
 GIConv g_iconv_open(const gchar *to_codeset, const gchar *← from_codeset);
```

Same as the standard UNIX routine iconv_open(), but may be implemented via libiconv on UNIX flavors that lack a native implementation.

GLib provides g_convert() and g_locale_to_utf8() which are likely more convenient than the raw iconv wrappers.

- **to_codeset**: destination codeset

² Despite the fact that bytes_read can return information about partial characters, the g_convert_with... functions are not generally suitable for streaming. If the underlying converter being used maintains internal state, then this won’t be preserved across successive calls to g_convert(), g_convert_with_iconv() or g_convert_with_fallback(). (An example of this is the GNU C converter for CP1255 which does not emit a base character until it knows that the next character is not a mark that could combine with the base character.)
CHAPTER 4. GLIB UTILITIES

4.2. CHARACTER SET CONVERSION

from_codeset: source codeset

Returns: a "conversion descriptor", or (GIConv)-1 if opening the converter failed.

\[\text{g_iconv}()\]

\[
\begin{align*}
gsize & \quad \text{g_iconv} \quad \text{(GIConv converter,} \\
& \quad \text{gchar **inbuf,} \\
& \quad \text{gsize *inbytes_left,} \\
& \quad \text{gchar **outbuf,} \\
& \quad \text{gsize *outbytes_left)}; \\
\end{align*}
\]

Same as the standard UNIX routine iconv(), but may be implemented via libiconv on UNIX flavors that lack a native implementation.

GLib provides g_convert() and g_locale_to_utf8() which are likely more convenient than the raw iconv wrappers.

converter: conversion descriptor from g_iconv_open()

inbuf: bytes to convert

inbytes_left: inout parameter, bytes remaining to convert in inbuf

outbuf: converted output bytes

outbytes_left: inout parameter, bytes available to fill in outbuf

Returns: count of non-reversible conversions, or -1 on error

\[\text{g_iconv_close}()\]

\[
\begin{align*}
gint & \quad \text{g_iconv_close} \quad \text{(GIConv converter)}; \\
\end{align*}
\]

Same as the standard UNIX routine iconv_close(), but may be implemented via libiconv on UNIX flavors that lack a native implementation. Should be called to clean up the conversion descriptor from g_iconv_open() when you are done converting things.

GLib provides g_convert() and g_locale_to_utf8() which are likely more convenient than the raw iconv wrappers.

converter: a conversion descriptor from g_iconv_open()

Returns: -1 on error, 0 on success

\[\text{g_locale_to_utf8}()\]

\[
\begin{align*}
gchar* & \quad \text{g_locale_to_utf8} \quad \text{(const gchar *opsysstring,} \\
& \quad \text{gssize len,} \\
& \quad \text{gsize *bytes_read,} \\
& \quad \text{gsize *bytes_written,} \\
& \quad \text{GError **error);} \\
\end{align*}
\]

Converts a string which is in the encoding used for strings by the C runtime (usually the same as that used by the operating system) in the current locale into a UTF-8 string.

opsysstring: a string in the encoding of the current locale. On Windows this means the system code-page.

len: the length of the string, or -1 if the string is nul-terminated\(^1\).

bytes_read: location to store the number of bytes in the input string that were successfully converted, or NULL. Even if the conversion was successful, this may be less than len if there were partial characters at the end of the input. If the error G_CONVERT_ERROR_ILLEGAL_SEQUENCE occurs, the value stored will the byte offset after the last valid input sequence.

\(^1\)
**bytes_written**: the number of bytes stored in the output buffer (not including the terminating nul).

**error**: location to store the error occuring, or NULL to ignore errors. Any of the errors in GConvertError may occur.

**Returns**: The converted string, or NULL on an error.

**g_filename_to_utf8 ()**

```c
const gchar * g_filename_to_utf8
(gchar *opsysstring, gssize len, gsize *bytes_read, gsize *bytes_written, GError **error);
```

Converts a string which is in the encoding used by GLib for filenames into a UTF-8 string. Note that on Windows GLib uses UTF-8 for filenames; on other platforms, this function indirectly depends on the current locale.

**opsysstring**: a string in the encoding for filenames

**len**: the length of the string, or -1 if the string is nul-terminated.

**bytes_read**: location to store the number of bytes in the input string that were successfully converted, or NULL. Even if the conversion was successful, this may be less than len if there were partial characters at the end of the input. If the error G_CONVERT_ERROR_ILLEGAL_SEQUENCE occurs, the value stored will the byte offset after the last valid input sequence.

**bytes_written**: the number of bytes stored in the output buffer (not including the terminating nul).

**error**: location to store the error occuring, or NULL to ignore errors. Any of the errors in GConvertError may occur.

**Returns**: The converted string, or NULL on an error.

**g_filename_from_utf8 ()**

```c
const gchar * g_filename_from_utf8
(gchar *utf8string, gssize len, gsize *bytes_read, gsize *bytes_written, GError **error);
```

Converts a string from UTF-8 to the encoding GLib uses for filenames. Note that on Windows GLib uses UTF-8 for filenames; on other platforms, this function indirectly depends on the current locale.

**utf8string**: a UTF-8 encoded string.

**len**: the length of the string, or -1 if the string is nul-terminated.

**bytes_read**: location to store the number of bytes in the input string that were successfully converted, or NULL. Even if the conversion was successful, this may be less than len if there were partial characters at the end of the input. If the error G_CONVERT_ERROR_ILLEGAL_SEQUENCE occurs, the value stored will the byte offset after the last valid input sequence.

**bytes_written**: the number of bytes stored in the output buffer (not including the terminating nul).

**error**: location to store the error occuring, or NULL to ignore errors. Any of the errors in GConvertError may occur.

**Returns**: The converted string, or NULL on an error.
CHAPTER 4. GLIB UTILITIES 4.2. CHARACTER SET CONVERSION

**g_filename_from_uri**

```c
 gchar * g_filename_from_uri (const gchar *uri,
                             gchar **hostname,
                             GError **error);
```

Converts an escaped ASCII-encoded URI to a local filename in the encoding used for filenames.

**uri**: a uri describing a filename (escaped, encoded in ASCII).

**hostname**: Location to store hostname for the URI, or NULL. If there is no hostname in the URI, NULL will be stored in this location.

**error**: location to store the error occurring, or NULL to ignore errors. Any of the errors in GConvertError may occur.

**Returns**: a newly-allocated string holding the resulting filename, or NULL on an error.

**g_filename_to_uri**

```c
 gchar * g_filename_to_uri (const gchar *filename,
                           const gchar *hostname,
                           GError **error);
```

Converts an absolute filename to an escaped ASCII-encoded URI, with the path component following Section 3.3. of RFC 2396.

**filename**: an absolute filename specified in the GLib file name encoding, which is the on-disk file name bytes on Unix, and UTF-8 on Windows

**hostname**: A UTF-8 encoded hostname, or NULL for none.

**error**: location to store the error occurring, or NULL to ignore errors. Any of the errors in GConvertError may occur.

**Returns**: a newly-allocated string holding the resulting URI, or NULL on an error.

**g_get_filename_charsets**

```c
 gboolean g_get_filename_charsets (G_CONST_RETURN gchar ***charsets);
```

Determines the preferred character sets used for filenames. The first character set from the *charsets* is the filename encoding, the subsequent character sets are used when trying to generate a displayable representation of a filename, see `g_filename_display_name()`.

On Unix, the character sets are determined by consulting the environment variables `G_FILENAME_ENCODING` and `G_BROKEN_FILENAMES`. On Windows, the character set used in the GLib API is always UTF-8 and said environment variables have no effect.

`G_FILENAME_ENCODING` may be set to a comma-separated list of character set names. The special token "@locale" is taken to mean the character set for the current locale. If `G_FILENAME_ENCODING` is not set, but `G_BROKEN_FILENAMES` is, the character set of the current locale is taken as the filename encoding. If neither environment variable is set, UTF-8 is taken as the filename encoding, but the character set of the current locale is also put in the list of encodings.

The returned *charsets* belong to GLib and must not be freed.

Note that on Unix, regardless of the locale character set or `G_FILENAME_ENCODING` value, the actual file names present on a system might be in any random encoding or just gibberish.

**charsets**: return location for the NULL-terminated list of encoding names

**Returns**: TRUE if the filename encoding is UTF-8.

Since 2.6

220
g_filename_display_name()

```c
 gchar * g_filename_display_name (const gchar *filename);
```

Converts a filename into a valid UTF-8 string. The conversion is not necessarily reversible, so you should keep the original around and use the return value of this function only for display purposes. Unlike `g_filename_to_utf8()`, the result is guaranteed to be non-NULL even if the filename actually isn’t in the GLib file name encoding.

If GLib can not make sense of the encoding of `filename`, as a last resort it replaces unknown characters with U+FFFD, the Unicode replacement character. You can search the result for the UTF-8 encoding of this character (which is "\357\277\275" in octal notation) to find out if `filename` was in an invalid encoding.

If you know the whole pathname of the file you should use `g_filename_display_basename()`, since that allows location-based translation of filenames.

**filename**: a pathname hopefully in the GLib file name encoding

**Returns**: a newly allocated string containing a rendition of the filename in valid UTF-8

Since 2.6

goingame_display_basename()

```c
 gchar * g_filename_display_basename (const gchar *filename);
```

Returns the display basename for the particular filename, guaranteed to be valid UTF-8. The display name might not be identical to the filename, for instance there might be problems converting it to UTF-8, and some files can be translated in the display.

If GLib can not make sense of the encoding of `filename`, as a last resort it replaces unknown characters with U+FFFD, the Unicode replacement character. You can search the result for the UTF-8 encoding of this character (which is "\357\277\275" in octal notation) to find out if `filename` was in an invalid encoding.

You must pass the whole absolute pathname to this functions so that translation of well known locations can be done.

This function is preferred over `g_filename_display_name()` if you know the whole path, as it allows translation.

**filename**: an absolute pathname in the GLib file name encoding

**Returns**: a newly allocated string containing a rendition of the basename of the filename in valid UTF-8

Since 2.6

g_uri_list_extract_uris()

```c
 gchar ** g_uri_list_extract_uris (const gchar *uri_list);
```

Splits an URI list conforming to the text/uri-list mime type defined in RFC 2483 into individual URIs, discarding any comments. The URIs are not validated.

**uri_list**: an URI list

**Returns**: a newly allocated NULL-terminated list of strings holding the individual URIs. The array should be freed with `g_strfreev()`.

Since 2.6
CHAPTER 4. GLIB UTILITIES

4.2. CHARACTER SET CONVERSION

g_locale_from_utf8 ()

```c
char* g_locale_from_utf8 (const gchar *utf8string, gssize len, gsize *bytes_read, gsize *bytes_written, GError **error);
```

Converts a string from UTF-8 to the encoding used for strings by the C runtime (usually the same as that used by the operating system) in the current locale. On Windows this means the system codepage.

terraform : a UTF-8 encoded string

**len**: the length of the string, or -1 if the string is nul-terminated\(^1\).

**bytes_read**: location to store the number of bytes in the input string that were successfully converted, or NULL. Even if the conversion was successful, this may be less than \( \text{len} \) if there were partial characters at the end of the input. If the error \( \text{G_CONVERT_ERROR_ILLEGAL_SEQUENCE} \) occurs, the value stored will be the byte offset after the last valid input sequence.

**bytes_written**: the number of bytes stored in the output buffer (not including the terminating null).

**error**: location to store the error occurring, or NULL to ignore errors. Any of the errors in \( \text{GConvertError} \) may occur.

**Returns**: The converted string, or NULL on an error.

enum GConvertError

```c
typedef enum
{
   G_CONVERT_ERROR_NO_CONVERSION,
   G_CONVERT_ERROR_ILLEGAL_SEQUENCE,
   G_CONVERT_ERROR_FAILED,
   G_CONVERT_ERROR_PARTIAL_INPUT,
   G_CONVERT_ERROR_BAD_URI,
   G_CONVERT_ERROR_NOT_ABSOLUTE_PATH
} GConvertError;
```

Error codes returned by character set conversion routines.

**G_CONVERT_ERROR_NO_CONVERSION** Conversion between the requested character sets is not supported.

**G_CONVERT_ERROR_ILLEGAL_SEQUENCE** Invalid byte sequence in conversion input.

**G_CONVERT_ERROR_FAILED** Conversion failed for some reason.

**G_CONVERT_ERROR_PARTIAL_INPUT** Partial character sequence at end of input.

**G_CONVERT_ERROR_BAD_URI** URI is invalid.

**G_CONVERT_ERROR_NOT_ABSOLUTE_PATH** Pathname is not an absolute path.

g_get_charset ()

```c
gboolean g_get_charset (G_CONST_RETURN char **charset);
```

Obtains the character set for the current locale; you might use this character set as an argument to \( \text{g_convert}() \), to convert from the current locale’s encoding to some other encoding. (Frequently \( \text{g_locale_to_utf8()} \) and \( \text{g_locale_from_utf8()} \) are nice shortcuts, though.)

On Windows the character set returned by this function is the so-called system default ANSI codepage. That is the character set used by the "narrow" versions of C library and Win32 functions that handle file names. It might be different from the character set used by the C library’s current locale.
The return value is TRUE if the locale’s encoding is UTF-8, in that case you can perhaps avoid calling g_convert().

The string returned in charset is not allocated, and should not be freed.

charset: return location for character set name

Returns: TRUE if the returned charset is UTF-8

4.3 Unicode Manipulation

Name

Unicode Manipulation – functions operating on Unicode characters and UTF-8 strings

Synopsis

#include <glib.h>

typedef gunichar;
typedef gunichar2;
gboolean g_unichar_validate (gunichar ch);
gboolean g_unichar_isalnum (gunichar c);
gboolean g_unichar_isalpha (gunichar c);
gboolean g_unichar_isctrl (gunichar c);
gboolean g_unichar_isdefined (gunichar c);
gboolean g_unichar_isdigit (gunichar c);
gboolean g_unichar_ismark (gunichar c);
gboolean g_unichar_isprint (gunichar c);
gboolean g_unichar_ispunct (gunichar c);
gboolean g_unichar_isspace (gunichar c);
gboolean g_unichar_istitle (gunichar c);
gboolean g_unichar_isupper (gunichar c);
gboolean g_unichar_isxdigit (gunichar c);
gboolean g_unichar_iswide (gunichar c);
gboolean g_unichar_iswide_cjk (gunichar c);
gboolean g_unichar_iszerowidth (gunichar c);
gint g_unichar_digit_value (gunichar c);
gint g_unichar_xdigit_value (gunichar c);
enum GUnicodeType;
GUnicodeType g_unichar_type (gunichar c);
enum GUnicodeBreakType;
GUnicodeBreakType g_unichar_break_type (gunichar c);
gint g_unichar_combining_class (gunichar c);
void g_unicode_canonical_ordering (gunichar *string, gsize len);
gint g_unichar_digit_value (gunichar c);
gint g_unichar_xdigit_value (gunichar c);
enum GUnicodeScript;
GUnicodeScript g_unichar_get_script (gunichar c);
gboolean g_unichar_get_mirror_char (gunichar ch, gunichar *mirrored_ch);
gboolean g_unichar_isdefined (gunichar c);

223
CHAPTER 4. GLIB UTILITIES  
4.3. UNICODE MANIPULATION

#define  g_utf8_next_char (p)  
gunichar  g_utf8_get_char  (const gchar *p);  
gunichar  g_utf8_get_char_validated  (const gchar *p,  
gssize max_len);  
gchar*  g_utf8_offset_to_pointer  (const gchar *p,  
glong offset);  
glong  g_utf8_pointer_to_offset  (const gchar *str,  
const gchar *pos);  
gchar*  g_utf8_prev_char  (const gchar *p);  
gchar*  g_utf8_find_next_char  (const gchar *p,  
const gchar *end);  
gchar*  g_utf8_find_prev_char  (const gchar *str,  
const gchar *p);  
glong  g_utf8_strlen  (const gchar *p,  
g ssize max);  
gchar*  g_utf8_strncpy  (gchar *dest,  
const gchar *src,  
gsize n);  
gchar*  g_utf8_strchr  (const gchar *p,  
g ssize len,  
gunichar c);  
gchar*  g_utf8_strrchr  (const gchar *p,  
g ssize len,  
gunichar c);  
gchar*  g_utf8_strreverse  (const gchar *str,  
g ssize len);  
goboolean  g_utf8_validate  (const gchar *str,  
const gchar **end);  
gchar*  g_utf8_strup  (const gchar *str,  
g ssize len);  
gchar*  g_utf8_strdown  (const gchar *str,  
g ssize len);  
gchar*  g_utf8_casefold  (const gchar *str,  
g ssize len);  
gchar*  g_utf8_normalize  (const gchar *str,  
g ssize len,  
GNormalizeMode mode);  
enum  GNNormalizeMode;  
gint  g_utf8_collate  (const gchar *str1,  
const gchar *str2);  
gchar*  g_utf8_collate_key  (const gchar *str,  
g ssize len);  
gchar*  g_utf8_collate_key_for_filename  (const gchar *str,  
g ssize len);  
gunichar2*  g_utf8_to_utf16  (const gchar *str,  
g long len,  
g long *items_read,  
g long *items_written,  
GError **error);  
gunichar*  g_utf8_to_ucs4  (const gchar *str,  
g long len,  
g long *items_read,  
g long *items_written,  
GError **error);  
gunichar*  g_utf8_to_ucs4_fast  (const gchar *str,  
g long len,
CHAPTER 4. GLIB UTILITIES 4.3. UNICODE MANIPULATION

```c
typedef guint32 gunichar;

A type which can hold any UTF-32 or UCS-4 character code, also known as a Unicode code point.
To print/scan values of this type to/from text you need to convert to/from UTF-8, using g_utf32_to_utf8()/g_utf8_to_utf32().
To print/scan values of this type as integer, use G_GINT32_MODIFIER and/or G_GUINT32_FORMAT.
The notation to express a Unicode code point in running text is as a hexadecimal number with four to six digits and uppercase letters, prefixed by the string "U+". Leading zeros are omitted, unless the code point would have fewer than four hexadecimal digits. For example, "U+0041 LATIN CAPITAL LETTER A". To print a code point in the U+-notation, use the format string "U+04"G_GINT32_FORMAT"X". To scan, use the format string "U+06"G_GINT32_FORMAT"X".
```

gunichar c;
sscanf ("U+0041", "U+%06"G_GINT32_FORMAT"X", &c)
g_print ("Read U+%04"G_GINT32_FORMAT"X", c);

gunichar2

typedef guint16 gunichar2;
```

Description

This section describes a number of functions for dealing with Unicode characters and strings. There are analogues of the traditional <ctype.h> character classification and case conversion functions, UTF-8 analogues of some string utility functions, functions to perform normalization, case conversion and collation on UTF-8 strings and finally functions to convert between the UTF-8, UTF-16 and UCS-4 encodings of Unicode.

The implementations of the Unicode functions in GLib are based on the Unicode Character Data tables, which are available from www.unicode.org. GLib 2.8 supports Unicode 4.0, GLib 2.10 supports Unicode 4.1, GLib 2.12 supports Unicode 5.0, GLib 2.16.3 supports Unicode 5.1.

Details

gunichar

typedef guint32 gunichar;
```
A type which can hold any UTF-16 code point. To print/scan values of this type to/from text you need to convert to/from UTF-8, using g_utf16_to_utf8()/g_utf8_to_utf16(). To print/scan values of this type as integer, use G_GINT16_MODIFIER and/or G_GUINT16_FORMAT.

\[ g \_ \text{uchar} \_ \text{validate}() \]

```c
gboolean g_unichar_validate (gunichar ch);
```

Checks whether \textit{ch} is a valid Unicode character. Some possible integer values of \textit{ch} will not be valid. 0 is considered a valid character, though it’s normally a string terminator.

\textit{ch} : a Unicode character

\textit{Returns} : TRUE if \textit{ch} is a valid Unicode character

\[ g \_ \text{uchar} \_ \text{isalnum}() \]

```c
gboolean g_unichar_isalnum (gunichar c);
```

Determines whether a character is alphanumeric. Given some UTF-8 text, obtain a character value with \texttt{g_utf8_get_char}().

\textit{c} : a Unicode character

\textit{Returns} : TRUE if \textit{c} is an alphanumeric character

\[ g \_ \text{uchar} \_ \text{isalpha}() \]

```c
gboolean g_unichar_isalpha (gunichar c);
```

Determines whether a character is alphabetic (i.e. a letter). Given some UTF-8 text, obtain a character value with \texttt{g_utf8_get_char}().

\textit{c} : a Unicode character

\textit{Returns} : TRUE if \textit{c} is an alphabetic character

\[ g \_ \text{uchar} \_ \text{iscntrl}() \]

```c
gboolean g_unichar_iscntrl (gunichar c);
```

Determines whether a character is a control character. Given some UTF-8 text, obtain a character value with \texttt{g_utf8_get_char}().

\textit{c} : a Unicode character

\textit{Returns} : TRUE if \textit{c} is a control character

\[ g \_ \text{uchar} \_ \text{isdefined}() \]

```c
gboolean g_unichar_isdefined (gunichar c);
```

Determines if a given character is assigned in the Unicode standard.

\textit{c} : a Unicode character

\textit{Returns} : TRUE if the character has an assigned value

---

3UTF-16 also has so called \textit{surrogate pairs} to encode characters beyond the BMP as pairs of 16bit numbers. Surrogate pairs cannot be stored in a single gunichar field, but all GLib functions accepting gunichar2 arrays will correctly interpret surrogate pairs.
**g_unichar_isdigit ()**

```c
gboolean g_unichar_isdigit (gunichar c);
```

Determines whether a character is numeric (i.e., a digit). This covers ASCII 0-9 and also digits in other languages/scripts. Given some UTF-8 text, obtain a character value with `g_utf8_get_char()`.

- **c**: a Unicode character

**Returns**: TRUE if `c` is a digit

**g_unichar_isgraph ()**

```c
gboolean g_unichar_isgraph (gunichar c);
```

Determines whether a character is printable and not a space (returns FALSE for control characters, format characters, and spaces). `g_unichar_isprint()` is similar, but returns TRUE for spaces. Given some UTF-8 text, obtain a character value with `g_utf8_get_char()`.

- **c**: a Unicode character

**Returns**: TRUE if `c` is printable unless it’s a space

**g_unichar_islower ()**

```c
gboolean g_unichar_islower (gunichar c);
```

Determines whether a character is a lowercase letter. Given some UTF-8 text, obtain a character value with `g_utf8_get_char()`.

- **c**: a Unicode character

**Returns**: TRUE if `c` is a lowercase letter

**g_unichar_ismark ()**

```c
gboolean g_unichar_ismark (gunichar c);
```

Determines whether a character is a mark (non-spacing mark, combining mark, or enclosing mark in Unicode speak). Given some UTF-8 text, obtain a character value with `g_utf8_get_char()`.

Note: in most cases where isalpha characters are allowed, ismark characters should be allowed to as they are essential for writing most European languages as well as many non-Latin scripts.

- **c**: a Unicode character

**Returns**: TRUE if `c` is a mark character

Since 2.14

**g_unichar_isprint ()**

```c
gboolean g_unichar_isprint (gunichar c);
```

Determines whether a character is printable. Unlike `g_unichar_isgraph()`, returns TRUE for spaces. Given some UTF-8 text, obtain a character value with `g_utf8_get_char()`.

- **c**: a Unicode character

**Returns**: TRUE if `c` is printable
g_unichar_ispunct()

gboolean g_unichar_ispunct (gunichar c);

Determines whether a character is punctuation or a symbol. Given some UTF-8 text, obtain a character value with g_utf8_get_char().

c: a Unicode character

Returns: TRUE if c is a punctuation or symbol character

g_unichar_isspace()

gboolean g_unichar_isspace (gunichar c);

Determines whether a character is a space, tab, or line separator (newline, carriage return, etc.). Given some UTF-8 text, obtain a character value with g_utf8_get_char().

(Note: don’t use this to do word breaking; you have to use Pango or equivalent to get word breaking right, the algorithm is fairly complex.)

c: a Unicode character

Returns: TRUE if c is a space character

g_unichar_istitle()

gboolean g_unichar_istitle (gunichar c);

Determines if a character is titlecase. Some characters in Unicode which are composites, such as the DZ digraph have three case variants instead of just two. The titlecase form is used at the beginning of a word where only the first letter is capitalized. The titlecase form of the DZ digraph is U+01F2 LATIN CAPITAL LETTER D WITH SMALL LETTER Z.

c: a Unicode character

Returns: TRUE if the character is titlecase

g_unichar_isupper()

gboolean g_unichar_isupper (gunichar c);

Determines if a character is uppercase.

c: a Unicode character

Returns: TRUE if c is an uppercase character

g_unichar_isxdigit()

gboolean g_unichar_isxdigit (gunichar c);

Determines if a character is a hexidecimal digit.

c: a Unicode character.

Returns: TRUE if the character is a hexadecimal digit

g_unichar_iswide()

gboolean g_unichar_iswide (gunichar c);

Determines if a character is typically rendered in a double-width cell.

c: a Unicode character

Returns: TRUE if the character is wide
g_unichar_iswide_cjk

gboolean g_unichar_iswide_cjk (gunichar c);

Determines if a character is typically rendered in a double-width cell under legacy East Asian locales. If a character is wide according to g_unichar_iswide(), then it is also reported wide with this function, but the converse is not necessarily true. See the Unicode Standard Annex #11 for details.

If a character passes the g_unichar_iswide() test then it will also pass this test, but not the other way around. Note that some characters may pass both this test and g_unichar_iszerowidth().

\( c \): a Unicode character

**Returns**: TRUE if the character is wide in legacy East Asian locales

Since 2.12

---

g_unichar_iszerowidth

gboolean g_unichar_iszerowidth (gunichar c);

Determines if a given character typically takes zero width when rendered. The return value is TRUE for all non-spacing and enclosing marks (e.g., combining accents), format characters, zero-width space, but not U+00AD SOFT HYPHEN.

A typical use of this function is with one of g_unichar_iswide() or g_unichar_iswide_cjk() to determine the number of cells a string occupies when displayed on a grid display (terminals). However, note that not all terminals support zero-width rendering of zero-width marks.

\( c \): a Unicode character

**Returns**: TRUE if the character has zero width

Since 2.14

---

g_unichar_toupper

gunichar g_unichar_toupper (gunichar c);

Converts a character to uppercase.

\( c \): a Unicode character

**Returns**: the result of converting \( c \) to uppercase. If \( c \) is not an lowercase or titlecase character, or has no upper case equivalent \( c \) is returned unchanged.

---

g_unichar_tolower

gunichar g_unichar_tolower (gunichar c);

Converts a character to lower case.

\( c \): a Unicode character.

**Returns**: the result of converting \( c \) to lower case. If \( c \) is not an upperlower or titlecase character, or has no lowercase equivalent \( c \) is returned unchanged.

---

g_unichar_totitle

gunichar g_unichar_totitle (gunichar c);

Converts a character to the titlecase.

\( c \): a Unicode character

**Returns**: the result of converting \( c \) to titlecase. If \( c \) is not an uppercase or lowercase character, \( c \) is returned unchanged.
g_unichar_digit_value()

```c
int g_unichar_digit_value (gunichar c);
```

Determines the numeric value of a character as a decimal digit.

**c**: a Unicode character

**Returns**: If `c` is a decimal digit (according to `g_unichar_isdigit()`), its numeric value. Otherwise, -1.

---

g_unichar_xdigit_value()

```c
int g_unichar_xdigit_value (gunichar c);
```

Determines the numeric value of a character as a hexadecimal digit.

**c**: a Unicode character

**Returns**: If `c` is a hex digit (according to `g_unichar_isxdigit()`), its numeric value. Otherwise, -1.

---

enum GUnicodeType

```c
typedef enum 
{
    G_UNICODE_CONTROL,
    G_UNICODE_FORMAT,
    G_UNICODE_UNASSIGNED,
    G_UNICODE_PRIVATE_USE,
    G_UNICODE_SURROGATE,
    G_UNICODE_LOWERCASE_LETTER,
    G_UNICODE_MODIFIER_LETTER,
    G_UNICODE_OTHER_LETTER,
    G_UNICODE_TITLECASE_LETTER,
    G_UNICODE_UPPERCASE_LETTER,
    G_UNICODE_COMBINING_MARK,
    G_UNICODE_ENCLOSING_MARK,
    G_UNICODE_NON_SPACING_MARK,
    G_UNICODE_DECIMAL_NUMBER,
    G_UNICODE_LETTER_NUMBER,
    G_UNICODE_OTHER_NUMBER,
    G_UNICODE_CONNECT_PUNCTUATION,
    G_UNICODE_DASH_PUNCTUATION,
    G_UNICODE_CLOSE_PUNCTUATION,
    G_UNICODE_FINAL_PUNCTUATION,
    G_UNICODE_INITIAL_PUNCTUATION,
    G_UNICODE_OTHER_PUNCTUATION,
    G_UNICODE_OPEN_PUNCTUATION,
    G_UNICODE_CURRENCY_SYMBOL,
    G_UNICODE_MODIFIER_SYMBOL,
    G_UNICODE_MATH_SYMBOL,
    G_UNICODE_OTHER_SYMBOL,
    G_UNICODE_LINE_SEPARATOR,
    G_UNICODE_PARAGRAPH_SEPARATOR,
    G_UNICODE_SPACE_SEPARATOR
} GUnicodeType;
```

These are the possible character classifications from the Unicode specification. See [http://www.unicode.org/Public/UNIDATA/UnicodeData.html](http://www.unicode.org/Public/UNIDATA/UnicodeData.html).

**G_UNICODE_CONTROL** General category "Other, Control" (Cc)

**G_UNICODE_FORMAT** General category "Other, Format" (Cf)

**G_UNICODE_UNASSIGNED** General category "Other, Not Assigned" (Cn)
G_UNICODE_PRIVATE_USE General category "Other, Private Use" (Co)
G_UNICODE_SURROGATE General category "Other, Surrogate" (Cs)
G_UNICODE_LOWERCASE_LETTER General category "Letter, Lowercase" (Ll)
G_UNICODE_MODIFIER_LETTER General category "Letter, Modifier" (Lm)
G_UNICODE_OTHER_LETTER General category "Letter, Other" (Lo)
G_UNICODE_TITLECASE_LETTER General category "Letter, Titlecase" (Lt)
G_UNICODE_UPPERCASE_LETTER General category "Letter, Uppercase" (Lu)
G_UNICODE_COMBINING_MARK General category "Mark, Spacing Combining" (Mc)
G_UNICODE_ENCLOSING_MARK General category "Mark, Enclosing" (Me)
G_UNICODE_NON_SPACING_MARK General category "Mark, Nonspaceing" (Mn)
G_UNICODE_DECIMAL_NUMBER General category "Number, Decimal Digit" (Nd)
G_UNICODE_LETTER_NUMBER General category "Number, Letter" (Nl)
G_UNICODE_OTHER_NUMBER General category "Number, Other" (No)
G_UNICODE_CONNECT_PUNCTUATION General category "Punctuation, Connector" (Pc)
G_UNICODE_DASH_PUNCTUATION General category "Punctuation, Dash" (Pd)
G_UNICODE_CLOSE_PUNCTUATION General category "Punctuation, Close" (Pe)
G_UNICODE_FINAL_PUNCTUATION General category "Punctuation, Final quote" (Pf)
G_UNICODE_INITIAL_PUNCTUATION General category "Punctuation, Initial quote" (Pi)
G_UNICODE_OTHER_PUNCTUATION General category "Punctuation, Other" (Po)
G_UNICODE_OPEN_PUNCTUATION General category "Punctuation, Open" (Ps)
G_UNICODE_CURRENCY_SYMBOL General category "Symbol, Currency" (Sc)
G_UNICODE_MODIFIER_SYMBOL General category "Symbol, Modifier" (Sk)
G_UNICODE_MATH_SYMBOL General category "Symbol, Math" (Sm)
G_UNICODE_OTHER_SYMBOL General category "Symbol, Other" (So)
G_UNICODE_LINE_SEPARATOR General category "Separator, Line" (Zl)
G_UNICODE_PARAGRAPH_SEPARATOR General category "Separator, Paragraph" (Zp)
G_UNICODE_SPACE_SEPARATOR General category "Separator, Space" (Zs)

g_unichar_type ()

GUnicodeType g_unichar_type (gunichar c);

Classifies a Unicode character by type.

c : a Unicode character

Returns : the type of the character.
enum GUnicodeBreakType
{
    G_UNICODE_BREAK_MANDATORY,
    G_UNICODE_BREAK_CARRIAGE_RETURN,
    G_UNICODE_BREAK_LINE_FEED,
    G_UNICODE_BREAK_COMBINING_MARK,
    G_UNICODE_BREAK_SURROGATE,
    G_UNICODE_BREAK_ZERO_WIDTH_SPACE,
    G_UNICODE_BREAK_INSEPARABLE,
    G_UNICODE_BREAK_NON_BREAKING_GLUE,
    G_UNICODE_BREAK_CONTINGENT,
    G_UNICODE_BREAK_SPACE,
    G_UNICODE_BREAK_AFTER,
    G_UNICODE_BREAK_BEFORE,
    G_UNICODE_BREAK_BEFORE_AND_AFTER,
    G_UNICODE_BREAK_HYPHEN,
    G_UNICODE_BREAK_NON_STARTER,
    G_UNICODE_BREAK_OPEN_PUNCTUATION,
    G_UNICODE_BREAK_CLOSE_PUNCTUATION,
    G_UNICODE_BREAK_QUOTATION,
    G_UNICODE_BREAK_EXCLAMATION,
    G_UNICODE_BREAK IDEOGRAPHIC,
    G_UNICODE_BREAK_NUMERIC,
    G_UNICODE_BREAK_INFIX_SEPARATOR,
    G_UNICODE_BREAK_SYMBOL,
    G_UNICODE_BREAK_ALPHABETIC,
    G_UNICODE_BREAK_PREFIX,
    G_UNICODE_BREAK_POSTFIX,
    G_UNICODE_BREAK_COMPLEX_CONTEXT,
    G_UNICODE_BREAK_AMBIGUOUS,
    G_UNICODE_BREAK_UNKNOWN,
    G_UNICODE_BREAK_NEXT_LINE,
    G_UNICODE_BREAK_WORD_JOINER,
    G_UNICODE_BREAK_HANGUL_L_JAMO,
    G_UNICODE_BREAK_HANGUL_V_JAMO,
    G_UNICODE_BREAK_HANGUL_T_JAMO,
    G_UNICODE_BREAK_HANGUL_LV_SYLLABLE,
    G_UNICODE_BREAK_HANGUL_LVT_SYLLABLE
} GUnicodeBreakType;

These are the possible line break classifications. The five Hangul types were added in Unicode 4.1, so,
has been introduced in GLib 2.10. Note that new types may be added in the future. Applications should
be ready to handle unknown values. They may be regarded as G_UNICODE_BREAK_UNKNOWN. See http://www.unicode.org/unicode/reports/tr14/.

G_UNICODE_BREAK_MANDATORY Mandatory Break (BK)
G_UNICODE_BREAK_CARRIAGE_RETURN Carriage Return (CR)
G_UNICODE_BREAK_LINE_FEED Line Feed (LF)
G_UNICODE_BREAK_COMBINING_MARK Attached Characters and Combining Marks (CM)
G_UNICODE_BREAK_SURROGATE Surrogates (SG)
G_UNICODE_BREAK_ZERO_WIDTH_SPACE Zero Width Space (ZW)
G_UNICODE_BREAK_INSEPARABLE Inseparable (IN)
G_UNICODE_BREAK_NON_BREAKING_GLUE Non-breaking ("Glue") (GL)
G_UNICODE_BREAK_CONTINGENT Contingent Break Opportunity (CB)
G_UNICODE_BREAK_SPACE Space (SP)
Determines the break type of \texttt{c}. \texttt{c} should be a Unicode character (to derive a character from UTF-8 encoded text, use \texttt{g \_utf8 \_get \_char()}). The break type is used to find word and line breaks ("text boundaries"). Pango implements the Unicode boundary resolution algorithms and normally you would use a function such as \texttt{pango \_break()} instead of caring about break types yourself.

\texttt{c}: a Unicode character

\textit{Returns}: the break type of \texttt{c}
CHAPTER 4. GLIB UTILITIES

4.3. UNICODE MANIPULATION

**g_unichar_combining_class ()**

```c
qint g_unichar_combining_class (gunichar uc);
```

Determines the canonical combining class of a Unicode character.

**uc**: a Unicode character

**Returns**: the combining class of the character

Since 2.14

**g_unicode_canonical_ordering ()**

```c
void g_unicode_canonical_ordering (gunichar *string, gsize len);
```

Computes the canonical ordering of a string in-place. This rearranges decomposed characters in the string according to their combining classes. See the Unicode manual for more information.

**string**: a UCS-4 encoded string.

**len**: the maximum length of **string** to use.

**g_unicode_canonical_decomposition ()**

```c
gunichar * g_unicode_canonical_decomposition (gunichar ch, gsize *result_len);
```

Computes the canonical decomposition of a Unicode character.

**ch**: a Unicode character.

**result_len**: location to store the length of the return value.

**Returns**: a newly allocated string of Unicode characters. **result_len** is set to the resulting length of the string.

**g_unichar_get_mirror_char ()**

```c
gboolean g_unichar_get_mirror_char (gunichar ch, gunichar *mirrored_ch);
```

In Unicode, some characters are mirrored. This means that their images are mirrored horizontally in text that is laid out from right to left. For instance, "(" would become its mirror image, ")", in right-to-left text.

If **ch** has the Unicode mirrored property and there is another unicode character that typically has a glyph that is the mirror image of **ch**'s glyph and **mirrored_ch** is set, it puts that character in the address pointed to by **mirrored_ch**. Otherwise the original character is put.

**ch**: a Unicode character

**mirrored_ch**: location to store the mirrored character

**Returns**: TRUE if **ch** has a mirrored character, FALSE otherwise

Since 2.4
enum GUnicodeScript
{
    /* ISO 15924 code */
    G_UNICODE_SCRIPT_INVALID_CODE = -1,
    G_UNICODE_SCRIPT_COMMON = 0, /* Zyyy */
    G_UNICODE_SCRIPT_INHERITED, /* Qaai */
    G_UNICODE_SCRIPT_ARABIC, /* Arab */
    G_UNICODE_SCRIPT_ARMENIAN, /* Armn */
    G_UNICODE_SCRIPT_BENGALI, /* Beng */
    G_UNICODE_SCRIPT_BOPOMOFO, /* Bopo */
    G_UNICODE_SCRIPT_CHEROKEE, /* Cher */
    G_UNICODE_SCRIPT_COPTIC, /* Qaac */
    G_UNICODE_SCRIPT_CYRILLIC, /* Cyr1 (Cyrs) */
    G_UNICODE_SCRIPT_DESERET, /* Dart */
    G_UNICODE_SCRIPT_DEVANAGARI, /* Deva */
    G_UNICODE_SCRIPT_ETHIOPIAN, /* Ethi */
    G_UNICODE_SCRIPT_GEORGIAN, /* Geor (Geon, Geoa) */
    G_UNICODESCRIPT_GOTHIC, /* Goth */
    G_UNICODE_SCRIPT_GREEK, /* Grek */
    G_UNICODE_SCRIPT_GUJARATI, /* Gujr */
    G_UNICODE_SCRIPT_GURMUKHI, /* Guru */
    G_UNICODE_SCRIPT_HAN, /* Hani */
    G_UNICODE_SCRIPT_HANGUL, /* Hang */
    G_UNICODE_SCRIPT_HEBREW, /* Hebr */
    G_UNICODE_SCRIPT_HIRAGANA, /* Hira */
    G_UNICODE_SCRIPT_KANNADA, /* Knda */
    G_UNICODE_SCRIPT_KATAKANA, /* Kana */
    G_UNICODE_SCRIPT_KHMER, /* Khmr */
    G_UNICODE_SCRIPT_LAO, /* Lao */
    G_UNICODE_SCRIPT_LATIN, /* Latn (Latf, Latg) */
    G_UNICODE_SCRIPT_MALAYALAM, /* Mlym */
    G_UNICODE_SCRIPT_MONGOLIAN, /* Mong */
    G_UNICODE_SCRIPT_MYANMAR, /* Mymr */
    G_UNICODE_SCRIPT_OGHAM, /* Ogam */
    G_UNICODE_SCRIPT_OLD_ITALIC, /* Ital */
    G_UNICODE_SCRIPT_ORIYA, /* Orya */
    G_UNICODE_SCRIPT_RUNIC, /* Runr */
    G_UNICODE_SCRIPT_SINHALA, /* Sinh */
    G_UNICODE_SCRIPT_SYRIAC, /* Syrc (Syrj, Syrn, Syre) */
    G_UNICODE_SCRIPT_TAMIL, /* Tamil */
    G_UNICODE_SCRIPT_TELUGU, /* Telu */
    G_UNICODE_SCRIPT_THAANA, /* Thaa */
    G_UNICODE_SCRIPT_THAI, /* Thai */
    G_UNICODE_SCRIPT_TIBETAN, /* Tibt */
    G_UNICODE_SCRIPT_CANADIAN_ABORIGINAL, /* Cans */
    G_UNICODE_SCRIPT_YI, /* Yiii */
    G_UNICODE_SCRIPT_TAGALOG, /* Tglg */
    G_UNICODE_SCRIPT_HANUNOO, /* Hano */
    G_UNICODE_SCRIPT_BUHID, /* Buhd */
    G_UNICODE_SCRIPT_TAGBANWA, /* Tagb */

    /* Unicode-4.0 additions */
    G_UNICODE_SCRIPT_BRAILLE, /* Brai */
    G_UNICODE_SCRIPT_CYPRIOT, /* Cprt */
    G_UNICODE_SCRIPT_LIMBU, /* Limb */
    G_UNICODE_SCRIPT_OSMANYA, /* Osma */
    G_UNICODE_SCRIPT_SHAVIAN, /* Shaw */
    G_UNICODE_SCRIPT_LINEAR_B, /* Linb */
    G_UNICODE_SCRIPT_TAI_LE, /* Tale */
    G_UNICODE_SCRIPT_UGARITIC, /* Ugar */

    /* Unicode-4.1 additions */
    G_UNICODE_SCRIPT_NEW_TAI_LUE, /* Talu */
};
The `GUnicodeScript` enumeration identifies different writing systems. The values correspond to the names as defined in the Unicode standard. The enumeration has been added in GLib 2.14, and is interchangeable with PangoScript. Note that new types may be added in the future. Applications should be ready to handle unknown values. See Unicode Standard Annex #24: Script names.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>G_UNICODE_SCRIPT_INVALID_CODE</strong></td>
<td>a value never returned from <code>g_unichar_get_script()</code></td>
</tr>
<tr>
<td><strong>G_UNICODE_SCRIPT_COMMON</strong></td>
<td>a character used by multiple different scripts</td>
</tr>
<tr>
<td><strong>G_UNICODE_SCRIPT_INHERITED</strong></td>
<td>a mark glyph that takes its script from the base glyph to which it is attached</td>
</tr>
<tr>
<td><strong>G_UNICODE_SCRIPT_ARABIC</strong></td>
<td>Arabic</td>
</tr>
<tr>
<td><strong>G_UNICODE_SCRIPT_ARMENIAN</strong></td>
<td>Armenian</td>
</tr>
<tr>
<td><strong>G_UNICODE_SCRIPT_BENGALI</strong></td>
<td>Bengali</td>
</tr>
<tr>
<td><strong>G_UNICODE_SCRIPT_BOPOMOFO</strong></td>
<td>Bopomofo</td>
</tr>
<tr>
<td><strong>G_UNICODE_SCRIPT_CHEROKEE</strong></td>
<td>Cherokee</td>
</tr>
<tr>
<td><strong>G_UNICODE_SCRIPT_COPTIC</strong></td>
<td>Coptic</td>
</tr>
<tr>
<td><strong>G_UNICODE_SCRIPT_CYRILLIC</strong></td>
<td>Cyrillic</td>
</tr>
<tr>
<td><strong>G_UNICODE_SCRIPT_DESERET</strong></td>
<td>Deseret</td>
</tr>
<tr>
<td><strong>G_UNICODE_SCRIPT_DEVANAGARI</strong></td>
<td>Devanagari</td>
</tr>
<tr>
<td><strong>G_UNICODE_SCRIPT_ETHIOPIIC</strong></td>
<td>Ethiopic</td>
</tr>
<tr>
<td><strong>G_UNICODE_SCRIPT_GEORGIAN</strong></td>
<td>Georgian</td>
</tr>
<tr>
<td><strong>G_UNICODE_SCRIPT_GOTHIC</strong></td>
<td>Gothic</td>
</tr>
<tr>
<td><strong>G_UNICODE_SCRIPT_GREEK</strong></td>
<td>Greek</td>
</tr>
</tbody>
</table>
G_UNICODE_SCRIPT_GUJARATI Gujarati
G_UNICODE_SCRIPT_GURMUKHI Gurmukhi
G_UNICODE_SCRIPT_HAN Han
G_UNICODE_SCRIPT_HANGUL Hangul
G_UNICODE_SCRIPT_HEBREW Hebrew
G_UNICODE_SCRIPT_HIRAGANA Hiragana
G_UNICODE_SCRIPT_KANNADA Kannada
G_UNICODE_SCRIPT_KATAKANA Katakana
G_UNICODE_SCRIPT_KHMER Khmer
G_UNICODE_SCRIPT_LAO Lao
G_UNICODE_SCRIPT_LATIN Latin
G_UNICODE_SCRIPT MALAYALAM Malayalam
G_UNICODE_SCRIPT_MONGOLIAN Mongolian
G_UNICODE_SCRIPT_MYANMAR Myanmar
G_UNICODE_SCRIPT_OGHAM Ogham
G_UNICODE_SCRIPT_OLD_ITALIC Old Italic
G_UNICODE_SCRIPT_ORIYA Oriya
G_UNICODE_SCRIPT_RUNIC Runic
G_UNICODE_SCRIPT_SINHALA Sinhala
G_UNICODE_SCRIPT_SYRIAC Syriac
G_UNICODE_SCRIPT_TAMIL Tamil
G_UNICODE_SCRIPT_TELUGU Telugu
G_UNICODE_SCRIPT_THAANA Thaana
G_UNICODE_SCRIPT_THAI Thai
G_UNICODE_SCRIPT_TIBETAN Tibetan
G_UNICODE_SCRIPT_CANADIAN_ABORIGINAL Canadian Aboriginal
G_UNICODE_SCRIPT_YI Yi
G_UNICODE_SCRIPT_TAGALOG Tagalog
G_UNICODE_SCRIPT_HANUNOO Hanunoo
G_UNICODE_SCRIPT_BUHID Buhid
G_UNICODE_SCRIPT_TAGBANWA Tagbanwa
G_UNICODE_SCRIPT_BRAILLE Braille
G_UNICODE_SCRIPT_CYPRIOT Cypriot
G_UNICODE_SCRIPT_LIMBU Limbu
G_UNICODE_SCRIPT_OSMANYA Osmanya
G_UNICODE_SCRIPT_SHAIVAN Shavian
G_UNICODE_SCRIPT_LINEAR_B  Linear B
G_UNICODE_SCRIPT_TAI_LE  Tai Le
G_UNICODE_SCRIPT_UGARITIC  Ugaritic
G_UNICODE_SCRIPT_NEW_TAI_LUE  New Tai Lue
G_UNICODE_SCRIPT_BUGINESE  Buginese
G_UNICODE_SCRIPT_GLAGOLITIC  Glagolitic
G_UNICODE_SCRIPT_TIFINAGH  Tifinagh
G_UNICODE_SCRIPT_SYLOTI_NAGRI  Syloti Nagri
G_UNICODE_SCRIPT_OLD_PERSIAN  Old Persian
G_UNICODE_SCRIPT_KHAROSHTHI  Kharoshthi
G_UNICODE_SCRIPT_UNKNOWN  an unassigned code point
G_UNICODE_SCRIPT_BALINESE  Balinese
G_UNICODE_SCRIPT_CUNEIFORM  Cuneiform
G_UNICODE_SCRIPT_PHOENICIAN  Phoenician
G_UNICODE_SCRIPT_PHAGS_PA  Phags-pa
G_UNICODE_SCRIPT_NKO  N’Ko
G_UNICODE_SCRIPT_KAYAH_LI  Kayah Li. Since 2.16.3
G_UNICODE_SCRIPT_LEPCHA  Lepcha. Since 2.16.3
G_UNICODE_SCRIPT_REJANG  Rejang. Since 2.16.3
G_UNICODE_SCRIPT_SUNDANESE  Sundanese. Since 2.16.3
G_UNICODE_SCRIPT_SAURASHTRA  Saurashtra. Since 2.16.3
G_UNICODE_SCRIPT_CHAM  Cham. Since 2.16.3
G_UNICODE_SCRIPT_Ol_CHIKI  Ol Chiki. Since 2.16.3
G_UNICODE_SCRIPT_VAI  Vai. Since 2.16.3
G_UNICODE_SCRIPT_CARIAN  Carian. Since 2.16.3
G_UNICODE_SCRIPT_LYCIAN  Lycian. Since 2.16.3
G_UNICODE_SCRIPT_LYDIAN  Lydian. Since 2.16.3

_g_unichar_get_script()

GUnicodeScript _g_unichar_get_script (gunichar ch);

Looks up the GUnicodeScript for a particular character (as defined by Unicode Standard Annex 24). No check is made for ch being a valid Unicode character; if you pass in invalid character, the result is undefined.

This function is equivalent to pango_script_for_unichar() and the two are interchangeable.

ch : a Unicode character

Returns : the GUnicodeScript for the character.

Since 2.14
g_utf8_next_char()

#define g_utf8_next_char(p)

Skips to the next character in a UTF-8 string. The string must be valid; this macro is as fast as possible, and has no error-checking. You would use this macro to iterate over a string character by character. The macro returns the start of the next UTF-8 character. Before using this macro, use g_utf8_validate() to validate strings that may contain invalid UTF-8.

p : Pointer to the start of a valid UTF-8 character.

\textbf{g_utf8_get_char ()}

\begin{verbatim}
gunichar g_utf8_get_char (const gchar *p);
\end{verbatim}

Converts a sequence of bytes encoded as UTF-8 to a Unicode character. If \( p \) does not point to a valid UTF-8 encoded character, results are undefined. If you are not sure that the bytes are complete valid Unicode characters, you should use g_utf8_get_char_validated() instead.

p : a pointer to Unicode character encoded as UTF-8

\textbf{Returns} : the resulting character

\textbf{g_utf8_get_char_validated ()}

\begin{verbatim}
gunichar g_utf8_get_char_validated (const gchar *p, gssize max_len);
\end{verbatim}

Convert a sequence of bytes encoded as UTF-8 to a Unicode character. This function checks for incomplete characters, for invalid characters such as characters that are out of the range of Unicode, and for overlong encodings of valid characters.

p : a pointer to Unicode character encoded as UTF-8

\begin{itemize}
\item max_len : the maximum number of bytes to read, or -1, for no maximum or if \( p \) is nul-terminated
\end{itemize}

\textbf{Returns} : the resulting character. If \( p \) points to a partial sequence at the end of a string that could begin a valid character (or if max_len is zero), returns (gunichar)-2; otherwise, if \( p \) does not point to a valid UTF-8 encoded Unicode character, returns (gunichar)-1.

\textbf{g_utf8_offset_to_pointer ()}

\begin{verbatim}
gchar* g_utf8_offset_to_pointer (const gchar *str, glong offset);
\end{verbatim}

Converts from an integer character offset to a pointer to a position within the string.

Since 2.10, this function allows to pass a negative offset to step backwards. It is usually worth stepping backwards from the end instead of forwards if \( offset \) is in the last fourth of the string, since moving forward is about 3 times faster than moving backward.

\textbf{NOTE}

This function doesn’t abort when reaching the end of \( str \). Therefore you should be sure that offset is within string boundaries before calling that function. Call g_utf8_strlen() when unsure.

This limitation exists as this function is called frequently during text rendering and therefore has to be as fast as possible.
CHAPTER 4. GLIB UTILITIES

4.3. UNICODE MANIPULATION

\textit{str}: a UTF-8 encoded string  
\textit{offset}: a character offset within \textit{str}  
\textbf{Returns}: the resulting pointer

\textbf{g_utf8_pointer_to_offset ()}

\begin{verbatim}
glong g_utf8_pointer_to_offset (const gchar *str,
                               const gchar *pos);
\end{verbatim}

Converts from a pointer to position within a string to an integer character offset.  
Since 2.10, this function allows \textit{pos} to be before \textit{str}, and returns a negative offset in this case.

\textit{str}: a UTF-8 encoded string  
\textit{pos}: a pointer to a position within \textit{str}  
\textbf{Returns}: the resulting character offset

\textbf{g_utf8_prev_char ()}

\begin{verbatim}
gchar* g_utf8_prev_char (const gchar *p);
\end{verbatim}

Finds the previous UTF-8 character in the string before \textit{p}.  
\textit{p} does not have to be at the beginning of a UTF-8 character. No check is made to see if the character  
found is actually valid other than it starts with an appropriate byte. If \textit{p} might be the first character of  
the string, you must use \textbf{g_utf8_find_prev_char()} instead.

\textit{p}: a pointer to a position within a UTF-8 encoded string  
\textbf{Returns}: a pointer to the found character.

\textbf{g_utf8_find_next_char ()}

\begin{verbatim}
gchar* g_utf8_find_next_char (const gchar *p,
                               const gchar *end);
\end{verbatim}

Finds the start of the next UTF-8 character in the string after \textit{p}.  
\textit{p} does not have to be at the beginning of a UTF-8 character. No check is made to see if the character  
found is actually valid other than it starts with an appropriate byte.

\textit{p}: a pointer to a position within a UTF-8 encoded string  
\textit{end}: a pointer to the byte following the end of the string, or \textbf{NULL} to indicate that the string is nul-terminat ed.

\textbf{Returns}: a pointer to the found character or \textbf{NULL}

\textbf{g_utf8_find_prev_char ()}

\begin{verbatim}
gchar* g_utf8_find_prev_char (const gchar *str,
                               const gchar *p);
\end{verbatim}

Given a position \textit{p} with a UTF-8 encoded string \textit{str}, find the start of the previous UTF-8 character  
starting before \textit{p}. Returns \textbf{NULL} if no UTF-8 characters are present in \textit{str} before \textit{p}.  
\textit{p} does not have to be at the beginning of a UTF-8 character. No check is made to see if the character  
found is actually valid other than it starts with an appropriate byte.

\textit{str}: pointer to the beginning of a UTF-8 encoded string  
\textit{p}: pointer to some position within \textit{str}  
\textbf{Returns}: a pointer to the found character or \textbf{NULL}.
4.3. UNICODE MANIPULATION

**g_utf8_strlen ()**

```c
glong g_utf8_strlen (const gchar *p, gssize max);
```

Returns the length of the string in characters.

- **p**: pointer to the start of a UTF-8 encoded string.
- **max**: the maximum number of bytes to examine. If *max* is less than 0, then the string is assumed to be null-terminated. If *max* is 0, *p* will not be examined and may be NULL.

**Returns**: the length of the string in characters

**g_utf8_strncpy ()**

```c
gchar* g_utf8_strncpy (gchar *dest, const gchar *src, gsize n);
```

Like the standard C `strncpy()` function, but copies a given number of characters instead of a given number of bytes. The `src` string must be valid UTF-8 encoded text. (Use `g_utf8_validate()` on all text before trying to use UTF-8 utility functions with it.)

- **dest**: buffer to fill with characters from `src`
- **src**: UTF-8 encoded string
- **n**: character count

**Returns**: `dest`

**g_utf8_strchr ()**

```c
gchar* g_utf8_strchr (const gchar *p, gssize len, gunichar c);
```

Finds the leftmost occurrence of the given Unicode character in a UTF-8 encoded string, while limiting the search to `len` bytes. If `len` is -1, allow unbounded search.

- **p**: a null-terminated UTF-8 encoded string
- **len**: the maximum length of `p`
- **c**: a Unicode character

**Returns**: `NULL` if the string does not contain the character, otherwise, a pointer to the start of the leftmost occurrence of the character in the string.

**g_utf8_strrchr ()**

```c
gchar* g_utf8_strrchr (const gchar *p, gssize len, gunichar c);
```

Find the rightmost occurrence of the given Unicode character in a UTF-8 encoded string, while limiting the search to `len` bytes. If `len` is -1, allow unbounded search.

- **p**: a null-terminated UTF-8 encoded string
- **len**: the maximum length of `p`
- **c**: a Unicode character

**Returns**: `NULL` if the string does not contain the character, otherwise, a pointer to the start of the rightmost occurrence of the character in the string.
### g_utf8_strreverse()

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gchar*</td>
<td>Reverses a UTF-8 string. <strong>str</strong> must be valid UTF-8 encoded text. (Use g_utf8_validate() on all text before trying to use UTF-8 utility functions with it.)</td>
</tr>
<tr>
<td>g_size</td>
<td>This function is intended for programmatic uses of reversed strings. It pays no attention to decomposed characters, combining marks, byte order marks, directional indicators (LRM, LRO, etc) and similar characters which might need special handling when reversing a string for display purposes.</td>
</tr>
<tr>
<td></td>
<td>Note that unlike g_strreverse(), this function returns newly-allocated memory, which should be freed with g_free() when no longer needed.</td>
</tr>
</tbody>
</table>

**str**: a UTF-8 encoded string  
**len**: the maximum length of **str** to use, in bytes. If len < 0, then the string is null-terminated.  

**Returns**: a newly-allocated string which is the reverse of **str**.  
Since 2.2

### g_utf8_validate()

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gboolean</td>
<td>Validates UTF-8 encoded text. <strong>str</strong> is the text to validate; if <strong>str</strong> is null-terminated, then max_len can be -1, otherwise max_len should be the number of bytes to validate. If end is non-NULL, then the end of the valid range will be stored there (i.e. the start of the first invalid character if some bytes were invalid, or the end of the text being validated otherwise).</td>
</tr>
</tbody>
</table>
|             | Note that g_utf8_validate() returns FALSE if max_len is positive and NUL is met before max_len bytes have been read.  
|             | Returns TRUE if all of **str** was valid. Many GLib and GTK+ routines require valid UTF-8 as input; so data read from a file or the network should be checked with g_utf8_validate() before doing anything else with it. |
| **str**     | a pointer to character data |
| **max_len** | max bytes to validate, or -1 to go until NUL |
| **end**     | return location for end of valid data |

**Returns**: TRUE if the text was valid UTF-8

### g_utf8_strup()

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gchar*</td>
<td>Converts all Unicode characters in the string that have a case to uppercase. The exact manner that this is done depends on the current locale, and may result in the number of characters in the string increasing. (For instance, the German ess-zet will be changed to SS.)</td>
</tr>
<tr>
<td>g_size</td>
<td></td>
</tr>
</tbody>
</table>

**str**: a UTF-8 encoded string  
**len**: length of **str**, in bytes, or -1 if **str** is null-terminated.  

**Returns**: a newly allocated string, with all characters converted to uppercase.
CHAPTER 4. GLIB UTILITIES 4.3. UNICODE MANIPULATION

**g_utf8_strdown ()**

```c
char * g_utf8_strdown (const char *str,
int len);
```

Converts all Unicode characters in the string that have a case to lowercase. The exact manner that this is done depends on the current locale, and may result in the number of characters in the string changing.

***str***: a UTF-8 encoded string

***len***: length of *str*, in bytes, or -1 if *str* is nul-terminated.

**Returns**: a newly allocated string, with all characters converted to lowercase.

**g_utf8_casefold ()**

```c
char * g_utf8_casefold (const char *str,
int len);
```

Converts a string into a form that is independent of case. The result will not correspond to any particular case, but can be compared for equality or ordered with the results of calling *g_utf8_casefold()* on other strings.

Note that calling *g_utf8_casefold()* followed by *g_utf8_collate()* is only an approximation to the correct linguistic case insensitive ordering, though it is a fairly good one. Getting this exactly right would require a more sophisticated collation function that takes case sensitivity into account. GLib does not currently provide such a function.

***str***: a UTF-8 encoded string

***len***: length of *str*, in bytes, or -1 if *str* is nul-terminated.

**Returns**: a newly allocated string, that is a case independent form of *str*.

**g_utf8_normalize ()**

```c
char * g_utf8_normalize (const char *str,
int len,
unsigned char mode);
```

Converts a string into canonical form, standardizing such issues as whether a character with an accent is represented as a base character and combining accent or as a single precomposed character. The string has to be valid UTF-8, otherwise NULL is returned. You should generally call *g_utf8_normalize()* before comparing two Unicode strings.

The normalization mode **G_NORMALIZE_DEFAULT** only standardizes differences that do not affect the text content, such as the above-mentioned accent representation. **G_NORMALIZE_ALL** also standardizes the "compatibility" characters in Unicode, such as SUPERSCRIPT THREE to the standard forms (in this case DIGIT THREE). Formatting information may be lost but for most text operations such characters should be considered the same.

**G_NORMALIZE_DEFAULT_COMPOSE** and **G_NORMALIZE_ALL_COMPOSE** are like **G_NORMALIZE_DEFAULT** and **G_NORMALIZE_ALL**, but returned a result with composed forms rather than a maximally decomposed form. This is often useful if you intend to convert the string to a legacy encoding or pass it to a system with less capable Unicode handling.

***str***: a UTF-8 encoded string.

***len***: length of *str*, in bytes, or -1 if *str* is nul-terminated.

***mode***: the type of normalization to perform.

**Returns**: a newly allocated string, that is the normalized form of *str*, or NULL if *str* is not valid UTF-8.
CHAPTER 4. GLIB UTILITIES 4.3. UNICODE MANIPULATION

**enum GNormalizeMode**

```c
typedef enum {
    G_NORMALIZE_DEFAULT,
    G_NORMALIZE_NFD = G_NORMALIZE_DEFAULT,
    G_NORMALIZE_DEFAULT_COMPOSE,
    G_NORMALIZE_NFC = G_NORMALIZE_DEFAULT_COMPOSE,
    G_NORMALIZE_ALL,
    G_NORMALIZE_NFKD = G_NORMALIZE_ALL,
    G_NORMALIZE_ALL_COMPOSE,
    G_NORMALIZE_NFKC = G_NORMALIZE_ALL_COMPOSE
} GNormalizeMode;
```

Defines how a Unicode string is transformed in a canonical form, standardizing such issues as whether a character with an accent is represented as a base character and combining accent or as a single precomposed character. Unicode strings should generally be normalized before comparing them.

**G_NORMALIZE_DEFAULT** standardize differences that do not affect the text content, such as the above-mentioned accent representation.

**G_NORMALIZE_NFD** another name for **G_NORMALIZE_DEFAULT**.

**G_NORMALIZE_DEFAULT_COMPOSE** like **G_NORMALIZE_DEFAULT**, but with composed forms rather than a maximally decomposed form.

**G_NORMALIZE_NFC** another name for **G_NORMALIZE_DEFAULT_COMPOSE**.

**G_NORMALIZE_ALL** beyond **G_NORMALIZE_DEFAULT** also standardize the "compatibility" characters in Unicode, such as SUPERSCRIPT THREE to the standard forms (in this case DIGIT THREE). Formatting information may be lost but for most text operations such characters should be considered the same.

**G_NORMALIZE_NFKD** another name for **G_NORMALIZE_ALL**.

**G_NORMALIZE_ALL_COMPOSE** like **G_NORMALIZE_ALL**, but with composed forms rather than a maximally decomposed form.

**G_NORMALIZE_NFKC** another name for **G_NORMALIZE_ALL_COMPOSE**.

**g_utf8_collate ()**

```c
gint g_utf8_collate (const gchar *str1,
                    const gchar *str2);
```

Compares two strings for ordering using the linguistically correct rules for the current locale. When sorting a large number of strings, it will be significantly faster to obtain collation keys with **g_utf8_collate_key()** and compare the keys with strcmp() when sorting instead of sorting the original strings.

**str1**: a UTF-8 encoded string

**str2**: a UTF-8 encoded string

**Returns**: < 0 if str1 compares before str2, 0 if they compare equal, > 0 if str1 compares after str2.

**g_utf8_collate_key ()**

```c
gchar * g_utf8_collate_key (const gchar *str,
                            gssize len);
```

Converts a string into a collation key that can be compared with other collation keys produced by the same function using strcmp().

The results of comparing the collation keys of two strings with strcmp() will always be the same as comparing the two original keys with g_utf8_collate().

Note that this function depends on the current locale.
**str**: a UTF-8 encoded string.

**len**: length of **str**, in bytes, or -1 if **str** is nul-terminated.

**Returns**: a newly allocated string. This string should be freed with g_free() when you are done with it.

---

### `g_utf8_collate_key_for_filename ()`

```c
 gchar * g_utf8_collate_key_for_filename (const gchar *str, gssize len);
```

Converts a string into a collation key that can be compared with other collation keys produced by the same function using strcmp().

In order to sort filenames correctly, this function treats the dot `'.'` as a special case. Most dictionary orderings seem to consider it insignificant, thus producing the ordering "event.c" "eventgenerator.c" "event.h" instead of "event.c" "event.h" "eventgenerator.c". Also, we would like to treat numbers intelligently so that "file1" "file10" "file5" is sorted as "file1" "file5" "file10".

Note that this function depends on the current locale.

**str**: a UTF-8 encoded string.

**len**: length of **str**, in bytes, or -1 if **str** is nul-terminated.

**Returns**: a newly allocated string. This string should be freed with g_free() when you are done with it.

Since 2.8

### `g_utf8_to_utf16 ()`

```c
 gunichar2 * g_utf8_to_utf16 (const gchar *str, glong len, glong *items_read, glong *items_written, GError **error);
```

Convert a string from UTF-8 to UTF-16. A 0 character will be added to the result after the converted text.

**str**: a UTF-8 encoded string

**len**: the maximum length (number of bytes) of **str** to use. If **len** < 0, then the string is nul-terminated.

**items_read**: location to store number of bytes read, or NULL. If NULL, then G_CONVERT_ERROR_PARTIAL_INPUT will be returned in case **str** contains a trailing partial character. If an error occurs then the index of the invalid input is stored here.

**items_written**: location to store number of gunichar2 written, or NULL. The value stored here does not include the trailing 0.

**error**: location to store the error occurring, or NULL to ignore errors. Any of the errors in GConvertError other than G_CONVERT_ERROR_NO_CONVERSION may occur.

**Returns**: a pointer to a newly allocated UTF-16 string. This value must be freed with g_free(). If an error occurs, NULL will be returned and **error** set.

### `g_utf8_to_ucs4 ()`

```c
 gunichar * g_utf8_to_ucs4 (const gchar *str, glong len, glong *items_read, glong *items_written, GError **error);
```
CHAPTER 4. GLIB UTILITIES 4.3. UNICODE MANIPULATION

Convert a string from UTF-8 to a 32-bit fixed width representation as UCS-4. A trailing 0 will be added to the string after the converted text.

\textit{str}: a UTF-8 encoded string

\textit{len}: the maximum length of \textit{str} to use, in bytes. If \textit{len} < 0, then the string is nul-terminated.

\textit{items_read}: location to store number of bytes read, or NULL. If NULL, then G_CONVERT_ERROR_PARTIAL_INPUT will be returned in case \textit{str} contains a trailing partial character. If an error occurs then the index of the invalid input is stored here.

\textit{items_written}: location to store number of characters written or NULL. The value here stored does not include the trailing 0 character.

\textit{error}: location to store the error occurring, or NULL to ignore errors. Any of the errors in GConvertError other than G_CONVERT_ERROR_NO_CONVERSION may occur.

\textbf{Returns}: a pointer to a newly allocated UCS-4 string. This value must be freed with \texttt{g_free()}. If an error occurs, NULL will be returned and \textit{error} set.

\texttt{g_utf8_to_ucs4_fast ()}

\begin{verbatim}
gunichar * g_utf8_to_ucs4_fast (const gchar *str, glong len, glong *items_written);
\end{verbatim}

Convert a string from UTF-8 to a 32-bit fixed width representation as UCS-4, assuming valid UTF-8 input. This function is roughly twice as fast as \texttt{g_utf8_to_ucs4()} but does no error checking on the input.

\textit{str}: a UTF-8 encoded string

\textit{len}: the maximum length of \textit{str} to use, in bytes. If \textit{len} < 0, then the string is nul-terminated.

\textbf{Returns}: a pointer to a newly allocated UCS-4 string. This value must be freed with \texttt{g_free()}. 

\texttt{g_utf16_to_ucs4 ()}

\begin{verbatim}
gunichar * g_utf16_to_ucs4 (const gunichar2 *str, glong len, glong *items_read, glong *items_written, GError **error);
\end{verbatim}

Convert a string from UTF-16 to UCS-4. The result will be nul-terminated. 

\textit{str}: a UTF-16 encoded string

\textit{len}: the maximum length (number of gunichar2) of \textit{str} to use. If \textit{len} < 0, then the string is nul-terminated.

\textit{items_read}: location to store number of words read, or NULL. If NULL, then G_CONVERT_ERROR_PARTIAL_INPUT will be returned in case \textit{str} contains a trailing partial character. If an error occurs then the index of the invalid input is stored here.

\textit{items_written}: location to store number of characters written, or NULL. The value stored here does not include the trailing 0 character.

\textit{error}: location to store the error occurring, or NULL to ignore errors. Any of the errors in GConvertError other than G_CONVERT_ERROR_NO_CONVERSION may occur.

\textbf{Returns}: a pointer to a newly allocated UCS-4 string. This value must be freed with \texttt{g_free()}. If an error occurs, NULL will be returned and \textit{error} set.
g_utf16_to_utf8 ()

```
 gchar* g_utf16_to_utf8 (const gunichar2 *str, glong len, glong *items_read, glong *items_written, GError **error);
```

Convert a string from UTF-16 to UTF-8. The result will be terminated with a 0 byte.
Note that the input is expected to be already in native endianness, an initial byte-order-mark character is not handled specially. g_convert() can be used to convert a byte buffer of UTF-16 data of ambiguous endianness.

**str**: a UTF-16 encoded string

**len**: the maximum length (number of gunichar2) of str to use. If len < 0, then the string is nul-terminated.

**items_read**: location to store number of words read, or NULL. If NULL, then G_CONVERT_ERROR_PARTIAL_INPUT will be returned in case str contains a trailing partial character. If an error occurs then the index of the invalid input is stored here.

**items_written**: location to store number of bytes written, or NULL. The value stored here does not include the trailing 0 byte.

**error**: location to store the error occurring, or NULL to ignore errors. Any of the errors in GConvertError other than G_CONVERT_ERROR_NO_CONVERSION may occur.

**Returns**: a pointer to a newly allocated UTF-8 string. This value must be freed with g_free(). If an error occurs, NULL will be returned and error set.

---

g_ucs4_to_utf16 ()

```
 gunichar2 * g_ucs4_to_utf16 (const gunichar *str, glong len, glong *items_read, glong *items_written, GError **error);
```

Convert a string from UCS-4 to UTF-16. A 0 character will be added to the result after the converted text.

**str**: a UCS-4 encoded string

**len**: the maximum length (number of characters) of str to use. If len < 0, then the string is nul-terminated.

**items_read**: location to store number of bytes read, or NULL. If an error occurs then the index of the invalid input is stored here.

**items_written**: location to store number of gunichar2 written, or NULL. The value stored here does not include the trailing 0.

**error**: location to store the error occurring, or NULL to ignore errors. Any of the errors in GConvertError other than G_CONVERT_ERROR_NO_CONVERSION may occur.

**Returns**: a pointer to a newly allocated UTF-16 string. This value must be freed with g_free(). If an error occurs, NULL will be returned and error set.
CHAPTER 4. GLIB UTILITIES

4.4. BASE64 ENCODING

**g_ucs4_to_utf8 ()**

```c
qchar* g_ucs4_to_utf8 (const gunichar *str,
glong len,
glong *items_read,
glong *items_written,
GError **error);
```

Convert a string from a 32-bit fixed width representation as UCS-4 to UTF-8. The result will be terminated with a 0 byte.

*str*: a UCS-4 encoded string

*len*: the maximum length (number of characters) of *str* to use. If *len* < 0, then the string is nul-terminated.

*items_read*: location to store number of characters read, or NULL.

*items_written*: location to store number of bytes written or NULL. The value here stored does not include the trailing 0 byte.

*error*: location to store the error occurring, or NULL to ignore errors. Any of the errors in GConvertError other than G_CONVERT_ERROR_NO_CONVERSION may occur.

**Returns**: a pointer to a newly allocated UTF-8 string. This value must be freed with g_free(). If an error occurs, NULL will be returned and *error* set. In that case, *items_read* will be set to the position of the first invalid input character.

**g_unichar_to_utf8 ()**

```c
qint g_unichar_to_utf8 (gunichar c,
gchar *outbuf);
```

Converts a single character to UTF-8.

*c*: a Unicode character code

*outbuf*: output buffer, must have at least 6 bytes of space. If NULL, the length will be computed and returned and nothing will be written to *outbuf*.

**Returns**: number of bytes written

See Also

g_locale_to_utf8(), g_locale_from_utf8() Convenience functions for converting between UTF-8 and the locale encoding.

4.4 Base64 Encoding

**Name**

Base64 Encoding – encodes and decodes data in Base64 format

**Synopsis**

```c
#include <glib.h>

gsizen g_base64_encode_step (const guint8 *in,
gsize len,
gboolean break_lines,
```

```c
```
Description

Base64 is an encoding that allows to encode a sequence of arbitrary bytes as a sequence of printable ASCII characters. For the definition of Base64, see RFC 1421 or RFC 2045. Base64 is most commonly used as a MIME transfer encoding for email.

GLib supports incremental encoding using `g_base64_encode_step()` and `g_base64_encode_close()`. Incremental decoding can be done with `g_base64_decode_step()`. To encode or decode data in one go, use `g_base64_encode()` or `g_base64_decode()`. To avoid memory allocation when decoding, you can use `g_base64_decode_inplace()`.

Support for Base64 encoding has been added in GLib 2.12.

Details

`g_base64_encode_step()`

Incrementally encode a sequence of binary data into its Base-64 stringified representation. By calling this function multiple times you can convert data in chunks to avoid having to have the full encoded data in memory.

When all of the data has been converted you must call `g_base64_encode_close()` to flush the saved state.

The output buffer must be large enough to fit all the data that will be written to it. Due to the way base64 encodes you will need at least: \((\text{len} / 3 + 1) * 4 + 4\) bytes (+ 4 may be needed in case of non-zero state). If you enable line-breaking you will need at least: \(((\text{len} / 3 + 1) * 4 + 4) / 72 + 1\) bytes of extra space.

`break_lines` is typically used when putting base64-encoded data in emails. It breaks the lines at 72 columns instead of putting all of the text on the same line. This avoids problems with long lines in the email system.

**in**: the binary data to encode

**len**: the length of **in**

**break_lines**: whether to break long lines

**out**: pointer to destination buffer
CHAPTER 4. GLIB UTILITIES

4.4. BASE64 ENCODING

**state**: Saved state between steps, initialize to 0

**save**: Saved state between steps, initialize to 0

**Returns**: The number of bytes of output that was written

Since 2.12

### g_base64_encode_close ()

<table>
<thead>
<tr>
<th>gsize</th>
<th>g_base64_encode_close</th>
</tr>
</thead>
<tbody>
<tr>
<td>(gboolean break_lines, gchar *out, gint *state, gint *save);</td>
<td></td>
</tr>
</tbody>
</table>

Flush the status from a sequence of calls to `g_base64_encode_step()`.

**break_lines**: whether to break long lines

**out**: pointer to destination buffer

**state**: Saved state from `g_base64_encode_step()`

**save**: Saved state from `g_base64_encode_step()`

**Returns**: The number of bytes of output that was written

Since 2.12

### g_base64_encode ()

<table>
<thead>
<tr>
<th>gchar*</th>
<th>g_base64_encode</th>
</tr>
</thead>
<tbody>
<tr>
<td>(const gchar *in, gsize len);</td>
<td></td>
</tr>
</tbody>
</table>

Encode a sequence of binary data into its Base-64 stringified representation.

**data**: the binary data to encode

**len**: the length of `data`

**Returns**: a newly allocated, zero-terminated Base-64 encoded string representing `data`. The returned string must be freed with `g_free()`.

Since 2.12

### g_base64_decode_step ()

<table>
<thead>
<tr>
<th>gsize</th>
<th>g_base64_decode_step</th>
</tr>
</thead>
<tbody>
<tr>
<td>(const gchar *in, gsize len, gchar *out, gint *state, guint *save);</td>
<td></td>
</tr>
</tbody>
</table>

Incrementally decode a sequence of binary data from its Base-64 stringified representation. By calling this function multiple times you can convert data in chunks to avoid having to have the full encoded data in memory.

The output buffer must be large enough to fit all the data that will be written to it. Since base64 encodes 3 bytes in 4 chars you need at least: `(len / 4) * 3 + 3` bytes (+ 3 may be needed in case of non-zero state).

**in**: binary input data

**len**: max length of `in` data to decode

**out**: output buffer
4.5 Data Checksums

Name
Data Checksums – Computes the checksum for data

Synopsis

```
#include <glib.h>

enum GChecksumType;

GChecksum * g_checksum_new (GChecksumType checksum_type);
void g_checksum_free (GChecksum *checksum);
void g_checksum_reset (GChecksum *checksum);
void g_checksum_update (GChecksum *checksum, const guchar *data, gssize length);
const gchar * g_checksum_get_string (GChecksum *checksum);
```
CHAPTER 4. GLIB UTILITIES  4.5. DATA CHECKSUMS

Description

GLib provides a generic API for computing checksums (or "digests") for a sequence of arbitrary bytes, using various hashing algorithms like MD5, SHA-1 and SHA-256. Checksums are commonly used in various environments and specifications.

GLib supports incremental checksums using the GChecksum data structure, by calling g_checksum_update() as long as there’s data available and then using g_checksum_get_string() or g_checksum_get_digest() to compute the checksum and return it either as a string in hexadecimal form, or as a raw sequence of bytes. To compute the checksum for binary blobs and NUL-terminated strings in one go, use the convenience functions g_compute_checksum_for_data() and g_compute_checksum_for_string(), respectively.

Support for checksums has been added in GLib 2.16

Details

enum GChecksumType

typedef enum {
    G_CHECKSUM_MD5,
    G_CHECKSUM_SHA1,
    G_CHECKSUM_SHA256
} GChecksumType;

The hashing algorithm to be used by GChecksum when performing the digest of some data.

Note that the GChecksumType enumeration may be extended at a later date to include new hashing algorithm types.

G_CHECKSUM_MD5 Use the MD5 hashing algorithm

G_CHECKSUM_SHA1 Use the SHA-1 hashing algorithm

G_CHECKSUM_SHA256 Use the SHA-256 hashing algorithm

Since 2.16

g_checksum_type_get_length ()

qssize g_checksum_type_get_length (GChecksumType checkpoint_type);

Gets the length in bytes of digests of type checkpoint_type

check_sum_type: a GChecksumType

Returns: the checksum length, or -1 if checkpoint_type is not supported.

Since 2.16

GChecksum

typedef struct _GChecksum GChecksum;

An opaque structure representing a checksumming operation. To create a new GChecksum, use g_checksum_new(). To free a GChecksum, use g_checksum_free().

Since 2.16
4.5. DATA CHECKSUMS

`sChecksum_new()`

GChecksum * sChecksum_new (GChecksumType checksum_type);

Creates a new GChecksum, using the checksum algorithm `checksum_type`. If the `checksum_type` is not known, NULL is returned. A GChecksum can be used to compute the checksum, or digest, of an arbitrary binary blob, using different hashing algorithms.

A GChecksum works by feeding a binary blob through `sChecksum_update()` until there is data to be checked; the digest can then be extracted using `sChecksum_get_string()`, which will return the checksum as a hexadecimal string; or `sChecksum_get_digest()`, which will return a vector of raw bytes. Once either `sChecksum_get_string()` or `sChecksum_get_digest()` have been called on a GChecksum, the checksum will be closed and it won’t be possible to call `sChecksum_update()` on it anymore.

`checksum_type`: the desired type of checksum

**Returns**: the newly created GChecksum, or NULL. Use `sChecksum_free()` to free the memory allocated by it.

Since 2.16

`sChecksum_copy()`

GChecksum * sChecksum_copy (const GChecksum * checksum);

Copies a GChecksum. If `checksum` has been closed, by calling `sChecksum_get_string()` or `sChecksum_get_digest()`, the copied checksum will be closed as well.

`checksum`: the GChecksum to copy

**Returns**: the copy of the passed GChecksum. Use `sChecksum_free()` when finished using it.

Since 2.16

`sChecksum_free()`

void sChecksum_free (GChecksum *checksum);

Frees the memory allocated for `checksum`.

`checksum`: a GChecksum

Since 2.16

`sChecksum_reset()`

void sChecksum_reset (GChecksum *checksum);

Resets the state of the `checksum` back to its initial state.

`checksum`: the GChecksum to reset

Since 2.18
CHAPTER 4. GLIB UTILITIES

4.5. DATA CHECKSUMS

`g_checksum_update()`

```c
void g_checksum_update (GChecksum *checksum,
                        const guchar *data,
                        gssize length);
```

Feeds `data` into an existing GChecksum. The checksum must still be open, that is `g_checksum_get_string()` or `g_checksum_get_digest()` must not have been called on `checksum`.

- **checksum**: a GChecksum
- **data**: buffer used to compute the checksum
- **length**: size of the buffer, or -1 if it is a null-terminated string.

Since 2.16

`g_checksum_get_string()`

```c
const gchar * g_checksum_get_string (GChecksum *checksum);
```

Gets the digest as an hexadecimal string. Once this function has been called the GChecksum can no longer be updated with `g_checksum_update()`.

- **checksum**: a GChecksum

**Returns**: the hexadecimal representation of the checksum. The returned string is owned by the checksum and should not be modified or freed.

Since 2.16

`g_checksum_get_digest()`

```c
void g_checksum_get_digest (GChecksum *checksum,
                            guint8 *buffer,
                            gsize *digest_len);
```

Gets the digest from `checksum` as a raw binary vector and places it into `buffer`. The size of the digest depends on the type of checksum.

Once this function has been called, the GChecksum is closed and can no longer be updated with `g_checksum_update()`.

- **checksum**: a GChecksum
- **buffer**: output buffer
- **digest_len**: an inout parameter. The caller initializes it to the size of `buffer`. After the call it contains the length of the digest.

Since 2.16

`g_compute_checksum_for_data()`

```c
gchar * g_compute_checksum_for_data (GChecksumType checksum_type,
                                      const guchar *data,
                                      gsize length);
```

Computes the checksum for a binary `data` of `length`. This is a convenience wrapper for `g_checksum_new()`, `g_checksum_get_string()` and `g_checksum_free()`.

The hexadecimal string returned will be in lower case.

- **checksum_type**: a GChecksumType
data: binary blob to compute the digest of

length: length of data

Returns: the digest of the binary data as a string in hexadecimal. The returned string should be freed with g_free() when done using it.

Since 2.16

g_compute_checksum_for_string()

    gchar * g_compute_checksum_for_string (GChecksumType checksum_type, const gchar *str, gssize length);

Computes the checksum of a string.
The hexadecimal string returned will be in lower case.

checksum_type: a GChecksumType
str: the string to compute the checksum of
length: the length of the string, or -1 if the string is null-terminated.

Returns: the checksum as a hexadecimal string. The returned string should be freed with g_free() when done using it.

Since 2.16

4.6 Internationalization

Name

Internationalization – gettext support macros

Synopsis

#include <glib.h>
#include <glib/gi18n.h>

#define Q_(String) (String)
#define C_(Context,String) (Context,String)
#define N_(String) (String)
#define NC_(Context, String) (Context, String)

const gchar * g_dgettext (const gchar *domain, const gchar *msgid);
const gchar * g_dngettext (const gchar *domain, const gchar *msgid, const gchar *msgid_plural, gulong n);
const gchar * g_dpgettext (const gchar *domain, const gchar *msgctxtid, gsize msgidoffset);
const gchar * g_dpgettext2 (const gchar *domain, const gchar *context, const gchar *msgid);
const gchar * g_strip_context (const gchar *msgid, const gchar *msgval);
const gchar* const * g_get_language_names (void);

255
Description

GLib doesn’t force any particular localization method upon its users. But since GLib itself is localized using the gettext() mechanism, it seems natural to offer the de-facto standard gettext() support macros in an easy-to-use form.

In order to use these macros in an application, you must include glib/gi18n.h. For use in a library, must include glib/gi18n-lib.h after defining the GETTEXT_PACKAGE macro suitably for your library:

```c
#define GETTEXT_PACKAGE "gtk20"
#include <glib/gi18n-lib.h>
```

The gettext manual covers details of how to set up message extraction with xgettext.

Details

**Q_()**

```c
#define Q_(String)
```

Like _(), but handles context in message ids. This has the advantage that the string can be adorned with a prefix to guarantee uniqueness and provide context to the translator.

One use case given in the gettext manual is GUI translation, where one could e.g. disambiguate two "Open" menu entries as "File|Open" and "Printer|Open". Another use case is the string "Russian" which may have to be translated differently depending on whether it’s the name of a character set or a language. This could be solved by using "charset|Russian" and "language|Russian".

See the C_() macro for a different way to mark up translatable strings with context.

**NOTE**

If you are using the Q_() macro, you need to make sure that you pass --keyword=Q_ to xgettext when extracting messages. If you are using GNU gettext >= 0.15, you can also use --keyword=Q_:1g to let xgettext split the context string off into a msgctxt line in the po file.

**String** : the string to be translated, with a '|'-separated prefix which must not be translated

**Returns** : the translated message

Since 2.4

**C_()**

```c
#define C_(Context,String)
```

Uses gettext to get the translation for msgid.msgctxt is used as a context. This is mainly useful for short strings which may need different translations, depending on the context in which they are used.

```c
label1 = C_("Navigation", "Back");
label2 = C_("Body part", "Back");
```

**NOTE**

If you are using the C_() macro, you need to make sure that you pass --keyword=C_-lc,2 to xgettext when extracting messages. Note that this only works with GNU gettext >= 0.15.
Context: a message context, must be a string literal

String: a message id, must be a string literal

Returns: the translated message

Since 2.16

N_()

#define N_(String)

Only marks a string for translation. This is useful in situations where the translated strings can’t be directly used, e.g. in string array initializers. To get the translated string, call gettext() at runtime.

```c
{  
    static const char *messages[] = {  
        N_("some very meaningful message"),  
        N_("and another one")  
    };  
    const char *string;  
    ...  
    string  
        = index > 1 ? N_("a default message") : gettext (messages[index]);  
    ...  
}
```

String: the string to be translated

Since 2.4

NC_()

#define NC_(Context, String)

Only marks a string for translation, with context. This is useful in situations where the translated strings can’t be directly used, e.g. in string array initializers. To get the translated string, you should call g_dpgettext2() at runtime.

```c
{  
    static const char *messages[] = {  
        NC_("some context", "some very meaningful message"),  
        NC_("some context", "and another one")  
    };  
    const char *string;  
    ...  
    string  
        = index > 1 ? g_dpgettext2 (NULL, "some context", "a default message") : g_dpgettext2 (NULL, "some context", messages[index]);  
    ...  
}
```

Note

If you are using the NC_() macro, you need to make sure that you pass --keyword=NC_:1c,2 to xgettext when extracting messages. Note that this only works with GNU gettext >= 0.15. Intltool has support for the NC_() macro since version 0.40.1.
**Context**: a message context, must be a string literal

**String**: a message id, must be a string literal

Since 2.18

### g_dgettext ()

```c
const gchar * g_dgettext (const gchar *domain,
                          const gchar *msgid);
```

This function is a wrapper of dgettext() which does not translate the message if the default domain as set with textdomain() has no translations for the current locale.

The advantage of using this function over dgettext() proper is that libraries using this function (like GTK+) will not use translations if the application using the library does not have translations for the current locale. This results in a consistent English-only interface instead of one having partial translations. For this feature to work, the call to textdomain() and setlocale() should precede any g_dgettext() invocations. For GTK+, it means calling textdomain() before gtk_init or its variants.

This function disables translations if and only if upon its first call all the following conditions hold:

- **domain** is not NULL
- textdomain() has been called to set a default text domain
- there is no translations available for the default text domain and the current locale
- current locale is not 'C' or any English locales (those starting with "en_")

Note that this behavior may not be desired for example if an application has its untranslated messages in a language other than English. In those cases the application should call textdomain() after initializing GTK+.

Applications should normally not use this function directly, but use the _() macro for translations.

**domain**: the translation domain to use, or NULL to use the domain set with textdomain()

**msgid**: message to translate

**Returns**: The translated string

Since 2.18

### g_dngettext ()

```c
const gchar * g_dngettext (const gchar *domain,
                          const gchar *msgid,
                          const gchar *msgid_plural,
                          gulong n);
```

This function is a wrapper of dngettext() which does not translate the message if the default domain as set with textdomain() has no translations for the current locale.

See g_dgettext() for details of how this differs from dngettext() proper.

**domain**: the translation domain to use, or NULL to use the domain set with textdomain()

**msgid**: message to translate

**msgid_plural**: plural form of the message

**n**: the quantity for which translation is needed

**Returns**: The translated string

Since 2.18
g_dpgettext()

(const gchar *domain, const gchar *msgctxtid, gsize msgidoffset);

This function is a variant of g_dgettext() which supports a disambiguating message context. GNU gettext uses the ‘\004’ character to separate the message context and message id in msgctxtid. If 0 is passed as msgidoffset, this function will fall back to trying to use the deprecated convention of using “|” as a separation character.

This uses g_dgettext() internally. See that functions for differences with dgettext() proper.

Applications should normally not use this function directly, but use the C_() macro for translations with context.

domain: the translation domain to use, or NULL to use the domain set with textdomain()

msgctxtid: a combined message context and message id, separated by a \004 character

msgidoffset: the offset of the message id in msgctxid

Returns: The translated string

Since 2.16

g_dpgettext2 ()

(const gchar *domain, const gchar *context, const gchar *msgid);

This function is a variant of g_dgettext() which supports a disambiguating message context. GNU gettext uses the ‘\004’ character to separate the message context and message id in msgctxtid.

This uses g_dgettext() internally. See that functions for differences with dgettext() proper.

This function differs from C_() in that it is not a macro and thus you may use non-string-literals as context and msgid arguments.

domain: the translation domain to use, or NULL to use the domain set with textdomain()

context: the message context

msgid: the message

Returns: The translated string

Since 2.18

g_strip_context ()

(const gchar *msgid, const gchar *msgval);

An auxiliary function for gettext() support (see Q_()).

msgid: a string

msgval: another string

Returns: msgval, unless msgval is identical to msgid and contains a ‘|’ character, in which case a pointer to the substring of msgid after the first ‘|’ character is returned.

Since 2.4

259
CHAPTER 4. GLIB UTILITIES

4.7. DATE AND TIME FUNCTIONS

\section*{g_get_language_names ()}

\begin{verbatim}
const gchar* const * g_get_language_names (void);
\end{verbatim}

Computes a list of applicable locale names, which can be used to e.g. construct locale-dependent filenames or search paths. The returned list is sorted from most desirable to least desirable and always contains the default locale "C".

For example, if \textsc{LANGUAGE}=de:en_US, then the returned list is "de", "en_US", "en", "C".

This function consults the environment variables \textsc{LANGUAGE}, \textsc{LC_ALL}, \textsc{LC_MESSAGES} and \textsc{LANG} to find the list of locales specified by the user.

\textbf{Returns} : a NULL-terminated array of strings owned by GLib that must not be modified or freed.

Since 2.6

\section*{See Also}

The gettext manual.

\section*{4.7 Date and Time Functions}

\section*{Name}

Date and Time Functions – calendrical calculations and miscellaneous time stuff

\section*{Synopsis}

\begin{verbatim}
#include <glib.h>
#define G_USEC_PER_SEC
void g_get_current_time (GTimeVal *result);
void g_usleep (gulong microseconds);
void g_time_val_add (GTimeVal *time_,
                    glong microseconds);
gboolean g_time_val_from_iso8601 (const gchar *iso_date,
                                   GTimeVal *time_);
gchar* g_time_val_to_iso8601 (GTimeVal *time_);

typedef GTimeVal;
typedef GDate;
typedef GTime;
typedef GDateDay;
typedef GDateMonth;
typedef GDateYear;
typedef GDateWeekday;
#define G_DATE_BAD_DAY
#define G_DATE_BAD_JULIAN
#define G_DATE_BAD_YEAR
GDate* g_date_new (void);
GDate* g_date_new_dmy (GDateDay day,
                        GDateMonth month,
                        GDateYear year);
GDate* g_date_new_julian (guint32 julian_day);
void g_date_clear (GDate *date,
                   guint n_dates);
\end{verbatim}

260
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>g_date_free</code></td>
<td>Frees a date object</td>
</tr>
<tr>
<td><code>g_date_set_day</code></td>
<td>Sets the day of a date</td>
</tr>
<tr>
<td><code>g_date_set_month</code></td>
<td>Sets the month of a date</td>
</tr>
<tr>
<td><code>g_date_set_year</code></td>
<td>Sets the year of a date</td>
</tr>
<tr>
<td><code>g_date_set_dmy</code></td>
<td>Sets the day, month, and year of a date</td>
</tr>
<tr>
<td><code>g_date_set_julian</code></td>
<td>Sets the Julian day of a date</td>
</tr>
<tr>
<td><code>g_date_set_time</code></td>
<td>Sets the time of a date</td>
</tr>
<tr>
<td><code>g_date_set_time_t</code></td>
<td>Sets the time_t of a date</td>
</tr>
<tr>
<td><code>g_date_set_time_val</code></td>
<td>Sets the timeval of a date</td>
</tr>
<tr>
<td><code>g_date_set_parse</code></td>
<td>Parses a string into a date</td>
</tr>
<tr>
<td><code>g_date_add_days</code></td>
<td>Adds days to a date</td>
</tr>
<tr>
<td><code>g_date_subtract_days</code></td>
<td>Subtracts days from a date</td>
</tr>
<tr>
<td><code>g_date_add_months</code></td>
<td>Adds months to a date</td>
</tr>
<tr>
<td><code>g_date_subtract_months</code></td>
<td>Subtracts months from a date</td>
</tr>
<tr>
<td><code>g_date_add_years</code></td>
<td>Adds years to a date</td>
</tr>
<tr>
<td><code>g_date_subtract_years</code></td>
<td>Subtracts years from a date</td>
</tr>
<tr>
<td><code>g_date_days_between</code></td>
<td>Calculates the number of days between two dates</td>
</tr>
<tr>
<td><code>g_date_compare</code></td>
<td>Compares two dates</td>
</tr>
<tr>
<td><code>g_date_clamp</code></td>
<td>Clamps a date to a range</td>
</tr>
<tr>
<td><code>g_date_order</code></td>
<td>Orders two dates</td>
</tr>
<tr>
<td><code>g_date_get_day</code></td>
<td>Gets the day of a date</td>
</tr>
<tr>
<td><code>g_date_get_month</code></td>
<td>Gets the month of a date</td>
</tr>
<tr>
<td><code>g_date_get_year</code></td>
<td>Gets the year of a date</td>
</tr>
<tr>
<td><code>g_date_get_julian</code></td>
<td>Gets the Julian day of a date</td>
</tr>
<tr>
<td><code>g_date_get_weekday</code></td>
<td>Gets the weekday of a date</td>
</tr>
<tr>
<td><code>g_date_get_day_of_year</code></td>
<td>Gets the day of the year</td>
</tr>
<tr>
<td><code>g_date_get_days_in_month</code></td>
<td>Gets the number of days in a month</td>
</tr>
<tr>
<td><code>g_date_is_first_of_month</code></td>
<td>Checks if a date is the first of the month</td>
</tr>
<tr>
<td><code>g_date_is_last_of_month</code></td>
<td>Checks if a date is the last of the month</td>
</tr>
<tr>
<td><code>g_date_is_leap_year</code></td>
<td>Checks if a year is a leap year</td>
</tr>
<tr>
<td><code>g_date_get_monday_week_of_year</code></td>
<td>Gets the Monday week of the year</td>
</tr>
<tr>
<td><code>g_date_get_monday_weeks_in_year</code></td>
<td>Gets the number of Monday weeks in the year</td>
</tr>
</tbody>
</table>
CHAPTER 4. GLIB UTILITIES 4.7. DATE AND TIME FUNCTIONS

The GDate data structure represents a day between January 1, Year 1, and sometime a few thousand years in the future (right now it will go to the year 65535 or so, but g_date_set_parse() only parses up to the year 8000 or so - just count on "a few thousand"). GDate is meant to represent everyday dates, not astronomical dates or historical dates or ISO timestamps or the like. It extrapolates the current Gregorian calendar forward and backward in time; there is no attempt to change the calendar to match time periods or locations. GDate does not store time information; it represents a day.

The GDate implementation has several nice features; it is only a 64-bit struct, so storing large numbers of dates is very efficient. It can keep both a Julian and day-month-year representation of the date, since some calculations are much easier with one representation or the other. A Julian representation is simply a count of days since some fixed day in the past; for GDate the fixed day is January 1, 1 AD. ("Julian" dates in the GDate API aren’t really Julian dates in the technical sense; technically, Julian dates count from the start of the Julian period, Jan 1, 4713 BC).

GDate is simple to use. First you need a "blank" date; you can get a dynamically allocated date from g_date_new(), or you can declare an automatic variable or array and initialize it to a sane state by calling g_date_clear(). A cleared date is sane; it’s safe to call g_date_set_dmy() and the other mutator functions to initialize the value of a cleared date. However, a cleared date is initially invalid, meaning that it doesn’t represent a day that exists. It is undefined to call any of the date calculation routines on an invalid date. If you obtain a date from a user or other unpredictable source, you should check its validity with the g_date_valid() predicate. g_date_valid() is also used to check for errors with g_date_set_parse() and other functions that can fail. Dates can be invalidated by calling g_date_clear() again.

It is very important to use the API to access the GDate struct. Often only the day-month-year or only the Julian representation is valid. Sometimes neither is valid. Use the API.

GLib doesn’t contain any time-manipulation functions; however, there is a GTime typedef and a GTimeVal struct which represents a more precise time (with microseconds). You can request the current time as a GTimeVal with g_get_current_time().

Details

G_USEC_PER_SEC

#define G_USEC_PER_SEC 1000000

Number of microseconds in one second (1 million). This macro is provided for code readability.

GTimeVal
typedef struct {
    glong tv_sec;
    glong tv_usec;
} GTimeVal;

Represents a precise time, with seconds and microseconds. Similar to the struct timeval returned by
the gettimeofday() UNIX call.

**glong tv_sec;** seconds

**glong tv_usec;** microseconds

### g_get_current_time ()

```c
void g_get_current_time (GTimeVal *result);
```

Equivalent to the UNIX gettimeofday() function, but portable.

**result**: GTimeVal structure in which to store current time.

### g_usleep ()

```c
void g_usleep (gulong microseconds);
```

Pauses the current thread for the given number of microseconds. There are 1 million microseconds
per second (represented by the G_USEC_PER_SEC macro). g_usleep() may have limited precision, de-
pending on hardware and operating system; don’t rely on the exact length of the sleep.

**microseconds**: number of microseconds to pause

### g_time_val_add ()

```c
void g_time_val_add (GTimeVal *time_,
                    glong microseconds);
```

Adds the given number of microseconds to **time_. microseconds** can also be negative to decrease
the value of **time_**.

**time_**: a GTimeVal

**microseconds**: number of microseconds to add to **time**

### g_time_val_from_iso8601 ()

```c
gboolean g_time_val_from_iso8601 (const gchar *iso_date,
                                 GTimeVal *time_);
```

Converts a string containing an ISO 8601 encoded date and time to a GTimeVal and puts it into
**time_.**

**iso_date**: an ISO 8601 encoded date string

**time_**: a GTimeVal

**Returns**: TRUE if the conversion was successful.

Since 2.12
**GDate**

```c
typedef struct {
    guint julian_days : 32; /* julian days representation - we use a
    * bitfield hoping that 64 bit platforms
    * will pack this whole struct in one big
    * int
    */
    guint julian : 1;  /* julian is valid */
    guint dmy : 1;     /* dmy is valid */
    /* DMY representation */
    guint day : 6;
    guint month : 4;
    guint year : 16;
} GDate;
```

Represents a day between January 1, Year 1 and a few thousand years in the future. None of its members should be accessed directly. If the GDate is obtained from `g_date_new()`, it will be safe to mutate but invalid and thus not safe for calendrical computations. If it’s declared on the stack, it will contain garbage so must be initialized with `g_date_clear()`. `g_date_clear()` makes the date invalid but sane. An invalid date doesn’t represent a day, it’s “empty.” A date becomes valid after you set it to a Julian day or you set a day, month, and year.

- **guint julian_days**: the Julian representation of the date
- **guint julian**: this bit is set if `julian_days` is valid
- **guint dmy**: this is set if `day, month` and `year` are valid
- **guint day**: the day of the day-month-year representation of the date, as a number between 1 and 31
- **guint month**: the day of the day-month-year representation of the date, as a number between 1 and 12
- **guint year**: the day of the day-month-year representation of the date

**GTime**

```c
typedef gint32 GTime;
```

Simply a replacement for `time_t`. It has been deprecated since it is *not* equivalent to `time_t` on 64-bit platforms with a 64-bit `time_t`. Unrelated to `GTimer`.

Note that `GTime` is defined to always be a 32bit integer, unlike `time_t` which may be 64bit on some systems. Therefore, `GTime` will overflow in the year 2038, and you cannot use the address of a `GTime` variable as argument to the UNIX `time()` function. Instead, do the following:

```c
time_t ttime;
GTime gtime;
time (&ttime);
gtime = (GTime)ttime;
```
CHAPTER 4. GLIB UTILITIES 4.7. DATE AND TIME FUNCTIONS

enum GDateDMY
{
    G_DATE_DAY = 0,
    G_DATE_MONTH = 1,
    G_DATE_YEAR = 2
} GDateDMY;

This enumeration isn’t used in the API, but may be useful if you need to mark a number as a day, month, or year.

G_DATE_DAY a day
G_DATE_MONTH a month
G_DATE_YEAR a year

GDateDay
typedef guint8 GDateDay; /* day of the month */

Integer representing a day of the month; between 1 and 31. G_DATE_BAD_DAY represents an invalid day of the month.

enum GDateMonth
typedef enum
{
    G_DATE_BAD_MONTH = 0,
    G_DATE_JANUARY = 1,
    G_DATE_FEBRUARY = 2,
    G_DATE_MARCH = 3,
    G_DATE_APRIL = 4,
    G_DATE_MAY = 5,
    G_DATE_JUNE = 6,
    G_DATE_JULY = 7,
    G_DATE_AUGUST = 8,
    G_DATE_SEPTEMBER = 9,
    G_DATE_OCTOBER = 10,
    G_DATE_NOVEMBER = 11,
    G_DATE_DECEMBER = 12
} GDateMonth;

Enumeration representing a month; values are G_DATE_JANUARY, G_DATE_FEBRUARY, etc. G_DATE_BAD_MONTH is the invalid value.

G_DATE_BAD_MONTH invalid value
G_DATE_JANUARY January
G_DATE_FEBRUARY February
G_DATE_MARCH March
G_DATE_APRIL April
G_DATE_MAY May
G_DATE_JUNE June
G_DATE_JULY July
G_DATE_AUGUST August
G_DATE_SEPTEMBER September
G_DATE_OCTOBER October
G_DATE_NOVEMBER November
G_DATE_DECEMBER December

GDateYear

typedef guint16 GDateYear;

Integer representing a year; G_DATE_BAD_YEAR is the invalid value. The year must be 1 or higher; negative (BC) years are not allowed. The year is represented with four digits.

enum GDateWeekday

typedef enum
{
    G_DATE_BAD_WEEKDAY = 0,
    G_DATE_MONDAY = 1,
    G_DATE_TUESDAY = 2,
    G_DATE_WEDNESDAY = 3,
    G_DATE_THURSDAY = 4,
    G_DATE_FRIDAY = 5,
    G_DATE_SATURDAY = 6,
    G_DATE_SUNDAY = 7
} GDateWeekday;

Enumeration representing a day of the week; G_DATE_MONDAY, G_DATE_TUESDAY, etc. G_DATE_BAD_WEEKDAY is an invalid weekday.

G_DATE_BAD_WEEKDAY invalid value
G_DATE_MONDAY Monday
G_DATE_TUESDAY Tuesday
G_DATE_WEDNESDAY Wednesday
G_DATE_THURSDAY Thursday
G_DATE_FRIDAY Friday
G_DATE_SATURDAY Saturday
G_DATE_SUNDAY Sunday

G_DATE_BAD_DAY

#define G_DATE_BAD_DAY 0U

Represents an invalid GDateDay.

G_DATE_BAD_JULIAN

#define G_DATE_BAD_JULIAN 0U

Represents an invalid Julian day number.

G_DATE_BAD_YEAR

#define G_DATE_BAD_YEAR 0U

Represents an invalid year.
CHAPTER 4. GLIB UTILITIES  

## 4.7. DATE AND TIME FUNCTIONS

### g_date_new()

```c
GDate* g_date_new (void);
```

Allocates a `GDate` and initializes it to a sane state. The new date will be cleared (as if you’d called `g_date_clear()`) but invalid (it won’t represent an existing day). Free the return value with `g_date_free()`.

**Returns**: a newly-allocated `GDate`

### g_date_new_dmy()

```c
GDate* g_date_new_dmy (GDateDay day,
                         GDateMonth month,
                         GDateYear year);
```

Like `g_date_new()`, but also sets the value of the date. Assuming the day-month-year triplet you pass in represents an existing day, the returned date will be valid.

- **day**: day of the month
- **month**: month of the year
- **year**: year

**Returns**: a newly-allocated `GDate` initialized with `day`, `month`, and `year`

### g_date_new_julian()

```c
GDate* g_date_new_julian (guint32 julian_day);
```

Like `g_date_new()`, but also sets the value of the date. Assuming the Julian day number you pass in is valid (greater than 0, less than an unreasonably large number), the returned date will be valid.

- **julian_day**: days since January 1, Year 1

**Returns**: a newly-allocated `GDate` initialized with `julian_day`

### g_date_clear()

```c
void g_date_clear (GDate *date,
                   guint n_dates);
```

Initializes one or more `GDate` structs to a sane but invalid state. The cleared dates will not represent an existing date, but will not contain garbage. Useful to init a date declared on the stack. Validity can be tested with `g_date_valid()`.

- **date**: pointer to one or more dates to clear
- **n_dates**: number of dates to clear

### g_date_free()

```c
void g_date_free (GDate *date);
```

Frees a `GDate` returned from `g_date_new()`.

- **date**: a `GDate`
### g_date_set_day()

```c
void g_date_set_day (GDate *date, GDateDay day);
```

Sets the day of the month for a GDate. If the resulting day-month-year triplet is invalid, the date will be invalid.

**date**: a GDate  
**day**: day to set

### g_date_set_month()

```c
void g_date_set_month (GDate *date, GDateMonth month);
```

Sets the month of the year for a GDate. If the resulting day-month-year triplet is invalid, the date will be invalid.

**date**: a GDate  
**month**: month to set

### g_date_set_year()

```c
void g_date_set_year (GDate *date, GDateYear year);
```

Sets the year for a GDate. If the resulting day-month-year triplet is invalid, the date will be invalid.

**date**: a GDate  
**year**: year to set

### g_date_set_dmy()

```c
void g_date_set_dmy (GDate *date, GDateDay day, GDateMonth month, GDateYear y);
```

Sets the value of a GDate from a day, month, and year. The day-month-year triplet must be valid; if you aren’t sure it is, call g_date_valid_dmy() to check before you set it.

**date**: a GDate  
**day**: day  
**month**: month  
**y**: year

### g_date_set_julian()

```c
void g_date_set_julian (GDate *date, guint32 julian_date);
```

Sets the value of a GDate from a Julian day number.

**date**: a GDate  
**julian_date**: Julian day number (days since January 1, Year 1)
CHAPTER 4. GLIB UTILITIES 4.7. DATE AND TIME FUNCTIONS

**g_date_set_time ()**

```c
void g_date_set_time (GDate *date, GTime time_);
```

**WARNING**

`g_date_set_time` is deprecated and should not be used in newly-written code.

Sets the value of a date from a GTime value. The time to date conversion is done using the user’s current timezone.

*Deprecated*: 2.10: Use `g_date_set_time_t()` instead.

**date**: a GDate.

**time_**: GTime value to set.

**g_date_set_time_t ()**

```c
void g_date_set_time_t (GDate *date, time_t timet);
```

Sets the value of a date to the date corresponding to a time specified as a time_t. The time to date conversion is done using the user’s current timezone.

To set the value of a date to the current day, you could write:

```c
g_date_set_time_t (date, time (NULL));
```

**date**: a GDate

**timet**: time_t value to set

Since 2.10

**g_date_set_time_val ()**

```c
void g_date_set_time_val (GDate *date, GTimeVal *timeval);
```

Sets the value of a date from a GTimeVal value. Note that the `tv_usec` member is ignored, because GDate can’t make use of the additional precision.

**date**: a GDate

**timeval**: GTimeVal value to set

Since 2.10

**g_date_set_parse ()**

```c
void g_date_set_parse (GDate *date, const gchar *str);
```

Parses a user-inputted string `str`, and try to figure out what date it represents, taking the current locale into account. If the string is successfully parsed, the date will be valid after the call. Otherwise, it will be invalid. You should check using `g_date_valid()` to see whether the parsing succeeded.

This function is not appropriate for file formats and the like; it isn’t very precise, and its exact behavior varies with the locale. It’s intended to be a heuristic routine that guesses what the user means by a given string (and it does work pretty well in that capacity).
**g_date_add_days**

```c
void g_date_add_days (GDate *date, guint n_days);
```

Increments a date some number of days. To move forward by weeks, add weeks*7 days. The date must be valid.

**date**: a GDate to increment

**n_days**: number of days to move the date forward

**g_date_subtract_days**

```c
void g_date_subtract_days (GDate *date, guint n_days);
```

Moves a date some number of days into the past. To move by weeks, just move by weeks*7 days. The date must be valid.

**date**: a GDate to decrement

**n_days**: number of days to move

**g_date_add_months**

```c
void g_date_add_months (GDate *date, guint n_months);
```

Increments a date by some number of months. If the day of the month is greater than 28, this routine may change the day of the month (because the destination month may not have the current day in it). The date must be valid.

**date**: a GDate to increment

**n_months**: number of months to move forward

**g_date_subtract_months**

```c
void g_date_subtract_months (GDate *date, guint n_months);
```

Moves a date some number of months into the past. If the current day of the month doesn’t exist in the destination month, the day of the month may change. The date must be valid.

**date**: a GDate to decrement

**n_months**: number of months to move

**g_date_add_years**

```c
void g_date_add_years (GDate *date, guint n_years);
```

Increments a date by some number of years. If the date is February 29, and the destination year is not a leap year, the date will be changed to February 28. The date must be valid.

**date**: a GDate to increment

**n_years**: number of years to move forward
### g_date_subtract_years

```c
void g_date_subtract_years (GDate *date,
                           guint n_years);
```

Moves a date some number of years into the past. If the current day doesn’t exist in the destination year (i.e. it’s February 29 and you move to a non-leap-year) then the day is changed to February 29. The date must be valid.

- **date**: a GDate to decrement
- **n_years**: number of years to move

### g_date_days_between

```c
gint g_date_days_between (const GDate *date1,
                          const GDate *date2);
```

Computes the number of days between two dates. If `date2` is prior to `date1`, the returned value is negative. Both dates must be valid.

- **date1**: the first date
- **date2**: the second date

**Returns**: the number of days between `date1` and `date2`

### g_date_compare

```c
gint g_date_compare (const GDate *lhs,
                     const GDate *rhs);
```

qsort()-style comparison function for dates. Both dates must be valid.

- **lhs**: first date to compare
- **rhs**: second date to compare

**Returns**: 0 for equal, less than zero if `lhs` is less than `rhs`, greater than zero if `lhs` is greater than `rhs`

### g_date_clamp

```c
void g_date_clamp (GDate *date,
                   const GDate *min_date,
                   const GDate *max_date);
```

If `date` is prior to `min_date`, sets `date` equal to `min_date`. If `date` falls after `max_date`, sets `date` equal to `max_date`. Otherwise, `date` is unchanged. Either of `min_date` and `max_date` may be NULL. All non-NULL dates must be valid.

- **date**: a GDate to clamp
- **min_date**: minimum accepted value for `date`
- **max_date**: maximum accepted value for `date`

### g_date_order

```c
void g_date_order (GDate *date1,
                   GDate *date2);
```

Checks if `date1` is less than or equal to `date2`, and swap the values if this is not the case.

- **date1**: the first date
- **date2**: the second date
g_date_get_day()

GDateDay g_date_get_day (const GDate *date);

Returns the day of the month. The date must be valid.

date: a GDate to extract the day of the month from

Returns: day of the month

g_date_get_month()

GDateMonth g_date_get_month (const GDate *date);

Returns the month of the year. The date must be valid.

date: a GDate to get the month from

Returns: month of the year as a GDateMonth

g_date_get_year()

GDateYear g_date_get_year (const GDate *date);

Returns the year of a GDate. The date must be valid.

date: a GDate

Returns: year in which the date falls

272
CHAPTER 4. GLIB UTILITIES

4.7. DATE AND TIME FUNCTIONS

**g_date_get_days_in_month ()**

```c
uint8 g_date_get_days_in_month (GDateMonth month, GDateYear year);
```

Returns the number of days in a month, taking leap years into account.

*month*: month

*year*: year

**Returns**: number of days in *month* during the *year*

**g_date_is_first_of_month ()**

```c
gboolean g_date_is_first_of_month (const GDate *date);
```

Returns **TRUE** if the date is on the first of a month. The date must be valid.

*date*: a GDate to check

**Returns**: **TRUE** if the date is the first of the month

**g_date_is_last_of_month ()**

```c
gboolean g_date_is_last_of_month (const GDate *date);
```

Returns **TRUE** if the date is the last day of the month. The date must be valid.

*date*: a GDate to check

**Returns**: **TRUE** if the date is the last day of the month

**g_date_is_leap_year ()**

```c
gboolean g_date_is_leap_year (GDateYear year);
```

Returns **TRUE** if the year is a leap year.

*year*: year to check

**Returns**: **TRUE** if the year is a leap year

**g_date_get_monday_week_of_year ()**

```c
guint g_date_get_monday_week_of_year (const GDate *date);
```

Returns the week of the year, where weeks are understood to start on Monday. If the date is before the first Monday of the year, return 0. The date must be valid.

*date*: a GDate

**Returns**: week of the year

---

* For the purposes of this function, leap year is every year divisible by 4 unless that year is divisible by 100. If it is divisible by 100 it would be a leap year only if that year is also divisible by 400.
CHAPTER 4. GLIB UTILITIES 4.7. DATE AND TIME FUNCTIONS

\textbf{g_date_get_monday_weeks_in_year ()}

\begin{Verbatim}
\textbf{guint8} \textbf{g_date_get_monday_weeks_in_year} (\textbf{GDateYear} year);
\end{Verbatim}

Returns the number of weeks in the year, where weeks are taken to start on Monday. Will be 52 or 53. The date must be valid. (Years always have 52 7-day periods, plus 1 or 2 extra days depending on whether it’s a leap year. This function is basically telling you how many Mondays are in the year, i.e. there are 53 Mondays if one of the extra days happens to be a Monday.)

\textit{year}: a year

\textit{Returns}: number of Mondays in the year

\textbf{g_date_get_sunday_week_of_year ()}

\begin{Verbatim}
\textbf{guint} \textbf{g_date_get_sunday_week_of_year} (const \textbf{GDate} *date);
\end{Verbatim}

Returns the week of the year during which this date falls, if weeks are understood to being on Sunday. The date must be valid. Can return 0 if the day is before the first Sunday of the year.

\textit{date}: a GDate

\textit{Returns}: week number

\textbf{g_date_get_sunday_weeks_in_year ()}

\begin{Verbatim}
\textbf{guint8} \textbf{g_date_get_sunday_weeks_in_year} (\textbf{GDateYear} year);
\end{Verbatim}

Returns the number of weeks in the year, where weeks are taken to start on Sunday. Will be 52 or 53. The date must be valid. (Years always have 52 7-day periods, plus 1 or 2 extra days depending on whether it’s a leap year. This function is basically telling you how many Sundays are in the year, i.e. there are 53 Sundays if one of the extra days happens to be a Sunday.)

\textit{year}: year to count weeks in

\textit{Returns}: number of weeks

\textbf{g_date_get_iso8601_week_of_year ()}

\begin{Verbatim}
\textbf{guint} \textbf{g_date_get_iso8601_week_of_year} (const \textbf{GDate} *date);
\end{Verbatim}

Returns the week of the year, where weeks are interpreted according to ISO 8601.

\textit{date}: a valid GDate

\textit{Returns}: ISO 8601 week number of the year.

Since 2.6

\textbf{g_date_strftime ()}

\begin{Verbatim}
\textbf{gsize} \textbf{g_date_strftime} (\textbf{gchar} *s, \textbf{gsize} slen, const \textbf{gchar} *format, const \textbf{GDate} *date);
\end{Verbatim}

Generates a printed representation of the date, in a locale-specific way. Works just like the platform’s C library \texttt{strftime()} function, but only accepts date-related formats; time-related formats give undefined results. Date must be valid. Unlike \texttt{strftime()} (which uses the locale encoding), works on a UTF-8 format string and stores a UTF-8 result.

This function does not provide any conversion specifiers in addition to those implemented by the platform’s C library. For example, don’t expect that using \texttt{g_date_strftime()} would make the F provided by the C99 \texttt{strftime()} work on Windows where the C library only complies to C89.
CHAPTER 4. GLIB UTILITIES 4.7. DATE AND TIME FUNCTIONS

\( s \): destination buffer

\( slen \): buffer size

\( format \): format string

\( date \): valid GDate

**Returns**: number of characters written to the buffer, or 0 the buffer was too small

**g_date_to_struct_tm()**

```c
void g_date_to_struct_tm (const GDate *date,
                          struct tm *tm);
```

Fills in the date-related bits of a struct tm using the \( date \) value. Initializes the non-date parts with something sane but meaningless.

\( date \): a GDate to set the struct tm from.

\( tm \): struct tm to fill.

**g_date_valid()**

```c
gboolean g_date_valid (const GDate *date);
```

Returns TRUE if the GDate represents an existing day. The date must not contain garbage; it should have been initialized with \( g_date_clear() \) if it wasn’t allocated by one of the \( g_date_new() \) variants.

\( date \): a GDate to check

**Returns**: Whether the date is valid

**g_date_valid_day()**

```c
gboolean g_date_valid_day (GDateDay day);
```

Returns TRUE if the day of the month is valid (a day is valid if it’s between 1 and 31 inclusive).

\( day \): day to check

**Returns**: %TRUE if the day is valid

**g_date_valid_month()**

```c
gboolean g_date_valid_month (GDateMonth month);
```

Returns TRUE if the month value is valid. The 12 GDateMonth enumeration values are the only valid months.

\( month \): month

**Returns**: %TRUE if the month is valid

**g_date_valid_year()**

```c
gboolean g_date_valid_year (GDateYear year);
```

Returns TRUE if the year is valid. Any year greater than 0 is valid, though there is a 16-bit limit to what GDate will understand.

\( year \): year

**Returns**: %TRUE if the year is valid
CHAPTER 4. GLIB UTILITIES

4.8 Random Numbers

Name
Random Numbers – pseudo-random number generator

Synopsis

```
#include <glib.h>

GRand;
GRand* g_rand_new_with_seed (guint32 seed);
GRand* g_rand_new_with_seed_array (const guint32 *seed, guint seed_length);
GRand* g_rand_new (void);
GRand* g_rand_copy (GRand *rand_);
void g_rand_free (GRand *rand_);
void g_rand_set_seed (GRand *rand_, guint32 seed);
void g_rand_set_seed_array (GRand *rand_, const guint32 *seed, guint seed_length);
```

---

**g_date_valid_dmy ()**

| gboolean | g_date_valid_dmy (GDateDay day, GDateMonth month, GDateYear year); |

Returns **TRUE** if the day-month-year triplet forms a valid, existing day in the range of days GDate understands (Year 1 or later, no more than a few thousand years in the future).

- **day**: day
- **month**: month
- **year**: year

**Returns**: %TRUE if the date is a valid one

**g_date_valid_julian ()**

| gboolean | g_date_valid_julian (guint32 julian_date); |

Returns **TRUE** if the Julian day is valid. Anything greater than zero is basically a valid Julian, though there is a 32-bit limit.

- **julian_date**: Julian day to check

**Returns**: %TRUE if the Julian day is valid

**g_date_valid_weekday ()**

| gboolean | g_date_valid_weekday (GDateWeekday weekday); |

Returns **TRUE** if the weekday is valid. The seven GDateWeekday enumeration values are the only valid weekdays.

- **weekday**: weekday

**Returns**: %TRUE if the weekday is valid

---

### 4.8 Random Numbers

**Name**
Random Numbers – pseudo-random number generator

**Synopsis**

```
#include <glib.h>

GRand;
GRand* g_rand_new_with_seed (guint32 seed);
GRand* g_rand_new_with_seed_array (const guint32 *seed, guint seed_length);
GRand* g_rand_new (void);
GRand* g_rand_copy (GRand *rand_);
void g_rand_free (GRand *rand_);
void g_rand_set_seed (GRand *rand_, guint32 seed);
void g_rand_set_seed_array (GRand *rand_, const guint32 *seed, guint seed_length);
```
CHAPTER 4. GLIB UTILITIES

4.8. RANDOM NUMBERS

```c
#define g_rand_boolean (rand_)

Guint g_rand_int (GRand *rand_);
Gint g_rand_int_range (GRand *rand_,
Gint begin,
Gint end);

Gdouble g_rand_double (GRand *rand_);
Gdouble g_rand_double_range (GRand *rand_,
Gdouble begin,
Gdouble end);

void g_random_set_seed (Guint32 seed);
#define g_random_boolean ()
Guint g_random_int (void);
Gint g_random_int_range (Gint begin,
Gint end);
Gdouble g_random_double (void);
Gdouble g_random_double_range (Gdouble begin,
Gdouble end);
```

Description

The following functions allow you to use a portable, fast and good pseudo-random number generator (PRNG). It uses the Mersenne Twister PRNG, which was originally developed by Makoto Matsumoto and Takuji Nishimura. Further information can be found at www.math.keio.ac.jp/~matumoto/emt.html.

If you just need a random number, you simply call the `g_random_*` functions, which will create a globally used `GRand` and use the according `g_rand_*` functions internally. Whenever you need a stream of reproducible random numbers, you better create a `GRand` yourself and use the `g_rand_*` functions directly, which will also be slightly faster. Initializing a `GRand` with a certain seed will produce exactly the same series of random numbers on all platforms. This can thus be used as a seed for e.g. games.

The `g_rand*_range` functions will return high quality equally distributed random numbers, whereas for example the `(g_random_int()%max)` approach often doesn’t yield equally distributed numbers.

GLib changed the seeding algorithm for the pseudo-random number generator Mersenne Twister, as used by `GRand` and `GRandom`. This was necessary, because some seeds would yield very bad pseudo-random streams. Also the pseudo-random integers generated by `g_rand*_int_range()` will have a slightly better equal distribution with the new version of GLib.

The original seeding and generation algorithms, as found in GLib 2.0.x, can be used instead of the new ones by setting the environment variable `G_RANDOM_VERSION` to the value of ‘2.0’. Use the GLib-2.0 algorithms only if you have sequences of numbers generated with Glib-2.0 that you need to reproduce exactly.

Details

GRand

```c
typedef struct _GRand GRand;
```

The `GRand` struct is an opaque data structure. It should only be accessed through the `g_rand_*` functions.

`g_rand_new_with_seed()`

```c
GRand* g_rand_new_with_seed (Guint32 seed);
```

Creates a new random number generator initialized with `seed`.

`seed` : a value to initialize the random number generator.

`Returns` : the new `GRand`.
CHAPTER 4. GLIB UTILITIES

4.8. RANDOM NUMBERS

```
g_rand_new_with_seed_array()

GRand* g_rand_new_with_seed_array (const guint32* seed, guint seed_length);
```

Creates a new random number generator initialized with `seed`.

**seed**: an array of seeds to initialize the random number generator.

**seed_length**: an array of seeds to initialize the random number generator.

**Returns**: the new GRand.

Since 2.4

```
g_rand_new()

GRand* g_rand_new (void);
```

Creates a new random number generator initialized with a seed taken either from `/dev/urandom` (if existing) or from the current time (as a fallback).

**Returns**: the new GRand.

```
g_rand_copy()

GRand* g_rand_copy (GRand* rand_);
```

Copies a GRand into a new one with the same exact state as before. This way you can take a snapshot of the random number generator for replaying later.

**rand_**: a GRand.

**Returns**: the new GRand.

Since 2.4

```
g_rand_free()

void g_rand_free (GRand* rand_);
```

Frees the memory allocated for the GRand.

**rand_**: a GRand.

```
g_rand_set_seed()

void g_rand_set_seed (GRand* rand_, guint32 seed);
```

Sets the seed for the random number generator GRand to `seed`.

**rand_**: a GRand.

**seed**: a value to reinitialize the random number generator.
g_rand_set_seed_array ()

```c
void g_rand_set_seed_array (GRand *rand_,
    const guint32 *seed,
    guint seed_length);
```

Initializes the random number generator by an array of longs. Array can be of arbitrary size, though only the first 624 values are taken. This function is useful if you have many low entropy seeds, or if you require more then 32bits of actual entropy for your application.

*rand_*: a GRand.

*seed*: array to initialize with

*seed_length*: length of array

Since 2.4

---

g_rand_boolean()

```c
#define g_rand_boolean(rand_)
```

Returns a random gboolean from *rand_. This corresponds to a unbiased coin toss.

*rand_*: a GRand.

Returns: a random gboolean.

---

g_rand_int ()

```c
guint32 g_rand_int (GRand *rand_);
```

Returns the next random guint32 from *rand_ equally distributed over the range \([0..2^{32}-1]\).

*rand_*: a GRand.

Returns: A random number.

---

g_rand_int_range ()

```c
gint32 g_rand_int_range (GRand *rand_,
    gint32 begin,
    gint32 end);
```

Returns the next random gint32 from *rand_ equally distributed over the range \([\text{begin}..\text{end}-1]\).

*rand_*: a GRand.

*begin*: lower closed bound of the interval.

*end*: upper open bound of the interval.

Returns: A random number.

---

g_rand_double ()

```c
gdouble g_rand_double (GRand *rand_);
```

Returns the next random gdouble from *rand_ equally distributed over the range \([0..1]\).

*rand_*: a GRand.

Returns: A random number.
CHAPTER 4. GLIB UTILITIES 4.8. RANDOM NUMBERS

**g_rand_double_range ()**

```c
gdouble g_rand_double_range (GRand *rand_,
gdouble begin,
gdouble end);
```

Returns the next random **gdouble** from **rand_** equally distributed over the range **[begin..end]**.

*rand_*: a **GRand**.

*begin*: lower closed bound of the interval.

*end*: upper open bound of the interval.

**Returns**: A random number.

**g_random_set_seed ()**

```c
void g_random_set_seed (guint32 seed);
```

Sets the seed for the global random number generator, which is used by the **g_random_*** functions, to **seed**.

*seed*: a value to reinitialize the global random number generator.

**g_random_boolean()**

```c
#define g_random_boolean()
```

Returns a random **gboolean**. This corresponds to a unbiased coin toss.

**Returns**: a random **gboolean**.

**g_random_int ()**

```c
guint32 g_random_int (void);
```

Return a random **guint32** equally distributed over the range **[0..2^32-1]**.

**Returns**: A random number.

**g_random_int_range ()**

```c
gint32 g_random_int_range (gint32 begin,
gint32 end);
```

Returns a random **gint32** equally distributed over the range **[begin..end-1]**.

*begin*: lower closed bound of the interval.

*end*: upper open bound of the interval.

**Returns**: A random number.

**g_random_double ()**

```c
gdouble g_random_double (void);
```

Returns a random **gdouble** equally distributed over the range **[0..1)**.

**Returns**: A random number.
CHAPTER 4. GLIB UTILITIES

4.9 Hook Functions

Name
Hook Functions – support for manipulating lists of hook functions

Synopsis

```c
#include <glib.h>

GHookList;
void (*GHookFinalizeFunc)(GHookList *hook_list, GHook *hook);
GHook;
void (*GHookFunc)(gpointer data);
gboolean (*GHookCheckFunc)(gpointer data);
void g_hook_list_init(GHookList *hook_list, guint hook_size);
void g_hook_list_invoke(GHookList *hook_list, gboolean may_recurse);
void g_hook_list_invoke_check(GHookList *hook_list, gboolean may_recurse);
void g_hook_list_marshal(GHookList *hook_list, gboolean may_recurse, GHookMarshaller marshaller, gpointer marshal_data);
void (*GHookMarshaller)(GHook *hook, gpointer marshal_data);
void g_hook_list_marshal_check(GHookList *hook_list, gboolean may_recurse, GHookCheckMarshaller marshaller, gpointer marshal_data);
gboolean (*GHookCheckMarshaller)(GHook *hook, gpointer marshal_data);
void g_hook_list_clear(GHookList *hook_list);
GHook* g_hook_alloc(GHookList *hook_list);
#define g_hook_append (hook_list, hook)
void g_hook_prepend(GHookList *hook_list, GHook *hook);
void g_hook_insert_before(GHookList *hook_list, GHook *sibling, GHook *hook);
void g_hook_insert_sorted(GHookList *hook_list, GHook *hook);
```
CHAPTER 4. GLIB UTILITIES

4.9. HOOK FUNCTIONS

The GHookList, GHook and their related functions provide support for lists of hook functions. Functions can be added and removed from the lists, and the list of hook functions can be invoked.

```c
GHookCompareFunc func);

Gint ( *GHookCompareFunc) (GHook *new_hook,
GHook *sibling);

Gint g_hook_compare_ids (GHook *new_hook,
GHook *sibling);

GHook* g_hook_get (GHookList *hook_list,
gulong hook_id);

GHook* g_hook_find (GHookList *hook_list,
gboolean need_valids,
GHookFindFunc func,
gpointer data);

gboolean ( *GHookFindFunc) (GHook *hook,
gpointer data);

GHook* g_hook_find_data (GHookList *hook_list,
gboolean need_valids,
gpointer data);

GHook* g_hook_find_func (GHookList *hook_list,
gboolean need_valids,
gpointer func);

GHook* g_hook_find_func_data (GHookList *hook_list,
gboolean need_valids,
gpointer func,
gpointer data);

GHook* g_hook_first_valid (GHookList *hook_list,
gboolean may_be_in_call);

GHook* g_hook_next_valid (GHookList *hook_list,
GHook *hook,
gboolean may_be_in_call);

enum GHookFlagMask;
#define G_HOOK_FLAGS (hook)
#define G_HOOK_FLAG_USER_SHIFT
#define G_HOOK (hook)
#define G_HOOK_IS_VALID (hook)
#define G_HOOK_ACTIVE (hook)
#define G_HOOK_IN_CALL (hook)
#define G_HOOK_IS_UNLINKED (hook)

GHook * g_hook_ref (GHookList *hook_list,
GHook *hook);

void g_hook_unref (GHookList *hook_list,
GHook *hook);

void g_hook_free (GHookList *hook_list,
GHook *hook);

gboolean g_hook_destroy (GHookList *hook_list,
gulong hook_id);

void g_hook_destroy_link (GHookList *hook_list,
GHook *hook);
```

Description

The GHookList, GHook and their related functions provide support for lists of hook functions. Functions can be added and removed from the lists, and the list of hook functions can be invoked.
Details

**GHookList**

typedef struct {
  gulong seq_id;
  guint hook_size : 16;
  guint is_setup : 1;
  GHook *hooks;
  gpointer dummy3;
  GHookFinalizeFunc finalize_hook;
  gpointer dummy[2];
} GHookList;

The GHookList struct represents a list of hook functions.

- **gulong seq_id**: the next free GHook id
- **guint hook_size**: the size of the GHookList elements, in bytes
- **guint is_setup**: 1; if the GHookList has been initialized
- **GHook *hooks**: the first GHook element in the list
- **gpointer dummy3**: unused
- **GHookFinalizeFunc finalize_hook**: the function to call to finalize a GHook element. The default behaviour is to call the hooks destroy function
- **gpointer dummy[2]**: unused

GHookFinalizeFunc ()

void (*GHookFinalizeFunc) (GHookList *hook_list, GHook *hook);

Defines the type of function to be called when a hook in a list of hooks gets finalized.

- **hook_list**: a GHookList
- **hook**: the hook in hook_list that gets finalized

**GHook**

typedef struct {
  gpointer data;
  GHook *next;
  GHook *prev;
  guint ref_count;
  gulong hook_id;
  guint flags;
  gpointer func;
  GDestroyNotify destroy;
} GHook;

The GHook struct represents a single hook function in a GHookList.

- **gpointer data**: data which is passed to func when this hook is invoked
- **GHook *next**: pointer to the next hook in the list
- **GHook *prev**: pointer to the previous hook in the list
- **guint ref_count**: the reference count of this hook
- **gulong hook_id**: the id of this hook, which is unique within its list
CHAPTER 4. GLIB UTILITIES 4.9. HOOK FUNCTIONS

**guint flags;** flags which are set for this hook. See `GHookFlagMask` for predefined flags.

**gpointer func;** the function to call when this hook is invoked. The possible signatures for this function are `GHookFunc` and `GHookCheckFunc`.

**GDestroyNotify destroy;** the default `finalize_hook` function of a `GHookList` calls this member of the hook that is being finalized.

**GHookFunc ()**

```c
void (GHookFunc)(gpointer data);
```

Defines the type of a hook function that can be invoked by `g_hook_list_invoke()`. 

**data:** the data field of the `GHook` is passed to the hook function here.

**GHookCheckFunc ()**

```c
gboolean (GHookCheckFunc)(gpointer data);
```

Defines the type of a hook function that can be invoked by `g_hook_list_invoke_check()`. 

**data:** the data field of the `GHook` is passed to the hook function here.

**Returns:** %FALSE if the `GHook` should be destroyed.

**g_hook_list_init ()**

```c
void g_hook_list_init (GHookList *hook_list, guint hook_size);
```

Initializes a `GHookList`. This must be called before the `GHookList` is used. 

**hook_list:** a `GHookList`

**hook_size:** the size of each element in the `GHookList`, typically `sizeof(GHook)`.

**g_hook_list_invoke ()**

```c
void g_hook_list_invoke (GHookList *hook_list, gboolean may_recurse);
```

Calls all of the `GHook` functions in a `GHookList`. 

**hook_list:** a `GHookList`

**may_recurse:** %TRUE if functions which are already running (e.g. in another thread) can be called. If set to FALSE, these are skipped.

**g_hook_list_invoke_check ()**

```c
void g_hook_list_invoke_check (GHookList *hook_list, gboolean may_recurse);
```

Calls all of the `GHook` functions in a `GHookList`. Any function which returns FALSE is removed from the `GHookList`. 

**hook_list:** a `GHookList`

**may_recurse:** %TRUE if functions which are already running (e.g. in another thread) can be called. If set to FALSE, these are skipped.
**g_hook_list_marshal()**

```c
void g_hook_list_marshal (GHookList *hook_list, gboolean may_recurse, GHookMarshaller marshaller, gpointer marshal_data);
```

Calls a function on each valid GHook.

- **hook_list**: a GHookList
- **may_recurse**: %TRUE if hooks which are currently running (e.g. in another thread) are considered valid. If set to FALSE, these are skipped
- **marshaller**: the function to call for each GHook
- **marshal_data**: data to pass to marshaller

**GHookMarshaller()**

```c
void (*GHookMarshaller) (GHook *hook, gpointer marshal_data);
```

Defines the type of function used by **g_hook_list_marshal()**.

- **hook**: a GHook
- **marshal_data**: user data

**g_hook_list_marshal_check()**

```c
void g_hook_list_marshal_check (GHookList *hook_list, gboolean may_recurse, GHookCheckMarshaller marshaller, gpointer marshal_data);
```

Calls a function on each valid GHook and destroys it if the function returns FALSE.

- **hook_list**: a GHookList
- **may_recurse**: %TRUE if hooks which are currently running (e.g. in another thread) are considered valid. If set to FALSE, these are skipped
- **marshaller**: the function to call for each GHook
- **marshal_data**: data to pass to marshaller

**GHookCheckMarshaller()**

```c
gboolean (*GHookCheckMarshaller) (GHook *hook, gpointer marshal_data);
```

Defines the type of function used by **g_hook_list_marshal_check()**.

- **hook**: a GHook
- **marshal_data**: user data

**Returns**: %FALSE if hook should be destroyed
CHAPTER 4. GLIB UTILITIES 4.9. HOOK FUNCTIONS

g_hook_list_clear()

```c
void g_hook_list_clear (GHookList *hook_list);
```

Removes all the GHook elements from a GHookList.

**hook_list**: a GHookList

**Returns**: a new GHook

**g_hook_alloc()**

```c
GHook* g_hook_alloc (GHookList *hook_list);
```

Allocates space for a GHook and initializes it.

**hook_list**: a GHookList

**Returns**: a new GHook

**g_hook_append()**

```c
#define g_hook_append( hook_list, hook )
```

Appends a GHook onto the end of a GHookList.

**hook_list**: a GHookList

**hook**: the GHook to add to the end of **hook_list**

**g_hook_prepend()**

```c
void g_hook_prepend (GHookList *hook_list, GHook *hook);
```

Prepends a GHook on the start of a GHookList.

**hook_list**: a GHookList

**hook**: the GHook to add to the start of **hook_list**

**g_hook_insert_before()**

```c
void g_hook_insert_before (GHookList *hook_list, GHook *sibling, GHook *hook);
```

Inserts a GHook into a GHookList, before a given GHook.

**hook_list**: a GHookList

**sibling**: the GHook to insert the new GHook before

**hook**: the GHook to insert

**g_hook_insert_sorted()**

```c
void g_hook_insert_sorted (GHookList *hook_list, GHook *hook, GHookCompareFunc func);
```

Inserts a GHook into a GHookList, sorted by the given function.

**hook_list**: a GHookList

**hook**: the GHook to insert

**func**: the comparison function used to sort the GHook elements
CHAPTER 4. GLIB UTILITIES 4.9. HOOK FUNCTIONS

GHookCompareFunc ()

```c
int (GHookCompareFunc) (GHook *new_hook, GHook *sibling);
```

Defines the type of function used to compare GHook elements in `g_hook_insert_sorted()`.

- **new_hook**: the GHook being inserted
- **sibling**: the GHook to compare with `new_hook`
- **Returns**: a value <= 0 if `new_hook` should be before `sibling`

**g_hook_compare_ids ()**

```c
int g_hook_compare_ids (GHook *new_hook, GHook *sibling);
```

Compares the ids of two GHook elements, returning a negative value if the second id is greater than the first.

- **new_hook**: a GHook
- **sibling**: a GHook to compare with `new_hook`
- **Returns**: a value <= 0 if the id of `sibling` is >= the id of `new_hook`

**g_hook_get ()**

```c
GHook* g_hook_get (GHookList *hook_list, gulong hook_id);
```

Returns the GHook with the given id, or NULL if it is not found.

- **hook_list**: a GHookList
- **hook_id**: a hook id
- **Returns**: the GHook with the given id, or NULL if it is not found

**g_hook_find ()**

```c
GHook* g_hook_find (GHookList *hook_list, gboolean need_valids, GHookFindFunc func, gpointer data);
```

Finds a GHook in a GHookList using the given function to test for a match.

- **hook_list**: a GHookList
- **need_valids**: %TRUE if GHook elements which have been destroyed should be skipped
- **func**: the function to call for each GHook, which should return TRUE when the GHook has been found
- **data**: the data to pass to `func`
- **Returns**: the found GHook or NULL if no matching GHook is found
CHAPTER 4. GLIB UTILITIES 4.9. HOOK FUNCTIONS

GHookFindFunc

```c
gboolean ( *GHookFindFunc) (GHook *hook,
gpointer data);
```

Defines the type of the function passed to `g_hook_find()`.

**hook**: a GHook

**data**: user data passed to `g_hook_find_func()`

**Returns**: %TRUE if the required GHook has been found

G_hook_find_data

```c
GHook* g_hook_find_data (GHookList *hook_list,
gboolean need_valids,
gpointer data);
```

Finds a GHook in a GHookList with the given data.

**hook_list**: a GHookList

**need_valids**: %TRUE if GHook elements which have been destroyed should be skipped

**data**: the data to find

**Returns**: the GHook with the given data or NULL if no matching GHook is found

G_hook_find_func

```c
GHook* g_hook_find_func (GHookList *hook_list,
gboolean need_valids,
gpointer func);
```

Finds a GHook in a GHookList with the given function.

**hook_list**: a GHookList

**need_valids**: %TRUE if GHook elements which have been destroyed should be skipped

**func**: the function to find

**Returns**: the GHook with the given func or NULL if no matching GHook is found

G_hook_find_func_data

```c
GHook* g_hook_find_func_data (GHookList *hook_list,
gboolean need_valids,
gpointer func,
gpointer data);
```

Finds a GHook in a GHookList with the given function and data.

**hook_list**: a GHookList

**need_valids**: %TRUE if GHook elements which have been destroyed should be skipped

**func**: the function to find

**data**: the data to find

**Returns**: the GHook with the given func and data or NULL if no matching GHook is found
g_hook_first_valid ()

Returns the first GHook in a GHookList which has not been destroyed. The reference count for the GHook is incremented, so you must call g_hook_unref() to restore it when no longer needed. (Or call g_hook_next_valid() if you are stepping through the GHookList.)

hook_list: a GHookList

may_be_in_call: %TRUE if hooks which are currently running (e.g. in another thread) are considered valid. If set to FALSE, these are skipped

Returns: the first valid GHook, or NULL if none are valid

g_hook_next_valid ()

Returns the next GHook in a GHookList which has not been destroyed. The reference count for the GHook is incremented, so you must call g_hook_unref() to restore it when no longer needed. (Or continue to call g_hook_next_valid() until NULL is returned.)

hook_list: a GHookList

hook: the current GHook

may_be_in_call: %TRUE if hooks which are currently running (e.g. in another thread) are considered valid. If set to FALSE, these are skipped

Returns: the next valid GHook, or NULL if none are valid

defined

typedef enum

| G_HOOK_FLAG_ACTIVE    | = 1 << 0,         |
| G_HOOK_FLAG_IN_CALL   | = 1 << 1,         |
| G_HOOK_FLAG_MASK      | = 0x0f           |
} GHookFlagMask;

Flags used internally in the GHook implementation.

G_HOOK_FLAG_ACTIVE set if the hook has not been destroyed

G_HOOK_FLAG_IN_CALL set if the hook is currently being run

G_HOOK_FLAG_MASK A mask covering all bits reserved for hook flags; see G_HOOK_FLAGS_USER_SHIFT

G_HOOK_FLAGS()

#define G_HOOK_FLAGS(hook) (G_HOOK (hook)->flags)

Returns the flags of a hook.

hook: a GHook
#define G_HOOK_FLAG_USER_SHIFT (4)

The position of the first bit which is not reserved for internal use be the GHook implementation, i.e. 1 << G_HOOK_FLAG_USER_SHIFT is the first bit which can be used for application-defined flags.

G_HOOK()

#define G_HOOK(hook) ((GHook*) (hook))

Casts a pointer to a GHook*.

hook: a pointer

G_HOOK_IS_VALID()

#define G_HOOK_IS_VALID(hook)

Returns TRUE if the GHook is valid, i.e. it is in a GHookList, it is active and it has not been destroyed.

hook: a GHook

Returns: %TRUE if the GHook is valid

G_HOOK_ACTIVE()

#define G_HOOK_ACTIVE(hook)

Returns TRUE if the GHook is active, which is normally TRUE until the GHook is destroyed.

hook: a GHook

Returns: %TRUE if the GHook is active

G_HOOK_IN_CALL()

#define G_HOOK_IN_CALL(hook)

Returns TRUE if the GHook function is currently executing.

hook: a GHook

Returns: %TRUE if the GHook function is currently executing

G_HOOK_IS_UNLINKED()

#define G_HOOK_IS_UNLINKED(hook)

Returns TRUE if the GHook is not in a GHookList.

hook: a GHook

Returns: %TRUE if the GHook is not in a GHookList

g_hook_ref()

GHook* g_hook_ref (GHookList* hook_list, GHook* hook);

Increments the reference count for a GHook.

hook_list: a GHookList

hook: the GHook to increment the reference count of

Returns: the hook that was passed in (since 2.6)
### 4.10 Miscellaneous Utility Functions

#### g_hook_unref()

| void | g_hook_unref | (GHookList *hook_list, GHook *hook); |

Decrements the reference count of a GHook. If the reference count falls to 0, the GHook is removed from the GHookList and g_hook_free() is called to free it.

**hook_list**: a GHookList

**hook**: the GHook to unref

#### g_hook_free()

| void | g_hook_free | (GHookList *hook_list, GHook *hook); |

Calls the GHookList finalize_hook function if it exists, and frees the memory allocated for the GHook.

**hook_list**: a GHookList

**hook**: the GHook to free

#### g_hook_destroy()

| gboolean | g_hook_destroy | (GHookList *hook_list, gulong hook_id); |

Destroys a GHook, given its ID.

**hook_list**: a GHookList

**hook_id**: a hook ID

**Returns**: %TRUE if the GHook was found in the GHookList and destroyed

#### g_hook_destroy_link()

| void | g_hook_destroy_link | (GHookList *hook_list, GHook *hook); |

Removes one GHook from a GHookList, marking it inactive and calling g_hook_unref() on it.

**hook_list**: a GHookList

**hook**: the GHook to remove

### 4.10 Miscellaneous Utility Functions

**Name**

Miscellaneous Utility Functions – a selection of portable utility functions
Synopsis

#include <glib.h>

const gchar* g_get_application_name (void);
void g_set_application_name (const gchar *application_name);
gchar* g_get_prgname (void);
void g_set_prgname (const gchar *prgname);
const gchar* g_getenv (const gchar *variable);
gchar* g_get_prgname (void);
void g_set_prgname (const gchar *prgname);
const gchar* g_getenv (const gchar *variable);
gboolean g_setenv (const gchar *variable, const gchar *value, gboolean overwrite);
gchar** g_listenv (void);
g_unsetenv (const gchar *variable);
void g_unsetenv (void);
gchar* g_get_user_name (void);
g_get_user_name (void);
g_get_user_cache_dir (void);
g_get_user_data_dir (void);
g_get_user_config_dir (void);
enum GUserDirectory;
g_get_user_special_dir (GUserDirectory directory);
const gchar* const* g_get_system_data_dirs (void);
const gchar* const* g_get_system_config_dirs (void);

g_get_host_name (void);
g_get_home_dir (void);
g_get_tmp_dir (void);
g_get_current_dir (void);
g_get_basename (const gchar *file_name);
const gchar* g_get_user_name (void);
const gchar* g_get_real_name (void);
const gchar* g_get_user_cache_dir (void);
const gchar* g_get_user_data_dir (void);
const gchar* g_get_user_config_dir (void);
const gchar* g_get_user_special_dir (GUserDirectory directory);
const gchar* const* g_get_system_data_dirs (void);
const gchar* const* g_get_system_config_dirs (void);
g_get_host_name (void);
g_get_user_name (void);
g_get_real_name (void);
g_get_user_cache_dir (void);
g_get_user_data_dir (void);
g_get_user_config_dir (void);
g_get_user_special_dir (GUserDirectory directory);
g_get_system_data_dirs (void);
g_get_system_config_dirs (void);

g_path_is_absolute (const gchar *file_name);
g_path_skip_root (const gchar *file_name);
g_path_get_basename (const gchar *file_name);
g_path_get_dirname (const gchar *file_name);
g_build_filename (const gchar *first_element, ...);
g_build_filenamev (gchar **args);
g_build_path (const gchar *separator, const gchar *first_element, ...);
g_build_pathv (const gchar *separator, gchar **args);
g_format_size_for_display (goffset size);
g_find_program_in_path (const gchar *program);
g_bit_nth_lsf (gulong mask, gint nth_bit);
g_bit_nth_msf (gulong mask, gint nth_bit);
g_bit_storage (gulong number);
g_spaced_primes_closest (guint num);
g_atexit (GVoidFunc func);
g_parse_debug_string (const gchar *string, const GDebugKey *keys,
CHAPTER 4. GLIB UTILITIES

4.10. MISCELLANEOUS UTILITY FUNCTIONS

Guint nkeys);

void (*GVoidFunc);

void (*GFreeFunc)

void g_qsort_with_data

(const gpointer pbase,
 gint total_elems,
 gsize size,
 GCompareDataFunc compare_func,
 gpointer user_data);

void g_nullify_pointer

(gpointer *nullify_location);

Description
These are portable utility functions.

Details

g_get_application_name()

const gchar* g_get_application_name (void);

Gets a human-readable name for the application, as set by g_set_application_name(). This name
should be localized if possible, and is intended for display to the user. Contrast with g_get_prgrname(),
which gets a non-localized name. If g_set_application_name() has not been called, returns the result of
g_get_prgrname() (which may be NULL if g_set_prgrname() has also not been called).

Returns: human-readable application name. may return NULL
Since 2.2

g_set_application_name()

void g_set_application_name (const gchar *

application_name);

Sets a human-readable name for the application. This name should be localized if possible, and
is intended for display to the user. Contrast with g_set_prgrname(), which sets a non-localized name.
g_set_prgrname() will be called automatically by gtk_init(), but g_set_application_name() will not.
Note that for thread safety reasons, this function can only be called once.
The application name will be used in contexts such as error messages, or when displaying an applica-
tion’s name in the task list.

application_name: localized name of the application
Since 2.2

g_get_prgrname()

gchar* g_get_prgrname (void);

Gets the name of the program. This name should not be localized, contrast with g_get_application_name().
(If you are using GDK or GTK+ the program name is set in gdk_init(), which is called by gtk_init(). The
program name is found by taking the last component of argv[0].)

Returns: the name of the program. The returned string belongs to GLib and must not be modified or
freed.
**g_set_prgname ()**

```c
void g_set_prgname (const gchar *prgname);
```

Sets the name of the program. This name should not be localized, contrast with `g_set_application_name()`.

Note that for thread-safety reasons this function can only be called once.

**prgname**: the name of the program.

**g_getenv ()**

```c
const gchar* g_getenv (const gchar *variable);
```

Returns the value of an environment variable. The name and value are in the GLib file name encoding. On UNIX, this means the actual bytes which might or might not be in some consistent character set and encoding. On Windows, it is in UTF-8. On Windows, in case the environment variable’s value contains references to other environment variables, they are expanded.

**variable**: the environment variable to get, in the GLib file name encoding.

**Returns**: the value of the environment variable, or NULL if the environment variable is not found. The returned string may be overwritten by the next call to `g_getenv()`, `g_setenv()` or `g_unsetenv()`.

**g_setenv ()**

```c
gboolean g_setenv (const gchar *variable, const gchar *value, gboolean overwrite);
```

Sets an environment variable. Both the variable’s name and value should be in the GLib file name encoding. On UNIX, this means that they can be any sequence of bytes. On Windows, they should be in UTF-8.

Note that on some systems, when variables are overwritten, the memory used for the previous variables and its value isn’t reclaimed.

**variable**: the environment variable to set, must not contain ‘=’.

**value**: the value for to set the variable to.

**overwrite**: whether to change the variable if it already exists.

**Returns**: FALSE if the environment variable couldn’t be set.

Since 2.4

**g_unsetenv ()**

```c
void g_unsetenv (const gchar *variable);
```

Removes an environment variable from the environment.

Note that on some systems, when variables are overwritten, the memory used for the previous variables and its value isn’t reclaimed. Furthermore, this function can’t be guaranteed to operate in a thread-safe way.

**variable**: the environment variable to remove, must not contain ‘=’.

Since 2.4
CHAPTER 4. GLIB UTILITIES

4.10. MISCELLANEOUS UTILITY FUNCTIONS

**g_listenv()**

```c
 gchar** g_listenv (void);
```

Gets the names of all variables set in the environment.

**Returns**: a NULL-terminated list of strings which must be freed with `g_strfreev()`. Programs that want to be portable to Windows should typically use this function and `g_getenv()` instead of using the `environ` array from the C library directly. On Windows, the strings in the `environ` array are in system codepage encoding, while in most of the typical use cases for environment variables in GLib-using programs you want the UTF-8 encoding that this function and `g_getenv()` provide.

Since 2.8

**g_get_user_name()**

```c
 const gchar* g_get_user_name (void);
```

Gets the user name of the current user. The encoding of the returned string is system-defined. On UNIX, it might be the preferred file name encoding, or something else, and there is no guarantee that it is even consistent on a machine. On Windows, it is always UTF-8.

**Returns**: the user name of the current user.

**g_get_real_name()**

```c
 const gchar* g_get_real_name (void);
```

Gets the real name of the user. This usually comes from the user’s entry in the `passwd` file. The encoding of the returned string is system-defined. (On Windows, it is, however, always UTF-8.) If the real user name cannot be determined, the string "Unknown" is returned.

**Returns**: the user’s real name.

**g_get_user_cache_dir()**

```c
 const gchar* g_get_user_cache_dir (void);
```

Returns a base directory in which to store non-essential, cached data specific to particular user.

On UNIX platforms this is determined using the mechanisms described in the XDG Base Directory Specification

**Returns**: a string owned by GLib that must not be modified or freed.

Since 2.6

**g_get_user_data_dir()**

```c
 const gchar* g_get_user_data_dir (void);
```

Returns a base directory in which to access application data such as icons that is customized for a particular user.

On UNIX platforms this is determined using the mechanisms described in the XDG Base Directory Specification

**Returns**: a string owned by GLib that must not be modified or freed.

Since 2.6
**g_get_user_config_dir**()  

```c
const gchar* g_get_user_config_dir (void);
```

Returns a base directory in which to store user-specific application configuration information such as user preferences and settings.  
On UNIX platforms this is determined using the mechanisms described in the XDG Base Directory Specification

**Returns:** a string owned by GLib that must not be modified or freed.

Since 2.6

**enum GUserDirectory**

```c
typedef enum {
    G_USER_DIRECTORY_DESKTOP,
    G_USER_DIRECTORY_DOCUMENTS,
    G_USER_DIRECTORY_DOWNLOAD,
    G_USER_DIRECTORY_MUSIC,
    G_USER_DIRECTORY_PICTURES,
    G_USER_DIRECTORY_PUBLIC_SHARE,
    G_USER_DIRECTORY_TEMPLATES,
    G_USER_DIRECTORY_VIDEOS,
    G_USER_N_DIRECTORIES
} GUserDirectory;
```

These are logical ids for special directories which are defined depending on the platform used. You should use `g_get_user_special_dir()` to retrieve the full path associated to the logical id.

The GUserDirectory enumeration can be extended at later date. Not every platform has a directory for every logical id in this enumeration.

**G_USER_DIRECTORY_DESKTOP** the user’s Desktop directory  
**G_USER_DIRECTORY_DOCUMENTS** the user’s Documents directory  
**G_USER_DIRECTORY_DOWNLOAD** the user’s Downloads directory  
**G_USER_DIRECTORY_MUSIC** the user’s Music directory  
**G_USER_DIRECTORY_PICTURES** the user’s Pictures directory  
**G_USER_DIRECTORY_PUBLIC_SHARE** the user’s shared directory  
**G_USER_DIRECTORY_TEMPLATES** the user’s Templates directory  
**G_USER_DIRECTORY_VIDEOS** the user’s Movies directory  
**G_USER_N_DIRECTORIES** the number of enum values

Since 2.14

**g_get_user_special_dir**()  

```c
const gchar* g_get_user_special_dir (GUserDirectory directory);
```

Returns the full path of a special directory using its logical id.  
On Unix this is done using the XDG special user directories. For compatibility with existing practise, **G_USER_DIRECTORY_DESKTOP** falls back to `$HOME/Desktop` when XDG special user directories have not been set up.

Depending on the platform, the user might be able to change the path of the special directory without requiring the session to restart; GLib will not reflect any change once the special directories are loaded.
directory: the logical id of special directory

Returns: the path to the specified special directory, or NULL if the logical id was not found. The returned string is owned by GLib and should not be modified or freed.

Since 2.14

g_get_system_data_dirs ()

const gchar* const * g_get_system_data_dirs (void);

Returns an ordered list of base directories in which to access system-wide application data. On UNIX platforms this is determined using the mechanisms described in the XDG Base Directory Specification.

On Windows the first elements in the list are the Application Data and Documents folders for All Users. (These can be determined only on Windows 2000 or later and are not present in the list on other Windows versions.) See documentation for CSIDL_COMMON_APPDATA and CSIDL_COMMON_DOCUMENTS.

Then follows the "share" subfolder in the installation folder for the package containing the DLL that calls this function, if it can be determined.

Finally the list contains the "share" subfolder in the installation folder for GLib, and in the installation folder for the package the application's .exe file belongs to.

The installation folders above are determined by looking up the folder where the module (DLL or EXE) in question is located. If the folder's name is "bin", its parent is used, otherwise the folder itself.

Note that on Windows the returned list can vary depending on where this function is called.

Returns: a NULL-terminated array of strings owned by GLib that must not be modified or freed.

Since 2.6

g_get_system_config_dirs ()

const gchar* const * g_get_system_config_dirs (void);

Returns an ordered list of base directories in which to access system-wide configuration information. On UNIX platforms this is determined using the mechanisms described in the XDG Base Directory Specification.

Returns: a NULL-terminated array of strings owned by GLib that must not be modified or freed.

Since 2.6

g_get_host_name ()

const gchar* g_get_host_name (void);

Return a name for the machine.

The returned name is not necessarily a fully-qualified domain name, or even present in DNS or some other name service at all. It need not even be unique on your local network or site, but usually it is. Callers should not rely on the return value having any specific properties like uniqueness for security purposes. Even if the name of the machine is changed while an application is running, the return value from this function does not change. The returned string is owned by GLib and should not be modified or freed. If no name can be determined, a default fixed string "localhost" is returned.

Returns: the host name of the machine.

Since 2.8
4.10. MISCELLANEOUS UTILITY FUNCTIONS

**g_get_home_dir()**

```c
const gchar* g_get_home_dir (void);
```

Gets the current user’s home directory as defined in the password database. Note that in contrast to traditional UNIX tools, this function prefers `passwd` entries over the `HOME` environment variable.

One of the reasons for this decision is that applications in many cases need special handling to deal with the case where `HOME` is:
- Not owned by the user
- Not writeable
- Not even readable

Since applications are in general not written to deal with these situations it was considered better to make `g_get_home_dir()` not pay attention to `HOME` and to return the real home directory for the user. If applications want to pay attention to `HOME`, they can do:

```c
const char *homedir = g_getenv ("HOME");
if (!homedir)
    homedir = g_get_home_dir ();
```

**Returns:** the current user’s home directory

**g_get_tmp_dir()**

```c
const gchar* g_get_tmp_dir (void);
```

Gets the directory to use for temporary files. This is found from inspecting the environment variables `TMPDIR`, `TMP`, and `TEMP` in that order. If none of those are defined `/tmp` is returned on UNIX and "C:" on Windows. The encoding of the returned string is system-defined. On Windows, it is always UTF-8. The return value is never NULL.

**Returns:** the directory to use for temporary files.

**g_get_current_dir()**

```c
gchar* g_get_current_dir (void);
```

Gets the current directory. The returned string should be freed when no longer needed. The encoding of the returned string is system defined. On Windows, it is always UTF-8.

**Returns:** the current directory.

**g_basename()**

```c
const gchar* g_basename (const gchar *file_name);
```

**WARNING**

`g_basename` has been deprecated since version 2.2 and should not be used in newly-written code. Use `g_path_get_basename()` instead, but notice that `g_path_get_basename()` allocates new memory for the returned string, unlike this function which returns a pointer into the argument.

Gets the name of the file without any leading directory components. It returns a pointer into the given file name string.

**file_name:** the name of the file.

**Returns:** the name of the file without any leading directory components.
CHAPTER 4. GLIB UTILITIES

4.10. MISCELLANEOUS UTILITY FUNCTIONS

\texttt{g dirname}

\begin{verbatim}
#define g dirname
\end{verbatim}

\textbf{WARNING}

\begin{verbatim}
\texttt{g dirname} is deprecated and should not be used in newly-written code.
\end{verbatim}

This function is deprecated and will be removed in the next major release of GLib. Use \texttt{g path get dirname()} instead.

\textbf{Gets the directory components of a file name. If the file name has no directory components "." is returned. The returned string should be freed when no longer needed.}

\textbf{Returns} : the directory components of the file.

\textbf{g path is absolute ()}

\begin{verbatim}
g gboolean g path is absolute (const gchar *file name);
\end{verbatim}

Returns \texttt{TRUE} if the given \texttt{file name} is an absolute file name, i.e. it contains a full path from the root directory such as "/usr/local" on UNIX or "C:\windows" on Windows systems.

\textbf{file name} : a file name.

\textbf{Returns} : \texttt{TRUE} if \texttt{file name} is an absolute path.

\textbf{g path skip root ()}

\begin{verbatim}
g const gchar* g path skip root (const gchar *file name);
\end{verbatim}

Returns a pointer into \texttt{file name} after the root component, i.e. after the "/" in UNIX or "C:\" under Windows. If \texttt{file name} is not an absolute path it returns \texttt{NULL}.

\textbf{file name} : a file name.

\textbf{Returns} : a pointer into \texttt{file name} after the root component.

\textbf{g path get basename ()}

\begin{verbatim}
g gchar* g path get basename (const gchar *file name);
\end{verbatim}

Gets the last component of the filename. If \texttt{file name} ends with a directory separator it gets the component before the last slash. If \texttt{file name} consists only of directory separators (and on Windows, possibly a drive letter), a single separator is returned. If \texttt{file name} is empty, it gets ".".

\textbf{file name} : the name of the file.

\textbf{Returns} : a newly allocated string containing the last component of the filename.

\textbf{g path get dirname ()}

\begin{verbatim}
g gchar* g path get dirname (const gchar *file name);
\end{verbatim}

Gets the directory components of a file name. If the file name has no directory components "." is returned. The returned string should be freed when no longer needed.

\textbf{file name} : the name of the file.

\textbf{Returns} : the directory components of the file.
CHAPTER 4. GLIB UTILITIES

4.10. MISCELLANEOUS UTILITY FUNCTIONS

\textbf{g\_build\_filename (0)}

\begin{Verbatim}
gchar * g_build_filename (const gchar * first_element, ...
\end{Verbatim}

Creates a filename from a series of elements using the correct separator for filenames.
On Unix, this function behaves identically to \texttt{g\_build\_path (G\_DIR\_SEPARATOR\_S, first\_element, ...)}.
On Windows, it takes into account that either the backslash (\ or slash (/) can be used as separator
in filenames, but otherwise behaves as on Unix. When file pathname separators need to be inserted, the
one that last previously occurred in the parameters (reading from left to right) is used.
No attempt is made to force the resulting filename to be an absolute path. If the first element is a
relative path, the result will be a relative path.

\textit{first\_element}: the first element in the path
\textit{...}: remaining elements in path, terminated by NULL

\textbf{Returns}: a newly-allocated string that must be freed with \texttt{g\_free()}. 

\textbf{g\_build\_filenamev (0)}

\begin{Verbatim}
gchar * g_build_filenamev (gchar **args);
\end{Verbatim}

Behaves exactly like \texttt{g\_build\_filename()}, but takes the path elements as a string array, instead of
varargs. This function is mainly meant for language bindings.

\textit{args}: NULL-terminated array of strings containing the path elements.

\textbf{Returns}: a newly-allocated string that must be freed with \texttt{g\_free()}. 

Since 2.8

\textbf{g\_build\_path (0)}

\begin{Verbatim}
gchar * g_build_path (const gchar *separator, const gchar * first_element, ...
\end{Verbatim}

Creates a path from a series of elements using \texttt{separator} as the separator between elements. At the
boundary between two elements, any trailing occurrences of separator in the first element, or leading
occurrences of separator in the second element are removed and exactly one copy of the separator is
inserted.
Empty elements are ignored.
The number of leading copies of the separator on the result is the same as the number of leading
copies of the separator on the first non-empty element.
The number of trailing copies of the separator on the result is the same as the number of trailing
copies of the separator on the last non-empty element. (Determination of the number of trailing copies
is done without stripping leading copies, so if the separator is ABA, ABABA has 1 trailing copy.)
However, if there is only a single non-empty element, and there are no characters in that element not
part of the leading or trailing separators, then the result is exactly the original value of that element.
Other than for determination of the number of leading and trailing copies of the separator, elements
consisting only of copies of the separator are ignored.

\textit{separator}: a string used to separator the elements of the path.

\textit{first\_element}: the first element in the path
\textit{...}: remaining elements in path, terminated by NULL

\textbf{Returns}: a newly-allocated string that must be freed with \texttt{g\_free()}. 

300
CHAPTER 4. GLIB UTILITIES 4.10. MISCELLANEOUS UTILITY FUNCTIONS

**g_build_pathv()**

| gchar* | g_build_pathv | (const gchar* separator, gchar** args); |

Behaves exactly like `g_build_path()`, but takes the path elements as a string array, instead of variargs. This function is mainly meant for language bindings.

- **separator**: a string used to separate the elements of the path.
- **args**: NULL-terminated array of strings containing the path elements.

**Returns**: a newly-allocated string that must be freed with `g_free()`.

Since 2.8

**g_format_size_for_display()**

| char* | g_format_size_for_display | (goffset size); |

Formats a size (for example the size of a file) into a human readable string. Sizes are rounded to the nearest size prefix (KB, MB, GB) and are displayed rounded to the nearest tenth. E.g. the file size 3292528 bytes will be converted into the string "3.1 MB".

The prefix units base is 1024 (i.e. 1 KB is 1024 bytes).

This string should be freed with `g_free()` when not needed any longer.

- **size**: a size in bytes.

**Returns**: a newly-allocated formatted string containing a human readable file size.

Since 2.16

**g_find_program_in_path()**

| gchar* | g_find_program_in_path | (const gchar* program); |

Locates the first executable named `program` in the user’s path, in the same way that `execvp()` would locate it. Returns an allocated string with the absolute path name, or NULL if the program is not found in the path. If `program` is already an absolute path, returns a copy of `program` if `program` exists and is executable, and NULL otherwise. On Windows, if `program` does not have a file type suffix, tries with the suffixes .exe, .cmd, .bat and .com, and the suffixes in the `PATHEXT` environment variable.

On Windows, it looks for the file in the same way as `CreateProcess()` would. This means first in the directory where the executing program was loaded from, then in the current directory, then in the Windows 32-bit system directory, then in the Windows directory, and finally in the directories in the `PATH` environment variable. If the program is found, the return value contains the full name including the type suffix.

- **program**: a program name in the GLib file name encoding

**Returns**: absolute path, or NULL

**g_bit_nth_lsf()**

| gint | g_bit_nth_lsf | (gulong mask, gint nth_bit); |

Find the position of the first bit set in `mask`, searching from (but not including) `nth_bit` upwards. Bits are numbered from 0 (least significant) to `sizeof(gulong) * 8 - 1` (31 or 63, usually). To start searching from the 0th bit, set `nth_bit` to -1.

- **mask**: a `gulong` containing flags.
- **nth_bit**: the index of the bit to start the search from.

**Returns**: the index of the first bit set which is higher than `nth_bit`. 

301
**g_bit_nth_msf ()**

```c
 quint g_bit_nth_msf (gulong mask, gint nth_bit);
```

Find the position of the first bit set in `mask`, searching from (but not including) `nth_bit` downwards. Bits are numbered from 0 (least significant) to sizeof(gulong) * 8 - 1 (31 or 63, usually). To start searching from the last bit, set `nth_bit` to -1 or GLIB_SIZEOF_LONG * 8.

- **mask**: a gulong containing flags.
- **nth_bit**: the index of the bit to start the search from.

**Returns**: the index of the first bit set which is lower than `nth_bit`.

**g_bit_storage ()**

```c
 guint g_bit_storage (gulong number);
```

Gets the number of bits used to hold `number`, e.g. if `number` is 4, 3 bits are needed.

- **number**: a guint.

**Returns**: the number of bits used to hold `number`.

**g_spaced_primes_closest ()**

```c
 guint g_spaced_primes_closest (guint num);
```

Gets the smallest prime number from a built-in array of primes which is larger than `num`. This is used within GLib to calculate the optimum size of a GHashTable.

- **num**: a guint.

**Returns**: the smallest prime number from a built-in array of primes which is larger than `num`.

**g_atexit ()**

```c
 void g_atexit (GVoidFunc func);
```

Specifies a function to be called at normal program termination.

Since GLib 2.8.2, on Windows `g_atexit()` actually is a preprocessor macro that maps to a call to the `atexit()` function in the C library. This means that in case the code that calls `g_atexit()`, i.e. `atexit()`, is in a DLL, the function will be called when the DLL is detached from the program. This typically makes more sense than that the function is called when the GLib DLL is detached, which happened earlier when `g_atexit()` was a function in the GLib DLL.

The behaviour of `atexit()` in the context of dynamically loaded modules is not formally specified and varies wildly.

On POSIX systems, calling `g_atexit()` (or `atexit()`) in a dynamically loaded module which is unloaded before the program terminates might well cause a crash at program exit.

Some POSIX systems implement `atexit()` like Windows, and have each dynamically loaded module maintain an own `atexit` chain that is called when the module is unloaded.

On other POSIX systems, before a dynamically loaded module is unloaded, the registered `atexit` functions (if any) residing in that module are called, regardless where the code that registered them resided. This is presumably the most robust approach.

As can be seen from the above, for portability it’s best to avoid calling `g_atexit()` (or `atexit()`) except in the main executable of a program.

- **func**: the function to call on normal program termination.
### g_parse_debug_string()

```c
uint g_parse_debug_string (const gchar *string,
                          const GDebugKey *keys,
                          guint nkeys);
```

Parses a string containing debugging options into a `uint` containing bit flags. This is used within GDK and GTK+ to parse the debug options passed on the command line or through environment variables.

If `string` is equal to "all", all flags are set. If `string` is equal to "help", all the available keys in `keys` are printed out to standard error.

- **string**: a list of debug options separated by colons, spaces, or commas, or NULL.
- **keys**: pointer to an array of `GDebugKey` which associate strings with bit flags.
- **nkeys**: the number of `GDebugKey`s in the array.

**Returns**: the combined set of bit flags.

#### GDebugKey

```c
typedef struct {
    const gchar *key;
    guint value;
} GDebugKey;
```

Associates a string with a bit flag. Used in `g_parse_debug_string()`.

- **key**: the string
- **value**: the flag

#### GVoidFunc()

```c
void (*GVoidFunc) (void);
```

Declares a type of function which takes no arguments and has no return value. It is used to specify the type function passed to `g_atexit()`.

#### GFreeFunc()

```c
void (*GFreeFunc) (gpointer data);
```

Declares a type of function which takes an arbitrary data pointer argument and has no return value. It is not currently used in GLib or GTK+.

- **data**: a data pointer.

### g_qsort_with_data()

```c
void g_qsort_with_data (gconstpointer pbase,
                        gint total_elems,
                        gsize size,
                        GCompareDataFunc compare_func,
                        gpointer user_data);
```

This is just like the standard C `qsort()` function, but the comparison routine accepts a user data argument.

- **pbase**: start of array to sort
total_elems: elements in the array
size: size of each element
compare_func: function to compare elements
user_data: data to pass to compare_func

**g_nullify_pointer()**

```c
void g_nullify_pointer (gpointer * nullify_location);
```

Set the pointer at the specified location to **NULL**.

**nullify_location**: the memory address of the pointer.

## 4.11 Lexical Scanner

### Name

Lexical Scanner – a general purpose lexical scanner

### Synopsis

```c
#include <glib.h>

GScanner;
GScannerConfig;
GScanner* g_scanner_new (const GScannerConfig* config_templ);
void g_scanner_destroy (GScanner* scanner);
void g_scanner_input_file (GScanner* scanner, gint input_fd);
void g_scanner_sync_file_offset (GScanner* scanner);
void g_scanner_input_text (GScanner* scanner, const gchar* text, guint text_len);
GTokenType g_scanner_peek_next_token (GScanner* scanner);
GTokenType g_scanner_get_next_token (GScanner* scanner);
gboolean g_scanner_eof (GScanner* scanner);
guint g_scanner_cur_line (GScanner* scanner);
guint g_scanner_cur_position (GScanner* scanner);
GTokenType g_scanner_cur_token (GScanner* scanner);
GTokenValue g_scanner_cur_value (GScanner* scanner);
guint g_scanner_set_scope (GScanner* scanner, guint scope_id);
void g_scanner_scope_add_symbol (GScanner* scanner, guint scope_id, const gchar* symbol, gpointer value);
void g_scanner_scope_foreach_symbol (GScanner* scanner, guint scope_id, GHFunc func, gpointer user_data);
gpointer g_scanner_scope_lookup_symbol (GScanner* scanner, guint scope_id);
```

---

304
void g_scanner_scope_remove_symbol (GScanner *scanner, guint scope_id, const gchar *symbol);
#define g_scanner_add_symbol (scanner, symbol, value)
#define g_scanner_remove_symbol (scanner, symbol)
#define g_scanner_foreach_symbol (scanner, func, data)
#define g_scanner_freeze_symbol_table (scanner)
#define g_scanner_thaw_symbol_table (scanner)
gpointer g_scanner_lookup_symbol (GScanner *scanner, const gchar *symbol);
void g_scanner_warn (GScanner *scanner, const gchar *format, ...);
void g_scanner_error (GScanner *scanner, const gchar *format, ...);
void g_scanner_unexp_token (GScanner *scanner, GTokenType expected_token, const gchar *identifier_spec, const gchar *symbol_spec, const gchar *symbol_name, const gchar *message, gint is_error);
void (*GScannerMsgFunc) (GScanner *scanner, gchar *message, gboolean error);
#define G_CSET_a_2_z
#define G_CSET_A_2_Z
#define G_CSET_DIGITS
#define G_CSET_LATINC
#define G_CSET_LATINS
enum GTokenType;
union GTokenValue;
enum GErrorType;

Description
The GScanner and its associated functions provide a general purpose lexical scanner.

Details
GScanner
typedef struct {
    /* unused fields */
gpointer user_data;
guint max_parse_errors;
    /* g_scanner_error() increments this field */
guint parse_errors;
    /* name of input stream, featured by the default message handler */
const gchar *input_name;
    /* quarked data */
GData *qdata;
}
The data structure representing a lexical scanner.

You should set `input_name` after creating the scanner, since it is used by the default message handler when displaying warnings and errors. If you are scanning a file, the file name would be a good choice.

The `user_data` and `max_parse_errors` fields are not used. If you need to associate extra data with the scanner you can place them here.

If you want to use your own message handler you can set the `msg_handler` field. The type of the message handler function is declared by `GScannerMsgFunc`.

```c
/* link into the scanner configuration */
GScannerConfig *config;

/* fields filled in after g_scanner_get_next_token */
GTokenType token;
GTokenValue value;
guint line;
guint position;

/* fields filled in after g_scanner_peek_next_token */
GTokenType next_token;
GTokenValue next_value;
guint next_line;
guint next_position;

/* to be considered private */
GHashTable *symbol_table;
gint input_fd;
const gchar *text;
const gchar *text_end;
gchar *buffer;
guint scope_id;

/* handler function for _warn and _error */
GScannerMsgFunc msg_handler;
)
GScanner;
```

gpointer user_data;
guint max_parse_errors;
guint parse_errors;
const gchar *input_name;
GData *qdata;
GScannerConfig *config;
GTokenType token; token parsed by the last `g_scanner_get_next_token`
GTokenValue value; value of the last token from `g_scanner_get_next_token`
guint line; line number of the last token from `g_scanner_get_next_token`
guint position; char number of the last token from `g_scanner_get_next_token`
GTokenType next_token; token parsed by the last `g_scanner_peek_next_token`
GTokenValue next_value; value of the last token from `g_scanner_peek_next_token`
guint next_line; line number of the last token from `g_scanner_peek_next_token`
guint next_position; char number of the last token from `g_scanner_peek_next_token`
GHashTable *symbol_table;
gint input_fd;
```
const gchar *text;
const gchar *text_end;
gchar *buffer;
guint scope_id;
GScannerMsgFunc msg_handler; // function to handle GScanner message output

GScannerConfig

typedef struct {
    /* Character sets */
    gchar *cset_skip_characters; /* default: " \t\n" */
    gchar *cset_identifier_first;
    gchar *cset_identifier_nth;
    gchar *cpair_comment_single; /* default: "#\n" */

    /* Should symbol lookup work case sensitive? */
    guint case_sensitive : 1;

    /* Boolean values to be adjusted "on the fly" */
    guint skip_comment_multi : 1; /* C like comment */
    guint skip_comment_single : 1; /* single line comment */
    guint scan_comment_multi : 1; /* scan multi line comments? */
    guint scan_identifier : 1;
    guint scan_identifier_1char : 1;
    guint scan_identifier_NULL : 1;
    guint scan_symbols : 1;
    guint scan_binary : 1;
    guint scan_octal : 1;
    guint scan_float : 1;
    guint scan_hex : 1; /* '0xff0' */
    guint scan_hex_dollar : 1; /* '$ff0' */
    guint scan_string_sq : 1; /* string: 'anything' */
    guint scan_string_dq : 1; /* string: "\-escapes!\n" */
    guint numbers_2_int : 1; /* bin, octal, hex -> int */
    guint int_2_float : 1; /* int -> G_TOKEN_FLOAT? */
    guint identifier_2_string : 1;
    guint char_2_token : 1; /* return G_TOKEN_CHAR? */
    guint symbol_2_token : 1;
    guint scope_0_fallback : 1; /* try scope 0 on lookups? */
    guint store_int64 : 1; /* use value.v_int64 rather than v_int */
    guint padding_dummy;
} GScannerConfig;

Specifies the GScanner parser configuration. Most settings can be changed during the parsing phase and will affect the lexical parsing of the next unpeeked token.

cset_skip_characters specifies which characters should be skipped by the scanner (the default is the whitespace characters: space, tab, carriage-return and line-feed).

cset_identifier_first specifies the characters which can start identifiers (the default is G_CSET_a_2_z, ",_", and G_CSET_A_2_Z).

cset_identifier_nth specifies the characters which can be used in identifiers, after the first character (the default is G_CSET_a_2_z,"_",".0123456789",G_CSET_A_2_Z,G_CSET_LATINS,G_CSET_LATINC).

cpair_comment_single specifies the characters at the start and end of single-line comments. The default is "/\n" which means that single-line comments start with a '/' and continue until a '\n' (end of line).

case_sensitive specifies if symbols are case sensitive (the default is FALSE).
**CHAPTER 4. GLIB UTILITIES 4.11. LEXICAL SCANNER**

- `skip_comment_multi` specifies if multi-line comments are skipped and not returned as tokens (the default is `TRUE`).
- `skip_comment_single` specifies if single-line comments are skipped and not returned as tokens (the default is `TRUE`).
- `scan_comment_multi` specifies if multi-line comments are recognized (the default is `TRUE`).
- `scan_identifier` specifies if identifiers are recognized (the default is `TRUE`).
- `scan_identifier_1char` specifies if single-character identifiers are recognized (the default is `FALSE`).
- `scan_identifier_NULL` specifies if `NULL` is reported as `G_TOKEN_IDENTIFIER_NULL` (the default is `FALSE`).
- `scan_symbols` specifies if symbols are recognized (the default is `TRUE`).
- `scan_binary` specifies if binary numbers are recognized (the default is `FALSE`).
- `scan_octal` specifies if octal numbers are recognized (the default is `TRUE`).
- `scan_float` specifies if floating point numbers are recognized (the default is `TRUE`).
- `scan_hex` specifies if hexadecimal numbers are recognized (the default is `TRUE`).
- `scan_hex_dollar` specifies if `$` is recognized as a prefix for hexadecimal numbers (the default is `FALSE`).
- `scan_string_sq` specifies if strings can be enclosed in single quotes (the default is `TRUE`).
- `scan_string_dq` specifies if strings can be enclosed in double quotes (the default is `TRUE`).
- `numbers_2_int` specifies if binary, octal and hexadecimal numbers are reported as `G_TOKEN_INT` (the default is `TRUE`).
- `int_2_float` specifies if all numbers are reported as `G_TOKEN_FLOAT` (the default is `FALSE`).
- `identifier_2_string` specifies if identifiers are reported as strings (the default is `FALSE`).
- `char_2_token` specifies if characters are reported by setting `token = ch` or as `G_TOKEN_CHAR` (the default is `TRUE`).
- `symbol_2_token` specifies if symbols are reported by setting `token = v_symbol` or as `G_TOKEN_SYMBOL` (the default is `FALSE`).
- `scope_0_fallback` specifies if a symbol is searched for in the default scope in addition to the current scope (the default is `FALSE`).

**`g_scanner_new()`**

```c
GScanner* g_scanner_new (const GScannerConfig * config_templ);
```

Creates a new `GScanner`. The `config_templ` structure specifies the initial settings of the scanner, which are copied into the `GScanner config` field. If you pass `NULL` then the default settings are used.

- `config_templ`: the initial scanner settings.

**Returns**: the new `GScanner`.

**`g_scanner_destroy()`**

```c
void g_scanner_destroy (GScanner *scanner);
```

Frees all memory used by the `GScanner`.

- `scanner`: a `GScanner`.

**`g_scanner_input_file()`**

```c
void g_scanner_input_file (GScanner *scanner, gint input_fd);
```

Prepares to scan a file.

- `scanner`: a `GScanner`.
- `input_fd`: a file descriptor.
CHAPTER 4. GLIB UTILITIES  4.11. LEXICAL SCANNER

**g_scanner_sync_file_offset ()**

```c
void g_scanner_sync_file_offset (GScanner *scanner);
```

Rewinds the filedescriptor to the current buffer position and blows the file read ahead buffer. This is useful for third party uses of the scanners filedescriptor, which hooks onto the current scanning position.

*scanner*: a GScanner.

**g_scanner_input_text ()**

```c
void g_scanner_input_text (GScanner *scanner, const gchar *text, guint text_len);
```

Prepares to scan a text buffer.

*scanner*: a GScanner.

*text*: the text buffer to scan.

*text_len*: the length of the text buffer.

**g_scanner_peek_next_token ()**

```c
GTokenType g_scanner_peek_next_token (GScanner *scanner);
```

Parses the next token, without removing it from the input stream. The token data is placed in the *next_token*, *next_value*, *next_line*, and *next_position* fields of the GScanner structure.

Note that, while the token is not removed from the input stream (i.e. the next call to *g_scanner_get_next_token()* will return the same token), it will not be reevaluated. This can lead to surprising results when changing scope or the scanner configuration after peeking the next token. Getting the next token after switching the scope or configuration will return whatever was peeked before, regardless of any symbols that may have been added or removed in the new scope.

*scanner*: a GScanner.

*Returns*: the type of the token.

**g_scanner_get_next_token ()**

```c
GTokenType g_scanner_get_next_token (GScanner *scanner);
```

Parses the next token just like *g_scanner_peek_next_token()* and also removes it from the input stream. The token data is placed in the *token*, *value*, *line*, and *position* fields of the GScanner structure.

*scanner*: a GScanner.

*Returns*: the type of the token.

**g_scanner_eof ()**

```c
gboolean g_scanner_eof (GScanner *scanner);
```

Returns TRUE if the scanner has reached the end of the file or text buffer.

*scanner*: a GScanner.

*Returns*: %TRUE if the scanner has reached the end of the file or text buffer.
g_scanner_cur_line ()

\texttt{guint g_scanner_cur_line (GScanner *scanner);} \hfill (4.11.1)

Returns the current line in the input stream (counting from 1). This is the line of the last token parsed via \texttt{g_scanner_get_next_token()}.

\textit{scanner:} a \texttt{GScanner}.

\textit{Returns:} the current line.

\section*{g_scanner_cur_position ()}

\texttt{guint g_scanner_cur_position (GScanner *scanner);} \hfill (4.11.2)

Returns the current position in the current line (counting from 0). This is the position of the last token parsed via \texttt{g_scanner_get_next_token()}.

\textit{scanner:} a \texttt{GScanner}.

\textit{Returns:} the current position on the line.

\section*{g_scanner_cur_token ()}

\texttt{GTokenType g_scanner_cur_token (GScanner *scanner);} \hfill (4.11.3)

Gets the current token type. This is simply the \texttt{token} field in the \texttt{GScanner} structure.

\textit{scanner:} a \texttt{GScanner}.

\textit{Returns:} the current token type.

\section*{g_scanner_cur_value ()}

\texttt{GTokenValue g_scanner_cur_value (GScanner *scanner);} \hfill (4.11.4)

Gets the current token value. This is simply the \texttt{value} field in the \texttt{GScanner} structure.

\textit{scanner:} a \texttt{GScanner}.

\textit{Returns:} the current token value.

\section*{g_scanner_set_scope ()}

\texttt{guint g_scanner_set_scope (GScanner *scanner,}\hfill (4.11.5)
\texttt{guint scope_id);} \hfill \texttt{(GScanner *scanner,}\hfill \texttt{guint scope_id);}

Sets the current scope.

\textit{scanner:} a \texttt{GScanner}.

\textit{scope_id:} the new scope id.

\textit{Returns:} the old scope id.
**g_scanner_scope_add_symbol ()**

```c
void g_scanner_scope_add_symbol (GScanner *scanner,
                                   guint scope_id,
                                   const gchar *symbol,
                                   gpointer value);
```

Adds a symbol to the given scope.

*scanner*: a **GScanner**.

*scope_id*: the scope id.

*symbol*: the symbol to add.

*value*: the value of the symbol.

**g_scanner_scope_foreach_symbol ()**

```c
void g_scanner_scope_foreach_symbol (GScanner *scanner,
                                     guint scope_id,
                                     GHFunc func,
                                     gpointer user_data);
```

Calls the given function for each of the symbol/value pairs in the given scope of the **GScanner**. The function is passed the symbol and value of each pair, and the given *user_data* parameter.

*scanner*: a **GScanner**.

*scope_id*: the scope id.

*func*: the function to call for each symbol/value pair.

*user_data*: user data to pass to the function.

**g_scanner_scope_lookup_symbol ()**

```c
gpointer g_scanner_scope_lookup_symbol (GScanner *scanner,
                                        guint scope_id,
                                        const gchar *symbol);
```

Looks up a symbol in a scope and return its value. If the symbol is not bound in the scope, **NULL** is returned.

*scanner*: a **GScanner**.

*scope_id*: the scope id.

*symbol*: the symbol to look up.

**Returns**: the value of *symbol* in the given scope, or **NULL** if *symbol* is not bound in the given scope.

**g_scanner_scope_remove_symbol ()**

```c
void g_scanner_scope_remove_symbol (GScanner *scanner,
                                   guint scope_id,
                                   const gchar *symbol);
```

Removes a symbol from a scope.

*scanner*: a **GScanner**.

*scope_id*: the scope id.

*symbol*: the symbol to remove.
g_scanner_add_symbol()

```c
#define g_scanner_add_symbol( scanner, symbol, value )
```

**WARNING**

`g_scanner_add_symbol` has been deprecated since version 2.2 and should not be used in newly-written code. Use `g_scanner_scope_add_symbol()` instead.

Adds a symbol to the default scope.

- **scanner**: a GScanner.
- **symbol**: the symbol to add.
- **value**: the value of the symbol.

---

g_scanner_remove_symbol()

```c
#define g_scanner_remove_symbol( scanner, symbol )
```

**WARNING**

`g_scanner_remove_symbol` has been deprecated since version 2.2 and should not be used in newly-written code. Use `g_scanner_scope_remove_symbol()` instead.

Removes a symbol from the default scope.

- **scanner**: a GScanner.
- **symbol**: the symbol to remove.

---

g_scanner_foreach_symbol()

```c
#define g_scanner_foreach_symbol( scanner, func, data )
```

**WARNING**

`g_scanner_foreach_symbol` has been deprecated since version 2.2 and should not be used in newly-written code. Use `g_scanner_scope_foreach_symbol()` instead.

Calls a function for each symbol in the default scope.

- **scanner**: a GScanner.
- **func**: the function to call with each symbol.
- **data**: data to pass to the function.
CHAPTER 4. GLIB UTILITIES

4.11. LEXICAL SCANNER

**g_scanner_freeze_symbol_table()**

```c
#define g_scanner_freeze_symbol_table(scanner)
```

**WARNING**

`g_scanner_freeze_symbol_table` has been deprecated since version 2.2 and should not be used in newly-written code. This macro does nothing.

There is no reason to use this macro, since it does nothing.

`scanner`: a `GScanner`.

**g_scanner_thaw_symbol_table()**

```c
#define g_scanner_thaw_symbol_table(scanner)
```

**WARNING**

`g_scanner_thaw_symbol_table` has been deprecated since version 2.2 and should not be used in newly-written code. This macro does nothing.

There is no reason to use this macro, since it does nothing.

`scanner`: a `GScanner`.

**g_scanner_lookup_symbol()**

```c
gpointer g_scanner_lookup_symbol (GScanner *scanner, const gchar *symbol);
```

Looks up a symbol in the current scope and return its value. If the symbol is not bound in the current scope, `NULL` is returned.

`scanner`: a `GScanner`.

`symbol`: the symbol to look up.

**Returns**: the value of `symbol` in the current scope, or `NULL` if `symbol` is not bound in the current scope.

**g_scanner_warn()**

```c
void g_scanner_warn (GScanner *scanner, const gchar *format, ...);
```

Outputs a warning message, via the `GScanner` message handler.

`scanner`: a `GScanner`.

`format`: the message format. See the `printf()` documentation.

`...`: the parameters to insert into the format string.
CHAPTER 4. GLIB UTILITIES

4.11. LEXICAL SCANNER

**g_scanner_error ()**

```c
void g_scanner_error (GScanner *scanner,
                      const gchar *format,
                      ...);
```

Outputs an error message, via the GScanner message handler.

*scanner*: a GScanner.

*format*: the message format. See the `printf()` documentation.

...: the parameters to insert into the format string.

**g_scanner_unexp_token ()**

```c
void g_scanner_unexp_token (GScanner *scanner,
                            GTokenType expected_token,
                            const gchar *identifier_spec,
                            const gchar *symbol_spec,
                            const gchar *symbol_name,
                            const gchar *message,
                            gint is_error);
```

Outputs a message through the scanner’s msg_handler, resulting from an unexpected token in the input stream. Note that you should not call `g_scanner_peek_next_token()` followed by `g_scanner_unexp_token()` without an intermediate call to `g_scanner_get_next_token()`, as `g_scanner_unexp_token()` evaluates the scanner’s current token (not the peeked token) to construct part of the message.

*scanner*: a GScanner.

*expected_token*: the expected token.

*identifier_spec*: a string describing how the scanner’s user refers to identifiers (NULL defaults to "identifier"). This is used if *expected_token* is `G_TOKEN_IDENTIFIER` or `G_TOKEN_IDENTIFIER_NULL`.

*symbol_spec*: a string describing how the scanner’s user refers to symbols (NULL defaults to "symbol"). This is used if *expected_token* is `G_TOKEN_SYMBOL` or any token value greater than `G_TOKEN_LAST`.

*symbol_name*: the name of the symbol, if the scanner’s current token is a symbol.

*message*: a message string to output at the end of the warning/error, or NULL.

*is_error*: if `TRUE` it is output as an error. If `FALSE` it is output as a warning.

**GScannerMsgFunc ()**

```c
void (*GScannerMsgFunc) (GScanner *scanner,
                          gchar *message,
                          gboolean error);
```

Specifies the type of the message handler function.

*scanner*: a GScanner.

*message*: the message.

*error*: %TRUE if the message signals an error, FALSE if it signals a warning.
The set of lowercase ASCII alphabet characters. Used for specifying valid identifier characters in `GScannerConfig`.

The set of uppercase ASCII alphabet characters. Used for specifying valid identifier characters in `GScannerConfig`.

The set of digits. Used for specifying valid identifier characters in `GScannerConfig`.

The set of uppercase ISO 8859-1 alphabet characters which are not ASCII characters. Used for specifying valid identifier characters in `GScannerConfig`.

The set of lowercase ISO 8859-1 alphabet characters which are not ASCII characters. Used for specifying valid identifier characters in `GScannerConfig`.

```
typedef enum
{
    G_TOKEN_EOF = 0,
    G_TOKEN_LEFT_PAREN = '(',
    G_TOKEN_RIGHT_PAREN = ')',
    G_TOKEN_LEFT_CURLY = '{',
    G_TOKEN_RIGHT_CURLY = '}',
    G_TOKEN_LEFT_BRACE = '[',
    G_TOKEN_RIGHT_BRACE = ']',
    G_TOKEN_EQUAL_SIGN = '=",
    G_TOKEN_COMMA = ',',
    G_TOKEN_NONE = 256,
    G_TOKEN_ERROR,
    G_TOKEN_CHAR,
    G_TOKEN_BINARY,
    G_TOKEN_OCTAL,
    G_TOKEN_INT,
    G_TOKEN_HEX,
    G_TOKEN_FLOAT,
    G_TOKEN_STRING,
} TokenType;
```
The possible types of token returned from each `g_scanner_get_next_token()` call.

- **G_TOKEN_EOF** the end of the file.
- **G_TOKEN_LEFT_PAREN** a `(` character.
- **G_TOKEN_LEFT_CURLY** a `{` character.
- **G_TOKEN_RIGHT_CURLY** a `}` character.

union GTokenValue

```c
union GTokenValue
{
    gpointer v_symbol;
    gchar *v_identifier;
    gulong v_binary;
    gulong v_octal;
    gulong v_int;
    guint64 v_int64;
    gdouble v_float;
    gulong v_hex;
    gchar *v_string;
    gchar *v_comment;
    guchar v_char;
    guint v_error;
};
```

A union holding the value of the token.

enum GErrorType

```c
typedef enum
{
    G_ERR_UNKNOWN,
    G_ERR_UNEXP_EOF,
    G_ERR_UNEXP_EOF_IN_STRING,
    G_ERR_UNEXP_EOF_IN_COMMENT,
    G_ERR_NON_DIGIT_IN_CONST,
    G_ERR_DIGIT_RADIX,
    G_ERR_FLOAT_RADIX,
    G_ERR_FLOAT_MALFORMED
} GErrorType;
```

The possible errors, used in the `v_error` field of GTokenValue, when the token is a **G_TOKEN_ERROR**.

- **G_ERR_UNKNOWN** unknown error.
- **G_ERR_UNEXP_EOF** unexpected end of file.
- **G_ERR_UNEXP_EOF_IN_STRING** unterminated string constant.
- **G_ERR_UNEXP_EOF_IN_COMMENT** unterminated comment.
- **G_ERR_NON_DIGIT_IN_CONST** non-digit character in a number.
**CHAPTER 4. GLIB UTILITIES**

**4.12. AUTOMATIC STRING COMPLETION**

**G_ERR_DIGIT_RADIX** digit beyond radix in a number.

**G_ERR_FLOAT_RADIX** non-decimal floating point number.

**G_ERR_FLOAT_MALFORMED** malformed floating point number.

### 4.12 Automatic String Completion

**Name**

Automatic String Completion – support for automatic completion using a group of target strings

**Synopsis**

```c
#include <glib.h>

GCompletion;
GCompletion* g_completion_new (GCompletionFunc func);
gchar * (*GCompletionFunc) (gpointer);
void g_completion_add_items (GCompletion *cmp, GList *items);
void g_completion_remove_items (GCompletion *cmp, GList *items);
void g_completion_clear_items (GCompletion *cmp);
GList* g_completion_complete (GCompletion *cmp, const gchar *prefix, gchar **new_prefix);
GList* g_completion_complete_utf8 (GCompletion *cmp, const gchar *prefix, gchar **new_prefix);
void g_completion_set_compare (GCompletion *cmp, GCompletionStrncmpFunc strncmp_func);
gint (*GCompletionStrncmpFunc) (const gchar *s1, const gchar *s2, gsize n);
void g_completion_free (GCompletion *cmp);
```

**Description**

**GCompletion** provides support for automatic completion of a string using any group of target strings. It is typically used for file name completion as is common in many UNIX shells.

A **GCompletion** is created using **g_completion_new()**. Target items are added and removed with **g_completion_add_items()**, **g_completion_remove_items()** and **g_completion_clear_items()**. A completion attempt is requested with **g_completion_complete()** or **g_completion_complete_utf8()**. When no longer needed, the **GCompletion** is freed with **g_completion_free()**.

Items in the completion can be simple strings (e.g. filenames), or pointers to arbitrary data structures. If data structures are used you must provide a **GCompletionFunc** in **g_completion_new()**, which retrieves the item’s string from the data structure. You can change the way in which strings are compared by setting a different **GCompletionStrncmpFunc** in **g_completion_set_compare()**.

**Details**

**GCompletion**

```c
typedef struct {
    GList* items;
    GCompletionFunc func;
} GCompletion;
```
The data structure used for automatic completion.

**GList** *items*: list of target items (strings or data structures).

**GCompletionFunc** *func*: function which is called to get the string associated with a target item. It is NULL if the target items are strings.

**gchar** *prefix*: the last prefix passed to `g_completion_complete()` or `g_completion_complete_utf8()`.

**GList** *cache*: the list of items which begin with `prefix`.

**GCompletionStrncmpFunc** *strncmp_func*: The function to use when comparing strings. Use `g_completion_set_compare()` to modify this function.

### g_completion_new() function

```c
GCompletion* g_completion_new (GCompletionFunc func);
```

Creates a new GCompletion.

*func*: the function to be called to return the string representing an item in the GCompletion, or NULL if strings are going to be used as the GCompletion items.

**Returns**: the new GCompletion.

### GCompletionFunc ()

```c
gchar * (*GCompletionFunc) (gpointer);
```

Specifies the type of the function passed to `g_completion_new()`. It should return the string corresponding to the given target item. This is used when you use data structures as GCompletion items.

**Param1**: the completion item.

**Returns**: the string corresponding to the item.

### g_completion_add_items ()

```c
void g_completion_add_items (GCompletion *cmp,
                             GList *items);
```

Adds items to the GCompletion.

**cmp**: the GCompletion.

**items**: the list of items to add.

### g_completion_remove_items ()

```c
void g_completion_remove_items (GCompletion *cmp,
                                GList *items);
```

Removes items from a GCompletion.

**cmp**: the GCompletion.

**items**: the items to remove.
g_completion_clear_items ()

```c
void g_completion_clear_items (GCompletion *cmp);
```

Removes all items from the GCompletion.

**cmp**: the GCompletion.

---

g_completion_complete ()

```c
GList* g_completion_complete (GCompletion *cmp, const gchar *prefix, gchar **new_prefix);
```

Attempts to complete the string `prefix` using the GCompletion target items.

**cmp**: the GCompletion.

**prefix**: the prefix string, typically typed by the user, which is compared with each of the items.

**new_prefix**: if non-NULL, returns the longest prefix which is common to all items that matched `prefix`, or NULL if no items matched `prefix`. This string should be freed when no longer needed.

**Returns**: the list of items whose strings begin with `prefix`. This should not be changed.

---

g_completion_complete_utf8 ()

```c
GList* g_completion_complete_utf8 (GCompletion *cmp, const gchar *prefix, gchar **new_prefix);
```

Attempts to complete the string `prefix` using the GCompletion target items. In contrast to `g_completion_complete()`, this function returns the largest common prefix that is a valid UTF-8 string, omitting a possible common partial character.

You should use this function instead of `g_completion_complete()` if your items are UTF-8 strings.

**cmp**: the GCompletion

**prefix**: the prefix string, typically used by the user, which is compared with each of the items.

**new_prefix**: if non-NULL, returns the longest prefix which is common to all items that matched `prefix`, or NULL if no items matched `prefix`. This string should be freed when no longer needed.

**Returns**: the list of items whose strings begin with `prefix`. This should not be changed.

Since 2.4

---

g_completion_set_compare ()

```c
void g_completion_set_compare (GCompletion *cmp, GCompletionStrncmpFunc strncmp_func);
```

Sets the function to use for string comparisons. The default string comparison function is `strn cmp()`.

**cmp**: a GCompletion.

**strncmp_func**: the string comparison function.
4.13 Timers

Name
Timers – keep track of elapsed time

Synopsis

```c
#include <glib.h>

GTimer;
void g_timer_new (void);
void g_timer_start (GTimer *timer);
void g_timer_stop (GTimer *timer);
void g_timer_continue (GTimer *timer);
gdouble g_timer_elapsed (GTimer *timer, gulong *microseconds);
void g_timer_reset (GTimer *timer);
void g_timer_destroy (GTimer *timer);
```

Description

GTimer records a start time, and counts microseconds elapsed since that time. This is done somewhat differently on different platforms, and can be tricky to get exactly right, so GTimer provides a portable/convenient interface.

**NOTE**

GTimer uses a higher-quality clock when thread support is available. Therefore, calling g_thread_init() while timers are running may lead to unreliable results. It is best to call g_thread_init() before starting any timers, if you are using threads at all.
Details

GTimer

typedef struct _GTimer GTimer;

Opaque datatype that records a start time.

\textbf{g\_timer\_new ()}

\begin{verbatim}
GTimer* g_timer_new (void);
\end{verbatim}

Creates a new timer, and starts timing (i.e. \texttt{g\_timer\_start()} is implicitly called for you).

\textbf{Returns} : a new GTimer.

\textbf{g\_timer\_start ()}

\begin{verbatim}
void g_timer_start (GTimer *timer);
\end{verbatim}

Marks a start time, so that future calls to \texttt{g\_timer\_elapsed()} will report the time since \texttt{g\_timer\_start()} was called. \texttt{g\_timer\_new()} automatically marks the start time, so no need to call \texttt{g\_timer\_start()} immediately after creating the timer.

\texttt{timer} : a GTimer.

\textbf{g\_timer\_stop ()}

\begin{verbatim}
void g_timer_stop (GTimer *timer);
\end{verbatim}

Marks an end time, so calls to \texttt{g\_timer\_elapsed()} will return the difference between this end time and the start time.

\texttt{timer} : a GTimer.

\textbf{g\_timer\_continue ()}

\begin{verbatim}
void g_timer_continue (GTimer *timer);
\end{verbatim}

Resumes a timer that has previously been stopped with \texttt{g\_timer\_stop()}. \texttt{g\_timer\_stop()} must be called before using this function.

\texttt{timer} : a GTimer.

Since 2.4

\textbf{g\_timer\_elapsed ()}

\begin{verbatim}
gdouble g_timer_elapsed (GTimer *timer,
gulong *microseconds);
\end{verbatim}

If \texttt{timer} has been started but not stopped, obtains the time since the timer was started. If \texttt{timer} has been stopped, obtains the elapsed time between the time it was started and the time it was stopped. The return value is the number of seconds elapsed, including any fractional part. The \texttt{microseconds} out parameter is essentially useless.

\textbf{WARNING}

\begin{quote}
Calling initialization functions, in particular \texttt{g\_thread\_init()}, while a timer is running will cause invalid return values from this function.
\end{quote}
timer: a GTimer.

microseconds: return location for the fractional part of seconds elapsed, in microseconds (that is, the total number of microseconds elapsed, modulo 1000000), or NULL.

Returns: seconds elapsed as a floating point value, including any fractional part.

g_timer_reset()

```c
void g_timer_reset (GTimer *timer);
```

This function is useless; it’s fine to call g_timer_start() on an already-started timer to reset the start time, so g_timer_reset() serves no purpose.

timer: a GTimer.

g_timer_destroy()

```c
void g_timer_destroy (GTimer *timer);
```

Destroys a timer, freeing associated resources.

timer: a GTimer to destroy.

### 4.14 Spawning Processes

#### Name

Spawning Processes – process launching

#### Synopsis

```c
#include <glib.h>
enum GSpawnError;
#define G_SPAWN_ERROR
enum GSpawnFlags;
void ( *GSpawnChildSetupFunc) (gpointer user_data);
gboolean g_spawn_async_with_pipes
  (const gchar *working_directory,
   gchar **argv,
   gchar **envp,
   GSpawnFlags flags,
   GSpawnChildSetupFunc child_setup,
   gpointer user_data,
   GPid *child_pid,
   gint *standard_input,
   gint *standard_output,
   gint *standard_error,
   GError **error);
gboolean g_spawn_async
  (const gchar *working_directory,
   gchar **argv,
   gchar **envp,
   GSpawnFlags flags,
   GSpawnChildSetupFunc child_setup,
   gpointer user_data,
   GPid *child_pid,
   GError **error);
gboolean g_spawn_sync
  (const gchar *working_directory,
   gchar **argv,
   gchar **envp,
   GSpawnFlags flags,
   GSpawnChildSetupFunc child_setup,
   gpointer user_data,
   GPid *child_pid,
   GError **error);
```
CHAPTER 4. GLIB UTILITIES 4.14. SPAWNING PROCESSES

gchar **argv, gchar **envp,
GSpawnFlags flags,
GSpawnChildSetupFunc child_setup,
gpointer user_data,
gchar **standard_output,
gchar **standard_error,
gint *exit_status,
GError **error);

gboolean g_spawn_command_line_async (const gchar *command_line,
GError **error);

gboolean g_spawn_command_line_sync (const gchar *command_line,
gchar **standard_output,
gchar **standard_error,
gint *exit_status,
GError **error);

void g_spawn_close_pid (GPid pid);

Description

Details

enum GSpawnError

typedef enum
{
  G_SPAWN_ERROR_FORK, /* fork failed due to lack of memory */
  G_SPAWN_ERROR_READ, /* read or select on pipes failed */
  G_SPAWN_ERROR_CHDIR, /* changing to working dir failed */
  G_SPAWN_ERROR_ACCES, /* execv() returned EACCES */
  G_SPAWN_ERROR_PERM, /* execv() returned EPERM */
  G_SPAWN_ERROR_2BIG, /* execv() returned E2BIG */
  G_SPAWN_ERROR_NOEXEC, /* execv() returned ENOEXEC */
  G_SPAWN_ERROR_NAMETOOLONG, /* ENAMETOOLONG */
  G_SPAWN_ERROR_NOENT, /* ENOENT */
  G_SPAWN_ERROR_NOMEM, /* ENOMEM */
  G_SPAWN_ERROR_NOTDIR, /* ENOTDIR */
  G_SPAWN_ERROR_LOOP, /* ELOOP */
  G_SPAWN_ERROR_TXTBUSY, /* ETXTBUSY */
  G_SPAWN_ERROR_IO, /* EIO */
  G_SPAWN_ERROR_NFILE, /* ENFILE */
  G_SPAWN_ERROR_MFILE, /* EMFILE */
  G_SPAWN_ERROR_INVAL, /* EINVAL */
  G_SPAWN_ERROR_ISDIR, /* EISDIR */
  G_SPAWN_ERROR_LIBBAD, /* ELIBBAD */
  G_SPAWN_ERROR_FAILED /* other fatal failure, error->message */
} GSpawnError;

Error codes returned by spawning processes.

G_SPAWN_ERROR_FORK Fork failed due to lack of memory.

G_SPAWN_ERROR_READ Read or select on pipes failed.

G_SPAWN_ERROR_CHDIR Changing to working directory failed.

G_SPAWN_ERROR_ACCES execv() returned EACCES.

G_SPAWN_ERROR_PERM execv() returned EPERM.

G_SPAWN_ERROR_2BIG execv() returned E2BIG.
CHAPTER 4. GLIB UTILITIES

4.14. SPAWNING PROCESSES

G_SPAWN_ERROR_NOEXEC execv() returned ENOEXEC.
G_SPAWN_ERROR_NAMETOOLONG execv() returned ENAMETOOLONG.
G_SPAWN_ERROR_NOENT execv() returned ENOENT.
G_SPAWN_ERROR_NOMEM execv() returned ENOMEM.
G_SPAWN_ERROR_NOTDIR execv() returned ENOTDIR.
G_SPAWN_ERROR_LOOP execv() returned ELOOP.
G_SPAWN_ERROR_TXTBUSY execv() returned ETXTBUSY.
G_SPAWN_ERROR_IO execv() returned EIO.
G_SPAWN_ERROR_NFILE execv() returned ENFILE.
G_SPAWN_ERROR_MFILE execv() returned EMFILE.
G_SPAWN_ERROR_INVAL execv() returned EINVAL.
G_SPAWN_ERROR_LIBBAD execv() returned EISDIR.
G_SPAWN_ERROR_FAILED Some other fatal failure, error->message should explain.

G_SPAWN_ERROR

#define G_SPAWN_ERROR g_spawn_error_quark ()

Error domain for spawning processes. Errors in this domain will be from the GSpawnError enumeration. See GError for information on error domains.

enum GSpawnFlags

typedef enum
{
  G_SPAWN_LEAVE_DESCRIPTORS_OPEN = 1 << 0,
  G_SPAWN_DO_NOT_REAP_CHILD = 1 << 1,
  /* look for argv[0] in the path i.e. use execvp() */
  G_SPAWN_SEARCH_PATH = 1 << 2,
  /* Dump output to /dev/null */
  G_SPAWN_STDOUT_TO_DEV_NULL = 1 << 3,
  G_SPAWN_STDERR_TO_DEV_NULL = 1 << 4,
  G_SPAWN_CHILD_INHERITS_STDIN = 1 << 5,
  G_SPAWN_FILE_AND_ARGV_ZERO = 1 << 6
} GSpawnFlags;

Flags passed to g_spawn_sync(), g_spawn_async() and g_spawn_async_with_pipes().

G_SPAWN_LEAVE_DESCRIPTORS_OPEN the parent's open file descriptors will be inherited by the child; otherwise all descriptors except stdin/stdout/stderr will be closed before calling exec() in the child.

G_SPAWN_DO_NOT_REAP_CHILD the child will not be automatically reaped; you must use g_child_watch_add() yourself (or call waitpid() or handle SIGCHLD yourself), or the child will become a zombie.

G_SPAWN_SEARCH_PATH argv[0] need not be an absolute path, it will be looked for in the user’s PATH.

G_SPAWN_STDOUT_TO_DEV_NULL the child’s standard output will be discarded, instead of going to the same location as the parent’s standard output.

G_SPAWN_STDERR_TO_DEV_NULL the child’s standard error will be discarded.
CHAPTER 4. GLIB UTILITIES

4.14. SPAWNING PROCESSES

**G_SPAWN_CHILD_INHERITS_STDIN** the child will inherit the parent’s standard input (by default, the child’s standard input is attached to /dev/null).

**G_SPAWN_FILE_AND_ARGV_ZERO** the first element of argv is the file to execute, while the remaining elements are the actual argument vector to pass to the file. Normally `g_spawn_async_with_pipes()` uses argv[0] as the file to execute, and passes all of argv to the child.

GSpawnChildSetupFunc ()

```c
void (*GSpawnChildSetupFunc) (gpointer user_data);
```

Specifies the type of the setup function passed to `g_spawn_async()`, `g_spawn_sync()` and `g_spawn_async_with_pipes()`. On POSIX platforms it is called in the child after GLib has performed all the setup it plans to perform but before calling exec(). On POSIX actions taken in this function will thus only affect the child, not the parent.

Note that POSIX allows only async-signal-safe functions (see signal(7)) to be called in the child between fork() and exec(), which drastically limits the usefulness of child setup functions.

Also note that modifying the environment from the child setup function may not have the intended effect, since it will get overridden by a non-NULL env argument to the `g_spawn...` functions.

On Windows the function is called in the parent. Its usefulness on Windows is thus questionable. In many cases executing the child setup function in the parent can have ill effects, and you should be very careful when porting software to Windows that uses child setup functions.

**user_data**: user data to pass to the function.

**g_spawn_async_with_pipes ()**

```c
gboolean g_spawn_async_with_pipes (const gchar *working_directory, gchar **argv, gchar **envp, GSpawnFlags flags, GSpawnChildSetupFunc ← child_setup, gpointer user_data, GPid *child_pid, gint *standard_input, gint *standard_output, gint *standard_error, GError **error);
```

Executes a child program asynchronously (your program will not block waiting for the child to exit). The child program is specified by the only argument that must be provided, `argv`. `argv` should be a NULL-terminated array of strings, to be passed as the argument vector for the child. The first string in `argv` is of course the name of the program to execute. By default, the name of the program must be a full path; the PATH shell variable will only be searched if you pass the `G_SPAWN_SEARCH_PATH` flag.

On Windows, note that all the string or string vector arguments to this function and the other `g_spawn*()` functions are in UTF-8, the GLib file name encoding. Unicode characters that are not part of the system codepage passed in these arguments will be correctly available in the spawned program only if it uses wide character API to retrieve its command line. For C programs built with Microsoft’s tools it is enough to make the program have a `wmain()` instead of `main()`. `wmain()` has a wide character argument vector as parameter.

At least currently, mingw doesn’t support `wmain()`, so if you use mingw to develop the spawned program, it will have to call the undocumented function `__wgetmainargs()` to get the wide character argument vector and environment. See gspawn-win32-helper.c in the GLib sources or init.c in the mingw runtime sources for a prototype for that function. Alternatively, you can retrieve the Win32 system level wide character command line passed to the spawned program using the `GetCommandLineW()` function.

On Windows the low-level child process creation API `CreateProcess()` doesn’t use argument vectors, but a command line. The C runtime library’s `spawn*()` family of functions (which `g_spawn_async_with_pipes()`
eventually calls) paste the argument vector elements together into a command line, and the C runtime
startup code does a corresponding reconstruction of an argument vector from the command line, to be
passed to main(). Complications arise when you have argument vector elements that contain spaces of
double quotes. The spawn*() functions don’t do any quoting or escaping, but on the other hand the
startup code does unquoting and unescaping in order to enable receiving arguments with embedded
spaces or double quotes. To work around this asymmetry, g_spawn_async_with_pipes() will do quoting
and escaping on argument vector elements that need it before calling the C runtime spawn() function.

The returned child_pid on Windows is a handle to the child process, not its identifier. Process
handles and process identifiers are different concepts on Windows.

envp is a NULL-terminated array of strings, where each string has the form KEY=VALUE. This will
become the child’s environment. If envp is NULL, the child inherits its parent’s environment.

flags should be the bitwise OR of any flags you want to affect the function’s behaviour. The
G_SPAWN_DO_NOT_REAP_CHILD means that the child will not automatically be reaped; you must
use a GChildWatch source to be notified about the death of the child process. Eventually you must call
g_spawn_close_pid() on the child_pid, in order to free resources which may be associated with the
child process. (On Unix, using a GChildWatch source is equivalent to calling waitpid() or handling the
SIGCHLD signal manually. On Windows, calling g_spawn_close_pid() is equivalent to calling Close-
Handle() on the process handle returned in child_pid).

G_SPAWN_LEAVE_DESCRIPTORS_OPEN means that the parent’s open file descriptors will be in-
herited by the child; otherwise all descriptors except stdin/stdout/stderr will be closed before calling
exec() in the child. G_SPAWN_SEARCH_PATH means that argv[0] need not be an absolute path, it
will be looked for in the user’s PATH. G_SPAWN_STDOUT_TO_DEV_NULL means that the child’s stan-
dard output will be discarded, instead of going to the same location as the parent’s standard output. If
you use this flag, standard_output must be NULL. G_SPAWN_STDERR_TO_DEV_NULL means that
the child’s standard error will be discarded, instead of going to the same location as the parent’s stan-
dard error. If you use this flag, standard_error must be NULL. G_SPAWN_CHILD_INHERITS_STDIN
means that the child will inherit the parent’s standard input (by default, the child’s standard input is at-
tached to /dev/null). If you use this flag, standard_input must be NULL. G_SPAWN_FILE_AND_ARGV_ZERO
means that the first element of argv is the file to execute, while the remaining elements are the actual
argument vector to pass to the file. Normally g_spawn_async_with_pipes() uses argv[0] as the file to
eexecute, and passes all of argv to the child.

child_setup and user_data are a function and user data. On POSIX platforms, the function is
called in the child after GLib has performed all the setup it plans to perform (including creating pipes,
closing file descriptors, etc.) but before calling exec(). That is, child_setup is called just before calling
exec() in the child. Obviously actions taken in this function will only affect the child, not the parent.

On Windows, there is no separate fork() and exec() functionality. Child processes are created and
run with a single API call, CreateProcess(). There is no sensible thing child_setup could be used for on
Windows so it is ignored and not called.

If non-NULL, child_pid will on Unix be filled with the child’s process ID. You can use the pro-
cess ID to send signals to the child, or to use g_child_watch_add() (or waitpid()) if you specified the
G_SPAWN_DO_NOT_REAP_CHILD flag. On Windows, child_pid will be filled with a handle to the
child process only if you specified the G_SPAWN_DO_NOT_REAP_CHILD flag. You can then access the
child process using the Win32 API, for example wait for its termination with the WaitFor*() functions,
or examine its exit code with GetExitCodeProcess(). You should close the handle with CloseHandle() or
g_spawn_close_pid() when you no longer need it.

If non-NULL, the standard_input, standard_output, standard_error locations will be filled with
file descriptors for writing to the child’s standard input or reading from its standard output or standard
error. The caller of g_spawn_async_with_pipes() must close these file descriptors when they are no
longer in use. If these parameters are NULL, the corresponding pipe won’t be created.

If standard_input is NULL, the child’s standard input is attached to /dev/null unless G_SPAWN_CHILD_INHERITS_STDIN
is set.

If standard_error is NULL, the child’s standard error goes to the same location as the parent’s
standard error unless G_SPAWN_STDERR_TO_DEV_NULL is set.

If standard_output is NULL, the child’s standard output goes to the same location as the parent’s
standard output unless G_SPAWN_STDOUT_TO_DEV_NULL is set.

e error can be NULL to ignore errors, or non-NULL to report errors. If an error is set, the function
returns FALSE. Errors are reported even if they occur in the child (for example if the executable in a-
argv[0] is not found). Typically the message field of returned errors should be displayed to users.
Possible errors are those from the G_SPAWN_ERROR domain.
If an error occurs, \texttt{child\_pid, standard\_input, standard\_output, and standard\_error} will not be filled with valid values.

If \texttt{child\_pid} is not \texttt{NULL} and an error does not occur then the returned process reference must be closed using \texttt{g\_spawn\_close\_pid()}. 

\textbf{Note}

If you are writing a GTK+ application, and the program you are spawning is a graphical application, too, then you may want to use \texttt{gdk\_spawn\_on\_screen\_with\_pipes()} instead to ensure that the spawned program opens its windows on the right screen.

\texttt{working\_directory}: child’s current working directory, or \texttt{NULL} to inherit parent’s, in the GLib file name encoding

\texttt{argv}: child’s argument vector, in the GLib file name encoding

\texttt{envp}: child’s environment, or \texttt{NULL} to inherit parent’s, in the GLib file name encoding

\texttt{flags}: flags from \texttt{GSpawnFlags}

\texttt{child\_setup}: function to run in the child just before \texttt{exec()}

\texttt{user\_data}: user data for \texttt{child\_setup}

\texttt{child\_pid}: return location for child process ID, or \texttt{NULL}

\texttt{standard\_input}: return location for file descriptor to write to child’s stdin, or \texttt{NULL}

\texttt{standard\_output}: return location for file descriptor to read child’s stdout, or \texttt{NULL}

\texttt{standard\_error}: return location for file descriptor to read child’s stderr, or \texttt{NULL}

\texttt{error}: return location for error

\textbf{Returns}: \texttt{TRUE} on success, \texttt{FALSE} if an error was set

\texttt{g\_spawn\_async()}

\begin{verbatim}
gboolean g_spawn_async (const gchar * ←
  working_directory,
  gchar **argv, ←
  gchar **envp, ←
  GSpawnFlags flags, ←
  GSpawnChildSetupFunc ←
  child_setup, ←
  gpointer user_data, ←
  GPid *child_pid, ←
  GError **error);
\end{verbatim}

See \texttt{g\_spawn\_async\_with\_pipes()} for a full description; this function simply calls the \texttt{g\_spawn\_async\_with\_pipes()} without any pipes.

You should call \texttt{g\_spawn\_close\_pid()} on the returned child process reference when you don’t need it any more.

\textbf{Note}

If you are writing a GTK+ application, and the program you are spawning is a graphical application, too, then you may want to use \texttt{gdk\_spawn\_on\_screen()} instead to ensure that the spawned program opens its windows on the right screen.
CHAPTER 4. GLIB UTILITIES
4.14. SPAWNING PROCESSES

**NOTE**

Note that the returned `child_pid` on Windows is a handle to the child process and not its identifier. Process handles and process identifiers are different concepts on Windows.

**working_directory**: child’s current working directory, or `NULL` to inherit parent’s

**argv**: child’s argument vector

**envp**: child’s environment, or `NULL` to inherit parent’s

**flags**: flags from `GSpawnFlags`

**child_setup**: function to run in the child just before `exec()`

**user_data**: user data for `child_setup`

**child_pid**: return location for child process reference, or `NULL`

**error**: return location for error

*Returns*: `TRUE` on success, `FALSE` if error is set

```c
gboolean g_spawn_sync (const gchar *working_directory, gchar **argv, gchar **envp, GSpawnFlags flags, GSpawnChildSetupFunc child_setup, gpointer user_data, gchar **standard_output, gchar **standard_error, gint *exit_status, GError **error);
```

Executes a child synchronously (waits for the child to exit before returning). All output from the child is stored in `standard_output` and `standard_error`, if those parameters are non-NULL. Note that you must set the `G_SPAWN_STDOUT_TO_DEV_NULL` and `G_SPAWN_STDERR_TO_DEV_NULL` flags when passing `NULL` for `standard_output` and `standard_error`. If `exit_status` is non-NULL, the exit status of the child is stored there as it would be returned by `waitpid()`. Standard UNIX macros such as `WIFEXITED()` and `WEXITSTATUS()` must be used to evaluate the exit status. Note that this function calls `waitpid()` even if `exit_status` is `NULL`, and does not accept the `G_SPAWN_DO_NOT_REAP_CHILD` flag. If an error occurs, no data is returned in `standard_output`, `standard_error`, or `exit_status`.

This function calls `g_spawn_async_with_pipes()` internally; see that function for full details on the other parameters and details on how these functions work on Windows.

**working_directory**: child’s current working directory, or `NULL` to inherit parent’s

**argv**: child’s argument vector

**envp**: child’s environment, or `NULL` to inherit parent’s

**flags**: flags from `GSpawnFlags`

**child_setup**: function to run in the child just before `exec()`

**user_data**: user data for `child_setup`
standard_output: return location for child output, or NULL
standard_error: return location for child error messages, or NULL
exit_status: return location for child exit status, as returned by waitpid(), or NULL
error: return location for error, or NULL

Returns: TRUE on success, FALSE if an error was set.

g_spawn_command_line_async()

gboolean g_spawn_command_line_async (const gchar *command_line,
                                       GError **error);

A simple version of g_spawn_async() that parses a command line with g_shell_parse_argv() and
passes it to g_spawn_async(). Runs a command line in the background. Unlike g_spawn_async(), the
G_SPAWN_SEARCH_PATH flag is enabled, other flags are not. Note that G_SPAWN_SEARCH_PATH
can have security implications, so consider using g_spawn_async() directly if appropriate. Possible
errors are those from g_shell_parse_argv() and g_spawn_async().

The same concerns on Windows apply as for g_spawn_command_line_sync().

command_line: a command line
error: return location for errors

Returns: TRUE on success, FALSE if error is set.

g_spawn_command_line_sync()

gboolean g_spawn_command_line_sync (const gchar *command_line,
                                     gchar **standard_output,
                                     gchar **standard_error,
                                     gint *exit_status,
                                     GError **error);

A simple version of g_spawn_sync() with little-used parameters removed, taking a command line
instead of an argument vector. See g_spawn_sync() for full details. command_line will be parsed by
g_shell_parse_argv(). Unlike g_spawn_sync(), the G_SPAWN_SEARCH_PATH flag is enabled. Note
that G_SPAWN_SEARCH_PATH can have security implications, so consider using g_spawn_sync() di-
rectly if appropriate. Possible errors are those from g_spawn_sync() and those from g_shell_parse_argv().

If exit_status is non-NULL, the exit status of the child is stored there as it would be returned by
waitpid(); standard UNIX macros such as WIFEXITED() and WEXITSTATUS() must be used to evaluate
the exit status.

On Windows, please note the implications of g_shell_parse_argv() parsing command_line. Parsing
is done according to Unix shell rules, not Windows command interpreter rules. Space is a separa-
tor, and backslashes are special. Thus you cannot simply pass a command_line containing canonical
Windows paths, like "c:\\program files\\app\\app.exe", as the backslashes will be eaten, and the
space will act as a separator. You need to enclose such paths with single quotes, like "'c:\\program
files\\app\\app.exe' 'e:\\folder\\argument.txt'".

command_line: a command line
standard_output: return location for child output
standard_error: return location for child errors
exit_status: return location for child exit status, as returned by waitpid()
error: return location for errors

Returns: TRUE on success, FALSE if an error was set
g_spawn_close_pid()

```c
void g_spawn_close_pid (Gpid pid);
```

On some platforms, notably Windows, the GPid type represents a resource which must be closed to prevent resource leaking. `g_spawn_close_pid()` is provided for this purpose. It should be used on all platforms, even though it doesn’t do anything under UNIX.

**pid**: The process reference to close

## 4.15 File Utilities

### Name

File Utilities – various file-related functions

### Synopsis

```c
#include <glib.h>
#include <glib/gstdio.h>

enum GFileError;
#define G_FILE_ERROR
enum GFileTest;
GFileError g_file_error_from_errno (gint err_no);
gboolean g_file_get_contents (const gchar *filename, gchar **contents, gsize *length, GError **error);
gboolean g_file_set_contents (const gchar *filename, const gchar *contents, gssize length, GError **error);
gboolean g_file_test (const gchar *filename, GFileTest test);
gint g_mkstemp (gchar *tmpl);
gint g_file_open_tmp (const gchar *tmpl, gchar **name_used, GError **error);
gchar * g_file_read_link (const gchar *filename, GError **error);
int g_mkdir_with_parents (const gchar *pathname, int mode);

GDir;
GDir * g_dir_open (const gchar *path, guint flags, GError **error);
const gchar * g_dir_read_name (GDir *dir);
void g_dir_rewind (GDir *dir);
void g_dir_close (GDir *dir);

GMappedFile;
GMappedFile * g_mapped_file_new (const gchar *filename, gboolean writable, GError **error);
void g_mapped_file_free (GMappedFile *file);
gsize g_mapped_file_get_length (GMappedFile *file);
```

330
CHAPTER 4. GLIB UTILITIES 4.15. FILE UTILITIES

```c

gchar * g_mapped_file_get_contents (GMappedFile *file);

int g_open (const gchar *filename, int flags, int mode);

int g_rename (const gchar *oldfilename, const gchar *newfilename);

int g_mkdir (const gchar *filename, int mode);

int g_stat (const gchar *filename, struct stat *buf);

int g_lstat (const gchar *filename, struct stat *buf);

int g_unlink (const gchar *filename);

int g_remove (const gchar *filename);

int g_rmdir (const gchar *filename);

FILE * g_fopen (const gchar *filename, const gchar *mode);

FILE * g_freopen (const gchar *filename, const gchar *mode, FILE *stream);

int g_chmod (const gchar *filename, int mode);

int g_access (const gchar *filename, int mode);

int g_creat (const gchar *filename, int mode);

int g_chdir (const gchar *path);

int g_utime (const gchar *filename, struct utimbuf *utb);
```

**Description**

There is a group of functions which wrap the common POSIX functions dealing with filenames (g_open(), g_rename(), g_mkdir(), g_stat(), g_unlink(), g_remove(), g_fopen(), g_freopen()). The point of these wrappers is to make it possible to handle file names with any Unicode characters in them on Windows without having to use ifdefs and the wide character API in the application code.

The pathname argument should be in the GLib file name encoding. On POSIX this is the actual on-disk encoding which might correspond to the locale settings of the process (or the G_FILENAME_ENCODING environment variable), or not.

On Windows the GLib file name encoding is UTF-8. Note that the Microsoft C library does not use UTF-8, but has separate APIs for current system code page and wide characters (UTF-16). The GLib wrappers call the wide character API if present (on modern Windows systems), otherwise convert to/from the system code page.

Another group of functions allows to open and read directories in the GLib file name encoding. These are g_dir_open(), g_dir_read_name(), g_dir_rewind(), g_dir_close().

**Details**

```c

typedef enum
{
    G_FILE_ERROR_EXIST,
    G_FILE_ERROR_ISDIR,
    G_FILE_ERROR_ACCES,
    G_FILE_ERROR>NamaTooLong,
    G_FILE_ERROR_NOENT,
    G_FILE_ERROR_NOTDIR,
};
```
Values corresponding to `errno` codes returned from file operations on UNIX. Unlike `errno` codes, `GFileError` values are available on all systems, even Windows. The exact meaning of each code depends on what sort of file operation you were performing; the UNIX documentation gives more details. The following error code descriptions come from the GNU C Library manual, and are under the copyright of that manual.

It's not very portable to make detailed assumptions about exactly which errors will be returned from a given operation. Some errors don't occur on some systems, etc., sometimes there are subtle differences in when a system will report a given error, etc.

- **G_FILE_ERROR_EXIST**: Operation not permitted; only the owner of the file (or other resource) or processes with special privileges can perform the operation.
- **G_FILE_ERROR_ISDIR**: File is a directory; you cannot open a directory for writing, or create or remove hard links to it.
- **G_FILE_ERROR_ACCESS**: Permission denied; the file permissions do not allow the attempted operation.
- **G_FILE_ERROR_NAME_TOO_LONG**: Filename too long.
- **G_FILE_ERROR_NO_ENTRY**: No such file or directory. This is a "file doesn’t exist" error for ordinary files that are referenced in contexts where they are expected to already exist.
- **G_FILE_ERROR_NOT_DIRECTORY**: A file that isn’t a directory was specified when a directory is required.
- **G_FILE_ERROR_SYS_IO**: No such device or address. The system tried to use the device represented by a file you specified, and it couldn’t find the device. This can mean that the device file was installed incorrectly, or that the physical device is missing or not correctly attached to the computer.
- **G_FILE_ERROR_NODEV**: This file is of a type that doesn’t support mapping.
- **G_FILE_ERROR_RDONLY**: The directory containing the new link can’t be modified because it’s on a read-only file system.
- **G_FILE_ERROR_TEXT_BUSY**: Text file busy.
- **G_FILE_ERROR_FAULT**: You passed in a pointer to bad memory. (GLib won’t reliably return this, don’t pass in pointers to bad memory.)
- **G_FILE_ERROR_LOOP**: Too many levels of symbolic links were encountered in looking up a file name. This often indicates a cycle of symbolic links.
- **G_FILE_ERROR_NO_SPACE**: No space left on device; write operation on a file failed because the disk is full.
- **G_FILE_ERROR_NO_MEMORY**: No memory available. The system cannot allocate more virtual memory because its capacity is full.
G_FILE_ERROR_MFILE The current process has too many files open and can’t open any more. Duplicate descriptors do count toward this limit.

G_FILE_ERROR_NFILE There are too many distinct file openings in the entire system.

G_FILE_ERROR_BADF Bad file descriptor; for example, I/O on a descriptor that has been closed or reading from a descriptor open only for writing (or vice versa).

G_FILE_ERROR_INVAIL Invalid argument. This is used to indicate various kinds of problems with passing the wrong argument to a library function.

G_FILE_ERROR_PIPE Broken pipe; there is no process reading from the other end of a pipe. Every library function that returns this error code also generates a ‘SIGPIPE’ signal; this signal terminates the program if not handled or blocked. Thus, your program will never actually see this code unless it has handled or blocked ‘SIGPIPE’.

G_FILE_ERROR_AGAIN Resource temporarily unavailable; the call might work if you try again later.

G_FILE_ERROR_INTR Interrupted function call; an asynchronous signal occurred and prevented completion of the call. When this happens, you should try the call again.

G_FILE_ERROR_IO Input/output error; usually used for physical read or write errors. i.e. the disk or other physical device hardware is returning errors.

G_FILE_ERROR_PERM Operation not permitted; only the owner of the file (or other resource) or processes with special privileges can perform the operation.

G_FILE_ERROR_NOSYS Function not implemented; this indicates that the system is missing some functionality.

G_FILE_ERROR_FAILED Does not correspond to a UNIX error code; this is the standard “failed for unspecified reason” error code present in all GError error code enumerations. Returned if no specific code applies.

G_FILE_ERROR

#define G_FILE_ERROR g_file_error_quark ()

Error domain for file operations. Errors in this domain will be from the GFileError enumeration. See GError for information on error domains.

enum GFileTest

typedef enum
{
   G_FILE_TEST_IS_REGULAR = 1 << 0,
   G_FILE_TEST_IS_SYMLINK = 1 << 1,
   G_FILE_TEST_IS_DIR = 1 << 2,
   G_FILE_TEST_IS_EXECUTABLE = 1 << 3,
   G_FILE_TEST_EXISTS = 1 << 4
} GFileTest;

A test to perform on a file using g_file_test().

G_FILE_TEST_IS_REGULAR %TRUE if the file is a regular file (not a directory). Note that this test will also return TRUE if the tested file is a symlink to a regular file.

G_FILE_TEST_IS_SYMLINK %TRUE if the file is a symlink.

G_FILE_TEST_IS_DIR %TRUE if the file is a directory.

G_FILE_TEST_IS_EXECUTABLE %TRUE if the file is executable.

G_FILE_TEST_EXISTS %TRUE if the file exists. It may or may not be a regular file.
CHAPTER 4. GLIB UTILITIES

4.15. FILE UTILITIES

**g_file_error_from_errno()**

```c
GFileError g_file_error_from_errno (gint err_no);
```

Gets a `GFileError` constant based on the passed-in `errno`. For example, if you pass in `EEXIST` this function returns `G_FILE_ERROR_EXIST`. Unlike `errno` values, you can portably assume that all `GFileError` values will exist.

Normally a `GFileError` value goes into a `GError` returned from a function that manipulates files. So you would use `g_file_error_from_errno()` when constructing a `GError`.

*err_no*: an "errno" value

*Returns*: `GFileError` corresponding to the given `errno`

**g_file_get_contents()**

```c
gboolean g_file_get_contents (const gchar *filename, gchar **contents, gsize *length, GError **error);
```

Reads an entire file into allocated memory, with good error checking.

If the call was successful, it returns `TRUE` and sets `contents` to the file contents and `length` to the length of the file contents in bytes. The string stored in `contents` will be null-terminated, so for text files you can pass `NULL` for the `length` argument. If the call was not successful, it returns `FALSE` and sets `error`. The error domain is `G_FILE_ERROR`. Possible error codes are those in the `GFileError` enumeration. In the error case, `contents` is set to `NULL` and `length` is set to zero.

*filename*: name of a file to read contents from, in the GLib file name encoding

*contents*: location to store an allocated string, use `g_free()` to free the returned string

*length*: location to store length in bytes of the contents, or `NULL`

*error*: return location for a `GError`, or `NULL`

*Returns*: `TRUE` on success, `FALSE` if an error occurred

**g_file_set_contents()**

```c
gboolean g_file_set_contents (const gchar *filename, const gchar *contents, gssize length, GError **error);
```

Writes all of `contents` to a file named `filename`, with good error checking. If a file called `filename` already exists it will be overwritten.

This write is atomic in the sense that it is first written to a temporary file which is then renamed to the final name. Notes:

- On Unix, if `filename` already exists hard links to `filename` will break. Also since the file is recreated, existing permissions, access control lists, metadata etc. may be lost. If `filename` is a symbolic link, the link itself will be replaced, not the linked file.

- On Windows renaming a file will not remove an existing file with the new name, so on Windows there is a race condition between the existing file being removed and the temporary file being renamed.

- On Windows there is no way to remove a file that is open to some process, or mapped into memory. Thus, this function will fail if `filename` already exists and is open.

If the call was successful, it returns `TRUE`. If the call was not successful, it returns `FALSE` and sets `error`. The error domain is `G_FILE_ERROR`. Possible error codes are those in the `GFileError` enumeration.
**filename**: name of a file to write `contents` to, in the GLib file name encoding

**contents**: string to write to the file

**length**: length of `contents`, or -1 if `contents` is a null-terminated string

**error**: return location for a GError, or NULL

**Returns**: TRUE on success, FALSE if an error occurred

Since 2.8

```c
gboolean g_file_test (const gchar *filename, GFileTest test);
```

Returns TRUE if any of the tests in the bitfield `test` are TRUE. For example, `(G_FILE_TEST_EXISTS | G_FILE_TEST_IS_DIR)` will return TRUE if the file exists; the check whether it’s a directory doesn’t matter since the existence test is TRUE. With the current set of available tests, there’s no point passing in more than one test at a time.

Apart from `G_FILE_TEST_IS_SYMLINK` all tests follow symbolic links, so for a symbolic link to a regular file `g_file_test()` will return TRUE for both `G_FILE_TEST_IS_SYMLINK` and `G_FILE_TEST_IS_REGULAR`.

Note, that for a dangling symbolic link `g_file_test()` will return TRUE for `G_FILE_TEST_IS_SYMLINK` and FALSE for all other flags.

You should never use `g_file_test()` to test whether it is safe to perform an operation, because there is always the possibility of the condition changing before you actually perform the operation. For example, you might think you could use `G_FILE_TEST_IS_SYMLINK` to know whether it is safe to write to a file without being tricked into writing into a different location. It doesn’t work!

```c
/* DON’T DO THIS */
if (!g_file_test (filename, G_FILE_TEST_IS_SYMLINK))
{
    fd = g_open (filename, O_WRONLY);
    /* write to fd */
}
```

Another thing to note is that `G_FILE_TEST_EXISTS` and `G_FILE_TEST_IS_EXECUTABLE` are implemented using the `access()` system call. This usually doesn’t matter, but if your program is setuid or setgid it means that these tests will give you the answer for the real user ID and group ID, rather than the effective user ID and group ID.

On Windows, there are no symlinks, so testing for `G_FILE_TEST_IS_SYMLINK` will always return FALSE. Testing for `G_FILE_TEST_IS_EXECUTABLE` will just check that the file exists and its name indicates that it is executable, checking for well-known extensions and those listed in the `PATHEXT` environment variable.

**filename**: a filename to test in the GLib file name encoding

**test**: bitfield of GFileTest flags

**Returns**: whether a test was TRUE

```c
gint g_mkstemp (gchar *tmpl);
```

Opens a temporary file. See the `mkstemp()` documentation on most UNIX-like systems.

The parameter is a string that should follow the rules for `mkstemp()` templates, i.e. contain the string "XXXXXX". `g_mkstemp()` is slightly more flexible than `mkstemp()` in that the sequence does not have to occur at the very end of the template. The X string will be modified to form the name of a file that didn’t exist. The string should be in the GLib file name encoding. Most importantly, on Windows it should be in UTF-8.
**tmpl**: template filename

*Returns*: A file handle (as from `open()`) to the file opened for reading and writing. The file is opened in binary mode on platforms where there is a difference. The file handle should be closed with `close()`. In case of errors, -1 is returned.

**g_file_open_tmp**

```c
gint g_file_open_tmp (const gchar *tmpl, gchar **name_used, GError **error);
```

Opens a file for writing in the preferred directory for temporary files (as returned by `g_get_tmp_dir()`). `tmpl` should be a string in the GLib file name encoding containing a sequence of six ‘X’ characters, as the parameter to `g_mkstemp()`. However, unlike these functions, the template should only be a basename, no directory components are allowed. If `tmpl` is NULL, a default template is used.

Note that in contrast to `g_mkstemp()` (and `mkstemp()`) `tmpl` is not modified, and might thus be a read-only literal string.

The actual name used is returned in `name_used` if non-NULL. This string should be freed with `g_free()` when not needed any longer. The returned name is in the GLib file name encoding.

- `tmpl`: Template for file name, as in `g_mkstemp()`, basename only, or NULL, to a default template
- `name_used`: location to store actual name used, or NULL
- `error`: return location for a GError

*Returns*: A file handle (as from `open()`) to the file opened for reading and writing. The file is opened in binary mode on platforms where there is a difference. The file handle should be closed with `close()`. In case of errors, -1 is returned and `error` will be set.

**g_file_read_link**

```c
gchar * g_file_read_link (const gchar *filename, GError **error);
```

Reads the contents of the symbolic link `filename` like the POSIX `readlink()` function. The returned string is in the encoding used for filenames. Use `g_filename_to_utf8()` to convert it to UTF-8.

- `filename`: the symbolic link
- `error`: return location for a GError

*Returns*: A newly-allocated string with the contents of the symbolic link, or NULL if an error occurred.

Since 2.4

**g_mkdir_with_parents**

```c
int g_mkdir_with_parents (const gchar *pathname, int mode);
```

Create a directory if it doesn’t already exist. Create intermediate parent directories as needed, too.

- `pathname`: a pathname in the GLib file name encoding
- `mode`: permissions to use for newly created directories

*Returns*: 0 if the directory already exists, or was successfully created. Returns -1 if an error occurred, with `errno` set.

Since 2.8
GDirm

typedef struct _GDir GDir;

An opaque structure representing an opened directory.

g_dir_open()

GDirm * g_dir_open (const gchar *path, guint flags, GError **error);

Opens a directory for reading. The names of the files in the directory can then be retrieved using g_dir_read_name().

path: the path to the directory you are interested in. On Unix in the on-disk encoding. On Windows in UTF-8

flags: Currently must be set to 0. Reserved for future use.

error: return location for a GError, or NULL. If non-NULL, an error will be set if and only if g_dir_open() fails.

Returns: a newly allocated GDir on success, NULL on failure. If non-NULL, you must free the result with g_dir_close() when you are finished with it.

g_dir_read_name()

const gchar * g_dir_read_name (GDir *dir);

Retrieves the name of the next entry in the directory. The ‘.’ and ‘..’ entries are omitted. On Windows, the returned name is in UTF-8. On Unix, it is in the on-disk encoding.

dir: a GDir* created by g_dir_open()

Returns: The entry’s name or NULL if there are no more entries. The return value is owned by GLib and must not be modified or freed.

g_dir_rewind()

void g_dir_rewind (GDir *dir);

Resets the given directory. The next call to g_dir_read_name() will return the first entry again.

dir: a GDir* created by g_dir_open()

g_dir_close()

void g_dir_close (GDir *dir);

Closes the directory and deallocates all related resources.

dir: a GDir* created by g_dir_open()

GMappedFile

typedef struct _GMappedFile GMappedFile;

The GMappedFile represents a file mapping created with g_mapped_file_new(). It has only private members and should not be accessed directly.
**g_mapped_file_new()**

Maps a file into memory. On UNIX, this is using the `mmap()` function. If `writable` is `TRUE`, the mapped buffer may be modified, otherwise it is an error to modify the mapped buffer. Modifications to the buffer are not visible to other processes mapping the same file, and are not written back to the file.

Note that modifications of the underlying file might affect the contents of the GMappedFile. Therefore, mapping should only be used if the file will not be modified, or if all modifications of the file are done atomically (e.g. using `g_file_set_contents()`).

**filename**: The path of the file to load, in the GLib filename encoding

**writable**: whether the mapping should be writable

**error**: return location for a GError, or NULL

**Returns**: a newly allocated GMappedFile which must be freed with `g_mapped_file_free()`, or NULL if the mapping failed.

Since 2.8

**g_mapped_file_free()**

Unmaps the buffer of `file` and frees it.

**file**: a GMappedFile

Since 2.8

**g_mapped_file_get_length()**

Returns the length of the contents of a GMappedFile.

**file**: a GMappedFile

**Returns**: the length of the contents of `file`.

Since 2.8

**g_mapped_file_get_contents()**

Returns the contents of a GMappedFile.

Note that the contents may not be zero-terminated, even if the GMappedFile is backed by a text file.

**file**: a GMappedFile

**Returns**: the contents of `file`.

Since 2.8
CHAPTER 4. GLIB UTILITIES

4.15. FILE UTILITIES

**g_open()**

```c
int g_open (const gchar *filename, int flags, int mode);
```

A wrapper for the POSIX `open()` function. The `open()` function is used to convert a pathname into a file descriptor.

On POSIX systems file descriptors are implemented by the operating system. On Windows, it’s the C library that implements `open()` and file descriptors. The actual Win32 API for opening files is quite different, see MSDN documentation for `CreateFile()`. The Win32 API uses file handles, which are more randomish integers, not small integers like file descriptors.

Because file descriptors are specific to the C library on Windows, the file descriptor returned by this function makes sense only to functions in the same C library. Thus if the GLib-using code uses a different C library than GLib does, the file descriptor returned by this function cannot be passed to C library functions like `write()` or `read()`.

See your C library manual for more details about `open()`.

- **filename**: a pathname in the GLib file name encoding (UTF-8 on Windows)
- **flags**: as in `open()`
- **mode**: as in `open()`

**Returns**: a new file descriptor, or -1 if an error occurred. The return value can be used exactly like the return value from `open()`.

Since 2.6

**g_rename()**

```c
int g_rename (const gchar *oldfilename, const gchar *newfilename);
```

A wrapper for the POSIX `rename()` function. The `rename()` function renames a file, moving it between directories if required.

See your C library manual for more details about how `rename()` works on your system. It is not possible in general on Windows to rename a file that is open to some process.

- **oldfilename**: a pathname in the GLib file name encoding (UTF-8 on Windows)
- **newfilename**: a pathname in the GLib file name encoding

**Returns**: 0 if the renaming succeeded, -1 if an error occurred

Since 2.6

**g_mkdir()**

```c
int g_mkdir (const gchar *filename, int mode);
```

A wrapper for the POSIX `mkdir()` function. The `mkdir()` function attempts to create a directory with the given name and permissions. The mode argument is ignored on Windows.

See your C library manual for more details about `mkdir()`.

- **filename**: a pathname in the GLib file name encoding (UTF-8 on Windows)
- **mode**: permissions to use for the newly created directory

**Returns**: 0 if the directory was successfully created, -1 if an error occurred

Since 2.6
CHAPTER 4. GLIB UTILITIES

4.15. FILE UTILITIES

**g_stat ()**

```c
int g_stat (const gchar *filename,
            struct stat *buf);
```

A wrapper for the POSIX stat() function. The stat() function returns information about a file. On Windows the stat() function in the C library checks only the FAT-style READONLY attribute and does not look at the ACL at all. Thus on Windows the protection bits in the st_mode field are a fabrication of little use.

See your C library manual for more details about stat().

*filename*: a pathname in the GLib file name encoding (UTF-8 on Windows)

*buf*: a pointer to a stat struct, which will be filled with the file information

**Returns**: 0 if the information was successfully retrieved, -1 if an error occurred

Since 2.6

**g_lstat ()**

```c
int g_lstat (const gchar *filename,
            struct stat *buf);
```

A wrapper for the POSIX lstat() function. The lstat() function is like stat() except that in the case of symbolic links, it returns information about the symbolic link itself and not the file that it refers to. If the system does not support symbolic links g_lstat() is identical to g_stat().

See your C library manual for more details about lstat().

*filename*: a pathname in the GLib file name encoding (UTF-8 on Windows)

*buf*: a pointer to a stat struct, which will be filled with the file information

**Returns**: 0 if the information was successfully retrieved, -1 if an error occurred

Since 2.6

**g_unlink ()**

```c
int g_unlink (const gchar *filename);
```

A wrapper for the POSIX unlink() function. The unlink() function deletes a name from the filesystem. If this was the last link to the file and no processes have it opened, the disk space occupied by the file is freed.

See your C library manual for more details about unlink(). Note that on Windows, it is in general not possible to delete files that are open to some process, or mapped into memory.

*filename*: a pathname in the GLib file name encoding (UTF-8 on Windows)

**Returns**: 0 if the name was successfully deleted, -1 if an error occurred

Since 2.6

**g_remove ()**

```c
int g_remove (const gchar *filename);
```

A wrapper for the POSIX remove() function. The remove() function deletes a name from the filesystem.

See your C library manual for more details about how remove() works on your system. On Unix, remove() removes also directories, as it calls unlink() for files and rmdir() for directories. On Windows, although remove() in the C library only works for files, this function tries first remove() and then if that
fails `rmdir()`, and thus works for both files and directories. Note however, that on Windows, it is in general not possible to remove a file that is open to some process, or mapped into memory.

If this function fails on Windows you can’t infer too much from the `errno` value. `rmdir()` is tried regardless of what caused `remove()` to fail. Any `errno` value set by `remove()` will be overwritten by that set by `rmdir()`.

**filename**: a pathname in the GLib file name encoding (UTF-8 on Windows)

**Returns**: 0 if the file was successfully removed, -1 if an error occurred

Since 2.6

**g_rmdir()**

```
int g_rmdir (const gchar *filename);
```

A wrapper for the POSIX `rmdir()` function. The `rmdir()` function deletes a directory from the filesystem.

See your C library manual for more details about how `rmdir()` works on your system.

**filename**: a pathname in the GLib file name encoding (UTF-8 on Windows)

**Returns**: 0 if the directory was successfully removed, -1 if an error occurred

Since 2.6

**g_fopen()**

```
FILE * g_fopen (const gchar *filename,
const gchar *mode);
```

A wrapper for the stdio `fopen()` function. The `fopen()` function opens a file and associates a new stream with it.

Because file descriptors are specific to the C library on Windows, and a file descriptor is part of the FILE struct, the FILE pointer returned by this function makes sense only to functions in the same C library. Thus if the GLib-using code uses a different C library than GLib does, the FILE pointer returned by this function cannot be passed to C library functions like `fprintf()` or `fread()`.

See your C library manual for more details about `fopen()`.

**filename**: a pathname in the GLib file name encoding (UTF-8 on Windows)

**mode**: a string describing the mode in which the file should be opened

**Returns**: A FILE pointer if the file was successfully opened, or `NULL` if an error occurred

Since 2.6

**g_freopen()**

```
FILE * g_freopen (const gchar *filename,
const gchar *mode,
FILE *stream);
```

A wrapper for the POSIX `freopen()` function. The `freopen()` function opens a file and associates it with an existing stream.

See your C library manual for more details about `freopen()`.

**filename**: a pathname in the GLib file name encoding (UTF-8 on Windows)

**mode**: a string describing the mode in which the file should be opened

**stream**: an existing stream which will be reused, or `NULL`

**Returns**: A FILE pointer if the file was successfully opened, or `NULL` if an error occurred.

Since 2.6
g_chmod ()

int g_chmod (const gchar *filename, int mode);

A wrapper for the POSIX chmod() function. The chmod() function is used to set the permissions of a file system object.

On Windows the file protection mechanism is not at all POSIX-like, and the underlying chmod() function in the C library just sets or clears the FAT-style READONLY attribute. It does not touch any ACL. Software that needs to manage file permissions on Windows exactly should use the Win32 API.

See your C library manual for more details about chmod().

filename: a pathname in the GLib file name encoding (UTF-8 on Windows)

mode: as in chmod()

Returns: zero if the operation succeeded, -1 on error.

Since 2.8

g_access ()

int g_access (const gchar *filename, int mode);

A wrapper for the POSIX access() function. This function is used to test a pathname for one or several of read, write or execute permissions, or just existence.

On Windows, the file protection mechanism is not at all POSIX-like, and the underlying function in the C library only checks the FAT-style READONLY attribute, and does not look at the ACL of a file at all. This function is this in practise almost useless on Windows. Software that needs to handle file permissions on Windows more exactly should use the Win32 API.

See your C library manual for more details about access().

filename: a pathname in the GLib file name encoding (UTF-8 on Windows)

mode: as in access()

Returns: zero if the pathname refers to an existing file system object that has all the tested permissions, or -1 otherwise or on error.

Since 2.8

g_creat ()

int g_creat (const gchar *filename, int mode);

A wrapper for the POSIX creat() function. The creat() function is used to convert a pathname into a file descriptor, creating a file if necessary.

On POSIX systems file descriptors are implemented by the operating system. On Windows, it's the C library that implements creat() and file descriptors. The actual Windows API for opening files is different, see MSDN documentation for CreateFile(). The Win32 API uses file handles, which are more randomish integers, not small integers like file descriptors.

Because file descriptors are specific to the C library on Windows, the file descriptor returned by this function makes sense only to functions in the same C library. Thus if the GLib-using code uses a different C library than GLib does, the file descriptor returned by this function cannot be passed to C library functions like write() or read().

See your C library manual for more details about creat().

filename: a pathname in the GLib file name encoding (UTF-8 on Windows)

mode: as in creat()

Returns: a new file descriptor, or -1 if an error occurred. The return value can be used exactly like the return value from creat().

Since 2.8
4.16. URI FUNCTIONS

**g_chdir ()**

```c
int g_chdir (const gchar *path);
```

A wrapper for the POSIX `chdir()` function. The function changes the current directory of the process to `path`.
See your C library manual for more details about `chdir()`.

**Path**: a pathname in the GLib file name encoding (UTF-8 on Windows)

**Returns**: 0 on success, -1 if an error occurred.

Since 2.8

**g_utime ()**

```c
int g_utime (const gchar *filename, struct utimbuf *utb);
```

A wrapper for the POSIX `utime()` function. The `utime()` function sets the access and modification timestamps of a file.
See your C library manual for more details about how `utime()` works on your system.

**Filename**: a pathname in the GLib file name encoding (UTF-8 on Windows)

**Utbd**: a pointer to a struct utimbuf.

**Returns**: 0 if the operation was successful, -1 if an error occurred.

Since 2.18

### 4.16 URI Functions

**Name**

URI Functions – URI Functions

**Synopsis**

```c
#include <glib.h>
```

```c
#define G_URI_RESERVED_CHARS_ALLOWED_IN_PATH
#define G_URI_RESERVED_CHARS_ALLOWED_IN_PATH_ELEMENT
#define G_URI_RESERVED_CHARS_ALLOWED_IN_USERINFO
#define G_URI_RESERVED_CHARS_GENERIC_DELIMITERS
#define G_URI_RESERVED_CHARS_SUBCOMPONENT_DELIMITERS

char * g_uri_parse_scheme (const char *uri);
char * g_uri_escape_string (const char *unescaped, const char *reserved_chars_allowed, gboolean allow_utf8);
char * g_uri_unescape_string (const char *escaped_string, const char *illegal_characters);
char * g_uri_unescape_segment (const char *escaped_string, const char *escaped_string_end, const char *illegal_characters);
```

**Description**

Functions for manipulating Universal Resource Identifiers (URIs) as defined by RFC 3986. It is highly recommended that you have read and understand RFC 3986 for understanding this API.
Details

G_URI_RESERVED_CHARS_ALLOWED_IN_PATH

#define G_URI_RESERVED_CHARS_ALLOWED_IN_PATH
G_URI_RESERVED_CHARS_ALLOWED_IN_PATH_ELEMENT "/"

Allowed characters in a path. Includes "!$&'(*+,;=@"/".  

G_URI_RESERVED_CHARS_ALLOWED_IN_PATH_ELEMENT

#define G_URI_RESERVED_CHARS_ALLOWED_IN_PATH_ELEMENT
G_URI_RESERVED_CHARS_SUBCOMPONENT_DELIMITERS ":@"

Allowed characters in path elements. Includes "!$&'(*+,;=@".  

G_URI_RESERVED_CHARS_ALLOWED_IN_USERINFO

#define G_URI_RESERVED_CHARS_ALLOWED_IN_USERINFO
G_URI_RESERVED_CHARS_SUBCOMPONENT_DELIMITERS ":"

Allowed characters in userinfo as defined in RFC 3986. Includes "!$&'(*+,;=".  

G_URI_RESERVED_CHARS_GENERIC_DELIMITERS

#define G_URI_RESERVED_CHARS_GENERIC_DELIMITERS
G_URI_RESERVED_CHARS_SUBCOMPONENT_DELIMITERS ":/?#[]@"

Generic delimiters characters as defined in RFC 3986. Includes "!:/?#[@".  

G_URI_RESERVED_CHARS_SUBCOMPONENT_DELIMITERS

#define G_URI_RESERVED_CHARS_SUBCOMPONENT_DELIMITERS
G_URI_RESERVED_CHARS_GENERIC_DELIMITERS ":!?&'(*+,;="

Subcomponent delimiter characters as defined in RFC 3986. Includes "!$&'(*+,;=".  

**g_uri_parse_scheme()**

```c
char * g_uri_parse_scheme (const char *uri);
```

Gets the scheme portion of a URI string. RFC 3986 decodes the scheme as:

URI = scheme ":" hier-part [ "?" query ] [ "#" fragment ]

Common schemes include "file", "http", "svn+ssh", etc.

**uri**: a valid URI.

**Returns**: The “Scheme” component of the URI, or NULL on error. The returned string should be freed when no longer needed.

Since 2.16

**g_uri_escape_string()**

```c
char * g_uri_escape_string (const char *unescaped,
const char *reserved_chars_allowed
, gboolean allow_utf8);
```
CHAPTER 4. GLIB UTILITIES

4.16. URI FUNCTIONS

Escapes a string for use in a URI.

Normally all characters that are not "unreserved" (i.e. ASCII alphanumerical characters plus dash, dot, underscore and tilde) are escaped. But if you specify characters in reserved_chars_allowed they are not escaped. This is useful for the "reserved" characters in the URI specification, since those are allowed unescaped in some portions of a URI.

escaped: the unescaped input string.

reserved_chars_allowed: a string of reserved characters that are allowed to be used.

allow_utf8: TRUE if the result can include UTF-8 characters.

Returns: an escaped version of escaped. The returned string should be freed when no longer needed.

Since 2.16

g_uri_unescape_string ()

char * g_uri_unescape_string (const char * escaped_string,
const char * reserved_chars_allowed,
const char * illegal_characters);

Unescapes a whole escaped string.

If any of the characters in illegal_characters or the character zero appears as an escaped character in escaped_string then that is an error and NULL will be returned. This is useful if you want to avoid for instance having a slash being expanded in an escaped path element, which might confuse pathname handling.

escaped_string: an escaped string to be unescaped.

illegal_characters: an optional string of illegal characters not to be allowed.

Returns: an unescaped version of escaped_string. The returned string should be freed when no longer needed.

Since 2.16

g_uri_unescape_segment ()

char * g_uri_unescape_segment (const char * escaped_string,
const char * escaped_string_end,
const char * illegal_characters);

Unescapes a segment of an escaped string.

If any of the characters in illegal_characters or the character zero appears as an escaped character in escaped_string then that is an error and NULL will be returned. This is useful if you want to avoid for instance having a slash being expanded in an escaped path element, which might confuse pathname handling.

escaped_string: a string.

escaped_string_end: a string.

illegal_characters: an optional string of illegal characters not to be allowed.

Returns: an unescaped version of escaped_string or NULL on error. The returned string should be freed when no longer needed.

Since 2.16
4.17 Shell-related Utilities

Name
Shell-related Utilities – shell-like commandline handling

Synopsis

```c
#include <glib.h>
enum GShellError;
#define G_SHELL_ERROR
gboolean g_shell_parse_argv (const gchar *command_line,
gint *argcp,
gchar ***argvp,
GError **error);
gchar* g_shell_quote (const gchar *unquoted_string);
gchar* g_shell_unquote (const gchar *quoted_string,
GError **error);
```

Description

Details

```c
enum GShellError
{
    /* mismatched or otherwise mangled quoting */
    G_SHELL_ERROR_BAD_QUOTING,
    /* string to be parsed was empty */
    G_SHELL_ERROR_EMPTY_STRING,
    G_SHELL_ERROR_FAILED
} GShellError;
```

Error codes returned by shell functions.

**G_SHELL_ERROR_BAD_QUOTING** Mismatched or otherwise mangled quoting.

**G_SHELL_ERROR_EMPTY_STRING** String to be parsed was empty.

**G_SHELL_ERROR_FAILED** Some other error.

**G_SHELL_ERROR**

```c
#define G_SHELL_ERROR g_shell_error_quark ()
```

Error domain for shell functions. Errors in this domain will be from the GShellError enumeration. See GError for information on error domains.

**g_shell_parse_argv ()**

```c
gboolean g_shell_parse_argv (const gchar *command_line,
gint *argcp,
gchar ***argvp,
GError **error);
```
CHAPTER 4. GLIB UTILITIES

4.18. COMMANDLINE OPTION PARSER

Parses a command line into an argument vector, in much the same way the shell would, but without many of the expansions the shell would perform (variable expansion, globs, operators, filename expansion, etc. are not supported). The results are defined to be the same as those you would get from a UNIX98 /bin/sh, as long as the input contains none of the unsupported shell expansions. If the input does contain such expansions, they are passed through literally. Possible errors are those from the G_SHELL_ERROR domain. Free the returned vector with g_strfreev().

command_line: command line to parse
argc: return location for number of args
argv: return location for array of args
error: return location for error

Returns: TRUE on success, FALSE if error set

\texttt{g\_shell\_quote ()}

\begin{verbatim}
gchar* g_shell_quote (const gchar * unquoted_string);
\end{verbatim}

Quotes a string so that the shell (/bin/sh) will interpret the quoted string to mean \texttt{unquoted_string}. If you pass a filename to the shell, for example, you should first quote it with this function. The return value must be freed with \texttt{g\_free()}. The quoting style used is undefined (single or double quotes may be used).

unquoted_string: a literal string

Returns: quoted string

\texttt{g\_shell\_unquote ()}

\begin{verbatim}
gchar* g_shell_unquote (const gchar * quoted_string, GError **error);
\end{verbatim}

Unquotes a string as the shell (/bin/sh) would. Only handles quotes; if a string contains file globs, arithmetic operators, variables, backticks, redirections, or other special-to-the-shell features, the result will be different from the result a real shell would produce (the variables, backticks, etc. will be passed through literally instead of being expanded). This function is guaranteed to succeed if applied to the result of \texttt{g\_shell\_quote()}. If it fails, it returns NULL and sets the error. The \texttt{quoted_string} need not actually contain quoted or escaped text; \texttt{g\_shell\_unquote()} simply goes through the string and unquotes/unescape anything that the shell would. Both single and double quotes are handled, as are escapes including escaped newlines. The return value must be freed with \texttt{g\_free()}. Possible errors are in the G_SHELL_ERROR domain.

Shell quoting rules are a bit strange. Single quotes preserve the literal string exactly. escape sequences are not allowed; not even \texttt{\textbackslash -} if you want a \texttt{\textquoteleft} in the quoted text, you have to do something like \texttt{\textquotefoo\textquoteleft bar}. Double quotes allow \$\texttt{, , \textbackslash}, \texttt{\textbackslash}, and newline to be escaped with backslash. Otherwise double quotes preserve things literally.

quoted_string: shell-quoted string
error: error return location or NULL

Returns: an unquoted string

4.18 Commandline option parser

Name

Commandline option parser – parses commandline options
Synopsis

#include <glib.h>

enum GOptionError;
#define G_OPTION_ERROR
 gboolean (*GOptionArgFunc) (const gchar *option_name,
 const gchar *value,
 gpointer data,
 GError **error);
 gboolean (*GOptionArgFunc) (const gchar *parameter_string);
 gboolean g_option_context_new (const gchar *parameter_string);
 void g_option_context_set_summary (GOptionContext *context,
 const gchar *summary);
 const gchar * g_option_context_get_summary (GOptionContext *context);
 gboolean g_option_context_parse (GOptionContext *context,
 gint *argc,
 gchar ***argv,
 GError **error);
 gboolean g_option_context_parse (GOptionContext *context,
 gint *argc,
 gchar ***argv,
 GError **error);
 gboolean g_option_context_get_main_group (GOptionContext *context);
 gboolean g_option_context_get_main_group (GOptionContext *context);
 gboolean g_option_context_get_help_enabled (GOptionContext *context);
 gboolean g_option_context_get_help_enabled (GOptionContext *context);
 gboolean g_option_context_set_translation_domain (GOptionContext *context,
 const gchar *domain);
 gboolean g_option_context_set_translation_domain (GOptionContext *context,
 const gchar *domain);
 gboolean g_option_context_set_help_enabled (GOptionContext *context,
 gboolean help_enabled);
 gboolean g_option_context_set_help_enabled (GOptionContext *context,
 gboolean help_enabled);
 gboolean g_option_context_set_ignore_unknown_options (GOptionContext *context,
 gboolean ignore_unknown);
 gboolean g_option_context_set_ignore_unknown_options (GOptionContext *context,
 gboolean ignore_unknown);
 gchar * g_option_context_get_help (GOptionContext *context,
 gboolean main_help,
 GOptionGroup *group);
 gchar * g_option_context_get_help (GOptionContext *context,
 gboolean main_help,
 GOptionGroup *group);

#define G_OPTION_REMAINING
 enum GOptionArg;
 enum GOptionFlags;
 void g_option_context_add_main_entries (GOptionContext *context,
 const GOptionEntry *entries,
 const gchar *translation_domain);
 void g_option_context_add_main_entries (GOptionContext *context,
 const GOptionEntry *entries,
 const gchar *translation_domain);
 void g_option_context_add_group (GOptionContext *context,
 GOptionGroup *group);
 void g_option_context_add_group (GOptionContext *context,
 GOptionGroup *group);
 void g_option_context_set_main_group (GOptionContext *context,
 GOptionGroup *group);
 void g_option_context_set_main_group (GOptionContext *context,
 GOptionGroup *group);
 GOptionGroup * g_option_context_get_main_group (GOptionContext *context);
 GOptionGroup * g_option_context_get_main_group (GOptionContext *context);
Description

The GOption commandline parser is intended to be a simpler replacement for the popt library. It supports short and long commandline options, as shown in the following example:

```
testtreemodel -r 1 --max-size 20 --rand --display=:1.0 -vb -- file1 file2
```

The example demonstrates a number of features of the GOption commandline parser:

- Options can be single letters, prefixed by a single dash. Multiple short options can be grouped behind a single dash.
- Long options are prefixed by two consecutive dashes.
- Options can have an extra argument, which can be a number, a string or a filename. For long options, the extra argument can be appended with an equals sign after the option name.
- Non-option arguments are returned to the application as rest arguments.
- An argument consisting solely of two dashes turns off further parsing, any remaining arguments (even those starting with a dash) are returned to the application as rest arguments.

Another important feature of GOption is that it can automatically generate nicely formatted help output. Unless it is explicitly turned off with `g_option_context_set_help_enabled()`, GOption will recognize the `--help`, `--help-all` and `--help-groupname` options (where `groupname` is the name of a `GOptionGroup`) and write a text similar to the one shown in the following example to stdout.

Usage:

```
testtreemodel [OPTION...] - test tree model performance
```

Help Options:

```
-?, --help Show help options
--help-all Show all help options
--help-gtk Show GTK+ Options
```

Application Options:

```
-r, --repeats=N Average over N repetitions
```

```c
const gchar *help_description,
gpointer user_data,
GDestroyNotify destroy);
void g_option_group_free (GOptionGroup *group);
void g_option_group_add_entries (GOptionGroup *group,
const GOptionEntry *entries);
gboolean (*GOptionParseFunc) (GOptionContext *context,
GOptionGroup *group,
gpointer data,
GError **error);
void g_option_group_set_parse_hooks (GOptionGroup *group,
GOptionParseFunc pre_parse_func,
GOptionParseFunc post_parse_func);
void (*GOptionErrorFunc) (GOptionContext *context,
GOptionGroup *group,
gpointer data,
GError **error);
void g_option_group_set_error_hook (GOptionGroup *group,
GOptionErrorFunc error_func);
void g_option_group_set_translate_func (GOptionGroup *group,
GTranslateFunc func,
gpointer data,
GDestroyNotify destroy_notify);
void g_option_group_set_translation_domain (GOptionGroup *group,
const gchar *domain);
```
GOption groups options in GOptionGroups, which makes it easy to incorporate options from multiple sources. The intended use for this is to let applications collect option groups from the libraries it uses, add them to their GOptionContext, and parse all options by a single call to g_option_context_parse(). See gtk_get_option_group() for an example.

If an option is declared to be of type string or filename, GOption takes care of converting it to the right encoding; strings are returned in UTF-8, filenames are returned in the GLib filename encoding. Note that this only works if setlocale() has been called before g_option_context_parse().

Here is a complete example of setting up GOption to parse the example commandline above and produce the example help output.

```c
static gint repeats = 2;
static gint max_size = 8;
static gboolean verbose = FALSE;
static gboolean beep = FALSE;
static gboolean rand = FALSE;
static GOptionEntry entries[] =
{
    { "repeats", 'r', 0, G_OPTION_ARG_INT, &repeats, "Average over N repetitions", "N" },
    { "max-size", 'm', 0, G_OPTION_ARG_INT, &max_size, "Test up to 2^M items", "M" },
    { "verbose", 'v', 0, G_OPTION_ARG_NONE, &verbose, "Be verbose", NULL },
    { "beep", 'b', 0, G_OPTION_ARG_NONE, &beep, "Beep when done", NULL },
    { "rand", 0, 0, G_OPTION_ARG_NONE, &rand, "Randomize the data", NULL },
    { NULL }
};

int
main (int argc, char *argv[])
{
    GError *error = NULL;
    GOptionContext *context;

    context = g_option_context_new ("- test tree model performance");
    g_option_context_add_main_entries (context, entries, GETTEXT_PACKAGE);
    g_option_context_add_group (context, gtk_get_option_group (TRUE));
    if (!g_option_context_parse (context, &argc, &argv, &error))
    {
        g_print ("option parsing failed: %s\n", error->message);
        exit (1);
    }

    // ...
}
```

**Details**

enum GOptionError

```c
typedef enum
{
    G_OPTION_ERROR_UNKNOWN_OPTION,
    G_OPTION_ERROR_BAD_VALUE,
    G_OPTION_ERROR_FAILED
};
```
CHAPTER 4. GLIB UTILITIES

4.18. COMMANDLINE OPTION PARSER

GOptionError;

Error codes returned by option parsing.

G_OPTION_ERROR_UNKNOWN_OPTION An option was not known to the parser. This error will only be
reported, if the parser hasn’t been instructed to ignore unknown options, see g_option_context_set_ignore_unknown_options().

G_OPTION_ERROR_BAD_VALUE A value couldn’t be parsed.

G_OPTION_ERROR_FAILED A GOptionArgFunc callback failed.

G_OPTION_ERROR

#define G_OPTION_ERROR (g_option_error_quark ())

Error domain for option parsing. Errors in this domain will be from the GOptionError enumeration. See GError for information on error domains.

GOptionArgFunc()

gboolean (*GOptionArgFunc)(const gchar *option_name, const gchar *value, gpointer data, GError **error);

The type of function to be passed as callback for G_OPTION_ARG_CALLBACK options.

option_name: The name of the option being parsed. This will be either a single dash followed by a
single letter (for a short name) or two dashes followed by a long option name.

value: The value to be parsed.

data: User data added to the GOptionGroup containing the option when it was created with g_option_group_new().

error: A return location for errors. The error code G_OPTION_ERROR_FAILED is intended to be used
for errors in GOptionArgFunc callbacks.

Returns: TRUE if the option was successfully parsed, FALSE if an error occurred, in which case error
should be set with g_set_error().

GOptionContext

typedef struct _GOptionContext GOptionContext;

A GOptionContext struct defines which options are accepted by the commandline option parser. The
struct has only private fields and should not be directly accessed.

g_option_context_new()

GOptionContext * g_option_context_new(const gchar * parameter_string);

Creates a new option context.

The parameter_string can serve multiple purposes. It can be used to add descriptions for "rest" arguments, which are not parsed by the GOptionContext, typically something like "FILES" or "FILE1 FILE2...". If you are using G_OPTION_REMAINING for collecting "rest" arguments, GLib handles this automatically by using the arg_description of the corresponding GOptionEntry in the usage summary.

Another usage is to give a short summary of the program functionality, like " - frob the strings", which
will be displayed in the same line as the usage. For a longer description of the program functionality
that should be displayed as a paragraph below the usage line, use g_option_context_set_summary().

Note that the parameter_string is translated using the function set with g_option_context_set_translate_func(),
so it should normally be passed untranslated.
**parameter_string**: a string which is displayed in the first line of `--help` output, after the usage summary *programname* [OPTION...]  

**Returns**: a newly created `GOptionContext`, which must be freed with `g_option_context_free()` after use.

Since 2.6

**g_option_context_set_summary ()**

```c
void g_option_context_set_summary (GOptionContext *context, const gchar *summary);
```

Adds a string to be displayed in `--help` output before the list of options. This is typically a summary of the program functionality.

Note that the summary is translated (see `g_option_context_set_translate_func()` and `g_option_context_set_translation_domain()`).

**context**: a `GOptionContext`

**summary**: a string to be shown in `--help` output before the list of options, or `NULL`

Since 2.12

**g_option_context_get_summary ()**

```c
const gchar * g_option_context_get_summary (GOptionContext *context);
```

Returns the summary. See `g_option_context_set_summary()`.

**context**: a `GOptionContext`

**Returns**: the summary

Since 2.12

**g_option_context_set_description ()**

```c
void g_option_context_set_description (GOptionContext *context, const gchar *description);
```

Adds a string to be displayed in `--help` output after the list of options. This text often includes a bug reporting address.

Note that the summary is translated (see `g_option_context_set_translate_func()`).

**context**: a `GOptionContext`

**description**: a string to be shown in `--help` output after the list of options, or `NULL`

Since 2.12

**g_option_context_get_description ()**

```c
const gchar * g_option_context_get_description (GOptionContext *context);
```

Returns the description. See `g_option_context_set_description()`.

**context**: a `GOptionContext`

**Returns**: the description

Since 2.12
CHAPTER 4. GLIB UTILITIES 4.18. COMMANDLINE OPTION PARSER

GTranslateFunc()

```c
const gchar * (*GTranslateFunc) (const gchar *str, gpointer data);
```

The type of functions which are used to translate user-visible strings, for --help output.

**str**: the untranslated string

**data**: user data specified when installing the function, e.g. in `g_option_group_set_translate_func()`

**Returns**: a translation of the string for the current locale. The returned string is owned by GLib and must not be freed.

**g_option_context_set_translate_func()**

```c
void g_option_context_set_translate_func (GOptionContext *context, GTranslateFunc func, gpointer data, GDestroyNotify destroy_notify);
```

Sets the function which is used to translate the contexts user-visible strings, for --help output. If **func** is NULL, strings are not translated.

Note that option groups have their own translation functions, this function only affects the parameter string (see `g_option_context_new()`), the summary (see `g_option_context_set_summary()`) and the description (see `g_option_context_set_description()`).

If you are using gettext(), you only need to set the translation domain, see `g_option_context_set_translation_domain()`.

**context**: a GOptionContext

**func**: the GTranslateFunc, or NULL

**data**: user data to pass to **func**, or NULL

**destroy_notify**: a function which gets called to free **data**, or NULL

Since 2.12

**g_option_context_set_translation_domain()**

```c
void g_option_context_set_translation_domain (GOptionContext *context, const gchar *domain);
```

A convenience function to use gettext() for translating user-visible strings.

**context**: a GOptionContext

**domain**: the domain to use

Since 2.12

**g_option_context_free()**

```c
void g_option_context_free (GOptionContext *context);
```

Frees context and all the groups which have been added to it.

Please note that parsed arguments need to be freed separately (see GOptionEntry).

**context**: a GOptionContext

Since 2.6
CHAPTER 4. GLIB UTILITIES 4.18. COMMANDLINE OPTION PARSER

**g_option_context_parse ()**

```c
gboolean g_option_context_parse (GOptionContext *context,
                                gint **argc,
                                gchar ***argv,
                                GError **error);
```

Parses the command line arguments, recognizing options which have been added to `context`. A side-effect of calling this function is that `g_set_prgname()` will be called.

If the parsing is successful, any parsed arguments are removed from the array and `argc` and `argv` are updated accordingly. A `'-'` option is stripped from `argv` unless there are unparsed options before and after it, or some of the options after it start with `-`. In case of an error, `argc` and `argv` are left unmodified.

If automatic `--help` support is enabled (see `g_option_context_set_help_enabled()`), and the `argv` array contains one of the recognized help options, this function will produce help output to stdout and call `exit (0)`.

Note that function depends on the current locale for automatic character set conversion of string and filename arguments.

- **context**: a GOptionContext
- **argc**: a pointer to the number of command line arguments
- **argv**: a pointer to the array of command line arguments
- **error**: a return location for errors

**Returns**: TRUE if the parsing was successful, FALSE if an error occurred

Since 2.6

**g_option_context_set_help_enabled ()**

```c
void g_option_context_set_help_enabled (GOptionContext *context,
                                       gboolean help_enabled);
```

Enables or disables automatic generation of `--help` output. By default, `g_option_context_parse()` recognizes `--help`, `-?`, `--help-all` and `--help-groupname` and creates suitable output to stdout.

- **context**: a GOptionContext
- **help_enabled**: TRUE to enable `--help`, FALSE to disable it

Since 2.6

**g_option_context_get_help_enabled ()**

```c
gboolean g_option_context_get_help_enabled (GOptionContext *context) ;
```

Returns whether automatic `--help` generation is turned on for `context`. See `g_option_context_set_help_enabled()`.

- **context**: a GOptionContext

**Returns**: TRUE if automatic help generation is turned on.

Since 2.6

354
g_option_context_set_ignore_unknown_options ()

```c
void g_option_context_set_ignore_unknown_options
    (GOptionContext *context,
     gboolean ignore_unknown) ←
```

Sets whether to ignore unknown options or not. If an argument is ignored, it is left in the argv array after parsing. By default, g_option_context_parse() treats unknown options as error.

This setting does not affect non-option arguments (i.e. arguments which don’t start with a dash). But note that GOption cannot reliably determine whether a non-option belongs to a preceding unknown option.

**context**: a GOptionContext

**ignore_unknown**: TRUE to ignore unknown options, FALSE to produce an error when unknown options are met

Since 2.6

**g_option_context_get_ignore_unknown_options ()**

```c
gboolean g_option_context_get_ignore_unknown_options
    (GOptionContext *context) ←
```

Returns whether unknown options are ignored or not. See g_option_context_set_ignore_unknown_options().

**context**: a GOptionContext

**Returns**: TRUE if unknown options are ignored.

Since 2.6

**g_option_context_get_help ()**

```c
gchar * g_option_context_get_help
    (GOptionContext *context,
     gboolean main_help,
     GOptionGroup *group);
```

Returns a formatted, translated help text for the given context. To obtain the text produced by --help, call g_option_context_get_help (context, TRUE, NULL). To obtain the text produced by --help-all, call g_option_context_get_help (context, FALSE, NULL). To obtain the help text for an option group, call g_option_context_get_help (context, FALSE, group).

**context**: a GOptionContext

**main_help**: if TRUE, only include the main group

**group**: the GOptionGroup to create help for, or NULL

**Returns**: A newly allocated string containing the help text

Since 2.14

**enum GOptionArg**

```c
typedef enum
{
    G_OPTION_ARG_NONE,
    G_OPTION_ARG_STRING,
    G_OPTION_ARG_INT,
    G_OPTION_ARG_CALLBACK,
    G_OPTION_ARG_FILENAME,
} 355
```
The `GOptionArg` enum values determine which type of extra argument the options expect to find. If an option expects an extra argument, it can be specified in several ways; with a short option: `-x` `arg`, with a long option: `--name arg` or combined in a single argument: `--name=arg`.

- **G_OPTION_ARG_NONE** No extra argument. This is useful for simple flags.
- **G_OPTION_ARG_STRING** The option takes a string argument.
- **G_OPTION_ARG_INT** The option takes an integer argument.
- **G_OPTION_ARG_CALLBACK** The option provides a callback to parse the extra argument.
- **G_OPTION_ARG_FILENAME** The option takes a filename as argument.
- **G_OPTION_ARG_STRING_ARRAY** The option takes a string argument, multiple uses of the option are collected into an array of strings.
- **G_OPTION_ARG_FILENAME_ARRAY** The option takes a filename as argument, multiple uses of the option are collected into an array of strings.
- **G_OPTION_ARG_DOUBLE** The option takes a double argument. The argument can be formatted either for the user’s locale or for the "C" locale. Since 2.12
- **G_OPTION_ARG_INT64** The option takes a 64-bit integer. Like **G_OPTION_ARG_INT** but for larger numbers. The number can be in decimal base, or in hexadecimal (when prefixed with `0x`, for example, `0xffffffff`). Since 2.12

```c
typedef enum
{
    G_OPTION_FLAG_HIDDEN = 1 << 0,
    G_OPTION_FLAG_IN_MAIN = 1 << 1,
    G_OPTION_FLAG_REVERSE = 1 << 2,
    G_OPTION_FLAG_NO_ARG = 1 << 3,
    G_OPTION_FLAG_FILENAME = 1 << 4,
    G_OPTION_FLAG_OPTIONAL_ARG = 1 << 5,
    G_OPTION_FLAG_NOALIAS = 1 << 6
} GOptionFlags;
```

Flags which modify individual options.

- **G_OPTION_FLAG_HIDDEN** The option doesn’t appear in `--help` output.
- **G_OPTION_FLAG_IN_MAIN** The option appears in the main section of the `--help` output, even if it is defined in a group.
- **G_OPTION_FLAG_REVERSE** For options of the **G_OPTION_ARG_NONE** kind, this flag indicates that the sense of the option is reversed.
- **G_OPTION_FLAG_NO_ARG** For options of the **G_OPTION_ARG_CALLBACK** kind, this flag indicates that the callback does not take any argument (like a **G_OPTION_ARG_NONE** option). Since 2.8
- **G_OPTION_FLAG_FILENAME** For options of the **G_OPTION_ARG_CALLBACK** kind, this flag indicates that the argument should be passed to the callback in the GLib filename encoding rather than UTF-8. Since 2.8
- **G_OPTION_FLAG_OPTIONAL_ARG** For options of the **G_OPTION_ARG_CALLBACK** kind, this flag indicates that the argument supply is optional. If no argument is given then data of GOptionParseFunc will be set to NULL. Since 2.8

---

356
**G_OPTION_FLAG_NOALIAS** This flag turns off the automatic conflict resolution which prefixes long option names with `groupname-` if there is a conflict. This option should only be used in situations where aliasing is necessary to model some legacy commandline interface. It is not safe to use this option, unless all option groups are under your direct control. Since 2.8.

**G_OPTION_REMAINING**

```c
#define G_OPTION_REMAINING ""
```

If a long option in the main group has this name, it is not treated as a regular option. Instead it collects all non-option arguments which would otherwise be left in `argv`. The option must be of type `G_OPTION_ARG_CALLBACK`, `G_OPTION_ARG_STRING_ARRAY` or `G_OPTION_ARG_FILENAME_ARRAY`. Using `G_OPTION_REMAINING` instead of simply scanning `argv` for leftover arguments has the advantage that `GOption` takes care of necessary encoding conversions for strings or filenames.

Since 2.6

**GOptionEntry**

```c
typedef struct {
    const gchar *long_name;
gchar short_name;
gint flags;

    GOptionArg arg;
gpointer arg_data;

    const gchar *description;
    const gchar *arg_description;
} GOptionEntry;
```

A `GOptionEntry` defines a single option. To have an effect, they must be added to a `GOptionGroup` with `g_option_context_add_main_entries()` or `g_option_group_add_entries()`.

**const gchar *long_name;** The long name of an option can be used to specify it in a commandline as `--long_name`. Every option must have a long name. To resolve conflicts if multiple option groups contain the same long name, it is also possible to specify the option as `--groupname-long_name`.

**gchar short_name;** If an option has a short name, it can be specified `--short_name` in a commandline. `short_name` must be a printable ASCII character different from ` '-'`, or zero if the option has no short name.

**gint flags;** Flags from `GOptionFlags`.

**GOptionArg arg;** The type of the option, as a `GOptionArg`.

**gpointer arg_data;** If the `arg` type is `G_OPTION_ARG_CALLBACK`, then `arg_data` must point to a `GOptionArgFunc` callback function, which will be called to handle the extra argument. Otherwise, `arg_data` is a pointer to a location to store the value, the required type of the location depends on the `arg` type:

- `G_OPTION_ARG_NONE` `gboolean`
- `G_OPTION_ARG_STRING` `gchar*`
- `G_OPTION_ARG_INT` `gint`
- `G_OPTION_ARG_FILENAME` `gchar*`
- `G_OPTION_ARG_STRING_ARRAY` `gchar**`
- `G_OPTION_ARG_FILENAME_ARRAY` `gchar**`
- `G_OPTION_ARG_DOUBLE` `gdouble`

357
If \texttt{arg} type is \texttt{G_OPTION_ARG_STRING} or \texttt{G_OPTION_ARG_FILENAME} the location will contain a newly allocated string if the option was given. That string needs to be freed by the callee using \texttt{g_free()}. Likewise if \texttt{arg} type is \texttt{G_OPTION_ARG_STRING_ARRAY} or \texttt{G_OPTION_ARG_FILENAME_ARRAY}, the data should be freed using \texttt{g_strfreev()}. 

\begin{verbatim}
const gchar * description; \end{verbatim}

the description for the option in \texttt{--help} output. The \texttt{description} is translated using the translate\_func of the group, see \texttt{g_option_group_set_translation_domain()}. 

\begin{verbatim}
const gchar * arg\_description; \end{verbatim}

The placeholder to use for the extra argument parsed by the option in \texttt{--help} output. The \texttt{arg\_description} is translated using the translate\_func of the group, see \texttt{g_option_group_set_translation_domain()}. 

\subsection*{g\_option\_context\_add\_main\_entries ()}

\begin{verbatim}
void g_option_context_add_main_entries (GOptionContext *context, const GOptionEntry * entries, const gchar * translation_domain);
\end{verbatim}

A convenience function which creates a main group if it doesn't exist, adds the \texttt{entries} to it and sets the translation domain. 

\begin{itemize}
  \item \texttt{context}: a \texttt{GOptionContext}
  \item \texttt{entries}: a NULL-terminated array of \texttt{GOptionEntry}s
  \item \texttt{translation\_domain}: a translation domain to use for translating the \texttt{--help} output for the options in \texttt{entries} with gettext(), or NULL
\end{itemize}

Since 2.6

\subsection*{GOptionGroup}

\begin{verbatim}
typedef struct _GOptionGroup GOptionGroup;
\end{verbatim}

A \texttt{GOptionGroup} struct defines the options in a single group. The struct has only private fields and should not be directly accessed.

All options in a group share the same translation function. Libraries which need to parse command-line options are expected to provide a function for getting a \texttt{GOptionGroup} holding their options, which the application can then add to its \texttt{GOptionContext}. 

\subsection*{g\_option\_context\_add\_group ()}

\begin{verbatim}
void g_option_context_add_group (GOptionContext *context, GOptionGroup *group);
\end{verbatim}

Adds a \texttt{GOptionGroup} to the \texttt{context}, so that parsing with \texttt{context} will recognize the options in the group. Note that the group will be freed together with the context when \texttt{g_option_context_free()} is called, so you must not free the group yourself after adding it to a context. 

\begin{itemize}
  \item \texttt{context}: a \texttt{GOptionContext}
  \item \texttt{group}: the group to add
\end{itemize}

Since 2.6
### 4.18. Commandline Option Parser

#### g_option_context_set_main_group

```c
void g_option_context_set_main_group (GOptionContext *context,
GOptionGroup *group);
```

Sets a GOptionGroup as main group of the `context`. This has the same effect as calling `g_option_context_add_group()`, the only difference is that the options in the main group are treated differently when generating --help output.

- **context**: a GOptionContext
- **group**: the group to set as main group

Since 2.6

#### g_option_context_get_main_group

```c
GOptionGroup * g_option_context_get_main_group (GOptionContext *context);
```

Returns a pointer to the main group of `context`.

- **context**: a GOptionContext

Returns:

- the main group of `context`, or NULL if `context` doesn’t have a main group. Note that group belongs to `context` and should not be modified or freed.

Since 2.6

#### g_option_group_new

```c
GOptionGroup * g_option_group_new (const gchar *name,
const gchar *description,
const gchar *help_description,
gpointer user_data,
GDestroyNotify destroy);
```

Creates a new GOptionGroup.

- **name**: the name for the option group, this is used to provide help for the options in this group with --help-name
- **description**: a description for this group to be shown in --help. This string is translated using the translation domain or translation function of the group
- **help_description**: a description for the --help-name option. This string is translated using the translation domain or translation function of the group
- **user_data**: user data that will be passed to the pre- and post-parse hooks, the error hook and to callbacks of G_OPTION_ARG_CALLBACK options, or NULL
- **destroy**: a function that will be called to free user_data, or NULL

Returns:

- a newly created option group. It should be added to a GOptionContext or freed with `g_option_group_free()`.

Since 2.6

#### g_option_group_free

```c
void g_option_group_free (GOptionGroup *group);
```

Frees a GOptionGroup. Note that you must not free groups which have been added to a GOptionContext.

- **group**: a GOptionGroup

Since 2.6
### g_option_group_add_entries ()

```c
void g_option_group_add_entries (GOptionGroup *group,
                                  const GOptionEntry *entries);
```

Adds the options specified in `entries` to `group`.

**group**: a `GOptionGroup`

**entries**: a NULL-terminated array of `GOptionEntry`s

Since 2.6

### GOptionParseFunc ()

```c
gboolean (*GOptionParseFunc) (GOptionContext *context,
                               GOptionGroup *group,
                               gpointer data,
                               GError **error);
```

The type of function that can be called before and after parsing.

**context**: The active `GOptionContext`

**group**: The group to which the function belongs

**data**: User data added to the `GOptionGroup` containing the option when it was created with `g_option_group_new()`

**error**: A return location for error details

**Returns**: `TRUE` if the function completed successfully, `FALSE` if an error occurred, in which case `error` should be set with `g_set_error()`

### g_option_group_set_parse_hooks ()

```c
void g_option_group_set_parse_hooks (GOptionGroup *group,
                                      GOptionParseFunc pre_parse_func,
                                      GOptionParseFunc post_parse_func);
```

 Associates two functions with `group` which will be called from `g_option_context_parse()` before the first option is parsed and after the last option has been parsed, respectively.

 Note that the user data to be passed to `pre_parse_func` and `post_parse_func` can be specified when constructing the group with `g_option_group_new()`.

**group**: a `GOptionGroup`

**pre_parse_func**: a function to call before parsing, or `NULL`

**post_parse_func**: a function to call after parsing, or `NULL`

Since 2.6

### GOptionErrorFunc ()

```c
void (*GOptionErrorFunc) (GOptionContext *context,
                           GOptionGroup *group,
                           gpointer data,
                           GError **error);
```

The type of function to be used as callback when a parse error occurs.

**context**: The active `GOptionContext`
**group**: The group to which the function belongs

**data**: User data added to the `GOptionGroup` containing the option when it was created with `g_option_group_new()`

**error**: The `GError` containing details about the parse error

### `g_option_group_set_error_hook ()`

```c
void g_option_group_set_error_hook (GOptionGroup *group,
        GOptionErrorFunc error_func);
```

Associates a function with `group` which will be called from `g_option_context_parse()` when an error occurs.

Note that the user data to be passed to `error_func` can be specified when constructing the group with `g_option_group_new()`.

**group**: a `GOptionGroup`

**error_func**: a function to call when an error occurs

Since 2.6

### `g_option_group_set_translate_func ()`

```c
void g_option_group_set_translate_func (GOptionGroup *group,
        GTranslateFunc func,
        gpointer data,
        GDestroyNotify destroy_notify);
```

Sets the function which is used to translate user-visible strings, for `--help` output. Different groups can use different `GTranslateFuncs`. If `func` is NULL, strings are not translated.

If you are using `gettext()`, you only need to set the translation domain, see `g_option_group_set_translation_domain()`.

**group**: a `GOptionGroup`

**func**: the `GTranslateFunc`, or NULL

**data**: user data to pass to `func`, or NULL

**destroy_notify**: a function which gets called to free `data`, or NULL

Since 2.6

### `g_option_group_set_translation_domain ()`

```c
void g_option_group_set_translation_domain (GOptionGroup *group,
        const gchar *domain);
```

A convenience function to use `gettext()` for translating user-visible strings.

**group**: a `GOptionGroup`

**domain**: the domain to use

Since 2.6

### 4.19 Glob-style pattern matching

**Name**

Glob-style pattern matching – matches strings against patterns containing ‘*’ (wildcard) and ‘?’ (joker)
Synopsis

#include <glib.h>

GPatternSpec;
GPatternSpec* g_pattern_spec_new (const gchar *pattern);
void g_pattern_spec_free (GPatternSpec *pspec);
gboolean g_pattern_spec_equal (GPatternSpec *pspec1,
                                   GPatternSpec *pspec2);
gboolean g_pattern_match (GPatternSpec *pspec,
                          guint string_length,
                          const gchar *string,
                          const gchar *string_reversed);
gboolean g_pattern_match_string (GPatternSpec *pspec,
                                   const gchar *string);
gboolean g_pattern_match_simple (const gchar *pattern,
                                   const gchar *string);

Description

The `g_pattern_match*` functions match a string against a pattern containing '*' and '?' wildcards
with similar semantics as the standard `glob()` function: '*' matches an arbitrary, possibly empty, string,
'?' matches an arbitrary character.

Note that in contrast to `glob()`, the '/' character can be matched by the wildcards, there are no '[]'
character ranges and '*' and '?' can not be escaped to include them literally in a pattern.

When multiple strings must be matched against the same pattern, it is better to compile the pattern to
a `GPatternSpec` using `g_pattern_spec_new()` and use `g_pattern_match_string()` instead of `g_pattern_match_simple()`.
This avoids the overhead of repeated pattern compilation.

Details

GPatternSpec

typedef struct _GPatternSpec GPatternSpec;

A GPatternSpec is the ‘compiled’ form of a pattern. This structure is opaque and its fields cannot be
accessed directly.

g_pattern_spec_new ()

GPatternSpec* g_pattern_spec_new (const gchar *pattern);

Compiles a pattern to a `GPatternSpec`.

`pattern`: a zero-terminated UTF-8 encoded string

Returns: a newly-allocated `GPatternSpec`

g_pattern_spec_free ()

void g_pattern_spec_free (GPatternSpec *pspec);

Frees the memory allocated for the `GPatternSpec`.

`pspec`: a `GPatternSpec`
CHAPTER 4. GLIB UTILITIES

4.19. GLOB-STYLE PATTERN MATCHING

**g_pattern_spec_equal ()**

```c
gboolean g_pattern_spec_equal (GPatternSpec *pspec1,
                               GPatternSpec *pspec2);
```

Compares two compiled pattern specs and returns whether they will match the same set of strings.

- **pspec1**: a GPatternSpec
- **pspec2**: another GPatternSpec

**Returns**: Whether the compiled patterns are equal

**g_pattern_match ()**

```c
gboolean g_pattern_match (GPatternSpec *pspec,
                          guint string_length,
                          const gchar *string,
                          const gchar *string_reversed);
```

Matches a string against a compiled pattern. Passing the correct length of the string given is mandatory. The reversed string can be omitted by passing NULL, this is more efficient if the reversed version of the string to be matched is not at hand, as `g_pattern_match()` will only construct it if the compiled pattern requires reverse matches.

Note that, if the user code will (possibly) match a string against a multitude of patterns containing wildcards, chances are high that some patterns will require a reversed string. In this case, it’s more efficient to provide the reversed string to avoid multiple constructions thereof in the various calls to `g_pattern_match()`.

Note also that the reverse of a UTF-8 encoded string can in general not be obtained by `g_strreverse()`. This works only if the string doesn’t contain any multibyte characters. GLib offers the `g_utf8_strreverse()` function to reverse UTF-8 encoded strings.

- **pspec**: a GPatternSpec
- **string_length**: the length of string (in bytes, i.e. `strlen`, not `g_utf8_strlen`)
- **string**: the UTF-8 encoded string to match
- **string_reversed**: the reverse of string or NULL

**Returns**: %TRUE if string matches `pspec`

**g_pattern_match_string ()**

```c
gboolean g_pattern_match_string (GPatternSpec *pspec,
                                 const gchar *string);
```

Matches a string against a compiled pattern. If the string is to be matched against more than one pattern, consider using `g_pattern_match()` instead while supplying the reversed string.

- **pspec**: a GPatternSpec
- **string**: the UTF-8 encoded string to match

**Returns**: %TRUE if string matches `pspec`
4.20 Perl-compatible regular expressions

Name
Perl-compatible regular expressions – matches strings against regular expressions

Synopsis

```c
#include <glib.h>

enum GRegexError;
#define G_REGEX_ERROR
enum GRegexCompileFlags;
enum GRegexMatchFlags;
GRegex * g_regex_new (const gchar *pattern,
const GRegexCompileFlags compile_options,
const GRegexMatchFlags match_options,
GError **error);
GRegex * g_regex_ref (GRegex *regex);
void g_regex_unref (GRegex *regex);
const gchar * g_regex_get_pattern (const GRegex *regex);
gint g_regex_get_max_backref (const GRegex *regex);
gint g_regex_get_capture_count (const GRegex *regex);
gint g_regex_get_string_number (const GRegex *regex,
const gchar *name);
gchar * g_regex_escape_string (const gchar *string,
gint length);
gboolean g_regex_match_simple (const gchar *pattern,
const gchar *string);
gboolean g_regex_match (const GRegex *regex,
const gchar *string,
const GRegexMatchFlags match_options,
GMatchInfo **match_info);
gboolean g_regex_match_full (const GRegex *regex,
const gchar *string,
gsize string_len,
const GRegexMatchFlags match_options,
GMatchInfo **match_info);
gboolean g_regex_match_full (const GRegex *regex,
const gchar *string,
gsize string_len,
const GRegexMatchFlags match_options,
GMatchInfo **match_info);
gboolean g_regex_match_full (const GRegex *regex,
const gchar *string,
gsize string_len,
const GRegexMatchFlags match_options,
GMatchInfo **match_info);
gboolean g_regex_match_full (const GRegex *regex,
const gchar *string,
gsize string_len,
const GRegexMatchFlags match_options,
GMatchInfo **match_info);
gboolean g_regex_match_full (const GRegex *regex,
const gchar *string,
gsize string_len,
const GRegexMatchFlags match_options,
GMatchInfo **match_info);
gboolean g_regex_match_full (const GRegex *regex,
const gchar *string,
gsize string_len,
const GRegexMatchFlags match_options,
GMatchInfo **match_info);
```
### Perl-Compatible Regular Expressions

- **gboolean** `g_regex_match_all` (const GRegex *regex, const gchar *string, GRegexMatchFlags match_options, GMatchInfo **match_info);
- **gboolean** `g_regex_match_all_full` (const GRegex *regex, const gchar *string, ssize string_len, gint start_position, GRegexMatchFlags match_options, GMatchInfo **match_info);
- **gchar** `g_regex_split_simple` (const gchar *pattern, const gchar *string, GRegexCompileFlags compile_options, GRegexMatchFlags match_options);
- **gchar** `g_regex_split` (const GRegex *regex, const gchar *string, GRegexMatchFlags match_options);
- **gchar** `g_regex_split_full` (const GRegex *regex, const gchar *string, ssize string_len, gint start_position, GRegexMatchFlags match_options, gint max_tokens, GError **error);
- **gchar** `g_regex_replace` (const GRegex *regex, const gchar *string, ssize string_len, gint start_position, const gchar *replacement, GRegexMatchFlags match_options, GError **error);
- **gchar** `g_regex_replace_literal` (const GRegex *regex, const gchar *string, ssize string_len, gint start_position, const gchar *replacement, GRegexMatchFlags match_options, GError **error);
- **gchar** `g_regex_replace_eval` (const GRegex *regex, const gchar *string, ssize string_len, gint start_position, GRegexMatchFlags match_options, GRegexEvalCallback eval, gpointer user_data, GError **error);
- **gboolean** `g_regex_check_replacement` (const gchar *replacement, gboolean *has_references, GError **error);
- **GMatchInfo** `g_match_info_get_regex` (GMatchInfo *match_info);
- **GMatchInfo** `g_match_info_get_string` (GMatchInfo *match_info);
- **void** `g_match_info_free` (GMatchInfo *match_info);
- **gboolean** `g_match_info_matches` (GMatchInfo *match_info);
- **gboolean** `g_match_info_next` (GMatchInfo *match_info, GError **error);
CHAPTER 4. GLIB UTILITIES 4.20. PERL-COMPATIBLE REGULAR...

The `g_regex_*()` functions implement regular expression pattern matching using syntax and semantics similar to Perl regular expression.

Some functions accept a `start_position` argument, setting it differs from just passing over a shortened string and setting `G_REGEX_MATCH_NOTBOL` in the case of a pattern that begins with any kind of lookbehind assertion. For example, consider the pattern "\Biss\B" which finds occurrences of "iss" in the middle of words. ("\B" matches only if the current position in the subject is not a word boundary.) When applied to the string "Mississipi" from the fourth byte, namely "issipi", it does not match, because "\B" is always false at the start of the subject, which is deemed to be a word boundary. However, if the entire string is passed, but with `start_position` set to 4, it finds the second occurrence of "iss" because it is able to look behind the starting point to discover that it is preceded by a letter.

Note that, unless you set the `G_REGEX_RAW` flag, all the strings passed to these functions must be encoded in UTF-8. The lengths and the positions inside the strings are in bytes and not in characters, so, for instance, "\xc3\xa0" (i.e. "à") is two bytes long but it is treated as a single character. If you set `G_REGEX_RAW` the strings can be non-valid UTF-8 strings and a byte is treated as a character, so "\xc3\xa0" is two bytes and two characters long.

When matching a pattern, "\n" matches only against a "\n" character in the string, and "\r" matches only a "\r" character. To match any newline sequence use "\R". This particular group matches either the two-character sequence CR + LF ("\r\n"), or one of the single characters LF (linefeed, U+000A, "\n"), VT (vertical tab, U+000B, "\v"), FF (formfeed, U+000C, "\f"), CR (carriage return, U+000D, "\r"), NEL (next line, U+0085), LS (line separator, U+2028), or PS (paragraph separator, U+2029).

The behaviour of the dot, circumflex, and dollar metacharacters are affected by newline characters, the default is to recognize any newline character (the same characters recognized by "\R"). This can be changed with `G_REGEX_NEWLINE_CR`, `G_REGEX_NEWLINE_LF` and `G_REGEX_NEWLINE_CRLF` compile options, and with `G_REGEX_MATCH_NEWLINE_ANY`, `G_REGEX_MATCH_NEWLINE_CR`, `G_REGEX_MATCH_NEWLINE_LF` and `G_REGEX_MATCH_NEWLINE_CRLF` match options. These settings are also relevant when compiling a pattern if `G_REGEX_EXTENDED` is set, and an unescaped “#” outside a character class is encountered. This indicates a comment that lasts until after the next newline.

Creating and manipulating the same GRegex structure from different threads is not a problem as GRegex does not modify its internal state between creation and destruction, on the other hand GMatchInfo is not threadsafe.

The regular expressions low level functionalities are obtained through the excellent PCRE library written by Philip Hazel.

Details

enum GRegexError
typedef enum
{
  G_REGEX_ERROR_COMPILE,
  G_REGEX_ERROR_OPTIMIZE,
  G_REGEX_ERROR_REPLACE,
  G_REGEX_ERROR_MATCH,
  G_REGEX_ERROR_INTERNAL,
  /* These are the error codes from PCRE + 100 */
  G_REGEX_ERROR_STRAY_BACKSLASH = 101,
  G_REGEX_ERROR MISSING_CONTROL_CHAR = 102,
  G_REGEX_ERROR UNRECOGNIZED_ESCAPE = 103,
  G_REGEX_ERROR QUANTIFIERS_OUT_OF_ORDER = 104,
  G_REGEX_ERROR QUANTIFIER_TOO_BIG = 105,
  G_REGEX_ERROR_UNTERMINATED_CHARACTER_CLASS = 106,
  G_REGEX_ERROR_INVALID_ESCAPE_IN_CHARACTER_CLASS = 107,
  G_REGEX_ERROR_RANGE_OUT_OF_ORDER = 108,
  G_REGEX_ERROR NOTHING_TO_REPEAT = 109,
  G_REGEX_ERROR UNRECOGNIZED_CHARACTER = 112,
  G_REGEX_ERROR POSIX_NAMED_CLASS_OUTSIDE_CLASS = 113,
  G_REGEX_ERROR_UNMATCHED_PARENTHESIS = 114,
  G_REGEX_ERROR INEXISTENT_SUBPATTERN_REFERENCE = 115,
  G_REGEX_ERROR UNTERMINATED_COMMENT = 118,
  G_REGEX_ERROR EXPRESSION_TOO_LARGE = 120,
  G_REGEX_ERROR MEMORY_ERROR = 121,
  G_REGEX_ERROR_VARIABLE_LENGTH_LOOKAHEAD = 125,
  G_REGEX_ERROR MALFORMED_CONDITION = 126,
  G_REGEX_ERROR TOO_MANY_CONDITIONAL_BRANCHES = 127,
  G_REGEX_ERROR ASSERTION_EXPECTED = 128,
  G_REGEX_ERROR UNKNOWN_POSIX_CLASS_NAME = 130,
  G_REGEX_ERROR POSIX_COLLATING_ELEMENTS_NOT_SUPPORTED = 131,
  G_REGEX_ERROR HEX_CODE_TOO_LARGE = 134,
  G_REGEX_ERROR INVALID_CONDITION = 135,
  G_REGEX_ERROR SINGLE_BYTE_MATCH_IN_LOOKBEHIND = 136,
  G_REGEX_ERROR INFINITE_LOOP = 140,
  G_REGEX_ERROR MISSING_SUBPATTERN_NAME_TERMINATOR = 142,
  G_REGEX_ERROR DUPLICATE_SUBPATTERN_NAME = 143,
  G_REGEX_ERROR MALFORMED_PROPERTY = 146,
  G_REGEX_ERROR UNKNOWN_PROPERTY = 147,
  G_REGEX_ERROR_SUBPATTERN_NAME_TOO_LONG = 148,
  G_REGEX_ERROR TOO_MANY_SUBPATTERNS = 149,
  G_REGEX_ERROR INVALID_OCTAL_VALUE = 151,
  G_REGEX_ERROR TOO_MANY_BRANCHES_IN_DEFINE = 154,
  G_REGEX_ERROR DEFINE_REPEITION = 155,
  G_REGEX_ERROR INCONSISTENT_NEWLINE_OPTIONS = 156,
  G_REGEX_ERROR MISSING_BACK_REFERENCE = 157
} GRegexError;

Error codes returned by regular expressions functions.

G_REGEX_ERROR_COMPILE Compilation of the regular expression failed.
G_REGEX_ERROR_OPTIMIZE Optimization of the regular expression failed.
G_REGEX_ERROR_REPLACE Replacement failed due to an ill-formed replacement string.
G_REGEX_ERROR_MATCH The match process failed.
G_REGEX_ERROR_INTERNAL Internal error of the regular expression engine. Since 2.16
G_REGEX_ERROR_STRAY_BACKSLASH "\\" at end of pattern. Since 2.16
G_REGEX_ERROR_MISSING_CONTROL_CHAR "\\e" at end of pattern. Since 2.16
G_REGEX_ERROR_UNRECOGNIZED_ESCAPE Unrecognized character follows "\\". Since 2.16
G_REGEX_ERROR_QUANTIFIERS_OUT_OF_ORDER Numbers out of order in "{|}" quantifier. Since 2.16
G_REGEX_ERROR_QUANTIFIER_TOO_BIG Number too big in "{|}" quantifier. Since 2.16
G_REGEX_ERROR_UNTERMINATED_CHARACTER_CLASS Missing terminating ’}’ for character class. Since 2.16
G_REGEX_ERROR_INVALID_ESCAPE_IN_CHARACTER_CLASS Invalid escape sequence in character class. Since 2.16
G_REGEX_ERROR_RANGE_OUT_OF_ORDER Range out of order in character class. Since 2.16
G_REGEX_ERROR NOTHING_TO_REPEAT Nothing to repeat. Since 2.16
G_REGEX_ERROR UNRECOGNIZED_CHARACTER Unrecognized character after ”(?”, ”?<” or ”(?P”. Since 2.16
G_REGEX_ERROR_POSIX NAMED_CLASS_OUTSIDE_CLASS POSIX named classes are supported only within a class. Since 2.16
G_REGEX_ERROR_UNMATCHED_PARENTHESIS Missing terminating ”)“ or ”)” without opening ”(“. Since 2.16
G_REGEX_ERROR_INEXISTENT_SUBPATTERN_REFERENCE Reference to non-existent subpattern. Since 2.16
G_REGEX_ERROR_UNTERMINATED_COMMENT Missing terminating ”)“ after comment. Since 2.16
G_REGEX_ERROR_EXPRESSION_TOO_LARGE Regular expression too large. Since 2.16
G_REGEX_ERROR_MEMORY_ERROR Failed to get memory. Since 2.16
G_REGEX_ERROR_VARIABLE_LENGTH_LOOKBEHIND Lookbehind assertion is not fixed length. Since 2.16
G_REGEX_ERROR_MALFORMED_CONDITION Malformed number or name after ”(?". Since 2.16
G_REGEX_ERROR_TOO_MANY_CONDITIONAL_BRANCHES Conditional group contains more than two branches. Since 2.16
G_REGEX_ERROR_ASSERTION_EXPECTED Assertion expected after ”(?“. Since 2.16
G_REGEX_ERROR_UNKNOWN_POSIX_CLASS_NAME Unknown POSIX class name. Since 2.16
G_REGEX_ERROR_POSIX_COLLATING_ELEMENTS_NOT_SUPPORTED POSIX collating elements are not supported. Since 2.16
G_REGEX_ERROR_HEX_CODE_TOO_LARGE Character value in ”\x{...}“ sequence is too large. Since 2.16
G_REGEX_ERROR_INVALID_CONDITION Invalid condition ”(?0). Since 2.16
G_REGEX_ERROR_SINGLE_BYTE_MATCH_IN_LOOKBEHIND \C not allowed in lookbehind assertion. Since 2.16
G_REGEX_ERROR_INFINITE_LOOP Recursive call could loop indefinitely. Since 2.16
G_REGEX_ERROR_MISSING_SUBPATTERN_NAME_TERMINATOR Missing terminator in subpattern name. Since 2.16
G_REGEX_ERROR_DUPLICATE_SUBPATTERN_NAME Two named subpatterns have the same name. Since 2.16
G_REGEX_ERROR_MALFORMED_PROPERTY Malformed ”\P“ or ”\p“ sequence. Since 2.16
G_REGEX_ERROR_UNKNOWN_PROPERTY Unknown property name after ”\P“ or ”\p“. Since 2.16
G_REGEX_ERROR_SUBPATTERN_NAME_TOO_LONG Subpattern name is too long (maximum 32 characters). Since 2.16
CHAPTER 4. GLIB UTILITIES

4.20. PERL-COMPATIBLE REGULAR EXPRESSIONS

G_REGEX_ERROR_TOO_MANY_SUBPATTERNS Too many named subpatterns (maximum 10,000). Since 2.16

G_REGEX_ERROR_INVALID_OCTAL_VALUE Octal value is greater than "\377". Since 2.16

G_REGEX_ERROR_TOO_MANY_BRANCHES_IN_DEFINE "DEFINE" group contains more than one branch. Since 2.16

G_REGEX_ERROR_DEFINE_REPETITION Repeating a "DEFINE" group is not allowed. Since 2.16

G_REGEX_ERROR_INCONSISTENT_NEWLINE_OPTIONS Inconsistent newline options. Since 2.16

G_REGEX_ERROR_MISSING_BACK_REFERENCE "\\g" is not followed by a braced name or an optionally braced non-zero number. Since 2.16

Since 2.14

G_REGEX_ERROR

#define G_REGEX_ERROR g_regex_error_quark ()

Error domain for regular expressions. Errors in this domain will be from the GRegexError enumeration. See GError for information on error domains.

Since 2.14

enum GRegexCompileFlags

typedef enum
{
    G_REGEX_CASELESS = 1 << 0,
    G_REGEX_MULTILINE = 1 << 1,
    G_REGEX_DOTALL = 1 << 2,
    G_REGEX_EXTENDED = 1 << 3,
    G_REGEX_ANCHORED = 1 << 4,
    G_REGEX_DOLLAR_ENDONLY = 1 << 5,
    G_REGEX_UNGREEDY = 1 << 9,
    G_REGEX_RAW = 1 << 11,
    G_REGEX_NO_AUTO_CAPTURE = 1 << 12,
    G_REGEX_OPTIMIZE = 1 << 13,
    G_REGEX_DUPNAMES = 1 << 19,
    G_REGEX_NEWLINE_CR = 1 << 20,
    G_REGEX_NEWLINE_LF = 1 << 21,
    G_REGEX_NEWLINE_CRLF = G_REGEX_NEWLINE_CR | G_REGEX_NEWLINE_LF
} GRegexCompileFlags;

Flags specifying compile-time options.

G_REGEX_CASELESS Letters in the pattern match both upper and lower case letters. It be changed within a pattern by a "(?i)" option setting.

G_REGEX_MULTILINE By default, GRegex treats the strings as consisting of a single line of characters (even if it actually contains newlines). The "start of line" metacharacter ("^") matches only at the start of the string, while the "end of line" metacharacter ("$" ) matches only at the end of the string, or before a terminating newline (unless G_REGEX_DOLLAR_ENDONLY is set). When G_REGEX_MULTILINE is set, the "start of line" and "end of line" constructs match immediately following or immediately before any newline in the string, respectively, as well as at the very start and end. This can be changed within a pattern by a "(?m)" option setting.

G_REGEX_DOTALL A dot metacharacter (\.) in the pattern matches all characters, including newlines. Without it, newlines are excluded. This option can be changed within a pattern by a "(?s)" option setting.
**G_REGEX_EXTENDED**  Whitespace data characters in the pattern are totally ignored except when escaped or inside a character class. Whitespace does not include the VT character (code 11). In addition, characters between an unescaped ‘#’ outside a character class and the next newline character, inclusive, are also ignored. This can be changed within a pattern by a ‘(?x)’ option setting.

**G_REGEX_ANCHORED**  The pattern is forced to be “anchored”, that is, it is constrained to match only at the first matching point in the string that is being searched. This effect can also be achieved by appropriate constructs in the pattern itself such as the ‘ˆ’ metacharacter.

**G_REGEX_DOLLAR_ENDONLY**  A dollar metacharacter (“$”) in the pattern matches only at the end of the string. Without this option, a dollar also matches immediately before the final character if it is a newline (but not before any other newlines). This option is ignored if G_REGEX_MULTILINE is set.

**G_REGEX_UNGREEDY**  Inverts the “greediness” of the quantifiers so that they are not greedy by default, but become greedy if followed by “?’”. It can also be set by a “(?U)” option setting within the pattern.

**G_REGEX_RAW**  Usually strings must be valid UTF-8 strings, using this flag they are considered as a raw sequence of bytes.

**G_REGEX_NO_AUTO_CAPTURE**  Disables the use of numbered capturing parentheses in the pattern. Any opening parenthesis that is not followed by “?’” behaves as if it were followed by “?’” but named parentheses can still be used for capturing (and they acquire numbers in the usual way).

**G_REGEX_OPTIMIZE**  Optimize the regular expression. If the pattern will be used many times, then it may be worth the effort to optimize it to improve the speed of matches.

**G_REGEX_DUPNAMES**  Names used to identify capturing subpatterns need not be unique. This can be helpful for certain types of pattern when it is known that only one instance of the named subpattern can ever be matched.

**G_REGEX_NEWLINE_CR**  Usually any newline character is recognized, if this option is set, the only recognized newline character is ‘\r’.

**G_REGEX_NEWLINE_LF**  Usually any newline character is recognized, if this option is set, the only recognized newline character is ‘\n’.

**G_REGEX_NEWLINE_CRLF**  Usually any newline character is recognized, if this option is set, the only recognized newline character sequence is ‘\r\n’.

Since 2.14

**enum GRegexMatchFlags**

```c
typedef enum
{
    G_REGEX_MATCH_ANCHORED  = 1 << 4,
    G_REGEX_MATCH_NOTBOL    = 1 << 7,
    G_REGEX_MATCH_NOTEOL    = 1 << 8,
    G_REGEX_MATCH_NOTEMPTY  = 1 << 10,
    G_REGEX_MATCH_PARTIAL   = 1 << 15,
    G_REGEX_MATCH_NEWLINE_CR = 1 << 20,
    G_REGEX_MATCH_NEWLINE_LF = 1 << 21,
    G_REGEX_MATCH_NEWLINE_CRLF = G_REGEX_MATCH_NEWLINE_CR | G_REGEX_MATCH_NEWLINE_LF,
    G_REGEX_MATCH_NEWLINE_ANY = 1 << 22
} GRegexMatchFlags;
```

Flags specifying match-time options.

**G_REGEX_MATCH_ANCHORED**  The pattern is forced to be “anchored”, that is, it is constrained to match only at the first matching point in the string that is being searched. This effect can also be achieved by appropriate constructs in the pattern itself such as the ‘ˆ’ metacharacter.
**G_REGEX_MATCH_NOTBOL** Specifies that first character of the string is not the beginning of a line, so the circumflex metacharacter should not match before it. Setting this without G_REGEX_MULTILINE (at compile time) causes circumflex never to match. This option affects only the behaviour of the circumflex metacharacter, it does not affect “\A”.

**G_REGEX_MATCH_NOTEOL** Specifies that the end of the subject string is not the end of a line, so the dollar metacharacter should not match it nor (except in multiline mode) a newline immediately before it. Setting this without G_REGEX_MULTILINE (at compile time) causes dollar never to match. This option affects only the behaviour of the dollar metacharacter; it does not affect “\Z” or “\z”.

**G_REGEX_MATCH_NOTEMPTY** An empty string is not considered to be a valid match if this option is set. If there are alternatives in the pattern, they are tried. If all the alternatives match the empty string, the entire match fails. For example, if the pattern “a?b?” is applied to a string not beginning with “a” or “b”, it matches the empty string at the start of the string. With this flag set, this match is not valid, so GRegex searches further into the string for occurrences of “a” or “b”.

**G_REGEX_MATCH_PARTIAL** Turns on the partial matching feature, for more documentation on partial matching see g_match_info_is_partial_match().

**G_REGEX_MATCH_NEWLINE_CR** Overrides the newline definition set when creating a new GRegex, setting the ‘\r’ character as line terminator.

**G_REGEX_MATCH_NEWLINE_LF** Overrides the newline definition set when creating a new GRegex, setting the ‘\n’ character as line terminator.

**G_REGEX_MATCH_NEWLINE_CRLF** Overrides the newline definition set when creating a new GRegex, setting the ‘\r\n’ characters as line terminator.

**G_REGEX_MATCH_NEWLINE_ANY** Overrides the newline definition set when creating a new GRegex, any newline character or character sequence is recognized.

Since 2.14

GRegex

typedef struct _GRegex GRegex;

A GRegex is the "compiled" form of a regular expression pattern. This structure is opaque and its fields cannot be accessed directly.

Since 2.14

GRegexEvalCallback()

```c
gboolean (*GRegexEvalCallback) (const GMatchInfo * match_info,
                                GString *result,
                                gpointer user_data);
```

Specifies the type of the function passed to g_regex_replace_eval(). It is called for each occurrence of the pattern in the string passed to g_regex_replace_eval(), and it should append the replacement to `result`.

**match_info**: the GMatchInfo generated by the match. Use g_match_info_get_regex() and g_match_info_get_string() if you need the GRegex or the matched string.

**result**: a GString containing the new string

**user_data**: user data passed to g_regex_replace_eval()

**Returns**: %FALSE to continue the replacement process, TRUE to stop it

Since 2.14
CHAPTER 4. GLIB UTILITIES

4.20. PERL-COMPATIBLE REGULAR EXPRESSIONS

4.20.1. Perl-Compatible Regular Expression Functions

**g_regex_new()**

```c
GRegex * g_regex_new (const gchar *pattern,
                       GRegexCompileFlags ← compile_options,
                       GRegexMatchFlags ← match_options,
                       GError **error);
```

Compiles the regular expression to an internal form, and does the initial setup of the `GRegex` structure.

- **pattern**: the regular expression
- **compile_options**: compile options for the regular expression, or 0
- **match_options**: match options for the regular expression, or 0
- **error**: return location for a `GError`

**Returns**: a `GRegex` structure. Call `g_regex_unref()` when you are done with it.

    Since 2.14

**g_regex_ref()**

```c
GRegex * g_regex_ref (GRegex *regex);
```

Increases reference count of `regex` by 1.

- **regex**: a `GRegex`

**Returns**: `regex`

    Since 2.14

**g_regex_unref()**

```c
void g_regex_unref (GRegex *regex);
```

Decreases reference count of `regex` by 1. When reference count drops to zero, it frees all the memory associated with the regex structure.

- **regex**: a `GRegex`

    Since 2.14

**g_regex_get_pattern()**

```c
const gchar * g_regex_get_pattern (const GRegex *regex);
```

Gets the pattern string associated with `regex`, i.e. a copy of the string passed to `g_regex_new()`.

- **regex**: a `GRegex` structure

**Returns**: the pattern of `regex`

    Since 2.14
CHAPTER 4. GLIB UTILITIES

4.20. PERL-COMPATIBLE REGULAR...

\textbf{g_regex_get_max_backref ()}

\begin{verbatim}
gint g_regex_get_max_backref (const GRegex *regex);
\end{verbatim}

Returns the number of the highest back reference in the pattern, or 0 if the pattern does not contain back references.

\textbf{regex} : a GRegex

\textbf{Returns} : the number of the highest back reference

Since 2.14

\textbf{g_regex_get_capture_count ()}

\begin{verbatim}
gint g_regex_get_capture_count (const GRegex *regex);
\end{verbatim}

Returns the number of capturing subpatterns in the pattern.

\textbf{regex} : a GRegex

\textbf{Returns} : the number of capturing subpatterns

Since 2.14

\textbf{g_regex_get_string_number ()}

\begin{verbatim}
gint g_regex_get_string_number (const GRegex *regex, const gchar *name);
\end{verbatim}

Retrieves the number of the subexpression named \textit{name}.

\textbf{regex} : GRegex structure

\textbf{name} : name of the subexpression

\textbf{Returns} : The number of the subexpression or -1 if \textit{name} does not exists

Since 2.14

\textbf{g_regex_escape_string ()}

\begin{verbatim}
gchar * g_regex_escape_string (const gchar *string, gint length);
\end{verbatim}

Escapes the special characters used for regular expressions in \textit{string}, for instance "a.b*c" becomes "a\b\c". This function is useful to dynamically generate regular expressions.

\textit{string} can contain nul characters that are replaced with "\0", in this case remember to specify the correct length of \textit{string} in \textit{length}.

\textbf{string} : the string to escape

\textbf{length} : the length of \textit{string}, or -1 if \textit{string} is nul-terminated

\textbf{Returns} : a newly-allocated escaped string

Since 2.14
g_regex_match_simple()

gboolean g_regex_match_simple (const gchar *pattern, const gchar *string, GRegexCompileFlags compile_options, GRegexMatchFlags match_options);

Scans for a match in string for pattern. This function is equivalent to g_regex_match() but it does not require to compile the pattern with g_regex_new(), avoiding some lines of code when you need just to do a match without extracting sub-strings, capture counts, and so on.

If this function is to be called on the same pattern more than once, it’s more efficient to compile the pattern once with g_regex_new() and then use g_regex_match().

pattern: the regular expression

string: the string to scan for matches

compile_options: compile options for the regular expression, or 0

match_options: match options, or 0

Returns: TRUE if the string matched, FALSE otherwise

Since 2.14

g_regex_match()

gboolean g_regex_match (const GRegex *regex, const gchar *string, GRegexMatchFlags match_options, GMatchInfo **match_info);

Scans for a match in string for the pattern in regex. The match_options are combined with the match options specified when the regex structure was created, letting you have more flexibility in reusing GRegex structures.

A GMatchInfo structure, used to get information on the match, is stored in match_info if not NULL. Note that if match_info is not NULL then it is created even if the function returns FALSE, i.e. you must free it regardless if regular expression actually matched.

To retrieve all the non-overlapping matches of the pattern in string you can use g_match_info_next().

static void print_uppercase_words (const gchar *string)
{
    /* Print all uppercase-only words. */
    GRegex *regex;
    GMatchInfo *match_info;
    ~
    regex = g_regex_new ("[A-Z]+", 0, 0, NULL);
    g_regex_match (regex, string, 0, &match_info);
    while (g_match_info_matches (match_info))
    {
        gchar *word = g_match_info_fetch (match_info, 0);
        g_print ("Found: %s\n", word);
        g_free (word);
        g_match_info_next (match_info, NULL);
    }
    g_match_info_free (match_info);
    g_regex_unref (regex);
}
string is not copied and is used in GMatchInfo internally. If you use any GMatchInfo method (except g_match_info_free()) after freeing or modifying string then the behaviour is undefined.

**regex**: a GRegex structure from g_regex_new()

**string**: the string to scan for matches

**match_options**: match options

**match_info**: pointer to location where to store the GMatchInfo, or NULL if you do not need it

**Returns**: TRUE is the string matched, FALSE otherwise

Since 2.14

---

```c
gboolean g_regex_match_full (const GRegex *regex,
const gchar *string,
gssize string_len,
gint start_position,
GRegexMatchFlags match_options,
GMatchInfo **match_info,
GError **error);
```

Scans for a match in string for the pattern in regex. The match_options are combined with the match options specified when the regex structure was created, letting you have more flexibility in reusing GRegex structures.

Setting start_position differs from just passing over a shortened string and setting G_REGEX_MATCH_NOTBOL in the case of a pattern that begins with any kind of lookbehind assertion, such as "\b".

A GMatchInfo structure, used to get information on the match, is stored in match_info if not NULL. Note that if match_info is not NULL then it is created even if the function returns FALSE, i.e. you must free it regardless if regular expression actually matched.

string is not copied and is used in GMatchInfo internally. If you use any GMatchInfo method (except g_match_info_free()) after freeing or modifying string then the behaviour is undefined.

To retrieve all the non-overlapping matches of the pattern in string you can use g_match_info_next().

```c
static void print_uppercase_words (const gchar *string)
{
  /* Print all uppercase-only words. */
  GRegex *regex;
  GMatchInfo *match_info;
  GError *error = NULL;
  ~
  regex = g_regex_new ("[A-Za-z]+", 0, 0, NULL);
  g_regex_match_full (regex, string, -1, 0, 0, &match_info, &error);
  while (g_match_info_matches (match_info))
  {
    gchar *word = g_match_info_fetch (match_info, 0);
    g_print ("Found: %s\n", word);
    g_free (word);
    g_match_info_next (match_info, &error);
  }
  g_match_info_free (match_info);
  g_regex_unref (regex);
  if (error != NULL)
  {
    g_printerr ("Error while matching: %s\n", error->message);
    g_error_free (error);
  }
}
regex: a GRegex structure from g_regex_new()

string: the string to scan for matches

string_len: the length of string, or -1 if string is nul-terminated

start_position: starting index of the string to match

match_options: match options

match_info: pointer to location where to store the GMatchInfo, or NULL if you do not need it

error: location to store the error occurring, or NULL to ignore errors

Returns: TRUE is the string matched, FALSE otherwise

Since 2.14

g_regex_match_all ()

gboolean g_regex_match_all (const GRegex *regex,
                             const gchar *string,
                             GRegexMatchFlags match_options,
                             GMatchInfo **match_info);

Using the standard algorithm for regular expression matching only the longest match in the string is retrieved. This function uses a different algorithm so it can retrieve all the possible matches. For more documentation see g_regex_match_all_full().

A GMatchInfo structure, used to get information on the match, is stored in match_info if not NULL. Note that if match_info is not NULL then it is created even if the function returns FALSE, i.e. you must free it regardless if regular expression actually matched.

string is not copied and is used in GMatchInfo internally. If you use any GMatchInfo method (except g_match_info_free()) after freeing or modifying string then the behaviour is undefined.

regex: a GRegex structure from g_regex_new()

string: the string to scan for matches

match_options: match options

match_info: pointer to location where to store the GMatchInfo, or NULL if you do not need it

Returns: TRUE is the string matched, FALSE otherwise

Since 2.14

g_regex_match_all_full ()

gboolean g_regex_match_all_full (const GRegex *regex,
                                  const gchar *string,
                                  guint start_position,
                                  GRegexMatchFlags match_options,
                                  GMatchInfo **match_info,
                                  GError **error);

Using the standard algorithm for regular expression matching only the longest match in the string is retrieved, it is not possible to obtain all the available matches. For instance matching "<a> <b> <c>" against the pattern "<.*>" you get "<a> <b> <c>".

This function uses a different algorithm (called DFA, i.e. deterministic finite automaton), so it can retrieve all the possible matches, all starting at the same point in the string. For instance matching "<a>
The number of matched strings is retrieved using g_match_info_get_match_count(). To obtain the matched strings and their position you can use, respectively, g_match_info_fetch() and g_match_info_fetch_pos(). Note that the strings are returned in reverse order of length; that is, the longest matching string is given first.

Note that the DFA algorithm is slower than the standard one and it is not able to capture substrings, so backreferences do not work.

Setting start_position differs from just passing over a shortened string and setting G_REGEX_MATCH_NOTBOL in the case of a pattern that begins with any kind of lookbehind assertion, such as "\b".

A GMatchInfo structure, used to get information on the match, is stored in match_info if not NULL. Note that if match_info is not NULL then it is created even if the function returns FALSE, i.e. you must free it regardless if regular expression actually matched.

string is not copied and is used in GMatchInfo internally. If you use any GMatchInfo method (except g_match_info_free()) after freeing or modifying string then the behaviour is undefined.

regex: a GRegex structure from g_regex_new()

string: the string to scan for matches

string_len: the length of string, or -1 if string is nul-terminated

start_position: starting index of the string to match

match_info: match options

match_options: pointer to location where to store the GMatchInfo, or NULL if you do not need it

error: location to store the error occurring, or NULL to ignore errors

Returns: TRUE is the string matched, FALSE otherwise

Since 2.14

g_regex_split_simple ()

Breaks the string on the pattern, and returns an array of the tokens. If the pattern contains capturing parentheses, then the text for each of the substrings will also be returned. If the pattern does not match anywhere in the string, then the whole string is returned as the first token.

This function is equivalent to g_regex_split() but it does not require to compile the pattern with g_regex_new(), avoiding some lines of code when you need just to do a split without extracting substrings, capture counts, and so on.

If this function is to be called on the same pattern more than once, it’s more efficient to compile the pattern once with g_regex_new() and then use g_regex_split().

A special case, the result of splitting the empty string "" is an empty vector, not a vector containing a single string. The reason for this special case is that being able to represent a empty vector is typically more useful than consistent handling of empty elements. If you do need to represent empty elements, you’ll need to check for the empty string before calling this function.

A pattern that can match empty strings splits string into separate characters wherever it matches the empty string between characters. For example splitting "ab c" using as a separator "\s*", you will get "a", "b" and "c".

pattern: the regular expression

string: the regular to scan for matches

compile_options: compile options for the regular expression, or 0
match_options: match options, or 0

Returns: a NULL-terminated array of strings. Free it using g_strfreev()

Since 2.14

g_regex_split ()

Breaks the string on the pattern, and returns an array of the tokens. If the pattern contains capturing parentheses, then the text for each of the substrings will also be returned. If the pattern does not match anywhere in the string, then the whole string is returned as the first token.

As a special case, the result of splitting the empty string "" is an empty vector, not a vector containing a single string. The reason for this special case is that being able to represent a empty vector is typically more useful than consistent handling of empty elements. If you do need to represent empty elements, you'll need to check for the empty string before calling this function.

A pattern that can match empty strings splits string into separate characters wherever it matches the empty string between characters. For example splitting "ab c" using as a separator "\s*", you will get "a", "b" and "c".

regex: a GRegex structure

string: the string to split with the pattern

match_options: match time option flags

Returns: a NULL-terminated gchar ** array. Free it using g_strfreev()

Since 2.14

g_regex_split_full ()

Breaks the string on the pattern, and returns an array of the tokens. If the pattern contains capturing parentheses, then the text for each of the substrings will also be returned. If the pattern does not match anywhere in the string, then the whole string is returned as the first token.

As a special case, the result of splitting the empty string "" is an empty vector, not a vector containing a single string. The reason for this special case is that being able to represent a empty vector is typically more useful than consistent handling of empty elements. If you do need to represent empty elements, you'll need to check for the empty string before calling this function.

A pattern that can match empty strings splits string into separate characters wherever it matches the empty string between characters. For example splitting "ab c" using as a separator "\s*", you will get "a", "b" and "c".

Setting start_position differs from just passing over a shortened string and setting G_REGEX_MATCH_NOTBOL in the case of a pattern that begins with any kind of lookbehind assertion, such as "\b".

regex: a GRegex structure

string: the string to split with the pattern

string_len: the length of string, or -1 if string is nul-terminated

378
**start_position**: starting index of the string to match

**match_options**: match time option flags

**max_tokens**: the maximum number of tokens to split `string` into. If this is less than 1, the string is split completely

**error**: return location for a `GError`

**Returns**: a NULL-terminated `gchar **` array. Free it using `g_strfreev()`

Since 2.14

```c
 gchar * g_regex_replace (const GRegex *regex, const gchar *string, gssize string_len, gint start_position, const gchar *replacement, GRegexMatchFlags match_options, GError **error);
```

Replaces all occurrences of the pattern in `regex` with the replacement text. Backreferences of the form \`\number` or \`\g<number>` in the replacement text are interpolated by the number-th captured subexpression of the match, \`\g<name>` refers to the captured subexpression with the given name. \`\0` refers to the complete match, but \`\0` followed by a number is the octal representation of a character. To include a literal \`\``` in the replacement, write \`\\```. There are also escapes that changes the case of the following text:

\`\l` Convert to lower case the next character

\`\u` Convert to upper case the next character

\`\L` Convert to lower case till \`\E`

\`\U` Convert to upper case till \`\E`

\`\E` End case modification

If you do not need to use backreferences use `g_regex_replace_literal()`.

The `replacement` string must be UTF-8 encoded even if `G_REXEG_RAW` was passed to `g_regex_new()`. If you want to use not UTF-8 encoded stings you can use `g_regex_replace_literal()`.

Setting `start_position` differs from just passing over a shortened string and setting `G_REGEX_MATCH_NOTBOL` in the case of a pattern that begins with any kind of lookbehind assertion, such as \`\b`.

**regex**: a `GRegex` structure

**string**: the string to perform matches against

**string_len**: the length of `string`, or -1 if `string` is null-terminated

**start_position**: starting index of the string to match

**replacement**: text to replace each match with

**match_options**: options for the match

**error**: location to store the error occurring, or `NULL` to ignore errors

**Returns**: a newly allocated string containing the replacements

Since 2.14
CHAPTER 4. GLIB UTILITIES

4.20. PERL-COMPATIBLE REGULAR...

\[\text{g_regex_replace_literal}\]

\[
\text{gchar * g_regex_replace_literal (const GRegex *regex, const gchar *string, gssize string_len, gint start_position, const gchar *replacement, GRegexMatchFlags match_options, GError **error);}\]

Replaces all occurrences of the pattern in \(\text{regex}\) with the replacement text. \(\text{replacement}\) is replaced literally, to include backreferences use \text{g_regex_replace()}.

Setting \(\text{start_position}\) differs from just passing over a shortened string and setting \text{G_REGEX_MATCH_NOTBOL} in the case of a pattern that begins with any kind of lookbehind assertion, such as "\b".

\text{regex}: a \text{GRegex} structure

\text{string}: the string to perform matches against

\text{string_len}: the length of \text{string}, or -1 if \text{string} is nul-terminated

\text{start_position}: starting index of the string to match

\text{replacement}: text to replace each match with

\text{match_options}: options for the match

\text{error}: location to store the error occurring, or NULL to ignore errors

\text{Returns}: a newly allocated string containing the replacements

Since 2.14

\[\text{g_regex_replace_eval}\]

\[
\text{gchar * g_regex_replace_eval (const GRegex *regex, const gchar *string, gssize string_len, gint start_position, GRegexMatchFlags match_options, GRegexEvalCallback eval, gpointer user_data, GError **error);}\]

Replaces occurrences of the pattern in \(\text{regex}\) with the output of \text{eval} for that occurrence.

Setting \(\text{start_position}\) differs from just passing over a shortened string and setting \text{G_REGEX_MATCH_NOTBOL} in the case of a pattern that begins with any kind of lookbehind assertion, such as "\b".

The following example uses \text{g_regex_replace_eval()} to replace multiple strings at once:

\[
\begin{align*}
\text{static gboolean eval_cb (const GMatchInfo *info, GString *res, gpointer data)} \{ \\
\text{ gchar *match; gchar *r;}
\end{align*}
\]

\[
\begin{align*}
\text{match = g_match_info_fetch (info, 0);} \\
\text{r = g_hash_table_lookup ((GHashTable *)data, match);} \\
\text{g_string_append (res, r);} \\
\text{g_free (match);} \\
\end{align*}
\]

380
return FALSE;
}
/* ... */
GRegex *reg;
GHashTable *h;
gchar *res;

h = g_hash_table_new (g_str_hash, g_str_equal);
g_hash_table_insert (h, "1", "ONE");
g_hash_table_insert (h, "2", "TWO");
g_hash_table_insert (h, "3", "THREE");
g_hash_table_insert (h, "4", "FOUR");

reg = g_regex_new ("1|2|3|4", 0, 0, NULL);
res = g_regex_replace_eval (reg, text, -1, 0, 0, eval_cb, h, NULL);
g_hash_table_destroy (h);
/* ... */

regex: a GRegex structure from g_regex_new()
string: string to perform matches against
string_len: the length of string, or -1 if string is nul-terminated
start_position: starting index of the string to match
match_options: options for the match
eval: a function to call for each match
user_data: user data to pass to the function
error: location to store the error occuring, or NULL to ignore errors

Returns: a newly allocated string containing the replacements

Since 2.14

g_regex_check_replacement ()

gboolean g_regex_check_replacement (const gchar *replacement
, gboolean *has_references
, GError **error);

Checks whether replacement is a valid replacement string (see g_regex_replace()), i.e. that all escape sequences in it are valid.

If has_references is not NULL then replacement is checked for pattern references. For instance, replacement text ‘foo\n’ does not contain references and may be evaluated without information about actual match, but ‘\0\1’ (whole match followed by first subpattern) requires valid GMatchInfo object.

replacement: the replacement string
has_references: location to store information about references in replacement or NULL
error: location to store error

Returns: whether replacement is a valid replacement string

Since 2.14
GMatchInfo

typedef struct _GMatchInfo GMatchInfo;

GMatchInfo is used to retrieve information about the regular expression match which created it. This structure is opaque and its fields cannot be accessed directly.
Since 2.14

g_match_info_get_regex()

GRegex * g_match_info_get_regex (const GMatchInfo * match_info);

Returns GRegex object used in match_info. It belongs to Glib and must not be freed. Use g_regex_ref() if you need to keep it after you free match_info object.

match_info: a GMatchInfo

Returns: GRegex object used in match_info
Since 2.14

g_match_info_get_string()

const gchar * g_match_info_get_string (const GMatchInfo * match_info);

Returns the string searched with match_info. This is the string passed to g_regex_match() or g_regex_replace() so you may not free it before calling this function.

match_info: a GMatchInfo

Returns: the string searched with match_info
Since 2.14

g_match_info_free()

void g_match_info_free (GMatchInfo *match_info);

Frees all the memory associated with the GMatchInfo structure.

match_info: a GMatchInfo

Since 2.14

g_match_info_matches()

gboolean g_match_info_matches (const GMatchInfo * match_info);

Returns whether the previous match operation succeeded.

match_info: a GMatchInfo structure

Returns: TRUE if the previous match operation succeeded, FALSE otherwise
Since 2.14
g_match_info_next ()

gboolean g_match_info_next (GMatchInfo *match_info, GError **error);

Scans for the next match using the same parameters of the previous call to g_regex_match_full() or g_regex_match() that returned match_info.

The match is done on the string passed to the match function, so you cannot free it before calling this function.

match_info: a GMatchInfo structure

error: location to store the error occurring, or NULL to ignore errors

Returns: TRUE is the string matched, FALSE otherwise

Since 2.14

g_match_info_get_match_count ()

gint g_match_info_get_match_count (const GMatchInfo *match_info);

Retrieves the number of matched substrings (including substring 0, that is the whole matched text), so 1 is returned if the pattern has no substrings in it and 0 is returned if the match failed.

If the last match was obtained using the DFA algorithm, that is using g_regex_match_all() or g_regex_match_all_full(), the retrieved count is not that of the number of capturing parentheses but that of the number of matched substrings.

match_info: a GMatchInfo structure

Returns: Number of matched substrings, or -1 if an error occurred

Since 2.14

g_match_info_is_partial_match ()

gboolean g_match_info_is_partial_match (const GMatchInfo *match_info);

Usually if the string passed to g_regex_match*() matches as far as it goes, but is too short to match the entire pattern, FALSE is returned. There are circumstances where it might be helpful to distinguish this case from other cases in which there is no match.

Consider, for example, an application where a human is required to type in data for a field with specific formatting requirements. An example might be a date in the form ddmmyy, defined by the pattern "\dинд\d(?jan|feb|mar|apr|may|jun|jul|aug|sep|oct|nov|dec)\d\d$". If the application sees the user’s keystrokes one by one, and can check that what has been typed so far is potentially valid, it is able to raise an error as soon as a mistake is made.

GRegex supports the concept of partial matching by means of the G_REGEX_MATCH_PARTIAL flag. When this is set the return code for g_regex_match() or g_regex_match_full() is, as usual, TRUE for a complete match, FALSE otherwise. But, when these functions return FALSE, you can check if the match was partial calling g_match_info_is_partial_match().

When using partial matching you cannot use g_match_info_fetch*().

Because of the way certain internal optimizations are implemented the partial matching algorithm cannot be used with all patterns. So repeated single characters such as "a{2,4}" and repeated single meta-sequences such as "\d+" are not permitted if the maximum number of occurrences is greater than one. Optional items such as "\d?" (where the maximum is one) are permitted. Quantifiers with any values are permitted after parentheses, so the invalid examples above can be coded thus "(a){2,4}" and "(\d+)". If G_REGEX_MATCH_PARTIAL is set for a pattern that does not conform to the restrictions, matching functions return an error.

match_info: a GMatchInfo structure

Returns: TRUE if the match was partial, FALSE otherwise

Since 2.14
**g_match_info_expand_references()**

```c
 gchar * g_match_info_expand_references (const GMatchInfo * match_info,
                                          const gchar * string_to_expand,
                                          GError **error);
```

Returns a new string containing the text in `string_to_expand` with references and escape sequences expanded. References refer to the last match done with `string` against `regex` and have the same syntax used by `g_regex_replace()`.

The `string_to_expand` must be UTF-8 encoded even if `G_REGEX_RAW` was passed to `g_regex_new()`. The backreferences are extracted from the string passed to the match function, so you cannot call this function after freeing the string.

*match_info*: a `GMatchInfo` or NULL

*string_to_expand*: the string to expand

*error*: location to store the error occurring, or NULL to ignore errors

>Returns*: the expanded string, or NULL if an error occurred

Since 2.14

**g_match_info_fetch()**

```c
 gchar * g_match_info_fetch (const GMatchInfo * match_info,
                            gint match_num);
```

Retrieves the text matching the `match_num`'th capturing parentheses. 0 is the full text of the match, 1 is the first paren set, 2 the second, and so on.

If `match_num` is a valid sub pattern but it didn’t match anything (e.g. sub pattern 1, matching "b" against "(a)?b") then an empty string is returned.

If the match was obtained using the DFA algorithm, that is using `g_regex_match_all()` or `g_regex_match_all_full()`, the retrieved string is not that of a set of parentheses but that of a matched substring. Substrings are matched in reverse order of length, so 0 is the longest match.

The string is fetched from the string passed to the match function, so you cannot call this function after freeing the string.

*match_info*: `GMatchInfo` structure

*match_num*: number of the sub expression

>Returns*: The matched substring, or NULL if an error occurred. You have to free the string yourself

Since 2.14

**g_match_info_fetch_pos()**

```c
 gboolean g_match_info_fetch_pos (const GMatchInfo * match_info,
                                  gint match_num,
                                  gint *start_pos,
                                  gint *end_pos);
```

Since 2.14
Retrieves the position in bytes of the `match_num`th capturing parentheses. 0 is the full text of the match, 1 is the first paren set, 2 the second, and so on.

If `match_num` is a valid sub pattern but it didn’t match anything (e.g., sub pattern 1, matching "b" against "(a)?b") then `start_pos` and `end_pos` are set to -1 and TRUE is returned.

If the match was obtained using the DFA algorithm, that is using `g_regex_match_all()` or `g_regex_match_all_full()`, the retrieved position is not that of a set of parentheses but that of a matched substring. Substrings are matched in reverse order of length, so 0 is the longest match.

**match_info**: GMatchInfo structure  
**match_num**: number of the sub expression  
**start_pos**: pointer to location where to store the start position  
**end_pos**: pointer to location where to store the end position  

**Returns**: TRUE if the position was fetched, FALSE otherwise. If the position cannot be fetched, `start_pos` and `end_pos` are left unchanged

Since 2.14

```c
gboolean g_match_info_fetch_named_pos (const GMatchInfo *match_info, const gchar *name, gint *start_pos, gint *end_pos);
```

Retrieves the position in bytes of the capturing parentheses named `name`.

If `name` is a valid sub pattern name but it didn’t match anything (e.g., sub pattern "X", matching "b" against "(?P<X>a)?b") then an empty string is returned.

The string is fetched from the string passed to the match function, so you cannot call this function after freeing the string.

**match_info**: GMatchInfo structure  
**name**: name of the subexpression  

**Returns**: The matched substring, or NULL if an error occurred. You have to free the string yourself

Since 2.14

```c
char * g_match_info_fetch_named (const GMatchInfo *match_info, const gchar *name);
```

Retrieves the text matching the capturing parentheses named `name`.

If `name` is a valid sub pattern name but it didn’t match anything (e.g., sub pattern "X", matching "b" against "(?P<X>a)?b") then an empty string is returned.

**match_info**: GMatchInfo structure  
**name**: name of the subexpression  

**Returns**: The matched substring, or NULL if an error occurred. You have to free the string yourself

Since 2.14
4.21. SIMPLE XML SUBSET PARSER

**g_match_info_fetch_all()**

```c
const gchar ** g_match_info_fetch_all (const GMatchInfo *match_info);
```

Bundles up pointers to each of the matching substrings from a match and stores them in an array of gchar pointers. The first element in the returned array is the match number 0, i.e. the entire matched text.

If a sub pattern didn’t match anything (e.g. sub pattern 1, matching "b" against "(a)?b") then an empty string is inserted.

If the last match was obtained using the DFA algorithm, that is using `g_regex_match_all()` or `g_regex_match_all_full()`, the retrieved strings are not that matched by sets of parentheses but that of the matched substring. Substrings are matched in reverse order of length, so the first one is the longest match.

The strings are fetched from the string passed to the match function, so you cannot call this function after freeing the string.

**match_info**: a GMatchInfo structure

**Returns**: a NULL-terminated array of gchar * pointers. It must be freed using `g_strfreev()`. If the previous match failed NULL is returned

Since 2.14

### 4.21 Simple XML Subset Parser

**Name**

Simple XML Subset Parser – parses a subset of XML

**Synopsis**

```c
#include <glib.h>
enum GMarkupError;
#define G_MARKUP_ERROR
enum GMarkupParseFlags;
enum GMarkupParseContext;
GMarkupParser;
gchar* g_markup_escape_text (const gchar *text, gssize length);
gchar * g_markup_printf_escaped (const char *format,...);
gchar * g_markup_vprintf_escaped (const char *format, va_list args);
gboolean g_markup_parse_context_end_parse (GMarkupParseContext *context, GError **error);
void g_markup_parse_context_free (GMarkupParseContext *context);
void g_markup_parse_context_get_position (GMarkupParseContext *context, gint *line_number, gint *char_number);
const gchar * g_markup_parse_context_get_element (GMarkupParseContext *context);
const GSList * g_markup_parse_context_get_element_stack
  (GMarkupParseContext *context);
gpointer g_markup_parse_context_get_user_data
  (GMarkupParseContext *context);
GMarkupParseContext * g_markup_parse_context_new (const GMarkupParser *parser, GMarkupParser Flags flags, gpointer user_data, GDestroyNotify user_data_dnotify);
```
CHAPTER 4. GLIB UTILITIES 4.21. SIMPLE XML SUBSET PARSER

gboolean g_markup_parse_context_parse (GMarkupParseContext *context, const gchar *text, gssize text_len, GError **error);

void g_markup_parse_context_push (GMarkupParseContext *context, GMarkupParser *parser, gpointer user_data);

gpointer g_markup_parse_context_pop (GMarkupParseContext *context);

enum GMarkupCollectType;

gboolean g_markup_collect_attributes (const gchar *element_name, const gchar **attribute_names, const gchar **attribute_values, GError **error, GMarkupCollectType first_type, const gchar *first_attr, ...);

Description

The "GMarkup" parser is intended to parse a simple markup format that’s a subset of XML. This is a small, efficient, easy-to-use parser. It should not be used if you expect to interoperate with other applications generating full-scale XML. However, it’s very useful for application data files, config files, etc. where you know your application will be the only one writing the file. Full-scale XML parsers should be able to parse the subset used by GMarkup, so you can easily migrate to full-scale XML at a later time if the need arises.

GMarkup is not guaranteed to signal an error on all invalid XML; the parser may accept documents that an XML parser would not. However, XML documents which are not well-formed\(^5\) are not considered valid GMarkup documents.

Simplifications to XML include:

- Only UTF-8 encoding is allowed.
- No user-defined entities.
- Processing instructions, comments and the doctype declaration are "passed through" but are not interpreted in any way.
- No DTD or validation.

The markup format does support:

- Elements
- Attributes
- 5 standard entities: &amp; &lt; &gt; &quot; &apos;
- Character references
- Sections marked as CDATA

Details

enum GMarkupError

```
typedef enum {
    G_MARKUP_ERROR_BAD_UTF8,
    G_MARKUP_ERROR_EMPTY,
    G_MARKUP_ERROR_PARSE,
    ...
} GMarkupError;
```

\(^5\) Being wellformed is a weaker condition than being valid. See the XML specification for definitions of these terms.
/* The following are primarily intended for specific GMarkupParser implementations to set. */
G_MARKUP_ERROR_UNKNOWN_ELEMENT,
G_MARKUP_ERROR_UNKNOWN_ATTRIBUTE,
G_MARKUP_ERROR_INVALID_CONTENT,
G_MARKUP_ERROR_MISSING_ATTRIBUTE
} GMarkupError;

Error codes returned by markup parsing.

G_MARKUP_ERROR_BAD_UTF8 text being parsed was not valid UTF-8
G_MARKUP_ERROR_EMPTY document contained nothing, or only whitespace
G_MARKUP_ERROR_PARSE document was ill-formed
G_MARKUP_ERROR_UNKNOWN_ELEMENT error should be set by GMarkupParser functions; element wasn’t known
G_MARKUP_ERROR_UNKNOWN_ATTRIBUTE error should be set by GMarkupParser functions; attribute wasn’t known
G_MARKUP_ERROR_INVALID_CONTENT error should be set by GMarkupParser functions; content was invalid
G_MARKUP_ERROR_MISSING_ATTRIBUTE error should be set by GMarkupParser functions; a required attribute was missing

G_MARKUP_ERROR

#define G_MARKUP_ERROR g_markup_error_quark ()

Error domain for markup parsing. Errors in this domain will be from the GMarkupError enumeration. See GError for information on error domains.

enum GMarkupParseFlags
typedef enum
{
  G_MARKUP_DO_NOT_USE_THIS_UNSUPPORTED_FLAG = 1 << 0,
  G_MARKUP_TREAT_CDATA_AS_TEXT = 1 << 1,
  G_MARKUP_PREFIX_ERROR_POSITION = 1 << 2
} GMarkupParseFlags;

Flags that affect the behaviour of the parser.

G_MARKUP_DO_NOT_USE_THIS_UNSUPPORTED_FLAG flag you should not use.

G_MARKUP_TREAT_CDATA_AS_TEXT When this flag is set, CDATA marked sections are not passed literally to the passthrough function of the parser. Instead, the content of the section (without the ![CDATA[ and ]]) is passed to the text function. This flag was added in GLib 2.12.

G_MARKUP_PREFIX_ERROR_POSITION Normally errors caught by GMarkup itself have line/column information prefixed to them to let the caller know the location of the error. When this flag is set the location information is also prefixed to errors generated by the GMarkupParser implementation functions.

GMarkupParseContext
typedef struct _GMarkupParseContext GMarkupParseContext;

A parse context is used to parse a stream of bytes that you expect to contain marked-up text. See gMarkup parses context new(), GMarkupParser, and so on for more details.
GMarkupParser
typedef struct {
    /* Called for open tags <foo bar="baz"> */
    void (*start_element) (GMarkupParseContext *context,
                            const gchar *element_name,
                            const gchar **attribute_names,
                            const gchar **attribute_values,
                            gpointer user_data,
                            GError **error);

    /* Called for close tags </foo> */
    void (*end_element) (GMarkupParseContext *context,
                          const gchar *element_name,
                          gpointer user_data,
                          GError **error);

    /* Called for character data */
    /* text is not nul-terminated */
    void (*text) (GMarkupParseContext *context,
                  const gchar *text,
                  gsize text_len,
                  gpointer user_data,
                  GError **error);

    /* Called for strings that should be re-saved verbatim in this same
     * position, but are not otherwise interpretable. At the moment
     * this includes comments and processing instructions.
     * text is not nul-terminated. */
    void (*passthrough) (GMarkupParseContext *context,
                         const gchar *passthrough_text,
                         gsize text_len,
                         gpointer user_data,
                         GError **error);

    /* Called on error, including one set by other
     * methods in the vtable. The GError should not be freed.
     */
    void (*error) (GMarkupParseContext *context,
                   GError *error,
                   gpointer user_data);
} GMarkupParser;

Any of the fields in GMarkupParser can be NULL, in which case they will be ignored. Except for the
error function, any of these callbacks can set an error; in particular the G_MARKUP_ERROR_UNKNOWN_ELEMENT,
G_MARKUP_ERROR_UNKNOWN_ATTRIBUTE, and G_MARKUP_ERROR_INVALID_CONTENT errors are intended to be set from these callbacks. If you set an error from a callback, g_markup_parse_context_parse() will report that error back to its caller.

start_element () Callback to invoke when the opening tag of an element is seen.

end_element () Callback to invoke when the closing tag of an element is seen. Note that this is also
called for empty tags like <empty/>.

text () Callback to invoke when some text is seen (text is always inside an element). Note that the text of
an element may be spread over multiple calls of this function. If the G_MARKUP_TREAT_CDATA_AS_TEXT flag is set, this function is also called for the content of CDATA marked sections.

passthrough () Callback to invoke for comments, processing instructions and doctype declarations; if
you’re re-writing the parsed document, write the passthrough text back out in the same position. If the
G_MARKUP_TREAT_CDATA_AS_TEXT flag is not set, this function is also called for CDATA marked sections.

error () Callback to invoke when an error occurs.
CHAPTER 4. GLIB UTILITIES

4.21. SIMPLE XML SUBSET PARSER

**g_markup_escape_text()**

```c
char* g_markup_escape_text (const gchar *text, gssize length);
```

Escapes text so that the markup parser will parse it verbatim. Less than, greater than, ampersand, etc. are replaced with the corresponding entities. This function would typically be used when writing out a file to be parsed with the markup parser.

Note that this function doesn’t protect whitespace and line endings from being processed according to the XML rules for normalization of line endings and attribute values.

Note also that if given a string containing them, this function will produce character references in the range of &x1; .. &x1f; for all control sequences except for tabstop, newline and carriage return. The character references in this range are not valid XML 1.0, but they are valid XML 1.1 and will be accepted by the GMarkup parser.

**text**: some valid UTF-8 text

**length**: length of text in bytes, or -1 if the text is nul-terminated

**Returns**: a newly allocated string with the escaped text

**g_markup_printf_escaped()**

```c
char * g_markup_printf_escaped (const char *format, ...);
```

Formats arguments according to format, escaping all string and character arguments in the fashion of `g_markup_escape_text()`. This is useful when you want to insert literal strings into XML-style markup output, without having to worry that the strings might themselves contain markup.

```c
const char *store = "Fortnum & Mason";
const char *item = "Tea";
char *output;
output = g_markup_printf_escaped ("<purchase>
  <store>%s</store>
  <item>%s</item>
</purchase>", store, item);
```

**format**: printf() style format string

...: the arguments to insert in the format string

**Returns**: newly allocated result from formatting operation. Free with g_free().

Since 2.4

**g_markup_vprintf_escaped()**

```c
char * g_markup_vprintf_escaped (const char *format, va_list args);
```

Formats the data in args according to format, escaping all string and character arguments in the fashion of `g_markup_escape_text()`. See `g_markup_printf_escaped()`.

**format**: printf() style format string

**args**: variable argument list, similar to vprintf()

**Returns**: newly allocated result from formatting operation. Free with g_free().

Since 2.4

390
g_markup_parse_context_end_parse ()

gboolean g_markup_parse_context_end_parse (GMarkupParseContext * context, GError **error);

Signals to the GMarkupParseContext that all data has been fed into the parse context with g_markup_parse_context_parse(). This function reports an error if the document isn’t complete, for example if elements are still open.

context: a GMarkupParseContext

error: return location for a GError

Returns: TRUE on success, FALSE if an error was set

g_markup_parse_context_free ()

void g_markup_parse_context_free (GMarkupParseContext * context);

Frees a GMarkupParseContext. Can’t be called from inside one of the GMarkupParser functions. Can’t be called while a subparser is pushed.

context: a GMarkupParseContext

g_markup_parse_context_get_position ()

void g_markup_parse_context_get_position (GMarkupParseContext * context, gint *line_number, gint *char_number);

Retrieves the current line number and the number of the character on that line. Intended for use in error messages; there are no strict semantics for what constitutes the "current" line number other than "the best number we could come up with for error messages."

context: a GMarkupParseContext

line_number: return location for a line number, or NULL

char_number: return location for a char-on-line number, or NULL

g_markup_parse_context_get_element ()

const gchar * g_markup_parse_context_get_element (GMarkupParseContext * context);

Retrieves the name of the currently open element. If called from the start_element or end_element handlers this will give the element_name as passed to those functions. For the parent elements, see g_markup_parse_context_get_element_stack().

context: a GMarkupParseContext

Returns: the name of the currently open element, or NULL

Since 2.2
CHAPTER 4. GLIB UTILITIES

4.21. SIMPLE XML SUBSET PARSER

**g_markup_parse_context_get_element_stack ()**

```c
const GSList * g_markup_parse_context_get_element_stack (GMarkupParseContext * context);
```

Retrieves the element stack from the internal state of the parser. The returned GSList is a list of strings where the first item is the currently open tag (as would be returned by `g_markup_parse_context_get_element()`) and the next item is its immediate parent.

This function is intended to be used in the start_element and end_element handlers where `g_markup_parse_context_get_element()` would merely return the name of the element that is being processed.

**context**: a GMarkupParseContext

**Returns**: the element stack, which must not be modified

Since 2.16

**g_markup_parse_context_get_user_data ()**

```c
gpointer g_markup_parse_context_get_user_data (GMarkupParseContext * context);
```

Returns the user_data associated with `context`. This will either be the user_data that was provided to `g_markup_parse_context_new()` or to the most recent call of `g_markup_parse_context_push()`.

**context**: a GMarkupParseContext

**Returns**: the provided user_data. The returned data belongs to the markup context and will be freed when `g_markup_context_free()` is called.

Since 2.18

**g_markup_parse_context_new ()**

```c
GMarkupParseContext * g_markup_parse_context_new (const GMarkupParser * parser, GMarkupParseFlags flags, gpointer user_data, GDestroyNotify user_data_dnotify);
```

Creates a new parse context. A parse context is used to parse marked-up documents. You can feed any number of documents into a context, as long as no errors occur; once an error occurs, the parse context can’t continue to parse text (you have to free it and create a new parse context).

**parser**: a GMarkupParser

**flags**: one or more GMarkupParseFlags

**user_data**: user data to pass to GMarkupParser functions

**user_data_dnotify**: user data destroy notifier called when the parse context is freed

**Returns**: a new GMarkupParseContext
CHAPTER 4. GLIB UTILITIES 4.21. SIMPLE XML SUBSET PARSER

`g_markup_parse_context_parse()`

```c
gboolean g_markup_parse_context_parse (GMarkupParseContext *context,
                                      const gchar *text,
                                      gssize text_len,
                                      GError **error);
```

Feed some data to the `GMarkupParseContext`. The data need not be valid UTF-8; an error will be signaled if it’s invalid. The data need not be an entire document; you can feed a document into the parser incrementally, via multiple calls to this function. Typically, as you receive data from a network connection or file, you feed each received chunk of data into this function, aborting the process if an error occurs. Once an error is reported, no further data may be fed to the `GMarkupParseContext`; all errors are fatal.

- **context**: a `GMarkupParseContext`
- **text**: chunk of text to parse
- **text_len**: length of `text` in bytes
- **error**: return location for a `GError`

>Returns*: FALSE if an error occurred, TRUE on success

`g_markup_parse_context_push()`

```c
void g_markup_parse_context_push (GMarkupParseContext *context,
                                  GMarkupParser *parser,
                                  gpointer user_data);
```

Temporarily redirects markup data to a sub-parser.

This function may only be called from the start_element handler of a `GMarkupParser`. It must be matched with a corresponding call to `g_markup_parse_context_pop()` in the matching end_element handler (except in the case that the parser aborts due to an error).

All tags, text and other data between the matching tags is redirected to the subparser given by `parser`. `user_data` is used as the user_data for that parser. `user_data` is also passed to the error callback in the event that an error occurs. This includes errors that occur in subparsers of the subparser.

The end tag matching the start tag for which this call was made is handled by the previous parser (which is given its own user_data) which is why `g_markup_parse_context_pop()` is provided to allow "one last access" to the `user_data` provided to this function. In the case of error, the `user_data` provided here is passed directly to the error callback of the subparser and `g_markup_parse_context()` should not be called. In either case, if `user_data` was allocated then it ought to be freed from both of these locations.

This function is not intended to be directly called by users interested in invoking subparsers. Instead, it is intended to be used by the subparsers themselves to implement a higher-level interface.

As an example, see the following implementation of a simple parser that counts the number of tags encountered.

```c
typedef struct {
    gint tag_count;
} CounterData;

static void counter_start_element (GMarkupParseContext *context,
                                   const gchar *element_name,
                                   const gchar **attribute_names,
                                   const gchar **attribute_values,
                                   gpointer user_data,
                                   GError **error)
{
    CounterData *data = user_data;
    /* Count the number of tags encountered. */
    data->tag_count++;
}```
data->tag_count++;
}

static void
counter_error (GMarkupParseContext *context,
GError *error,
gpointer user_data)
{
  CounterData *data = user_data;
  g_slice_free (CounterData, data);
}

static GMarkupParser counter_subparser =
{
  counter_start_element,
  NULL,
  NULL,
  NULL,
  counter_error
};

In order to allow this parser to be easily used as a subparser, the following interface is provided:

void
start_counting (GMarkupParseContext *context)
{
  CounterData *data = g_slice_new (CounterData);
  data->tag_count = 0;
  g_markup_parse_context_push (context, &counter_subparser, data);
}

gint
end_counting (GMarkupParseContext *context)
{
  CounterData *data = g_markup_parse_context_pop (context);
  int result;
  result = data->tag_count;
  g_slice_free (CounterData, data);
  return result;
}

The subparser would then be used as follows:

static void start_element (context, element_name, ...)
{
  if (strcmp (element_name, "count-these") == 0)
    start_counting (context);
  /* else, handle other tags... */
}

static void end_element (context, element_name, ...)
{
  if (strcmp (element_name, "count-these") == 0)
    g_print ("Counted %d tags\n", end_counting (context));
  /* else, handle other tags... */
}

counter : a GMarkupParseContext
### Simple XML Subset Parser

**parser**: a GMarkupParser

**user_data**: user data to pass to GMarkupParser functions

Since 2.18

**g_markup_parse_context_pop ()**

```c
 gpointer g_markup_parse_context_pop (GMarkupParseContext * context);
```

Completes the process of a temporary sub-parser redirection.

This function exists to collect the user_data allocated by a matching call to `g_markup_parse_context_push()`. It must be called in the end_element handler corresponding to the start_element handler during which `g_markup_parse_context_push()` was called. You must not call this function from the error callback -- the user_data is provided directly to the callback in that case.

This function is not intended to be directly called by users interested in invoking subparsers. Instead, it is intended to be used by the subparsers themselves to implement a higher-level interface.

**context**: a GMarkupParseContext

**Returns**: the user_data passed to `g_markup_parse_context_push()`.

Since 2.18

**enum GMarkupCollectType**

```c
typedef enum
{
    G_MARKUP_COLLECT_INVALID,
    G_MARKUP_COLLECT_STRING,
    G_MARKUP_COLLECT_STRDUP,
    G_MARKUP_COLLECT_BOOLEAN,
    G_MARKUP_COLLECT_TRISTATE,

    G_MARKUP_COLLECT_OPTIONAL = (1 << 16)
} GMarkupCollectType;
```

A mixed enumerated type and flags field. You must specify one type (string, strdup, boolean, tristate). Additionally, you may optionally bitwise OR the type with the flag `G_MARKUP_COLLECT_OPTIONAL`. It is likely that this enum will be extended in the future to support other types.

**G_MARKUP_COLLECT_INVALID** used to terminate the list of attributes to collect.

**G_MARKUP_COLLECT_STRING** collect the string pointer directly from the attribute_values[] array. Expects a parameter of type (const char **). If `G_MARKUP_COLLECT_OPTIONAL` is specified and the attribute isn’t present then the pointer will be set to NULL.

**G_MARKUP_COLLECT_STRDUP** as with `G_MARKUP_COLLECT_STRING`, but expects a parameter of type (char **) and `g_strdup()`s the returned pointer. The pointer must be freed with `g_free()`.

**G_MARKUP_COLLECT_BOOLEAN** expects a parameter of type (gboolean *) and parses the attribute value as a boolean. Sets `FALSE` if the attribute isn’t present. Valid boolean values consist of (case insensitive) "false", "f", "no", "n", "0" and "true", "t", "yes", "y", "1".

**G_MARKUP_COLLECT_TRISTATE** as with `G_MARKUP_COLLECT_BOOLEAN`, but in the case of a missing attribute a value is set that compares equal to neither `FALSE` nor `TRUE`. `G_MARKUP_COLLECT_OPTIONAL` is implied.

**G_MARKUP_COLLECT_OPTIONAL** can be bitwise ORed with the other fields. If present, allows the attribute not to appear. A default value is set depending on what value type is used.
g_markup_collect_attributes ()

Collects the attributes of the element from the data passed to the GMarkupParser start_element function, dealing with common error conditions and supporting boolean values.

This utility function is not required to write a parser but can save a lot of typing.

The element_name, attribute_names, attribute_values and error parameters passed to the start_element callback should be passed unmodified to this function.

Following these arguments is a list of "supported" attributes to collect. It is an error to specify multiple attributes with the same name. If any attribute not in the list appears in the attribute_names array then an unknown attribute error will result.

The GMarkupCollectType field allows specifying the type of collection to perform and if a given attribute must appear or is optional.

The attribute name is simply the name of the attribute to collect.

The pointer should be of the appropriate type (see the descriptions under GMarkupCollectType) and may be NULL in case a particular attribute is to be allowed but ignored.

This function deals with issuing errors for missing attributes (of type G_MARKUP_ERROR_MISSING_ATTRIBUTE), unknown attributes (of type G_MARKUP_ERROR_UNKNOWN_ATTRIBUTE) and duplicate attributes (of type G_MARKUP_ERROR_INVALID_CONTENT) as well as parse errors for boolean-valued attributes (again of type G_MARKUP_ERROR_INVALID_CONTENT). In all of these cases FALSE will be returned and error will be set as appropriate.

**element_name**: the current tag name

**attribute_names**: the attribute names

**attribute_values**: the attribute values

**error**: a pointer to a GError or NULL

**first_type**: the GMarkupCollectType of the first attribute

**first_attr**: the name of the first attribute

...: a pointer to the storage location of the first attribute (or NULL), followed by more types names and pointers, ending with G_MARKUP_COLLECT_INVALID.

**Returns**: TRUE if successful

Since 2.16

4.22 Key-value file parser

Name

Key-value file parser – parses .ini-like config files
Synopsis

#include <glib.h>

GKeyFile;
#define G_KEY_FILE_ERROR
enum GKeyFileError;
enum GKeyFileFlags;

GKeyFile * g_key_file_new (void);
void g_key_file_free (GKeyFile *key_file);
void g_key_file_set_list_separator (GKeyFile *key_file,
gchar separator);
gboolean g_key_file_load_from_file (GKeyFile *key_file,
const gchar *file,
GKeyFileFlags flags,
GError **error);
gboolean g_key_file_load_from_data (GKeyFile *key_file,
const gchar *data,
gsize length,
GKeyFileFlags flags,
GError **error);
gboolean g_key_file_load_from_data_dirs (GKeyFile *key_file,
const gchar *file,
gchar **full_path,
GKeyFileFlags flags,
GError **error);
gboolean g_key_file_load_from_dirs (GKeyFile *key_file,
const gchar *file,
const gchar **search_dirs,
gchar **full_path,
GKeyFileFlags flags,
GError **error);
gchar * g_key_file_to_data (GKeyFile *key_file,
gsize *length,
GError **error);
gchar * g_key_file_get_start_group (GKeyFile *key_file);
gchar ** g_key_file_get_groups (GKeyFile *key_file,
gsize *length);
gchar ** g_key_file_get_keys (GKeyFile *key_file,
const gchar *group_name,
gsize *length,
GError **error);
gboolean g_key_file_has_group (GKeyFile *key_file,
const gchar *group_name);
gboolean g_key_file_has_key (GKeyFile *key_file,
const gchar *group_name,
const gchar *key,
GError **error);
gchar * g_key_file_get_value (GKeyFile *key_file,
const gchar *group_name,
const gchar *key,
GError **error);
gchar * g_key_file_get_string (GKeyFile *key_file,
const gchar *group_name,
const gchar *key,
GError **error);
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gchar * g_key_file_get_locale_string</td>
<td>Gets a locale string from a key-value file.</td>
</tr>
<tr>
<td>gboolean g_key_file_get_boolean</td>
<td>Gets a boolean value from a key-value file.</td>
</tr>
<tr>
<td>gint g_key_file_get_integer</td>
<td>Gets an integer value from a key-value file.</td>
</tr>
<tr>
<td>gdouble g_key_file_get_double</td>
<td>Gets a double value from a key-value file.</td>
</tr>
<tr>
<td>gchar ** g_key_file_get_string_list</td>
<td>Gets a string list from a key-value file.</td>
</tr>
<tr>
<td>gchar ** g_key_file_get_locale_string_list</td>
<td>Gets a locale string list from a key-value file.</td>
</tr>
<tr>
<td>gboolean * g_key_file_get_boolean_list</td>
<td>Gets a boolean list from a key-value file.</td>
</tr>
<tr>
<td>gint * g_key_file_get_integer_list</td>
<td>Gets an integer list from a key-value file.</td>
</tr>
<tr>
<td>gdouble * g_key_file_get_double_list</td>
<td>Gets a double list from a key-value file.</td>
</tr>
<tr>
<td>gchar * g_key_file_get_comment</td>
<td>Gets a comment from a key-value file.</td>
</tr>
<tr>
<td>void g_key_file_set_value</td>
<td>Sets a value in a key-value file.</td>
</tr>
<tr>
<td>void g_key_file_set_string</td>
<td>Sets a string in a key-value file.</td>
</tr>
<tr>
<td>void g_key_file_set_locale_string</td>
<td>Sets a locale string in a key-value file.</td>
</tr>
</tbody>
</table>
void g_key_file_set_boolean (GKeyFile *key_file, const gchar *group_name, const gchar *key, gboolean value);

void g_key_file_set_integer (GKeyFile *key_file, const gchar *group_name, const gchar *key, gint value);

void g_key_file_set_double (GKeyFile *key_file, const gchar *group_name, const gchar *key, gdouble value);

void g_key_file_set_string_list (GKeyFile *key_file, const gchar *group_name, const gchar *key, const gchar * const list[], gsize length);

void g_key_file_set_locale_string_list (GKeyFile *key_file, const gchar *group_name, const gchar *key, const gchar *locale, const gchar * const list[], gsize length);

void g_key_file_set_boolean_list (GKeyFile *key_file, const gchar *group_name, const gchar *key, gboolean list[], gsize length);

void g_key_file_set_integer_list (GKeyFile *key_file, const gchar *group_name, const gchar *key, gint list[], gsize length);

void g_key_file_set_double_list (GKeyFile *key_file, const gchar *group_name, const gchar *key, gdouble list[], gsize length);

gboolean g_key_file_set_comment (GKeyFile *key_file, const gchar *group_name, const gchar *key, const gchar *comment, GError **error);

gboolean g_key_file_remove_group (GKeyFile *key_file, const gchar *group_name, GError **error);

gboolean g_key_file_remove_key (GKeyFile *key_file, const gchar *group_name, const gchar *key, GError **error);

gboolean g_key_file_remove_comment (GKeyFile *key_file, const gchar *group_name, const gchar *key, GError **error);

#define G_KEY_FILE_DESKTOP_GROUP
#define G_KEY_FILE_DESKTOP_KEY_TYPE

399
Description

GKeyFile lets you parse, edit or create files containing groups of key-value pairs, which we call key files for lack of a better name. Several freedesktop.org specifications use key files now, e.g. the Desktop Entry Specification and the Icon Theme Specification.

The syntax of key files is described in detail in the Desktop Entry Specification, here is a quick summary: Key files consist of groups of key-value pairs, interspersed with comments.

```
# this is just an example
# there can be comments before the first group
[First Group]
Name=Key File Example\this value shows\escaping
# localized strings are stored in multiple key-value pairs
Welcome=Hello
Welcome[de]=Hallo
Welcome[fr_FR]=Bonjour
Welcome[it]=Ciao
Welcome[be@latin]=Hello
[Another Group]
Numbers=2;20;-200;0
Booleans=true,false;true,true
```

Lines beginning with a ‘#’ and blank lines are considered comments.

Groups are started by a header line containing the group name enclosed in ‘[’ and ‘]’, and ended implicitly by the start of the next group or the end of the file. Each key-value pair must be contained in a group.

Key-value pairs generally have the form key=value, with the exception of localized strings, which have the form key[locale]=value, with a locale identifier of the form lang_COUNTRYMODIFIER where COUNTRY and MODIFIER are optional. Space before and after the ‘=’ character are ignored. Newline, tab, carriage return and backslash characters in value are escaped as \n, \t, \r, and \\, respectively. To preserve leading spaces in values, these can also be escaped as \s.

Key files can store strings (possibly with localized variants), integers, booleans and lists of these. Lists are separated by a separator character, typically ‘;’ or ‘,’. To use the list separator character in a value in a list, it has to be escaped by prefixing it with a backslash.

This syntax is obviously inspired by the .ini files commonly met on Windows, but there are some important differences:

- .ini files use the ‘;’ character to begin comments, key files use the ‘#’ character.
- Key files do not allow for ungrouped keys meaning only comments can precede the first group.
• Key files are always encoded in UTF-8.
• Key and Group names are case-sensitive, for example a group called `[GROUP]` is a different group from `[group]`.
• `.ini` files don’t have a strongly typed boolean entry type, they only have `GetProfileInt`. In GKeyFile only `true` and `false` (in lower case) are allowed.

Note that in contrast to the Desktop Entry Specification, groups in key files may contain the same key multiple times; the last entry wins. Key files may also contain multiple groups with the same name; they are merged together. Another difference is that keys and group names in key files are not restricted to ASCII characters.

Details

GKeyFile

typedef struct _GKeyFile GKeyFile;

The GKeyFile struct contains only private fields and should not be used directly.

G_KEY_FILE_ERROR
#define G_KEY_FILE_ERROR g_key_file_error_quark()

Error domain for key file parsing. Errors in this domain will be from the GKeyFileError enumeration. See GError for information on error domains.

enum GKeyFileError

typedef enum
{
    G_KEY_FILE_ERROR_UNKNOWN_ENCODING,
    G_KEY_FILE_ERROR_PARSE,
    G_KEY_FILE_ERROR_NOT_FOUND,
    G_KEY_FILE_ERROR_KEY_NOT_FOUND,
    G_KEY_FILE_ERROR_GROUP_NOT_FOUND,
    G_KEY_FILE_ERROR_INVALID_VALUE
} GKeyFileError;

Error codes returned by key file parsing.

G_KEY_FILE_ERROR_UNKNOWN_ENCODING the text being parsed was in an unknown encoding
G_KEY_FILE_ERROR_PARSE document was ill-formed
G_KEY_FILE_ERROR_NOT_FOUND the file was not found
G_KEY_FILE_ERROR_KEY_NOT_FOUND a requested key was not found
G_KEY_FILE_ERROR_GROUP_NOT_FOUND a requested group was not found
G_KEY_FILE_ERROR_INVALID_VALUE a value could not be parsed

enum GKeyFileFlags

typedef enum
{
    G_KEY_FILE_NONE = 0,
    G_KEY_FILE_KEEP_COMMENTS = 1 << 0,
    G_KEY_FILE_KEEP_TRANSLATIONS = 1 << 1
} GKeyFileFlags;
Flags which influence the parsing.

**G_KEY_FILE_NONE**  No flags, default behaviour

**G_KEY_FILE_KEEP_COMMENTS**  Use this flag if you plan to write the (possibly modified) contents of the key file back to a file; otherwise all comments will be lost when the key file is written back.

**G_KEY_FILE_KEEP_TRANSLATIONS**  Use this flag if you plan to write the (possibly modified) contents of the key file back to a file; otherwise only the translations for the current language will be written back.

### g_key_file_new()

```c
GKeyFile * g_key_file_new (void);
```

Creates a new empty GKeyFile object. Use `g_key_file_load_from_file()`, `g_key_file_load_from_data()`, `g_key_file_load_from_dirs()` or `g_key_file_load_from_data_dirs()` to read an existing key file.

**Returns**: an empty GKeyFile.

Since 2.6

### g_key_file_free()

```c
void g_key_file_free (GKeyFile *key_file);
```

Frees a GKeyFile.

**key_file**: a GKeyFile

Since 2.6

### g_key_file_set_list_separator()

```c
void g_key_file_set_list_separator (GKeyFile *key_file, gchar separator);
```

Sets the character which is used to separate values in lists. Typically ‘;’ or ‘,’ are used as separators. The default list separator is ‘;’.

**key_file**: a GKeyFile

**separator**: the separator

Since 2.6

### g_key_file_load_from_file()

```c
gboolean g_key_file_load_from_file (GKeyFile *key_file, const gchar *file, GKeyFileFlags flags, GError **error);
```

Loads a key file into an empty GKeyFile structure. If the file could not be loaded then `error` is set to either a GFileError or GKeyFileError.

**key_file**: an empty GKeyFile struct

**file**: the path of a filename to load, in the GLib filename encoding

**flags**: flags from GKeyFileFlags

**error**: return location for a GError, or NULL

**Returns**: TRUE if a key file could be loaded, FALSE otherwise

Since 2.6
g_key_file_load_from_data ()

 gboolean g_key_file_load_from_data (GKeyFile *key_file,
               const gchar *data,
               gsize length,
               GKeyFileFlags flags,
               GError **error);

Loads a key file from memory into an empty GKeyFile structure. If the object cannot be created then error is set to a GKeyFileError.

**key_file**: an empty GKeyFile struct  
**data**: key file loaded in memory  
**length**: the length of data in bytes  
**flags**: flags from GKeyFileFlags  
**error**: return location for a GError, or NULL  

**Returns**: TRUE if a key file could be loaded, FALSE otherwise  
Since 2.6

g_key_file_load_from_data_dirs ()

 gboolean g_key_file_load_from_data_dirs (GKeyFile *key_file,
               const gchar *file,
               gchar **full_path,
               GKeyFileFlags flags,
               GError **error);

This function looks for a key file named file in the paths returned from g_get_user_data_dir() and g_get_system_data_dirs(), loads the file into key_file and returns the file’s full path in full_path. If the file could not be loaded then an error is set to either a GFileError or GKeyFileError.

**key_file**: an empty GKeyFile struct  
**file**: a relative path to a filename to open and parse  
**full_path**: return location for a string containing the full path of the file, or NULL  
**flags**: flags from GKeyFileFlags  
**error**: return location for a GError, or NULL  

**Returns**: TRUE if a key file could be loaded, FALSE otherwise  
Since 2.6

g_key_file_load_from_dirs ()

 gboolean g_key_file_load_from_dirs (GKeyFile *key_file,
               const gchar *file,
               gchar **search_dirs,
               gchar **full_path,
               GKeyFileFlags flags,
               GError **error);

This function looks for a key file named file in the paths specified in search_dirs, loads the file into key_file and returns the file’s full path in full_path. If the file could not be loaded then an error is set to either a GFileError or GKeyFileError.
key_file: an empty GKeyFile struct

cache: a relative path to a filename to open and parse

search_dirs: NULL-terminated array of directories to search

full_path: return location for a string containing the full path of the file, or NULL

flags: flags from GKeyFileFlags

error: return location for a GError, or NULL

Returns: TRUE if a key file could be loaded, FALSE otherwise

Since 2.14

```
gchar * g_key_file_to_data (GKeyFile *key_file,
                          gchar **length,
                          GError **error);
```

This function outputs key_file as a string.
Note that this function never reports an error, so it is safe to pass NULL as error.

key_file: a GKeyFile

length: return location for the length of the returned string, or NULL

error: return location for a GError, or NULL

Returns: a newly allocated string holding the contents of the GKeyFile

Since 2.6

```
gchar * g_key_file_get_start_group (GKeyFile *key_file);
```

Returns the name of the start group of the file.

key_file: a GKeyFile

Returns: The start group of the key file.

Since 2.6

```
gchar ** g_key_file_get_groups (GKeyFile *key_file,
                                 gchar *length);
```

Returns all groups in the key file loaded with key_file. The array of returned groups will be NULL-terminated, so length may optionally be NULL.

key_file: a GKeyFile

length: return location for the number of returned groups, or NULL

Returns: a newly-allocated NULL-terminated array of strings. Use g_strfreev() to free it.

Since 2.6
g_key_file_get_keys()

```c
gboolean g_key_file_has_group (GKeyFile *key_file,
                              const gchar *group_name);
```

Looks whether the key file has the group `group_name`.  

**key_file**: a `GKeyFile`  

**group_name**: a group name  

**Returns**: `TRUE` if `group_name` is a part of `key_file`, `FALSE` otherwise.  

Since 2.6

### g_key_file_has_key()

```c
gboolean g_key_file_has_key (GKeyFile *key_file,
                             const gchar *group_name,
                             const gchar *key,
                             GError **error);
```

Looks whether the key file has the key `key` in the group `group_name`.  

**key_file**: a `GKeyFile`  

**group_name**: a group name  

**key**: a key name  

**error**: return location for a `GError`  

**Returns**: `TRUE` if `key` is a part of `group_name`, `FALSE` otherwise.  

Since 2.6
CHAPTER 4. GLIB UTILITIES 4.22. KEY-VALUE FILE PARSER

**g_key_file_get_value()**

```c
gboolean g_key_file_get_value (GKeyFile *key_file,
const gchar *group_name,
const gchar *key,
GError **error);
```

Returns the raw value associated with `key` under `group_name`. Use `g_key_file_get_string()` to retrieve an unescaped UTF-8 string.

In the event the key cannot be found, `NULL` is returned and `error` is set to `G_KEY_FILE_ERROR_KEY_NOT_FOUND`. In the event that the `group_name` cannot be found, `NULL` is returned and `error` is set to `G_KEY_FILE_ERROR_GROUP_NOT_FOUND`.

- **key_file**: a `GKeyFile`
- **group_name**: a group name
- **key**: a key
- **error**: return location for a `GError`, or `NULL`

Returns: a newly allocated string or `NULL` if the specified key cannot be found.

Since 2.6

**g_key_file_get_string()**

```c
gboolean g_key_file_get_string (GKeyFile *key_file,
const gchar *group_name,
const gchar *key,
GError **error);
```

Returns the string value associated with `key` under `group_name`. Unlike `g_key_file_get_value()`, this function handles escape sequences like `\s`.

In the event the key cannot be found, `NULL` is returned and `error` is set to `G_KEY_FILE_ERROR_KEY_NOT_FOUND`. In the event that the `group_name` cannot be found, `NULL` is returned and `error` is set to `G_KEY_FILE_ERROR_GROUP_NOT_FOUND`.

- **key_file**: a `GKeyFile`
- **group_name**: a group name
- **key**: a key
- **error**: return location for a `GError`, or `NULL`

Returns: a newly allocated string or `NULL` if the specified key cannot be found.

Since 2.6

**g_key_file_get_locale_string()**

```c
gboolean g_key_file_get_locale_string (GKeyFile *key_file,
const gchar *group_name,
const gchar *key,
const gchar *locale,
GError **error);
```

Returns the value associated with `key` under `group_name` translated in the given `locale` if available. If `locale` is `NULL` then the current locale is assumed.

If `key` cannot be found then `NULL` is returned and `error` is set to `G_KEY_FILE_ERROR_KEY_NOT_FOUND`. If the value associated with `key` cannot be interpreted or no suitable translation can be found then the untranslated value is returned.

- **key_file**: a `GKeyFile`
- **group_name**: a group name

Since 2.6
key: a key
locale: a locale identifier or NULL
error: return location for a GError, or NULL

Returns: a newly allocated string or NULL if the specified key cannot be found.

Since 2.6

g_key_file_get_boolean()

gboolean g_key_file_get_boolean (GKeyFile *key_file,
const gchar *group_name,
const gchar *key,
GError **error);

Returns the value associated with key under group_name as a boolean.
If key cannot be found then FALSE is returned and error is set to G_KEY_FILE_ERROR_KEY_NOT_FOUND.
Likewise, if the value associated with key cannot be interpreted as a boolean then FALSE is returned
and error is set to G_KEY_FILE_ERROR_INVALID_VALUE.

key_file: a GKeyFile

group_name: a group name

key: a key

error: return location for a GError

Returns: the value associated with the key as a boolean, or FALSE if the key was not found or could
not be parsed.

Since 2.6

g_key_file_get_integer()

gint g_key_file_get_integer (GKeyFile *key_file,
const gchar *group_name,
const gchar *key,
GError **error);

Returns the value associated with key under group_name as an integer.
If key cannot be found then 0 is returned and error is set to G_KEY_FILE_ERROR_KEY_NOT_FOUND.
Likewise, if the value associated with key cannot be interpreted as an integer then 0 is returned and e-
error is set to G_KEY_FILE_ERROR_INVALID_VALUE.

key_file: a GKeyFile

group_name: a group name

key: a key

error: return location for a GError

Returns: the value associated with the key as an integer, or 0 if the key was not found or could not be parsed.

Since 2.6
**g_key_file_get_double()**

```c
gdouble g_key_file_get_double (GKeyFile *key_file,
    const gchar *group_name,
    const gchar *key,
    GError **error);
```

Returns the value associated with `key` under `group_name` as a double. If `group_name` is NULL, the start group is used. If `key` cannot be found then 0.0 is returned and `error` is set to `G_KEY_FILE_ERROR_KEY_NOT_FOUND`. Likewise, if the value associated with `key` cannot be interpreted as a double then 0.0 is returned and `error` is set to `G_KEY_FILE_ERROR_INVALID_VALUE`.

- **key_file**: a GKeyFile
- **group_name**: a group name
- **key**: a key
- **error**: return location for a GError

**Returns**: the value associated with the key as a double, or 0.0 if the key was not found or could not be parsed.

Since 2.12

**g_key_file_get_string_list()**

```c
gchar ** g_key_file_get_string_list (GKeyFile *key_file,
    const gchar *group_name,
    const gchar *key,
    gsize *length,
    GError **error);
```

Returns the values associated with `key` under `group_name`. In the event the key cannot be found, NULL is returned and `error` is set to `G_KEY_FILE_ERROR_KEY_NOT_FOUND`. In the event that the `group_name` cannot be found, NULL is returned and `error` is set to `G_KEY_FILE_ERROR_GROUP_NOT_FOUND`.

- **key_file**: a GKeyFile
- **group_name**: a group name
- **key**: a key
- **length**: return location for the number of returned strings, or NULL
- **error**: return location for a GError, or NULL

**Returns**: a NULL-terminated string array or NULL if the specified key cannot be found. The array should be freed with g_strfreev().

Since 2.6

**g_key_file_get_locale_string_list()**

```c
gchar ** g_key_file_get_locale_string_list (GKeyFile *key_file,
    const gchar *group_name,
    const gchar *key,
    const gchar *locale,
    gsize *length,
    GError **error);
```
Returns the values associated with key under group_name translated in the given locale if available. If locale is NULL then the current locale is assumed. If key cannot be found then NULL is returned and error is set to G_KEY_FILE_ERROR_KEY_NOT_FOUND. If the values associated with key cannot be interpreted or no suitable translations can be found then the untranslated values are returned. The returned array is NULL-terminated, so length may optionally be NULL.

key_file: a GKeyFile

group_name: a group name

key: a key

locale: a locale identifier or NULL

length: return location for the number of returned strings or NULL

error: return location for a GError or NULL

Returns: a newly allocated NULL-terminated string array or NULL if the key isn’t found. The string array should be freed with g_strfreev(). Since 2.6

\textbf{\texttt{g_key_file_get_boolean_list()}}

| gboolean * | g_key_file_get_boolean_list | (GKeyFile *key_file, |
| const gchar *group_name, |
| const gchar *key, |
| gsize *length, |
| GError **error); |

Returns the values associated with key under group_name as booleans. If key cannot be found then NULL is returned and error is set to G_KEY_FILE_ERROR_KEY_NOT_FOUND. Likewise, if the values associated with key cannot be interpreted as booleans then NULL is returned and error is set to G_KEY_FILE_ERROR_INVALID_VALUE.

key_file: a GKeyFile

group_name: a group name

key: a key

length: the number of booleans returned

error: return location for a GError

Returns: the values associated with the key as a list of booleans, or NULL if the key was not found or could not be parsed. Since 2.6

\textbf{\texttt{g_key_file_get_integer_list()}}

| gint * | g_key_file_get_integer_list | (GKeyFile *key_file, |
| const gchar *group_name, |
| const gchar *key, |
| gsize *length, |
| GError **error); |

Returns the values associated with key under group_name as integers. If key cannot be found then NULL is returned and error is set to G_KEY_FILE_ERROR_KEY_NOT_FOUND. Likewise, if the values associated with key cannot be interpreted as integers then NULL is returned and error is set to G_KEY_FILE_ERROR_INVALID_VALUE.
key_file: a GKeyFile

group_name: a group name

key: a key

length: the number of integers returned

error: return location for a GError

Returns: the values associated with the key as a list of integers, or NULL if the key was not found or could not be parsed.

Since 2.6

g_key_file_get_double_list ()

```c
qdouble * g_key_file_get_double_list (GKeyFile *key_file,
const gchar *group_name,
const gchar *key,
gsize *length,
GError **error);
```

Returns the values associated with key under group_name as doubles. If key cannot be found then NULL is returned and error is set to G_KEY_FILE_ERROR_KEY_NOT_FOUND. Likewise, if the values associated with key cannot be interpreted as doubles then NULL is returned and error is set to G_KEY_FILE_ERROR_INVALID_VALUE.

key_file: a GKeyFile

group_name: a group name

key: a key

length: the number of doubles returned

error: return location for a GError

Returns: the values associated with the key as a list of doubles, or NULL if the key was not found or could not be parsed.

Since 2.12

g_key_file_get_comment ()

```c
gchar * g_key_file_get_comment (GKeyFile *key_file,
const gchar *group_name,
const gchar *key,
GError **error);
```

Retrieves a comment above key from group_name. If key is NULL then comment will be read from above group_name. If both key and group_name are NULL, then comment will be read from above the first group in the file.

key_file: a GKeyFile

group_name: a group name, or NULL

key: a key

error: return location for a GError

Returns: a comment that should be freed with g_free()

Since 2.6
g_key_file_set_value ()

```c
void g_key_file_set_value (GKeyFile *key_file,
        const gchar *group_name,
        const gchar *key,
        const gchar *value);
```

Associates a new value with `key` under `group_name`. If `key` cannot be found then it is created. If `group_name` cannot be found then it is created. To set an UTF-8 string which may contain characters that need escaping (such as newlines or spaces), use `g_key_file_set_string()`.

- **key_file**: a GKeyFile
- **group_name**: a group name
- **key**: a key
- **value**: a string
  - Since 2.6

---

**g_key_file_set_string ()**

```c
void g_key_file_set_string (GKeyFile *key_file,
        const gchar *group_name,
        const gchar *key,
        const gchar *string);
```

Associates a new string value with `key` under `group_name`. If `key` cannot be found then it is created. If `group_name` cannot be found then it is created. Unlike `g_key_file_set_value()`, this function handles characters that need escaping, such as newlines.

- **key_file**: a GKeyFile
- **group_name**: a group name
- **key**: a key
- **string**: a string
  - Since 2.6

---

**g_key_file_set_locale_string ()**

```c
void g_key_file_set_locale_string (GKeyFile *key_file,
        const gchar *group_name,
        const gchar *key,
        const gchar *locale,
        const gchar *string);
```

Associates a string value for `key` and `locale` under `group_name`. If the translation for `key` cannot be found then it is created.

- **key_file**: a GKeyFile
- **group_name**: a group name
- **key**: a key
- **locale**: a locale identifier
- **string**: a string
  - Since 2.6
g_key_file_set_boolean()

```c
void g_key_file_set_boolean (GKeyFile *key_file,
   const gchar *group_name,
   const gchar *key,
   gboolean value);
```

Associates a new boolean value with `key` under `group_name`. If `key` cannot be found then it is created.

**key_file**: a GKeyFile

**group_name**: a group name

**key**: a key

**value**: TRUE or FALSE

Since 2.6

---

g_key_file_set_integer()

```c
void g_key_file_set_integer (GKeyFile *key_file,
   const gchar *group_name,
   const gchar *key,
   gint value);
```

Associates a new integer value with `key` under `group_name`. If `key` cannot be found then it is created.

**key_file**: a GKeyFile

**group_name**: a group name

**key**: a key

**value**: an integer value

Since 2.6

---

g_key_file_set_double()

```c
void g_key_file_set_double (GKeyFile *key_file,
   const gchar *group_name,
   const gchar *key,
   gdouble value);
```

Associates a new double value with `key` under `group_name`. If `key` cannot be found then it is created.

**key_file**: a GKeyFile

**group_name**: a group name

**key**: a key

**value**: an double value

Since 2.12
### g_key_file_set_string_list()

```c
void g_key_file_set_string_list (GKeyFile *key_file,
    const gchar *group_name,
    const gchar *key,
    const gchar * const list[],
    gsize length);
```

Associates a list of string values for `key` under `group_name`. If `key` cannot be found then it is created. If `group_name` cannot be found then it is created.

- **key_file**: a `GKeyFile`
- **group_name**: a group name
- **key**: a key
- **list**: an array of string values
- **length**: number of string values in `list`

Since 2.6

### g_key_file_set_locale_string_list()

```c
void g_key_file_set_locale_string_list (GKeyFile *key_file,
    const gchar *group_name,
    const gchar *key,
    const gchar *locale,
    const gchar * const list[],
    gsize length);
```

Associates a list of string values for `key` and `locale` under `group_name`. If the translation for `key` cannot be found then it is created.

- **key_file**: a `GKeyFile`
- **group_name**: a group name
- **key**: a key
- **locale**: a locale identifier
- **list**: a NULL-terminated array of locale string values
- **length**: the length of `list`

Since 2.6

### g_key_file_set_boolean_list()

```c
void g_key_file_set_boolean_list (GKeyFile *key_file,
    const gchar *group_name,
    const gchar *key,
    gboolean list[],
    gsize length);
```

Associates a list of boolean values with `key` under `group_name`. If `key` cannot be found then it is created. If `group_name` is NULL, the start_group is used.

- **key_file**: a `GKeyFile`
- **group_name**: a group name
**key**: a key

**list**: an array of boolean values

**length**: length of list

Since 2.6

```c
void g_key_file_set_integer_list (GKeyFile *key_file, const gchar *group_name, const gchar *key, gint list[], gsize length);
```

Associates a list of integer values with **key** under **group_name**. If **key** cannot be found then it is created.

- **key_file**: a GKeyFile
- **group_name**: a group name
- **key**: a key
- **list**: an array of integer values
- **length**: number of integer values in **list**

Since 2.6

```c
void g_key_file_set_double_list (GKeyFile *key_file, const gchar *group_name, const gchar *key, gdouble list[], gsize length);
```

Associates a list of double values with **key** under **group_name**. If **key** cannot be found then it is created.

- **key_file**: a GKeyFile
- **group_name**: a group name
- **key**: a key
- **list**: an array of double values
- **length**: number of double values in **list**

Since 2.12

```c
gboolean g_key_file_set_comment (GKeyFile *key_file, const gchar *group_name, const gchar *key, const gchar *comment, GError **error);
```

Places a comment above **key** from **group_name**. If **key** is **NULL** then **comment** will be written above **group_name**. If both **key** and **group_name** are **NULL**, then **comment** will be written above the first group in the file.
**CHAPTER 4. GLIB UTILITIES**

**4.22. KEY-VALUE FILE PARSER**

**key_file**: a GKeyFile

**group_name**: a group name, or NULL

**key**: a key

**comment**: a comment

**error**: return location for a GError

**Returns**: TRUE if the comment was written, FALSE otherwise

Since 2.6

**g_key_file_remove_group()**

```c
gboolean g_key_file_remove_group (GKeyFile *key_file, const gchar *group_name, GError **error);
```

Removes the specified group, `group_name`, from the key file.

**key_file**: a GKeyFile

**group_name**: a group name

**error**: return location for a GError or NULL

**Returns**: TRUE if the group was removed, FALSE otherwise

Since 2.6

**g_key_file_remove_key()**

```c
gboolean g_key_file_remove_key (GKeyFile *key_file, const gchar *group_name, const gchar *key, GError **error);
```

Removes **key** in **group_name** from the key file.

**key_file**: a GKeyFile

**group_name**: a group name

**key**: a key name to remove

**error**: return location for a GError or NULL

**Returns**: TRUE if the key was removed, FALSE otherwise

Since 2.6

**g_key_file_remove_comment()**

```c
gboolean g_key_file_remove_comment (GKeyFile *key_file, const gchar *group_name, const gchar *key, GError **error);
```

Removes a comment above **key** from **group_name**. If **key** is NULL then **comment** will be removed above **group_name**. If both **key** and **group_name** are NULL, then **comment** will be removed above the first group in the file.

**key_file**: a GKeyFile
group_name: a group name, or NULL
key: a key
error: return location for a GError

Returns: TRUE if the comment was removed, FALSE otherwise

Since 2.6

G_KEY_FILE_DESKTOP_GROUP

#define G_KEY_FILE_DESKTOP_GROUP "Desktop Entry"

The name of the main group of a desktop entry file, as defined in the Desktop Entry Specification. Consult the specification for more details about the meanings of the keys below.

Since 2.14

G_KEY_FILE_DESKTOP_KEY_TYPE

#define G_KEY_FILE_DESKTOP_KEY_TYPE "Type"

A key under G_KEY_FILE_DESKTOP_GROUP whose value is a string giving the type of the desktop entry. Usually G_KEY_FILE_DESKTOP_TYPE_APPLICATION, G_KEY_FILE_DESKTOP_TYPE_LINK, or G_KEY_FILE_DESKTOP_TYPE_DIRECTORY.

Since 2.14

G_KEY_FILE_DESKTOP_KEY_VERSION

#define G_KEY_FILE_DESKTOP_KEY_VERSION "Version"

A key under G_KEY_FILE_DESKTOP_GROUP whose value is a string giving the version of the Desktop Entry Specification used for the desktop entry file.

Since 2.14

G_KEY_FILE_DESKTOP_KEY_NAME

#define G_KEY_FILE_DESKTOP_KEY_NAME "Name"

A key under G_KEY_FILE_DESKTOP_GROUP whose value is a localized string giving the specific name of the desktop entry.

Since 2.14

G_KEY_FILE_DESKTOP_KEY_GENERIC_NAME

#define G_KEY_FILE_DESKTOP_KEY_GENERIC_NAME "GenericName"

A key under G_KEY_FILE_DESKTOP_GROUP whose value is a localized string giving the generic name of the desktop entry.

Since 2.14

G_KEY_FILE_DESKTOP_KEY_NO_DISPLAY

#define G_KEY_FILE_DESKTOP_KEY_NO_DISPLAY "NoDisplay"

A key under G_KEY_FILE_DESKTOP_GROUP whose value is a boolean stating whether the desktop entry should be shown in menus.

Since 2.14
4.22. KEY-VALUE FILE PARSER

**G_KEY_FILE_DESKTOP_KEY_COMMENT**

```c
#define G_KEY_FILE_DESKTOP_KEY_COMMENT "Comment"
```

A key under `G_KEY_FILE_DESKTOP_GROUP` whose value is a localized string giving the tooltip for the desktop entry.

Since 2.14

**G_KEY_FILE_DESKTOP_KEY_ICON**

```c
#define G_KEY_FILE_DESKTOP_KEY_ICON "Icon"
```

A key under `G_KEY_FILE_DESKTOP_GROUP` whose value is a localized string giving the name of the icon to be displayed for the desktop entry.

Since 2.14

**G_KEY_FILE_DESKTOP_KEY_HIDDEN**

```c
#define G_KEY_FILE_DESKTOP_KEY_HIDDEN "Hidden"
```

A key under `G_KEY_FILE_DESKTOP_GROUP` whose value is a boolean stating whether the desktop entry has been deleted by the user.

Since 2.14

**G_KEY_FILE_DESKTOP_KEY_ONLY_SHOW_IN**

```c
#define G_KEY_FILE_DESKTOP_KEY_ONLY_SHOW_IN "OnlyShowIn"
```

A key under `G_KEY_FILE_DESKTOP_GROUP` whose value is a list of strings identifying the environments that should display the desktop entry.

Since 2.14

**G_KEY_FILE_DESKTOP_KEY_NOT_SHOW_IN**

```c
#define G_KEY_FILE_DESKTOP_KEY_NOT_SHOW_IN "NotShowIn"
```

A key under `G_KEY_FILE_DESKTOP_GROUP` whose value is a list of strings identifying the environments that should not display the desktop entry.

Since 2.14

**G_KEY_FILE_DESKTOP_KEY_TRY_EXEC**

```c
#define G_KEY_FILE_DESKTOP_KEY_TRY_EXEC "TryExec"
```

A key under `G_KEY_FILE_DESKTOP_GROUP` whose value is a string giving the file name of a binary on disk used to determine if the program is actually installed. It is only valid for desktop entries with the `Application` type.

Since 2.14

**G_KEY_FILE_DESKTOP_KEY_EXEC**

```c
#define G_KEY_FILE_DESKTOP_KEY_EXEC "Exec"
```

A key under `G_KEY_FILE_DESKTOP_GROUP` whose value is a string giving the command line to execute. It is only valid for desktop entries with the `Application` type.

Since 2.14
CHAPTER 4. GLIB UTILITIES

4.22. KEY-VALUE FILE PARSER

G_KEY_FILE_DESKTOP_KEY_PATH

#define G_KEY_FILE_DESKTOP_KEY_PATH "Path"

A key under G_KEY_FILE_DESKTOP_GROUP whose value is a string containing the working directory to run the program in. It is only valid for desktop entries with the Application type.
Since 2.14

G_KEY_FILE_DESKTOP_KEY_TERMINAL

#define G_KEY_FILE_DESKTOP_KEY_TERMINAL "Terminal"

A key under G_KEY_FILE_DESKTOP_GROUP whose value is a boolean stating whether the program should be run in a terminal window. It is only valid for desktop entries with the Application type.
Since 2.14

G_KEY_FILE_DESKTOP_KEY_MIME_TYPE

#define G_KEY_FILE_DESKTOP_KEY_MIME_TYPE "MimeType"

A key under G_KEY_FILE_DESKTOP_GROUP whose value is a list of strings giving the MIME types supported by this desktop entry.
Since 2.14

G_KEY_FILE_DESKTOP_KEY_CATEGORIES

#define G_KEY_FILE_DESKTOP_KEY_CATEGORIES "Categories"

A key under G_KEY_FILE_DESKTOP_GROUP whose value is a list of strings giving the categories in which the desktop entry should be shown in a menu.
Since 2.14

G_KEY_FILE_DESKTOP_KEY_STARTUP_NOTIFY

#define G_KEY_FILE_DESKTOP_KEY_STARTUP_NOTIFY "StartupNotify"

A key under G_KEY_FILE_DESKTOP_GROUP whose value is a boolean stating whether the application supports the Startup Notification Protocol Specification.
Since 2.14

G_KEY_FILE_DESKTOP_KEY_STARTUP_WM_CLASS

#define G_KEY_FILE_DESKTOP_KEY_STARTUP_WM_CLASS "StartupWMClass"

A key under G_KEY_FILE_DESKTOP_GROUP whose value is string identifying the WM class or name hint of a window that the application will create, which can be used to emulate Startup Notification with older applications.
Since 2.14

G_KEY_FILE_DESKTOP_KEY_URL

#define G_KEY_FILE_DESKTOP_KEY_URL "URL"

A key under G_KEY_FILE_DESKTOP_GROUP whose value is a string giving the URL to access. It is only valid for desktop entries with the Link type.
Since 2.14
G_KEY_FILE_DESKTOP_TYPE_APPLICATION

#define G_KEY_FILE_DESKTOP_TYPE_APPLICATION "Application"

The value of the G_KEY_FILE_DESKTOP_KEY_TYPE key for desktop entries representing applications.
Since 2.14

G_KEY_FILE_DESKTOP_TYPE_LINK

#define G_KEY_FILE_DESKTOP_TYPE_LINK "link"

The value of the G_KEY_FILE_DESKTOP_KEY_TYPE key for desktop entries representing links to documents.
Since 2.14

G_KEY_FILE_DESKTOP_TYPE_DIRECTORY

#define G_KEY_FILE_DESKTOP_TYPE_DIRECTORY "Directory"

The value of the G_KEY_FILE_DESKTOP_KEY_TYPE key for desktop entries representing directories.
Since 2.14

4.23 Bookmark file parser

Name
Bookmark file parser – parses files containing bookmarks

Synopsis

#include <glib.h>

GBookmarkFile;

#define G_BOOKMARK_FILE_ERROR
enum GBookmarkFileError;

GBookmarkFile * g_bookmark_file_new (void);
void g_bookmark_file_free (GBookmarkFile *bookmark);
gboolean g_bookmark_file_load_from_file (GBookmarkFile *bookmark,
const gchar *filename,
GError **error);
gboolean g_bookmark_file_load_from_data (GBookmarkFile *bookmark,
const gchar *data,
gsize length,
GError **error);
gboolean g_bookmark_file_load_from_data_dirs (GBookmarkFile *bookmark,
const gchar *file,
gchar **full_path,
GError **error);
gboolean g_bookmark_file_to_data (GBookmarkFile *bookmark,
gsize *length,
GError **error);
gboolean g_bookmark_file_to_file (GBookmarkFile *bookmark,
const gchar *filename,
GError **error);
gboolean g_bookmark_file_has_item (GBookmarkFile *bookmark,
CHAPTER 4. GLIB UTILITIES

4.23. BOOKMARK FILE PARSER

const gchar *uri);

gboolean g_bookmark_file_has_group (GBookmarkFile *bookmark,
const gchar *uri,
const gchar *group,
GError **error);

gboolean g_bookmark_file_has_application (GBookmarkFile *bookmark,
const gchar *uri,
const gchar *name,
GError **error);

gboolean g_bookmark_file_has_icon (GBookmarkFile *bookmark,
const gchar *uri,
gchar **href,
gchar **mime_type,
GError **error);

gint g_bookmark_file_get_size (GBookmarkFile *bookmark);

gchar ** g_bookmark_file_get_uris (GBookmarkFile *bookmark,
gsize *length);

gboolean g_bookmark_file_get_is_private (GBookmarkFile *bookmark,
const gchar *uri,
GError **error);

gboolean g_bookmark_file_get_icon (GBookmarkFile *bookmark,
const gchar *uri,
gchar **href,
gchar **mime_type,
GError **error);

gchar * g_bookmark_file_get_title (GBookmarkFile *bookmark,
const gchar *uri,
GError **error);

gchar * g_bookmark_file_get_description (GBookmarkFile *bookmark,
const gchar *uri,
GError **error);

gchar * g_bookmark_file_get_mime_type (GBookmarkFile *bookmark,
const gchar *uri,
GError **error);

gboolean g_bookmark_file_get_app_info (GBookmarkFile *bookmark,
const gchar *uri,
const gchar *name,
gchar **exec,
guint *count,
time_t *stamp,
GError **error);

void g_bookmark_file_set_title (GBookmarkFile *bookmark,
const gchar *uri,
const gchar *title);

void g_bookmark_file_set_description (GBookmarkFile *bookmark,
const gchar *uri,
const gchar *title);


420
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>g_bookmark_file_set_mime_type</code></td>
<td>Sets MIME type of the bookmark file</td>
</tr>
<tr>
<td><code>g_bookmark_file_set_is_private</code></td>
<td>Sets is private flag of the bookmark file</td>
</tr>
<tr>
<td><code>g_bookmark_file_set_icon</code></td>
<td>Sets icon of the bookmark file</td>
</tr>
<tr>
<td><code>g_bookmark_file_set_added</code></td>
<td>Sets added time of the bookmark file</td>
</tr>
<tr>
<td><code>g_bookmark_file_set_groups</code></td>
<td>Sets groups of the bookmark file</td>
</tr>
<tr>
<td><code>g_bookmark_file_set_modified</code></td>
<td>Sets modified time of the bookmark file</td>
</tr>
<tr>
<td><code>g_bookmark_file_set_visited</code></td>
<td>Sets visited time of the bookmark file</td>
</tr>
<tr>
<td><code>g_bookmark_file_set_app_info</code></td>
<td>Sets application information of the bookmark file</td>
</tr>
<tr>
<td><code>g_bookmark_file_add_group</code></td>
<td>Adds a group to the bookmark file</td>
</tr>
<tr>
<td><code>g_bookmark_file_add_application</code></td>
<td>Adds an application to the bookmark file</td>
</tr>
<tr>
<td><code>g_bookmark_file_remove_group</code></td>
<td>Removes a group from the bookmark file</td>
</tr>
<tr>
<td><code>g_bookmark_file_remove_application</code></td>
<td>Removes an application from the bookmark file</td>
</tr>
<tr>
<td><code>g_bookmark_file_remove_item</code></td>
<td>Removes an item from the bookmark file</td>
</tr>
<tr>
<td><code>g_bookmark_file_move_item</code></td>
<td>Moves an item within the bookmark file</td>
</tr>
</tbody>
</table>

**Description**

`GBookmarkFile` lets you parse, edit or create files containing bookmarks to URI, along with some meta-data about the resource pointed by the URI like its MIME type, the application that is registering the
bookmark and the icon that should be used to represent the bookmark. The data is stored using the Desktop Bookmark Specification.

The syntax of the bookmark files is described in detail inside the Desktop Bookmark Specification, here is a quick summary: bookmark files use a sub-class of the XML Bookmark Exchange Language specification, consisting of valid UTF-8 encoded XML, under the xbel root element; each bookmark is stored inside a bookmark element, using its URI: no relative paths can be used inside a bookmark file. The bookmark may have a user defined title and description, to be used instead of the URI. Under the metadata element, with its owner attribute set to http://freedesktop.org, is stored the metadata about a resource pointed by its URI. The meta-data consists of the resource’s MIME type; the applications that have registered a bookmark; the groups to which a bookmark belongs to; a visibility flag, used to set the bookmark as “private” to the applications and groups that has it registered; the URI and MIME type of an icon, to be used when displaying the bookmark inside a GUI.

```xml
<?xml version="1.0"?>
<!DOCTYPE xbel PUBLIC
"+//IDN python.org//DTD XML Bookmark Exchange Language 1.0//EN//XML"
"http://www.python.org/topics/xml/dtds/xbel-1.0.dtd">
<xbel version="1.0"
xmlns:mime="http://www.freedesktop.org/standards/shared-mime-info"
xmlns:bookmark="http://www.freedesktop.org/standards/desktop-bookmarks">
  <bookmark href="file:///home/ebassi/bookmark-spec/bookmark-spec.xml">
    <title>Desktop Bookmarks Spec</title>
    <info>
      <metadata owner="http://freedesktop.org">
        <mime:mime-type>text/xml</mime:mime-type>
        <bookmark:applications>
          <bookmark:application name="GEdit" count="2" exec="gedit %u" timestamp ← "1115726763"/>
          <bookmark:application name="GViM" count="7" exec="gvim %f" timestamp ← "1115726812"/>
        </bookmark:applications>
        <bookmark:groups>
          <bookmark:group>Editors</bookmark:group>
        </bookmark:groups>
      </metadata>
    </info>
  </bookmark>
</xbel>
```

A bookmark file might contain more than one bookmark; each bookmark is accessed through its URI.

The important caveat of bookmark files is that when you add a new bookmark you must also add the application that is registering it, using `g_bookmark_file_add_application()` or `g_bookmark_file_set_app_info()`. If a bookmark has no applications then it won’t be dumped when creating the on disk representation, using `g_bookmark_file_to_data()` or `g_bookmark_file_to_file()`.

The GBookmarkFile parser was added in GLib 2.12.

**Details**

**GBookmarkFile**

typedef struct _GBookmarkFile GBookmarkFile;

The GBookmarkFile struct contains only private data and should not be used directly.

**G_BOOKMARK_FILE_ERROR**

#define G_BOOKMARK_FILE_ERROR (g_bookmark_file_error_quark ())

Error domain for bookmark file parsing. Errors in this domain will be from the GBookmarkFileError enumeration. See GError for informations on error domains.
enum GBookmarkFileError

typedef enum
{
    G_BOOKMARK_FILE_ERROR_INVALID_URI,
    G_BOOKMARK_FILE_ERROR_INVALID_VALUE,
    G_BOOKMARK_FILE_ERROR_APP_NOT_REGISTERED,
    G_BOOKMARK_FILE_ERROR_URI_NOT_FOUND,
    G_BOOKMARK_FILE_ERROR_READ,
    G_BOOKMARK_FILE_ERROR_UNKNOWN_ENCODING,
    G_BOOKMARK_FILE_ERROR_WRITE,
    G_BOOKMARK_FILE_ERROR_FILE_NOT_FOUND
} GBookmarkFileError;

Error codes returned by bookmark file parsing.

G_BOOKMARK_FILE_ERROR_INVALID_URI URI was ill-formed
G_BOOKMARK_FILE_ERROR_INVALID_VALUE a requested field was not found
G_BOOKMARK_FILE_ERROR_APP_NOT_REGISTERED a requested application did not register a bookmark
G_BOOKMARK_FILE_ERROR_URI_NOT_FOUND a requested URI was not found
G_BOOKMARK_FILE_ERROR_READ document was ill formed
G_BOOKMARK_FILE_ERROR_UNKNOWN_ENCODING the text being parsed was in an unknown encoding
G_BOOKMARK_FILE_ERROR_WRITE an error occurred while writing
G_BOOKMARK_FILE_ERROR_FILE_NOT_FOUND requested file was not found

g_bookmark_file_new ()

GBookmarkFile * g_bookmark_file_new (void);

Creates a new empty GBookmarkFile object.
Use g_bookmark_file_load_from_file(), g_bookmark_file_load_from_data() or g_bookmark_file_load_from_data_dirs() to read an existing bookmark file.

Returns: an empty GBookmarkFile

Since 2.12

g_bookmark_file_free ()

void g_bookmark_file_free (GBookmarkFile *bookmark) ┆

Frees a GBookmarkFile.

bookmark: a GBookmarkFile

Since 2.12

g_bookmark_file_load_from_file ()

gboolean g_bookmark_file_load_from_file (GBookmarkFile *bookmark,
                                           const gchar *filename,
                                           GError **error);

Loads a desktop bookmark file into an empty GBookmarkFile structure. If the file could not be loaded then error is set to either a GFileError or GBookmarkFileError.
**bookmark**: an empty GBookmarkFile struct

**filename**: the path of a filename to load, in the GLib file name encoding

**error**: return location for a GError, or NULL

**Returns**: TRUE if a desktop bookmark file could be loaded

Since 2.12

### g_bookmark_file_load_from_data ()

```c
gboolean g_bookmark_file_load_from_data (GBookmarkFile *bookmark, const gchar *data, gsize length, GError **error);
```

Loads a bookmark file from memory into an empty GBookmarkFile structure. If the object cannot be created then error is set to a GBookmarkFileError.

**bookmark**: an empty GBookmarkFile struct

**data**: desktop bookmarks loaded in memory

**length**: the length of data in bytes

**error**: return location for a GError, or NULL

**Returns**: TRUE if a desktop bookmark could be loaded.

Since 2.12

### g_bookmark_file_load_from_data_dirs ()

```c
gboolean g_bookmark_file_load_from_data_dirs (GBookmarkFile *bookmark, const gchar *file, gchar **full_path, GError **error);
```

This function looks for a desktop bookmark file named file in the paths returned from g_get_user_data_dir() and g_get_system_data_dirs(), loads the file into bookmark and returns the file's full path in full_path. If the file could not be loaded then an error is set to either a GFileError or GBookmarkFileError.

**bookmark**: a GBookmarkFile

**file**: a relative path to a filename to open and parse

**full_path**: return location for a string containing the full path of the file, or NULL

**error**: return location for a GError, or NULL

**Returns**: TRUE if a key file could be loaded, FALSE otherwise

Since 2.12

### g_bookmark_file_to_data ()

```c
gchar * g_bookmark_file_to_data (GBookmarkFile *bookmark, gsize *length, GError **error);
```

This function outputs bookmark as a string.

**bookmark**: a GBookmarkFile

**length**: return location for the length of the returned string, or NULL

**error**: return location for a GError, or NULL

**Returns**: a newly allocated string holding the contents of the GBookmarkFile

Since 2.12
g_bookmark_file_to_file()

gboolean g_bookmark_file_to_file (GBookmarkFile *bookmark, const gchar *filename, GError **error);

This function outputs \textit{bookmark} into a file. The write process is guaranteed to be atomic by using \texttt{g_file_set_contents()} internally.

\textit{bookmark}: a GBookmarkFile

\textit{filename}: path of the output file

\textit{error}: return location for a GError, or NULL

\textit{Returns}: TRUE if the file was successfully written.

Since 2.12

g_bookmark_file_has_item()

gboolean g_bookmark_file_has_item (GBookmarkFile *bookmark, const gchar *uri);

Looks whether the desktop bookmark has an item with its URI set to \textit{uri}.

\textit{bookmark}: a GBookmarkFile

\textit{uri}: a valid URI

\textit{Returns}: TRUE if \textit{uri} is inside \textit{bookmark}, FALSE otherwise

Since 2.12

g_bookmark_file_has_group()

gboolean g_bookmark_file_has_group (GBookmarkFile *bookmark, const gchar *uri, const gchar *group, GError **error);

Checks whether \textit{group} appears in the list of groups to which the bookmark for \textit{uri} belongs to. In the event the URI cannot be found, FALSE is returned and \textit{error} is set to G_BOOKMARK_FILE_ERROR_URI_NOT_FOUND.

\textit{bookmark}: a GBookmarkFile

\textit{uri}: a valid URI

\textit{group}: the group name to be searched

\textit{error}: return location for a GError, or NULL

\textit{Returns}: TRUE if \textit{group} was found.

Since 2.12
## g_bookmark_file_has_application()

```
gboolean g_bookmark_file_has_application (GBookmarkFile *bookmark,  
    const gchar *uri,  
    const gchar *name,  
    GError **error);```

Checks whether the bookmark for `uri` inside `bookmark` has been registered by application `name`. In the event the URI cannot be found, `FALSE` is returned and `error` is set to `G_BOOKMARK_FILE_ERROR_URI_NOT_FOUND`.

- **bookmark**: a `GBookmarkFile`
- **uri**: a valid URI
- **name**: the name of the application
- **error**: return location for a `GError` or `NULL`

**Returns**: `TRUE` if the application `name` was found.

Since 2.12

## g_bookmark_file_get_size()

```
gint g_bookmark_file_get_size (GBookmarkFile *bookmark)```

Gets the number of bookmarks inside `bookmark`.

- **bookmark**: a `GBookmarkFile`

**Returns**: the number of bookmarks.

Since 2.12

## g_bookmark_file_get_uris()

```
gchar ** g_bookmark_file_get_uris (GBookmarkFile *bookmark,  
    gsize *length);```

Returns all URIs of the bookmarks in the bookmark file `bookmark`. The array of returned URIs will be `NULL`-terminated, so `length` may optionally be `NULL`.

- **bookmark**: a `GBookmarkFile`
- **length**: return location for the number of returned URIs, or `NULL`

**Returns**: a newly allocated `NULL`-terminated array of strings. Use `g_strfreev()` to free it.

Since 2.12

## g_bookmark_file_get_title()

```
gchar * g_bookmark_file_get_title (GBookmarkFile *bookmark,  
    const gchar *uri,  
    GError **error);```

Returns the title of the bookmark for `uri`. If `uri` is `NULL`, the title of `bookmark` is returned. In the event the URI cannot be found, `NULL` is returned and `error` is set to `G_BOOKMARK_FILE_ERROR_URI_NOT_FOUND`.

- **bookmark**: a `GBookmarkFile`
- **uri**: a valid URI or `NULL`
- **error**: return location for a `GError`, or `NULL`

**Returns**: a newly allocated string or `NULL` if the specified URI cannot be found.

Since 2.12
g_bookmark_file_get_description ()

| gchar * | g_bookmark_file_get_description | (GBookmarkFile *bookmark, const gchar *uri, GError **error); |

Retrieves the description of the bookmark for uri. In the event the URI cannot be found, NULL is returned and error is set to G_BOOKMARK_FILE_ERROR_URI_NOT_FOUND.

bookmark: a GBookmarkFile
uri: a valid URI
error: return location for a GError, or NULL

Returns: a newly allocated string or NULL if the specified URI cannot be found.

Since 2.12

g_bookmark_file_get_mime_type ()

| gchar * | g_bookmark_file_get_mime_type | (GBookmarkFile *bookmark, const gchar *uri, GError **error); |

Retrieves the MIME type of the resource pointed by uri. In the event the URI cannot be found, NULL is returned and error is set to G_BOOKMARK_FILE_ERROR_URI_NOT_FOUND. In the event that the MIME type cannot be found, NULL is returned and error is set to G_BOOKMARK_FILE_ERROR_INVALID_VALUE.

bookmark: a GBookmarkFile
uri: a valid URI
error: return location for a GError, or NULL

Returns: a newly allocated string or NULL if the specified URI cannot be found.

Since 2.12

g_bookmark_file_get_is_private ()

| gboolean | g_bookmark_file_get_is_private | (GBookmarkFile *bookmark, const gchar *uri, GError **error); |

Gets whether the private flag of the bookmark for uri is set. In the event the URI cannot be found, FALSE is returned and error is set to G_BOOKMARK_FILE_ERROR_URI_NOT_FOUND. In the event that the private flag cannot be found, FALSE is returned and error is set to G_BOOKMARK_FILE_ERROR_INVALID_VALUE.

bookmark: a GBookmarkFile
uri: a valid URI
error: return location for a GError, or NULL

Returns: TRUE if the private flag is set, FALSE otherwise.

Since 2.12
CHAPTER 4. GLIB UTILITIES

4.23. BOOKMARK FILE PARSER

```
gboolean g_bookmark_file_get_icon (GBookmarkFile *bookmark,
                                   const gchar *uri,
                                   gchar **href,
                                   gchar **mime_type,
                                   GError **error);
```

Gets the icon of the bookmark for `uri`. In the event the URI cannot be found, `FALSE` is returned and `error` is set to `G_BOOKMARK_FILE_ERROR_URI_NOT_FOUND`.

- `bookmark`: a `GBookmarkFile`
- `uri`: a valid URI
- `href`: return location for the icon’s location or `NULL`
- `mime_type`: return location for the icon’s MIME type or `NULL`
- `error`: return location for a `GError` or `NULL`

**Returns**: `TRUE` if the icon for the bookmark for the URI was found. You should free the returned strings.

Since 2.12

```
time_t g_bookmark_file_get_added (GBookmarkFile *bookmark,
                                   const gchar *uri,
                                   GError **error);
```

Gets the time the bookmark for `uri` was added to `bookmark`. In the event the URI cannot be found, `-1` is returned and `error` is set to `G_BOOKMARK_FILE_ERROR_URI_NOT_FOUND`.

- `bookmark`: a `GBookmarkFile`
- `uri`: a valid URI
- `error`: return location for a `GError`, or `NULL`

**Returns**: a `timestamp`

Since 2.12

```
time_t g_bookmark_file_get_modified (GBookmarkFile *bookmark,
                                     const gchar *uri,
                                     GError **error);
```

Gets the time when the bookmark for `uri` was last modified. In the event the URI cannot be found, `-1` is returned and `error` is set to `G_BOOKMARK_FILE_ERROR_URI_NOT_FOUND`.

- `bookmark`: a `GBookmarkFile`
- `uri`: a valid URI
- `error`: return location for a `GError`, or `NULL`

**Returns**: a `timestamp`

Since 2.12
CHAPTER 4. GLIB UTILITIES

4.23. BOOKMARK FILE PARSER

\textbf{g\_bookmark\_file\_get\_visited ()}

\begin{verbatim}
    gchar ** g_bookmark_file_get_groups (GBookmarkFile *bookmark,
                                           const gchar *uri,
                                           gsize *length,
                                           GError **error);
\end{verbatim}

Gets the time the bookmark for \texttt{uri} was last visited. In the event the URI cannot be found, -1 is returned and \texttt{error} is set to \texttt{G_BOOKMARK_FILE_ERROR_URI_NOT_FOUND}.

\textbf{bookmark} : a \texttt{GBookmarkFile}

\textbf{uri} : a valid URI

\textbf{error} : return location for a \texttt{GError}, or NULL

\textbf{Returns} : a timestamp.

Since 2.12

\textbf{g\_bookmark\_file\_get\_groups ()}

\begin{verbatim}
    gchar ** g_bookmark_file_get_applications (GBookmarkFile *bookmark,
                                                const gchar *uri,
                                                gsize *length,
                                                GError **error);
\end{verbatim}

Retrieves the list of group names of the bookmark for \texttt{uri}. In the event the URI cannot be found, \texttt{NULL} is returned and \texttt{error} is set to \texttt{G_BOOKMARK_FILE_ERROR_URI_NOT_FOUND}. The returned array is \texttt{NULL} terminated, so \texttt{length} may optionally be \texttt{NULL}.

\textbf{bookmark} : a \texttt{GBookmarkFile}

\textbf{uri} : a valid URI

\textbf{length} : return location for the length of the returned string, or \texttt{NULL}

\textbf{error} : return location for a \texttt{GError}, or NULL

\textbf{Returns} : a newly allocated \texttt{NULL}-terminated array of group names. Use \texttt{g\_strfreev() \_to \_free \_it}.

Since 2.12

\textbf{g\_bookmark\_file\_get\_applications ()}

\begin{verbatim}
    gchar ** g_bookmark_file_get_applications (GBookmarkFile *bookmark,
                                                const gchar *uri,
                                                gsize *length,
                                                GError **error);
\end{verbatim}

Retrieves the names of the applications that have registered the bookmark for \texttt{uri}. In the event the URI cannot be found, \texttt{NULL} is returned and \texttt{error} is set to \texttt{G_BOOKMARK_FILE_ERROR_URI_NOT_FOUND}.

\textbf{bookmark} : a \texttt{GBookmarkFile}

\textbf{uri} : a valid URI

\textbf{length} : return location of the length of the returned list, or \texttt{NULL}

\textbf{error} : return location for a \texttt{GError}, or NULL

\textbf{Returns} : a newly allocated \texttt{NULL}-terminated array of strings. Use \texttt{g\_strfreev() \_to \_free \_it}.

Since 2.12

\footnotesize{429}
4.23. BOOKMARK FILE PARSER

**g_bookmark_file_get_app_info ()**

```c
gboolean g_bookmark_file_get_app_info (GBookmarkFile *bookmark,
const gchar *uri,
const gchar *name,
gchar **exec,
guint *count,
time_t *stamp,
GError **error);
```

Gets the registration information of `app_name` for the bookmark for `uri`. See `g_bookmark_file_set_app_info()` for more information about the returned data.

The string returned in `app_exec` must be freed.

In the event the URI cannot be found, FALSE is returned and `error` is set to `G_BOOKMARK_FILE_ERROR_URI_NOT_FOUND`.

In the event that no application with name `app_name` has registered a bookmark for `uri`, FALSE is returned and `error` is set to `G_BOOKMARK_FILE_ERROR_APP_NOT_REGISTERED`. In the event that unquoting the command line fails, an error of the `G_SHELL_ERROR` domain is set and FALSE is returned.

**bookmark**: a `GBookmarkFile`

**uri**: a valid URI

**name**: an application’s name

**exec**: location for the command line of the application, or NULL

**count**: return location for the registration count, or NULL

**stamp**: return location for the last registration time, or NULL

**error**: return location for a `GError`, or NULL

**Returns**: TRUE on success.

Since 2.12

**g_bookmark_file_set_title ()**

```c
void g_bookmark_file_set_title (GBookmarkFile *bookmark,
const gchar *uri,
const gchar *title);
```

Sets `title` as the title of the bookmark for `uri` inside the bookmark file `bookmark`.

If `uri` is NULL, the title of `bookmark` is set.

If a bookmark for `uri` cannot be found then it is created.

**bookmark**: a `GBookmarkFile`

**uri**: a valid URI or NULL

**title**: a UTF-8 encoded string

Since 2.12

**g_bookmark_file_set_description ()**

```c
void g_bookmark_file_set_description (GBookmarkFile *bookmark,
const gchar *uri,
const gchar *description);
```

Sets `description` as the description of the bookmark for `uri`.

If `uri` is NULL, the description of `bookmark` is set.

If a bookmark for `uri` cannot be found then it is created.

Since 2.12
bookmark: a GBookmarkFile

uri: a valid URI or NULL

description: a string

Since 2.12

g_bookmark_file_set_mime_type ()

void g_bookmark_file_set_mime_type (GBookmarkFile *bookmark,
const gchar *uri,
const gchar *mime_type);

Sets mime_type as the MIME type of the bookmark for uri. If a bookmark for uri cannot be found then it is created.

bookmark: a GBookmarkFile

uri: a valid URI

mime_type: a MIME type

Since 2.12

g_bookmark_file_set_is_private ()

void g_bookmark_file_set_is_private (GBookmarkFile *bookmark,
const gchar *uri,
gboolean is_private);

Sets the private flag of the bookmark for uri. If a bookmark for uri cannot be found then it is created.

bookmark: a GBookmarkFile

uri: a valid URI

is_private: TRUE if the bookmark should be marked as private

Since 2.12

g_bookmark_file_set_icon ()

void g_bookmark_file_set_icon (GBookmarkFile *bookmark,
const gchar *uri,
const gchar *href,
const gchar *mime_type);

Sets the icon for the bookmark for uri. If href is NULL, unsets the currently set icon. href can either be a full URL for the icon file or the icon name following the Icon Naming specification. If no bookmark for uri is found one is created.

bookmark: a GBookmarkFile

uri: a valid URI

href: the URI of the icon for the bookmark, or NULL

mime_type: the MIME type of the icon for the bookmark

Since 2.12
CHAPTER 4. GLIB UTILITIES 4.23. BOOKMARK FILE PARSER

**g_bookmark_file_set_added()**

```c
void g_bookmark_file_set_added (GBookmarkFile *bookmark,
                              const gchar *uri,
                              time_t added);
```

Sets the time the bookmark for `uri` was added into `bookmark`. If no bookmark for `uri` is found then it is created.

*bookmark*: a `GBookmarkFile`

*uri*: a valid URI

*added*: a timestamp or -1 to use the current time

Since 2.12

**g_bookmark_file_set_groups()**

```c
void g_bookmark_file_set_groups (GBookmarkFile *bookmark,
                                const gchar *uri,
                                const gchar **groups,
                                gsize length);
```

Sets a list of group names for the item with URI `uri`. Each previously set group name list is removed. If `uri` cannot be found then an item for it is created.

*bookmark*: a `GBookmarkFile`

*uri*: an item’s URI

*groups*: an array of group names, or NULL to remove all groups

*length*: number of group name values in `groups`

Since 2.12

**g_bookmark_file_set_modified()**

```c
void g_bookmark_file_set_modified (GBookmarkFile *bookmark,
                                  const gchar *uri,
                                  time_t modified);
```

Sets the last time the bookmark for `uri` was last modified. If no bookmark for `uri` is found then it is created.

The "modified" time should only be set when the bookmark’s meta-data was actually changed. Every function of `GBookmarkFile` that modifies a bookmark also changes the modification time, except for `g_bookmark_file_set_visited()`.

*bookmark*: a `GBookmarkFile`

*uri*: a valid URI

*modified*: a timestamp or -1 to use the current time

Since 2.12
**g_bookmark_file_set_visited()**

```c
void g_bookmark_file_set_visited (GBookmarkFile *bookmark,
                                 const gchar *uri,
                                 time_t visited);
```

Sets the time the bookmark for `uri` was last visited. If no bookmark for `uri` is found then it is created.

The "visited" time should only be set if the bookmark was launched, either using the command line retrieved by `g_bookmark_file_get_app_info()` or by the default application for the bookmark’s MIME type, retrieved using `g_bookmark_file_get_mime_type()`. Changing the "visited" time does not affect the "modified" time.

- **bookmark**: a GBookmarkFile
- **uri**: a valid URI
- **visited**: a timestamp or -1 to use the current time
  
  Since 2.12

**g_bookmark_file_set_app_info()**

```c
gboolean g_bookmark_file_set_app_info (GBookmarkFile *bookmark,
                                       const gchar *uri,
                                       const gchar *name,
                                       const gchar *exec,
                                       gint count,
                                       time_t stamp,
                                       GError **error);
```

Sets the meta-data of application `name` inside the list of applications that have registered a bookmark for `uri` inside `bookmark`.

You should rarely use this function; use `g_bookmark_file_add_application()` and `g_bookmark_file_remove_application()` instead.

- **name** can be any UTF-8 encoded string used to identify an application. `exec` can have one of these two modifiers: "f", which will be expanded as the local file name retrieved from the bookmark’s URI; "u", which will be expanded as the bookmark’s URI. The expansion is done automatically when retrieving the stored command line using the `g_bookmark_file_get_app_info()` function. `count` is the number of times the application has registered the bookmark; if is < 0, the current registration count will be increased by one, if is 0, the application with `name` will be removed from the list of registered applications. `stamp` is the Unix time of the last registration; if it is -1, the current time will be used.

  If you try to remove an application by setting its registration count to zero, and no bookmark for `uri` is found, FALSE is returned and `error` is set to `G_BOOKMARK_FILE_ERROR_URI_NOT_FOUND`; similarly, in the event that no application `name` has registered a bookmark for `uri`, FALSE is returned and `error` is set to `G_BOOKMARK_FILE_ERROR_APP_NOT_REGISTERED`. Otherwise, if no bookmark for `uri` is found, one is created.

- **bookmark**: a GBookmarkFile
- **uri**: a valid URI
- **name**: an application’s name
- **exec**: an application’s command line
- **count**: the number of registrations done for this application
- **stamp**: the time of the last registration for this application
- **error**: return location for a GError or NULL

**Returns**: TRUE if the application’s meta-data was successfully changed.

Since 2.12
CHAPTER 4. GLIB UTILITIES

4.23. BOOKMARK FILE PARSER

**g_bookmark_file_add_group ()**

```c
void g_bookmark_file_add_group (GBookmarkFile *bookmark,
                              const gchar *uri,
                              const gchar *group);
```

Adds group to the list of groups to which the bookmark for uri belongs to.
If no bookmark for uri is found then it is created.

**bookmark**: a GBookmarkFile

**uri**: a valid URI

**group**: the group name to be added

Since 2.12

**g_bookmark_file_add_application ()**

```c
void g_bookmark_file_add_application (GBookmarkFile *bookmark,
                                      const gchar *uri,
                                      const gchar *name,
                                      const gchar *exec);
```

Adds the application with name and exec to the list of applications that have registered a bookmark for uri into bookmark.

Every bookmark inside a GBookmarkFile must have at least an application registered. Each application must provide a name, a command line useful for launching the bookmark, the number of times the bookmark has been registered by the application and the last time the application registered this bookmark.

If name is NULL, the name of the application will be the same returned by g_get_application_name(); if exec is NULL, the command line will be a composition of the program name as returned by g_get_prgname() and the "u" modifier, which will be expanded to the bookmark's URI.

This function will automatically take care of updating the registrations count and timestamping in case an application with the same name had already registered a bookmark for uri inside bookmark.

If no bookmark for uri is found, one is created.

**bookmark**: a GBookmarkFile

**uri**: a valid URI

**name**: the name of the application registering the bookmark or NULL

**exec**: command line to be used to launch the bookmark or NULL

Since 2.12

**g_bookmark_file_remove_group ()**

```c
gboolean g_bookmark_file_remove_group (GBookmarkFile *bookmark,
                                       const gchar *uri,
                                       const gchar *group,
                                       GError **error);
```

Removes group from the list of groups to which the bookmark for uri belongs to.
In the event the URI cannot be found, FALSE is returned and error is set to G_BOOKMARK_FILE_ERROR_URI_NOT_FOUND.
In the event no group was defined, FALSE is returned and error is set to G_BOOKMARK_FILE_ERROR_INVALID_VALUE.

**bookmark**: a GBookmarkFile

**uri**: a valid URI

**group**: the group name to be removed

**error**: return location for a GError, or NULL

**Returns**: TRUE if group was successfully removed.

Since 2.12
g_bookmark_file_remove_application ()

gboolean  g_bookmark_file_remove_application (GBookmarkFile *bookmark,
 const gchar *uri,
 const gchar *name,
 GError **error);

Removes application registered with name from the list of applications that have registered a bookmark for uri inside bookmark.

In the event the URI cannot be found, FALSE is returned and error is set to G_BOOKMARK_FILE_ERROR_URI_NOT_FOUND.

In the event that no application with name app_name has registered a bookmark for uri, FALSE is returned and error is set to G_BOOKMARK_FILE_ERROR_APP_NOT_REGISTERED.

bookmark: a GBookmarkFile
uri: a valid URI
name: the name of the application
error: return location for a GError or NULL

Returns: TRUE if the application was successfully removed.

Since 2.12

g_bookmark_file_remove_item ()

gboolean  g_bookmark_file_remove_item (GBookmarkFile *bookmark,
 const gchar *uri,
 GError **error);

Removes the bookmark for uri from the bookmark file bookmark.

bookmark: a GBookmarkFile
uri: a valid URI
error: return location for a GError, or NULL

Returns: TRUE if the bookmark was removed successfully.

Since 2.12

g_bookmark_file_move_item ()

gboolean  g_bookmark_file_move_item (GBookmarkFile *bookmark,
 const gchar *old_uri,
 const gchar *new_uri,
 GError **error);

Changes the URI of a bookmark item from old_uri to new_uri. Any existing bookmark for new_uri will be overwritten. If new_uri is NULL, then the bookmark is removed.

In the event the URI cannot be found, FALSE is returned and error is set to G_BOOKMARK_FILE_ERROR_URI_NOT_FOUND.

bookmark: a GBookmarkFile
old_uri: a valid URI
new_uri: a valid URI, or NULL
error: return location for a GError or NULL

Returns: TRUE if the URI was successfully changed.

Since 2.12
4.24 Testing

Name
Testing – a test framework

Synopsis

```c
#include <glib.h>

void g_test_minimized_result (double minimized_quantity, const char *format, ...);
void g_test_maximized_result (double maximized_quantity, const char *format, ...);
void g_test_init (int *argc, char ***argv, ...);
#define g_test_quick ()
#define g_test_slow ()
#define g_test_thorough ()
#define g_test_perf ()
#define g_test_verbose ()
#define g_test_quiet ()
int g_test_run (void);
void g_test_add_func (const char *testpath, void (test_funcvoid) ());
void g_test_add_data_func (const char *testpath, gconstpointer test_data, void (test_funcgconstpointer) ());
#define g_test_add (testpath, Fixture, tdata, fsetup, ftest, fteardown)
void g_test_message (const char *format, ...);
void g_test_bug_base (const char *uri_pattern);
void g_test_bug (const char *bug_uri_snippet);
void g_test_timer_start (void);
double g_test_timer_elapsed (void);
double g_test_timer_last (void);
void g_test_queue_free (gpointer gfree_pointer);
void g_test_queue_destroy (GDestroyNotify destroy_func, gpointer destroy_data);
#define g_test_queue_unref (gobject)
enum GTestTrapFlags;
gboolean g_test_trap_fork (guint64 usec_timeout, GTestTrapFlags test_trap_flags);
gboolean g_test_trap_has_passed (void);
gboolean g_test_trap_reached_timeout (void);
#define g_test_trap_assert_passed ()
#define g_test_trap_assert_failed ()
#define g_test_trap_assert_stdout (soutpattern)
#define g_test_trap_assert_stdout_unmatched (soutpattern)
#define g_test_trap_assert_stderr (serrpattern)
#define g_test_trap_assert_stderr_unmatched (serrpattern)
#define g_test_rand_bit ()
gint32 g_test_rand_int (void);
gint32 g_test_rand_int_range (gint32 begin, gint32 end);
```

436
### Description

GLib provides a framework for writing and maintaining unit tests in parallel to the code they are testing. The API is designed according to established concepts found in the other test frameworks (JUnit, NUnit, RUnit), which in turn is based on smalltalk unit testing concepts.

**Test case**  Tests (test methods) are grouped together with their fixture into test cases.

**Fixture**  A test fixture consists of fixture data and setup and teardown methods to establish the environment for the test functions. We use fresh fixtures, i.e. fixtures are newly set up and torn down around each test invocation to avoid dependencies between tests.

**Test suite**  Test cases can be grouped into test suites, to allow subsets of the available tests to be run. Test suites can be grouped into other test suites as well.

The API is designed to handle creation and registration of test suites and test cases implicitly. A simple call like

```c
    g_test_add_func ("/misc/assertions", test_assertions);
```

creates a test suite called "misc" with a single test case named "assertions", which consists of running the test_assertions function.

In addition to the traditional `g_assert()`, the test framework provides an extended set of assertions for string and numerical comparisons: `g_assert_cmpfloat()`, `g_assert_cmpint()`, `g_assert_cmpuint()`, `g_assert_cmphex()`, `g_assert_cmpstr()`. The advantage of these variants over plain `g_assert()` is that the assertion messages can be more elaborate, and include the values of the compared entities.

GLib ships with two utilities called gtester and gtester-report to facilitate running tests and producing nicely formatted test reports.

### Details

`g_test_minimized_result()`
Report the result of a performance or measurement test. The test should generally strive to minimize the reported quantities (smaller values are better than larger ones), this and \texttt{minimized\_quantity} can determine sorting order for test result reports.

\texttt{minimized\_quantity} : the reported value

\texttt{format} : the format string of the report message

\ldots : arguments to pass to the \texttt{printf()} function

Since 2.16

\texttt{g\_test\_maximized\_result} ()

Report the result of a performance or measurement test. The test should generally strive to maximize the reported quantities (larger values are better than smaller ones), this and \texttt{maximized\_quantity} can determine sorting order for test result reports.

\texttt{maximized\_quantity} : the reported value

\texttt{format} : the format string of the report message

\ldots : arguments to pass to the \texttt{printf()} function

Since 2.16

\texttt{g\_test\_init} ()

Initialize the GLib testing framework, e.g. by seeding the test random number generator, the name for \texttt{g\_get\_prgname()} and parsing test related command line args. So far, the following arguments are understood:

\texttt{-l} list test cases available in a test executable.

\texttt{--seed=RANDOMSEED} provide a random seed to reproduce test runs using random numbers.

\texttt{--verbose} run tests verbosely.

\texttt{--quiet} run tests quietly.

\texttt{-p TESTPATH} execute all tests matching \texttt{TESTPATH}.

\texttt{-m \{perf|slow|thorough|quick\}} execute tests according to these test modes:

\begin{itemize}
  \item \texttt{perf} performance tests, may take long and report results.
  \item \texttt{slow, thorough} slow and thorough tests, may take quite long and maximize coverage.
  \item \texttt{quick} quick tests, should run really quickly and give good coverage.
\end{itemize}

\texttt{--debug-log} debug test logging output.
-k, --keep-going  gtester-specific argument.
--GTestLogFD N  gtester-specific argument.
--GTestSkipCount N  gtester-specific argument.

**argc**: Address of the `argc` parameter of the `main()` function. Changed if any arguments were handled.

**argv**: Address of the `argv` parameter of `main()`. Any parameters understood by `g_test_init()` stripped before return.

...: Reserved for future extension. Currently, you must pass `NULL`.

Since 2.16

**g_test_quick()**

```c
#define g_test_quick()
```

Returns `TRUE` if tests are run in quick mode.

**g_test_slow()**

```c
#define g_test_slow()
```

Returns `TRUE` if tests are run in slow mode.

**g_test_thorough()**

```c
#define g_test_thorough()
```

Returns `TRUE` if tests are run in thorough mode.

**g_test_perf()**

```c
#define g_test_perf()
```

Returns `TRUE` if tests are run in performance mode.

**g_test_verbose()**

```c
#define g_test_verbose()
```

Returns `TRUE` if tests are run in verbose mode.

**g_test_quiet()**

```c
#define g_test_quiet()
```

Returns `TRUE` if tests are run in quiet mode.

**g_test_run()**

```c
int g_test_run (void);
```

Runs all tests under the toplevel suite which can be retrieved with `g_test_get_root()`. Similar to `g_test_run_suite()`, the test cases to be run are filtered according to test path arguments (`-p testpath`) as parsed by `g_test_init()`. `g_test_run_suite()` or `g_test_run()` may only be called once in a program.

**Returns**: 0 on success

Since 2.16
CHAPTER 4. GLIB UTILITIES  4.24. TESTING

\textbf{g_test_add_func}()

\begin{verbatim}
void g_test_add_func (const char *testpath,
    void (*test_func)(void) () ) ;
\end{verbatim}

Create a new test case, similar to \texttt{g_test_create_case()}. However the test is assumed to use no fixture, and test suites are automatically created on the fly and added to the root fixture, based on the slash-separated portions of \texttt{testpath}.

\textit{testpath}: Slash-separated test case path name for the test.

\textit{test_func}: The test function to invoke for this test.

Since 2.16

\textbf{g_test_add_data_func}()

\begin{verbatim}
void g_test_add_data_func (const char *testpath,
    gpointer test_data,
    gpointer (*test_func)(gpointer) () ) ;
\end{verbatim}

Create a new test case, similar to \texttt{g_test_create_case()}. However the test is assumed to use no fixture, and test suites are automatically created on the fly and added to the root fixture, based on the slash-separated portions of \texttt{testpath}. The \texttt{test_data} argument will be passed as first argument to \texttt{test_func}.

\textit{testpath}: Slash-separated test case path name for the test.

\textit{test_data}: Test data argument for the test function.

\textit{test_func}: The test function to invoke for this test.

Since 2.16

\textbf{g_test_add()}\n
\begin{verbatim}
#define g_test_add(testpath, Fixture, tdata, fsetup, ftest, fteardown)
#  g_test_add(testpath, Fixture, tdata, fsetup, ftest, fteardown)
\end{verbatim}

Hook up a new test case at \texttt{testpath}, similar to \texttt{g_test_add_func()}. A fixture data structure with setup and teardown function may be provided though, similar to \texttt{g_test_create_case()}. \texttt{g_test_add()} is implemented as a macro, so that the \texttt{fsetup()}, \texttt{ftest()} and \texttt{fteardown()} callbacks can expect a \texttt{Fixture} pointer as first argument in a type safe manner.

\textit{testpath}: The test path for a new test case.

\textit{Fixture}: The type of a fixture data structure.

\textit{tdata}: Data argument for the test functions.

\textit{fsetup}: The function to set up the fixture data.

\textit{ftest}: The actual test function.

\textit{fteardown}: The function to tear down the fixture data.

Since 2.16
**g_test_message()**

```c
void g_test_message (const char *format, ...);
```

Add a message to the test report.

*format*: the format string

*...*: printf-like arguments to *format*

Since 2.16

**g_test_bug_base()**

```c
void g_test_bug_base (const char *uri_pattern);
```

Specify the base URI for bug reports.

The base URI is used to construct bug report messages for `g_test_message()` when `g_test_bug()` is called. Calling this function outside of a test case sets the default base URI for all test cases. Calling it from within a test case changes the base URI for the scope of the test case only. Bug URIs are constructed by appending a bug specific URI portion to *uri_pattern*, or by replacing the special string ‘s’ within *uri_pattern* if that is present.

*uri_pattern*: the base pattern for bug URIs

Since 2.16

**g_test_bug()**

```c
void g_test_bug (const char *bug_uri_snippet);
```

This function adds a message to test reports that associates a bug URI with a test case. Bug URIs are constructed from a base URI set with `g_test_bug_base()` and *bug_uri_snippet*.

*bug_uri_snippet*: Bug specific bug tracker URI portion.

Since 2.16

**g_test_timer_start()**

```c
void g_test_timer_start (void);
```

Start a timing test. Call `g_test_timer_elapsed()` when the task is supposed to be done. Call this function again to restart the timer.

Since 2.16

**g_test_timer_elapsed()**

```c
double g_test_timer_elapsed (void);
```

Get the time since the last start of the timer with `g_test_timer_start()`.

*Returns*: the time since the last start of the timer, as a double

Since 2.16
CHAPTER 4. GLIB UTILITIES

4.24. TESTING

**g_test_timer_last ()**

```c
double g_test_timer_last (void);
```

Report the last result of `g_test_timer_elapsed()`.

**Returns :** the last result of `g_test_timer_elapsed()`, as a double

Since 2.16

**g_test_queue_free ()**

```c
void g_test_queue_free (gpointer gfree_pointer);
```

Enqueue a pointer to be released with `g_free()` during the next teardown phase. This is equivalent to calling `g_test_queue_destroy()` with a destroy callback of `g_free()`.

**gfree_pointer :** the pointer to be stored.

Since 2.16

**g_test_queue_destroy ()**

```c
void g_test_queue_destroy (GDestroyNotify destroy_func,
                         gpointer destroy_data);
```

This function enqueues a callback @destroy_func() to be executed during the next test case teardown phase. This is most useful to auto destruct allocated test resources at the end of a test run. Resources are released in reverse queue order, that means enqueuing callback A before callback B will cause B() to be called before A() during teardown.

**destroy_func :** Destroy callback for teardown phase.

**destroy_data :** Destroy callback data.

Since 2.16

**g_test_queue_unref()**

```c
#define g_test_queue_unref(gobject)
```

Enqueue an object to be released with `g_object_unref()` during the next teardown phase. This is equivalent to calling `g_test_queue_destroy()` with a destroy callback of `g_object_unref()`.

**gobject :** the object to unref

Since 2.16

**enum GTestTrapFlags**

```c
typedef enum {
    G_TEST_TRAP_SILENCE_STDOUT = 1 << 7,
    G_TEST_TRAP_SILENCE_STDERR = 1 << 8,
    G_TEST_TRAP_INHERIT_STDIN = 1 << 9
} GTestTrapFlags;
```

Test traps are guards around forked tests. These flags determine what traps to set.

**G_TEST_TRAP_SILENCE_STDOUT** Redirect stdout of the test child to `/dev/null` so it cannot be observed on the console during test runs. The actual output is still captured though to allow later tests with `g_test_trap_assert_stdout()`.
GTEST_TRAP_SILENCE_STDERR Redirect stderr of the test child to /dev/null so it cannot be observed on the console during test runs. The actual output is still captured though to allow later tests with g_test_trap_assert_stderr().

GTEST_TRAP_INHERIT_STDIN If this flag is given, stdin of the forked child process is shared with stdin of its parent process. It is redirected to /dev/null otherwise.

g_test_trap_fork()

gboolean g_test_trap_fork (guint64 usec_timeout, GTestTrapFlags test_trap_flags);

Fork the current test program to execute a test case that might not return or that might abort. The forked test case is aborted and considered failing if its run time exceeds usec_timeout.

The forking behavior can be configured with the GTestTrapFlags flags.

In the following example, the test code forks, the forked child process produces some sample output and exits successfully. The forking parent process then asserts successful child program termination and validates child program outputs.

```c
static void
  test_fork_patterns (void)
{
  if (g_test_trap_fork (0, G_TEST_TRAP_SILENCE_STDOUT | G_TEST_TRAP_SILENCE_STDERR))
  {
    g_print ("some stdout text: somagic17\n");
    g_printerr ("some stderr text: semagic43\n");
    exit (0); /* successful test run */
  }
  g_test_trap_assert_passed();
  g_test_trap_assert_stdout ("*somagic17*");
  g_test_trap_assert_stderr ("*semagic43*");
}
```

This function is implemented only on Unix platforms.

usec_timeout: Timeout for the forked test in micro seconds.

test_trap_flags: Flags to modify forking behaviour.

Returns: TRUE for the forked child and FALSE for the executing parent process.

Since 2.16

g_test_trap_has_passed()

gboolean g_test_trap_has_passed (void);

Check the result of the last g_test_trap_fork() call.

Returns: TRUE if the last forked child terminated successfully.

Since 2.16

g_test_trap_reached_timeout()

gboolean g_test_trap_reached_timeout (void);

Check the result of the last g_test_trap_fork() call.

Returns: TRUE if the last forked child got killed due to a fork timeout.

Since 2.16


**g_test_trap_assert_passed()**

```c
#define g_test_trap_assert_passed()
```

Assert that the last forked test passed. See `g_test_trap_fork()`. Since 2.16

**g_test_trap_assert_failed()**

```c
#define g_test_trap_assert_failed()
```

Assert that the last forked test failed. See `g_test_trap_fork()`. Since 2.16

**g_test_trap_assert_stdout()**

```c
#define g_test_trap_assert_stdout(soutpattern)
```

Assert that the stdout output of the last forked test matches `soutpattern`. See `g_test_trap_fork()`.

`soutpattern`: a glob-style pattern Since 2.16

**g_test_trap_assert_stdout_unmatched()**

```c
#define g_test_trap_assert_stdout_unmatched(soutpattern)
```

Assert that the stdout output of the last forked test does not match `soutpattern`. See `g_test_trap_fork()`.

`soutpattern`: a glob-style pattern Since 2.16

**g_test_trap_assert_stderr()**

```c
#define g_test_trap_assert_stderr(serrpattern)
```

Assert that the stderr output of the last forked test matches `serrpattern`. See `g_test_trap_fork()`.

`serrpattern`: a glob-style pattern Since 2.16

**g_test_trap_assert_stderr_unmatched()**

```c
#define g_test_trap_assert_stderr_unmatched(serrpattern)
```

Assert that the stderr output of the last forked test does not match `serrpattern`. See `g_test_trap_fork()`.

`serrpattern`: a glob-style pattern Since 2.16

**g_test_rand_bit()**

```c
#define g_test_rand_bit()
```

Get a reproducible random bit (0 or 1), see `g_test_rand_int()` for details on test case random numbers. Since 2.16
CHAPTER 4. GLIB UTILITIES

4.24. TESTING

g_test_rand_int()

 guint32 g_test_rand_int (void);

Get a reproducible random integer number.

The random numbers generated by the g_test_rand_*() family of functions change with every new test program start, unless the --seed option is given when starting test programs.

For individual test cases however, the random number generator is reseeded, to avoid dependencies between tests and to make --seed effective for all test cases.

Returns: a random number from the seeded random number generator.

Since 2.16

g_test_rand_int_range()

 guint32 g_test_rand_int_range (gint32 begin, gint32 end);

Get a reproducible random integer number out of a specified range, see g_test_rand_int() for details on test case random numbers.

begin: the minimum value returned by this function
end: the smallest value not to be returned by this function

Returns: a number with \(\text{begin} \leq \text{number} < \text{end}\).

Since 2.16

g_test_rand_double()

double g_test_rand_double (void);

Get a reproducible random floating point number, see g_test_rand_int() for details on test case random numbers.

Returns: a random number from the seeded random number generator.

Since 2.16

g_test_rand_double_range()

double g_test_rand_double_range (double range_start, double range_end);

Get a reproducible random floating pointer number out of a specified range, see g_test_rand_int() for details on test case random numbers.

range_start: the minimum value returned by this function
range_end: the minimum value not returned by this function

Returns: a number with \(\text{range_start} \leq \text{number} < \text{range_end}\).

Since 2.16

g_assert()

#define g_assert(expr)

Debugging macro to terminate the application if the assertion fails. If the assertion fails (i.e. the expression is not true), an error message is logged and the application is terminated.

The macro can be turned off in final releases of code by defining G_DISABLE_ASSERT when compiling the application.

expr: the expression to check.

445
CHAPTER 4. GLIB UTILITIES

4.24. TESTING

---

**g_assert_not_reached()**

```c
#define g_assert_not_reached()
```

Debugging macro to terminate the application if it is ever reached. If it is reached, an error message is logged and the application is terminated.

The macro can be turned off in final releases of code by defining `G_DISABLE_ASSERT` when compiling the application.

**g_assert_cmpstr()**

```c
#define g_assert_cmpstr(s1, cmp, s2)
```

Debugging macro to terminate the application with a warning message if a string comparison fails. The strings are compared using `g_strcmp0()`.

The effect of `g_assert_cmpstr (s1, op, s2)` is the same as `g_assert (g_strcmp0 (s1-, s2) op 0)`. The advantage of this macro is that it can produce a message that includes the actual values of `s1` and `s2`.

```c
g_assert_cmpstr (mystring, ==, "fubar");
```

- **s1**: a string (may be `NULL`)
- **cmp**: The comparison operator to use. One of `==`, `!=`, `<`, `>`, `<=`, `>=`.
- **s2**: another string (may be `NULL`)

Since 2.16

**g_assert_cmpint()**

```c
#define g_assert_cmpint(n1, cmp, n2)
```

Debugging macro to terminate the application with a warning message if an integer comparison fails.

The effect of `g_assert_cmpint (n1, op, n2)` is the same as `g_assert (n1 op n2)`. The advantage of this macro is that it can produce a message that includes the actual values of `n1` and `n2`.

- **n1**: an integer
- **cmp**: The comparison operator to use. One of `==`, `!=`, `<`, `>`, `<=`, `>=`.
- **n2**: another integer

Since 2.16

**g_assert_cmpuint()**

```c
#define g_assert_cmpuint(n1, cmp, n2)
```

Debugging macro to terminate the application with a warning message if an unsigned integer comparison fails.

The effect of `g_assert_cmpuint (n1, op, n2)` is the same as `g_assert (n1 op n2)`. The advantage of this macro is that it can produce a message that includes the actual values of `n1` and `n2`.

- **n1**: an unsigned integer
- **cmp**: The comparison operator to use. One of `==`, `!=`, `<`, `>`, `<=`, `>=`.
- **n2**: another unsigned integer

Since 2.16
g_assert_cmphex()

#define g_assert_cmphex(n1, cmp, n2)

Debugging macro to terminate the application with a warning message if an unsigned integer comparison fails. This is a variant of g_assert_cmpuint() that displays the numbers in hexadecimal notation in the message.

n1: an unsigned integer

cmp: The comparison operator to use. One of ==, !=, <, >, <=, >=.

n2: another unsigned integer

Since 2.16

g_assert_cmpfloat()

#define g_assert_cmpfloat(n1, cmp, n2)

Debugging macro to terminate the application with a warning message if a floating point number comparison fails.

The effect of g_assert_cmpfloat (n1, op, n2) is the same as g_assert (n1 op n2). The advantage of this function is that it can produce a message that includes the actual values of n1 and n2.

n1: an floating point number

cmp: The comparison operator to use. One of ==, !=, <, >, <=, >=.

n2: another floating point number

Since 2.16

g_assert_no_error()

#define g_assert_no_error(err)

Debugging macro to terminate the application with a warning message if a method has returned a GError.

The effect of g_assert_no_error (err) is the same as g_assert (err == NULL). The advantage of this macro is that it can produce a message that includes the error message and code.

err: a GError, possibly NULL

Since 2.20

g_assert_error()

#define g_assert_error(err, dom, c)

Debugging macro to terminate the application with a warning message if a method has not returned the correct GError.

The effect of g_assert_error (err, dom, c) is the same as g_assert (err != NULL && err->domain == dom && err->code == c). The advantage of this macro is that it can produce a message that includes the incorrect error message and code.

This can only be used to test for a specific error. If you want to test that err is set, but don’t care what it’s set to, just use g_assert (err != NULL)

err: a GError, possibly NULL

dom: the expected error domain (a GQuark)

c: the expected error code

Since 2.20
CHAPTER 4. GLIB UTILITIES

GTestCase

typedef struct GTestCase GTestCase;

An opaque structure representing a test case.

GTestSuite

typedef struct GTestSuite GTestSuite;

An opaque structure representing a test suite.

g_test_create_case()

GTestCase* g_test_create_case (const char *test_name, gsize data_size, gconstpointer test_data, void (data_setupvoid) (), void (data_testvoid) (), void (data_teardownvoid) ());

Create a new GTestCase, named test_name, this API is fairly low level, calling g_test_add() or g_test_add_func() is preferable. When this test is executed, a fixture structure of size data_size will be allocated and filled with 0s. Then data_setup() is called to initialize the fixture. After fixture setup, the actual test function data_test() is called. Once the test run completed, the fixture structure is torn down by calling data_teardown() and after that the memory is released.

Splitting up a test run into fixture setup, test function and fixture teardown is most useful if the same fixture is used for multiple tests. In this cases, g_test_create_case() will be called with the same fixture, but varying test_name and data_test arguments.

- **test_name**: the name for the test case
- **data_size**: the size of the fixture data structure
- **test_data**: test data argument for the test functions
- **data_setup**: the function to set up the fixture data
- **data_test**: the actual test function
- **data_teardown**: the function to teardown the fixture data

**Returns**: a newly allocated GTestCase.

Since 2.16

g_test_create_suite()

GTestSuite* g_test_create_suite (const char *suite_name);

Create a new test suite with the name suite_name.

- **suite_name**: a name for the suite

**Returns**: A newly allocated GTestSuite instance.

Since 2.16


### 4.25 Windows Compatibility Functions

**Name**

Windows Compatibility Functions – UNIX emulation on Windows

#### 4.25.1 g_test_get_root()

```c
GTestSuite* g_test_get_root (void);
```

Get the toplevel test suite for the test path API.

**Returns**: the toplevel GTestSuite

Since 2.16

#### 4.25.2 g_test_suite_add()

```c
void g_test_suite_add (GTestSuite *suite, GTestCase *test_case);
```

Adds `test_case` to `suite`.

**suite**: a GTestSuite

**test_case**: a GTestCase

Since 2.16

#### 4.25.3 g_test_suite_add_suite()

```c
void g_test_suite_add_suite (GTestSuite *suite, GTestSuite *nestedsuite);
```

Adds `nestedsuite` to `suite`.

**suite**: a GTestSuite

**nestedsuite**: another GTestSuite

Since 2.16

#### 4.25.4 g_test_run_suite()

```c
int g_test_run_suite (GTestSuite *suite);
```

Execute the tests within `suite` and all nested GTestSuites. The test suites to be executed are filtered according to test path arguments (`-p testpath`) as parsed by `g_test_init()`. `g_test_run_suite()` or `g_test_run()` may only be called once in a program.

**suite**: a GTestSuite

**Returns**: 0 on success

Since 2.16

**See Also**

gtester, gtester-report

**4.25 Windows Compatibility Functions**
Synopsis

#include <glib.h>
#define MAXPATHLEN

gchar* g_win32_error_message (gint error);
gchar* g_win32_getlocale (void);
gchar* g_win32_get_package_installation_directory
     (const gchar *package, const gchar *dll_name);
gchar* g_win32_get_package_installation_directory_of_module
     (gpointer hmodule);
gchar* g_win32_get_package_installation_subdirectory
     (const gchar *package, const gchar *dll_name, const gchar *subdir);
guint g_win32_get_windows_version (void);
gchar* g_win32_locale_filename_from_utf8 (const gchar *utf8filename);
#define G_WIN32_DLLMAIN_FOR_DLL_NAME (static, dll_name)
#define G_WIN32_HAVE_WIDECHAR_API ()
#define G_WIN32_IS_NT_BASED ()

Description

These functions provide some level of UNIX emulation on the Windows platform. If your application really needs the POSIX APIs, we suggest you try the Cygwin project.

Details

MAXPATHLEN

#define MAXPATHLEN 1024

Provided for UNIX emulation on Windows; equivalent to UNIX macro MAXPATHLEN, which is the maximum length of a filename (including full path).

g_win32_error_message ()

Translate a Win32 error code (as returned by GetLastError()) into the corresponding message. The message is either language neutral, or in the thread’s language, or the user’s language, the system’s language, or US English (see docs for FormatMessage()). The returned string is in UTF-8. It should be deallocated with g_free().

error: error code.

Returns: newly-allocated error message

g_win32_getlocale ()

The setlocale() function in the Microsoft C library uses locale names of the form "English_United States.1252" etc. We want the UNIXish standard form "en_US", "zh_TW" etc. This function gets the current thread locale from Windows - without any encoding info - and returns it as a string of the above form for use in forming file names etc. The returned string should be deallocated with g_free().

Returns: newly-allocated locale name.
CHAPTER 4. GLIB UTILITIES 4.25. WINDOWS COMPATIBILITY FUNCTIONS

G_Win32_GetPackageInstallationDirectory()

gchar* g_win32_getPackageInstallationDirectory
(const gchar *package,
 const gchar *dll_name);

WARNING

G_Win32_GetPackageInstallationDirectory is deprecated and should not be used in newly-written code.

Try to determine the installation directory for a software package. This function is deprecated. Use G_Win32_GetPackageInstallationDirectoryOfModule() instead.

The use of package is deprecated. You should always pass NULL. A warning is printed if non-NULL is passed as package.

The original intended use of package was for a short identifier of the package, typically the same identifier as used for GETTEXT_PACKAGE in software configured using GNU autotools. The function first looks in the Windows Registry for the value #InstallationDirectory in the key #HKLM\Software@package, and if that value exists and is a string, returns that.

It is strongly recommended that packagers of GLib-using libraries for Windows do not store installation paths in the Registry to be used by this function as that interferes with having several parallel installations of the library. Enabling multiple installations of different versions of some GLib-using library, or GLib itself, is desirable for various reasons.

For this reason it is recommended to always pass NULL as package to this function, to avoid the temptation to use the Registry. In version 2.20 of GLib the package parameter will be ignored and this function won’t look in the Registry at all.

If package is NULL, or the above value isn’t found in the Registry, but dll_name is non-NULL, it should name a DLL loaded into the current process. Typically that would be the name of the DLL calling this function, looking for its installation directory. The function then asks Windows what directory that DLL was loaded from. If that directory’s last component is “bin” or “lib”, the parent directory is returned, otherwise the directory itself. If that DLL isn’t loaded, the function proceeds as if dll_name was NULL.

If both package and dll_name are NULL, the directory from where the main executable of the process was loaded is used instead in the same way as above.

package: You should pass NULL for this.

dll_name: The name of a DLL that a package provides in UTF-8, or NULL.

Returns: a string containing the installation directory for package. The string is in the GLib file name encoding, i.e. UTF-8. The return value should be freed with g_free() when not needed any longer. If the function fails NULL is returned. Deprecated:2.18: Pass the HMODULE of a DLL or EXE to G_Win32_GetPackageInstallationDirectoryOfModule() instead.

G_Win32_GetPackageInstallationDirectoryOfModule()

gchar* g_win32_getPackageInstallationDirectoryOfModule
(gpointer hmodule);

This function tries to determine the installation directory of a software package based on the location of a DLL of the software package.

hmodule should be the handle of a loaded DLL or NULL. The function looks up the directory that DLL was loaded from. If hmodule is NULL, the directory the main executable of the current process is looked up. If that directory’s last component is “bin” or “lib”, its parent directory is returned, otherwise the directory itself.

It thus makes sense to pass only the handle to a “public” DLL of a software package to this function, as such DLLs typically are known to be installed in a “bin” or occasionally “lib” subfolder of the installation folder. DLLs that are of the dynamically loaded module or plugin variety are often located in more
private locations deeper down in the tree, from which it is impossible for GLib to deduce the root of the
package installation.

The typical use case for this function is to have a DllMain() that saves the handle for the DLL. Then
when code in the DLL needs to construct names of files in the installation tree it calls this function
passing the DLL handle.

\textbf{hmodule}: The Win32 handle for a DLL loaded into the current process, or \texttt{NULL}.

\textbf{Returns}: A string containing the guessed installation directory for the software package \texttt{hmodule} is
from. The string is in the GLib file name encoding, i.e. UTF-8. The return value should be freed
with \texttt{g_free()} when no longer needed. If the function fails \texttt{NULL} is returned.

Since 2.16

\texttt{g_win32_get_package_installation_subdirectory ()}

\begin{verbatim}
 gchar* g_win32_get_package_installation_subdirectory (const gchar *package,
 const gchar *dll_name,
 const gchar *subdir);
\end{verbatim}

\textbf{WARNING}

\texttt{g_win32_get_package_installation_subdirectory} is deprecated and
should not be used in newly-written code.

This function is deprecated. Use \texttt{g Win32 Get Package Installation Directory Of Module()} and \texttt{g Build Filename()} instead.

Returns a newly-allocated string containing the path of the subdirectory \texttt{subdir} in the return value
from calling \texttt{g Win32 Get Package Installation Directory()} with the \texttt{package} and \texttt{dll_name} parameters.
See the documentation for \texttt{g Win32 Get Package Installation Directory()} for more details. In particular,
note that it is deprecated to pass anything except \texttt{NULL} as \texttt{package}.

\textbf{package}: You should pass \texttt{NULL} for this.

\textbf{dll_name}: The name of a DLL that a package provides, in UTF-8, or \texttt{NULL}.

\textbf{subdir}: A subdirectory of the package installation directory, also in UTF-8.

\textbf{Returns}: A string containing the complete path to \texttt{subdir} inside the installation directory of \texttt{package}.
The returned string is in the GLib file name encoding, i.e. UTF-8. The return value should be freed
with \texttt{g_free()} when no longer needed. If something goes wrong, \texttt{NULL} is returned. \texttt{Deprecated-}
\texttt{d} 2.18: Pass the HMODULE of a DLL or EXE to \texttt{g Win32 Get Package Installation Directory Of Module()} instead,
and then construct a subdirectory pathname with \texttt{g Build Filename()}.

\texttt{g_win32_get_windows_version ()}

\begin{verbatim}
 guint g_win32_get_windows_version (void);
\end{verbatim}

\textbf{Returns}: The version information.

Since 2.6
CHAPTER 4. GLIB UTILITIES 4.25. WINDOWS COMPATIBILITY FUNCTIONS

**g_win32_locale_filename_from_utf8 ()**

```c
 gchar* g_win32_locale_filename_from_utf8 (const gchar * utf8filename);
```

Converts a filename from UTF-8 to the system codepage.

On NT-based Windows, on NTFS file systems, file names are in Unicode. It is quite possible that Unicode file names contain characters not representable in the system codepage. (For instance, Greek or Cyrillic characters on Western European or US Windows installations, or various less common CJK characters on CJK Windows installations.)

In such a case, and if the filename refers to an existing file, and the file system stores alternate short (8.3) names for directory entries, the short form of the filename is returned. Note that the "short" name might in fact be longer than the Unicode name if the Unicode name has very short pathname components containing non-ASCII characters. If no system codepage name for the file is possible, NULL is returned.

The return value is dynamically allocated and should be freed with g_free() when no longer needed.

`utf8filename`: a UTF-8 encoded filename.

**Returns**: The converted filename, or NULL on conversion failure and lack of short names.

Since 2.8

**G_WIN32_DLLMAIN_FOR_DLL_NAME()**

```c
#define G_WIN32_DLLMAIN_FOR_DLL_NAME(static, dll_name)
```

**WARNING**

```
G_WIN32_DLLMAIN_FOR_DLL_NAME is deprecated and should not be used in newly-written code.
```

On Windows, this macro defines a DllMain() function that stores the actual DLL name that the code being compiled will be included in.

On non-Windows platforms, expands to nothing.

`static`: empty or "static".

`dll_name`: the name of the (pointer to the) char array where the DLL name will be stored. If this is used, you must also include windows.h. If you need a more complex DLL entry point function, you cannot use this.

**G_WIN32_HAVE_WIDECHAR_API()**

```c
#define G_WIN32_HAVE_WIDECHAR_API() TRUE
```

On Windows, this macro defines an expression which evaluates to TRUE if the code is running on a version of Windows where the wide character versions of the Win32 API functions, and the wide character versions of the C library functions work. (They are always present in the DLLs, but don’t work on Windows 9x and Me.)

On non-Windows platforms, it is not defined.

Since 2.6
G_WIN32_IS_NT_BASED()

#define G_WIN32_IS_NT_BASED() TRUE

On Windows, this macro defines an expression which evaluates to \texttt{TRUE} if the code is running on an NT-based Windows operating system.
On non-Windows platforms, it is not defined.
Since 2.6
Chapter 5

GLib Data Types

5.1 Memory Slices

Name
Memory Slices – efficient way to allocate groups of equal-sized chunks of memory

Synopsis

```c
#include <glib.h>

gpointer g_slice_alloc (gsize block_size);
gpointer g_slice_alloc0 (gsize block_size);
gpointer g_slice_copy (gsize block_size, gconstpointer mem_block);
void g_slice_free1 (gsize block_size, gpointer mem_block);
void g_slice_free_chain_with_offset (gsize block_size, gpointer mem_chain, gsize next_offset);

#define g_slice_new (type)
#define g_slice_new0 (type)
#define g_slice_dup (type, mem)
#define g_slice_free (type, mem)
#define g_slice_free_chain (type, mem_chain, next)
```

Description

Memory slices provide a space-efficient and multi-processing scalable way to allocate equal-sized pieces of memory, just like the original GMemChunks (from Glib <= 2.8), while avoiding their excessive memory-waste, scalability and performance problems.

To achieve these goals, the slice allocator uses a sophisticated, layered design that has been inspired by Bonwick’s slab allocator \(^1\). It uses `posix_memalign()` to optimize allocations of many equally-sized chunks, and has per-thread free lists (the so-called magazine layer) to quickly satisfy allocation requests of already known structure sizes. This is accompanied by extra caching logic to keep freed memory around for some time before returning it to the system. Memory that is unused due to alignment constraints is used for cache colorization (random distribution of chunk addresses) to improve CPU cache utilization. The caching layer of the slice allocator adapts itself to high lock contention to improve scalability.

---

The slice allocator can allocate blocks as small as two pointers, and unlike malloc(), it does not reserve extra space per block. For large block sizes, g_slice_new() and g_slice_alloc() will automatically delegate to the system malloc() implementation. For newly written code it is recommended to use the new g_slice API instead of g_malloc() and friends, as long as objects are not resized during their lifetime and the object size used at allocation time is still available when freeing.

Example 5.1 Using the slice allocator

```c
 gchar *mem[10000];
 gint i;
 /* Allocate 10000 blocks. */
 for (i = 0; i < 10000; i++)
  { 
   mem[i] = g_slice_alloc (50);
   /* Fill in the memory with some junk. */
   for (j = 0; j < 50; j++)
     mem[i][j] = i * j;
  }
 /* Now free all of the blocks. */
 for (i = 0; i < 10000; i++)
  { 
   g_slice_free1 (50, mem[i]);
  }
```

Example 5.2 Using the slice allocator with data structures

```c
GRealArray *array;
 /* Allocate one block, using the g_slice_new() macro. */
 array = g_slice_new (GRealArray);
 /* We can now use array just like a normal pointer to a structure. */
 array->data = NULL;
 array->len = 0;
 array->alloc = 0;
 array->zero_terminated = (zero_terminated ? 1 : 0);
 array->clear = (clear ? 1 : 0);
 array->elt_size = elt_size;
 /* We can free the block, so it can be reused. */
 g_slice_free (GRealArray, array);
```

Details

**g_slice_alloc ()**

Allocates a block of memory from the slice allocator. The block adress handed out can be expected to be aligned to at least 1 + sizeof (void*), though in general slices are 2 * sizeof (void*) bytes aligned, if a malloc() fallback implementation is used instead, the alignment may be reduced in a libc dependent fashion. Note that the underlying slice allocation mechanism can be changed with the G_SLI CE=always-malloc environment variable.

**block_size:** the number of bytes to allocate

**Returns:** a pointer to the allocated memory block

Since 2.10

**g_slice_alloc0 ()**
5.1. MEMORY SLICES

`gpointer g_slice_alloc0 (gsize block_size);`

Allocates a block of memory via `g_slice_alloc()` and initialize the returned memory to 0. Note that the underlying slice allocation mechanism can be changed with the `G_SLICE=always-malloc` environment variable.

**block_size**: the number of bytes to allocate

**Returns**: a pointer to the allocated block

Since 2.10

`g_slice_copy ()`

`gpointer g_slice_copy (gsize block_size, gconstpointer mem_block);`

Allocates a block of memory from the slice allocator and copies `block_size` bytes into it from `mem_block`.

**block_size**: the number of bytes to allocate

**mem_block**: the memory to copy

**Returns**: a pointer to the allocated memory block

Since 2.14

`g_slice_free1 ()`

`void g_slice_free1 (gsize block_size, gpointer mem_block);`

Frees a block of memory. The memory must have been allocated via `g_slice_alloc()` or `g_slice_alloc0()` and the `block_size` has to match the size specified upon allocation. Note that the exact release behaviour can be changed with the `G_DEBUG=gc-friendly` environment variable, also see `G_SLICE` for related debugging options.

**block_size**: the size of the block

**mem_block**: a pointer to the block to free

Since 2.10

`g_slice_free_chain_with_offset ()`

`void g_slice_free_chain_with_offset (gsize block_size, gpointer mem_chain, gsize next_offset);`

Frees a linked list of memory blocks of structure type `type`. The memory blocks must be equal-sized, allocated via `g_slice_alloc()` or `g_slice_alloc0()` and linked together by a `next` pointer (similar to `GSList`). The offset of the `next` field in each block is passed as third argument. Note that the exact release behaviour can be changed with the `G_DEBUG=gc-friendly` environment variable, also see `G_SLICE` for related debugging options.

**block_size**: the size of the blocks

**mem_chain**: a pointer to the first block of the chain

**next_offset**: the offset of the `next` field in the blocks

Since 2.10
CHAPTER 5. GLIB DATA TYPES

5.1. MEMORY SLICES

g_slice_new()

#define g_slice_new(type)

A convenience macro to allocate a block of memory from the slice allocator. It calls g_slice_alloc() with sizeof (type) and casts the returned pointer to a pointer of the given type, avoiding a type cast in the source code. Note that the underlying slice allocation mechanism can be changed with the G_SLICE=always-malloc environment variable.

**type** : the type to allocate, typically a structure name

**Returns** : a pointer to the allocated block, cast to a pointer to type.

Since 2.10

---

g_slice_new0()

#define g_slice_new0(type)

A convenience macro to allocate a block of memory from the slice allocator and set the memory to 0. It calls g_slice_alloc0() with sizeof (type) and casts the returned pointer to a pointer of the given type, avoiding a type cast in the source code. Note that the underlying slice allocation mechanism can be changed with the G_SLICE=always-malloc environment variable.

**type** : the type to allocate, typically a structure name

**Returns** : a pointer to the allocated block, cast to a pointer to type.

Since 2.10

---

g_slice_dup()

#define g_slice_dup(type, mem)

A convenience macro to duplicate a block of memory using the slice allocator. It calls g_slice_copy() with sizeof (type) and casts the returned pointer to a pointer of the given type, avoiding a type cast in the source code. Note that the underlying slice allocation mechanism can be changed with the G_SLICE=always-malloc environment variable.

**type** : the type to duplicate, typically a structure name

**mem** : the memory to copy into the allocated block

**Returns** : a pointer to the allocated block, cast to a pointer to type.

Since 2.14

---

g_slice_free()

#define g_slice_free(type, mem)

A convenience macro to free a block of memory that has been allocated from the slice allocator. It calls g_slice_free1() using sizeof (type) as the block size. Note that the exact release behaviour can be changed with the G_DEBUG=gc-friendly environment variable, also see G_SLICE for related debugging options.

**type** : the type of the block to free, typically a structure name

**mem** : a pointer to the block to free

Since 2.10
g_slice_free_chain()

#define g_slice_free_chain(type, mem_chain, next)

Frees a linked list of memory blocks of structure type type. The memory blocks must be equal-sized, allocated via g_slice_alloc() or g_slice_alloc0() and linked together by a next pointer (similar to GSList). The name of the next field in type is passed as third argument. Note that the exact release behaviour can be changed with the G_DEBUG=gc-friendly environment variable, also see G_SLICE for related debugging options.

**type**: the type of the mem_chain blocks

**mem_chain**: a pointer to the first block of the chain

**next**: the field name of the next pointer in type

Since 2.10

## 5.2 Memory Chunks

### Name

Memory Chunks – deprecated way to allocate groups of equal-sized chunks of memory

### Synopsis

```
#include <glib.h>

GMemChunk;

#define G_ALLOC_AND_FREE
#define G_ALLOC_ONLY

GMemChunk* g_mem_chunk_new (const gchar *name, gint atom_size, gsize area_size, gint type);
gpointer g_mem_chunk_alloc (GMemChunk *mem_chunk);
gpointer g_mem_chunk_alloc0 (GMemChunk *mem_chunk);
void g_mem_chunk_free (GMemChunk *mem_chunk, gpointer mem);
void g_mem_chunk_destroy (GMemChunk *mem_chunk);

#define g_mem_chunk_create (type, pre_alloc, alloc_type)
#define g_chunk_new (type, chunk)
#define g_chunk_new0 (type, chunk)
#define g_chunk_free (mem, mem_chunk)

void g_mem_chunk_reset (GMemChunk *mem_chunk);
void g_mem_chunk_clean (GMemChunk *mem_chunk);
void g_blow_chunks (void);
void g_mem_chunk_info (void);
void g_mem_chunk_print (GMemChunk *mem_chunk);
```

### Description

Memory chunks provide an space-efficient way to allocate equal-sized pieces of memory, called atoms. However, due to the administrative overhead (in particular for G_ALLOC_AND_FREE, and when used
from multiple threads), they are in practise often slower than direct use of `g_malloc()`. Therefore, memory chunks have been deprecated in favor of the slice allocator, which has been added in 2.10. All internal uses of memory chunks in GLib have been converted to the `g_slice` API.

There are two types of memory chunks, `G_ALLOC_ONLY`, and `G_ALLOC_AND_FREE`.

- **`G_ALLOC_ONLY`** chunks only allow allocation of atoms. The atoms can never be freed individually. The memory chunk can only be free in its entirety.

- **`G_ALLOC_AND_FREE`** chunks do allow atoms to be freed individually. The disadvantage of this is that the memory chunk has to keep track of which atoms have been freed. This results in more memory being used and a slight degradation in performance.

To create a memory chunk use `g_mem_chunk_new()` or the convenience macro `g_mem_chunk_create()`.

To allocate a new atom use `g_mem_chunk_alloc()`, `g_mem_chunk_alloc0()`, or the convenience macros `g_chunk_new()` or `g_chunk_new0()`.

To free an atom use `g_mem_chunk_free()`, or the convenience macro `g_chunk_free()`. (Atoms can only be freed if the memory chunk is created with the type set to `G_ALLOC_AND_FREE`.)

To free any blocks of memory which are no longer being used, use `g_mem_chunk_clean()`. To clean all memory chunks, use `g_blow_chunks()`.

To reset the memory chunk, freeing all of the atoms, use `g_mem_chunk_reset()`.

To destroy a memory chunk, use `g_mem_chunk_destroy()`.

To help debug memory chunks, use `g_mem_chunk_info()` and `g_mem_chunk_print()`.

---

**Example 5.3 Using a GMemChunk**

```c
GMemChunk *mem_chunk;
gchar *mem[10000];
gint i;
/* Create a GMemChunk with atoms 50 bytes long, and memory blocks holding 100 bytes. Note that this means that only 2 atoms fit into each memory block and so isn't very efficient. */
mem_chunk = g_mem_chunk_new (*test mem chunk*, 50, 100, G_ALLOC_AND_FREE);
/* Now allocate 10000 atoms. */
for (i = 0; i < 10000; i++)
{
    mem[i] = g_chunk_new (gchar, mem_chunk);
    /* Fill in the atom memory with some junk. */
    for (j = 0; j < 50; j++)
        mem[i][j] = i * j;
}
/* Now free all of the atoms. Note that since we are going to destroy the GMemChunk, this wouldn't normally be used. */
for (i = 0; i < 10000; i++)
{
    g_mem_chunk_free (mem_chunk, mem[i]);
}
/* We are finished with the GMemChunk, so we destroy it. */
g_mem_chunk_destroy (mem_chunk);
```
Example 5.4 Using a GMemChunk with data structures

```c
GMemChunk *array_mem_chunk;
GRealArray *array;

/* Create a GMemChunk to hold GRealArray structures, using the 
g_mem_chunk_create() convenience macro. We want 1024 atoms in each 
memory block, and we want to be able to free individual atoms. */
array_mem_chunk = g_mem_chunk_create (GRealArray, 1024, G_ALLOC_AND_FREE);
/* Allocate one atom, using the g_chunk_new() convenience macro. */
array = g_chunk_new (GRealArray, array_mem_chunk);
/* We can now use array just like a normal pointer to a structure. */
array->data = NULL;
array->len = 0;
array->alloc = 0;
array->zero_terminated = (zero_terminated ? 1 : 0);
array->clear = (clear ? 1 : 0);
array->elt_size = elt_size;
/* We can free the element, so it can be reused. */
g_chunk_free (array, array_mem_chunk);
/* We destroy the GMemChunk when we are finished with it. */
g_mem_chunk_destroy (array_mem_chunk);
```

Details

GMemChunk

```c
typedef struct _GMemChunk GMemChunk;
```

**WARNING**

GMemChunk is deprecated and should not be used in newly-written code.

The GMemChunk struct is an opaque data structure representing a memory chunk. It should be accessed only through the use of the following functions.

**G_ALLOC_AND_FREE**

```c
#define G_ALLOC_AND_FREE 2
```

**WARNING**

G_ALLOC_AND_FREE is deprecated and should not be used in newly-written code.

Specifies the type of a GMemChunk. Used in g_mem_chunk_new() and g_mem_chunk_create() to specify that atoms will be freed individually.

**G_ALLOC_ONLY**

```c
#define G_ALLOC_ONLY 1
```
WARNING

G_ALLOC_ONLY is deprecated and should not be used in newly-written code.

Specifies the type of a GMemChunk. Used in g_mem_chunk_new() and g_mem_chunk_create() to specify that atoms will never be freed individually.

**g_mem_chunk_new ()**

```c
GMemChunk* g_mem_chunk_new (const gchar *name, gint atom_size, gsize area_size, gint type);
```

**WARNING**

g_mem_chunk_new has been deprecated since version 2.10 and should not be used in newly-written code. Use the slice allocator instead

Creates a new GMemChunk.

*name*: a string to identify the GMemChunk. It is not copied so it should be valid for the lifetime of the GMemChunk. It is only used in g_mem_chunk_print(), which is used for debugging.

*atom_size*: the size, in bytes, of each element in the GMemChunk.

*area_size*: the size, in bytes, of each block of memory allocated to contain the atoms.

*type*: the type of the GMemChunk. G_ALLOC_AND_FREE is used if the atoms will be freed individually. G_ALLOC_ONLY should be used if atoms will never be freed individually. G_ALLOC_ONLY is quicker, since it does not need to track free atoms, but it obviously wastes memory if you no longer need many of the atoms.

**Returns**: the new GMemChunk.

**g_mem_chunk_alloc ()**

```c
gpointer g_mem_chunk_alloc (GMemChunk *mem_chunk);
```

**WARNING**

g_mem_chunk_alloc has been deprecated since version 2.10 and should not be used in newly-written code. Use g_slice_alloc() instead

Allocates an atom of memory from a GMemChunk.

*mem_chunk*: a GMemChunk.

**Returns**: a pointer to the allocated atom.
CHAPTER 5. GLIB DATA TYPES

5.2. MEMORY CHUNKS

**g_mem_chunk_alloc0 ()**

```c
gpointer g_mem_chunk_alloc0 (GMemChunk *mem_chunk);
```

**WARNING**

`g_mem_chunk_alloc0` has been deprecated since version 2.10 and should not be used in newly-written code. Use `g_slice_alloc0()` instead.

Allocates an atom of memory from a GMemChunk, setting the memory to 0.

**mem_chunk**: a GMemChunk.

**Returns**: a pointer to the allocated atom.

**g_mem_chunk_free ()**

```c
void g_mem_chunk_free (GMemChunk *mem_chunk, gpointer mem);
```

**WARNING**

`g_mem_chunk_free` has been deprecated since version 2.10 and should not be used in newly-written code. Use `g_slice_free1()` instead.

Frees an atom in a GMemChunk. This should only be called if the GMemChunk was created with `G_ALLOC_AND_FREE`. Otherwise it will simply return.

**mem_chunk**: a GMemChunk.

**mem**: a pointer to the atom to free.

**g_mem_chunk_destroy ()**

```c
void g_mem_chunk_destroy (GMemChunk *mem_chunk);
```

**WARNING**

`g_mem_chunk_destroy` has been deprecated since version 2.10 and should not be used in newly-written code. Use the slice allocator instead.

Frees all of the memory allocated for a GMemChunk.

**mem_chunk**: a GMemChunk.
g_mem_chunk_create()

#define g_mem_chunk_create(type, pre_alloc, alloc_type)

**WARNING**

`g_mem_chunk_create` has been deprecated since version 2.10 and should not be used in newly-written code. Use the slice allocator instead.

A convenience macro for creating a new GMemChunk. It calls `g_mem_chunk_new()`, using the given type to create the GMemChunk name. The atom size is determined using `sizeof()`, and the area size is calculated by multiplying the `pre_alloc` parameter with the atom size.

- **type**: the type of the atoms, typically a structure name.
- **pre_alloc**: the number of atoms to store in each block of memory.
- **alloc_type**: the type of the GMemChunk. `G_ALLOC_AND_FREE` is used if the atoms will be freed individually. `G_ALLOC_ONLY` should be used if atoms will never be freed individually. `G_ALLOC_ONLY` is quicker, since it does not need to track free atoms, but it obviously wastes memory if you no longer need many of the atoms.

**Returns**: the new GMemChunk.

g_chunk_new()

#define g_chunk_new(type, chunk)

**WARNING**

`g_chunk_new` has been deprecated since version 2.10 and should not be used in newly-written code. Use `g_slice_new()` instead.

A convenience macro to allocate an atom of memory from a GMemChunk. It calls `g_mem_chunk_alloc()` and casts the returned atom to a pointer to the given type, avoiding a type cast in the source code.

- **type**: the type of the GMemChunk atoms, typically a structure name.
- **chunk**: a GMemChunk.

**Returns**: a pointer to the allocated atom, cast to a pointer to `type`.

g_chunk_new0()

#define g_chunk_new0(type, chunk)

**WARNING**

`g_chunk_new0` has been deprecated since version 2.10 and should not be used in newly-written code. Use `g_slice_new0()` instead.

A convenience macro to allocate an atom of memory from a GMemChunk. It calls `g_mem_chunk_alloc0()` and casts the returned atom to a pointer to the given type, avoiding a type cast in the source code.

- **type**: the type of the GMemChunk atoms, typically a structure name.
- **chunk**: a GMemChunk.

**Returns**: a pointer to the allocated atom, cast to a pointer to `type`. 
A convenience macro to allocate an atom of memory from a GMemChunk. It calls `g_mem_chunk_alloc0()` and casts the returned atom to a pointer to the given type, avoiding a type cast in the source code.

**type**: the type of the GMemChunk atoms, typically a structure name.

**chunk**: a GMemChunk.

**Returns**: a pointer to the allocated atom, cast to a pointer to `type`.

### `g_chunk_free()`

```c
#define g_chunk_free(mem, mem_chunk)
```

**WARNING**

`g_chunk_free` has been deprecated since version 2.10 and should not be used in newly-written code. Use `g_slice_free()` instead.

A convenience macro to free an atom of memory from a GMemChunk. It simply switches the arguments and calls `g_mem_chunk_free()` It is included simply to complement the other convenience macros, `g_chunk_new()` and `g_chunk_new0()`.

**mem**: a pointer to the atom to be freed.

**mem_chunk**: a GMemChunk.

### `g_mem_chunk_reset()`

```c
void g_mem_chunk_reset (GMemChunk *mem_chunk);
```

**WARNING**

`g_mem_chunk_reset` has been deprecated since version 2.10 and should not be used in newly-written code. Use the slice allocator instead.

Resets a GMemChunk to its initial state. It frees all of the currently allocated blocks of memory.

**mem_chunk**: a GMemChunk.

### `g_mem_chunk_clean()`

```c
void g_mem_chunk_clean (GMemChunk *mem_chunk);
```

**WARNING**

`g_mem_chunk_clean` has been deprecated since version 2.10 and should not be used in newly-written code. Use the slice allocator instead.

Frees any blocks in a GMemChunk which are no longer being used.

**mem_chunk**: a GMemChunk.
g_blow_chunks()

```c
void g_blow_chunks (void);
```

**WARNING**

`g_blow_chunks` has been deprecated since version 2.10 and should not be used in newly-written code. Use the slice allocator instead.

Calls `g_mem_chunk_clean()` on all GMemChunk objects.

---

g_mem_chunk_info()

```c
void g_mem_chunk_info (void);
```

**WARNING**

`g_mem_chunk_info` has been deprecated since version 2.10 and should not be used in newly-written code. Use the slice allocator instead.

Outputs debugging information for all GMemChunk objects currently in use. It outputs the number of GMemChunk objects currently allocated, and calls `g_mem_chunk_print()` to output information on each one.

---

g_mem_chunk_print()

```c
void g_mem_chunk_print (GMemChunk *mem_chunk);
```

**WARNING**

`g_mem_chunk_print` has been deprecated since version 2.10 and should not be used in newly-written code. Use the slice allocator instead.

Outputs debugging information for a GMemChunk. It outputs the name of the GMemChunk (set with `g_mem_chunk_new()`), the number of bytes used, and the number of blocks of memory allocated.

`mem_chunk`: a GMemChunk.

---

### 5.3 Doubly-Linked Lists

#### Name

Doubly-Linked Lists – linked lists containing integer values or pointers to data, with the ability to iterate over the list in both directions
CHAPTER 5. GLIB DATA TYPES

5.3. DOUBLY-LINKED LISTS

Synopsis

#include <glib.h>

GList;

GList* g_list_append (GList *list, gpointer data);
GList* g_list_prepend (GList *list, gpointer data);
GList* g_list_insert (GList *list, gpointer data, gint position);
GList* g_list_insert_before (GList *list, GList *sibling, gpointer data);
GList* g_list_insert_sorted (GList *list, gpointer data, GCompareFunc func);
GList* g_list_remove (GList *list, gconstpointer data);
GList* g_list_remove_link (GList *list, GList *llink);
GList* g_list_delete_link (GList *list, GList *link_);
GList* g_list_remove_all (GList *list, gconstpointer data);
void g_list_free (GList *list);
GList* g_list_alloc (void);
void g_list_free_1 (GList *list);
#define g_list_free1

guint g_list_length (GList *list);
GList* g_list_copy (GList *list);
GList* g_list_reverse (GList *list);
GList* g_list_sort (GList *list, GCompareFunc compare_func);

GList* g_list_insert_sorted_with_data (GList *list, gpointer data, GCompareDataFunc func, gpointer user_data);
GList* g_list_sort_with_data (GList *list, GCompareDataFunc compare_func, gpointer user_data);

gint (*GCompareFunc) (gconstpointer a, gconstpointer b);
GList* g_list_insert_sorted_with_data (GList *list, gpointer data, GCompareDataFunc func, gpointer user_data);
GList* g_list_sort_with_data (GList *list, GCompareDataFunc compare_func, gpointer user_data);

gint (*GCompareDataFunc) (gconstpointer a, gconstpointer b, gpointer user_data);
GList* g_list_concat (GList *list1, GList *list2);
void g_list_foreach (GList *list, GFunc func, gpointer user_data);
void (*GFunc) (gpointer data, gpointer user_data);
The GList structure and its associated functions provide a standard doubly-linked list data structure.

Each element in the list contains a piece of data, together with pointers which link to the previous and next elements in the list. Using these pointers it is possible to move through the list in both directions (unlike the Singly-Linked Lists which only allows movement through the list in the forward direction).

The data contained in each element can be either integer values, by using one of the Type Conversion Macros, or simply pointers to any type of data.

List elements are allocated from the slice allocator, which is more efficient than allocating elements individually.

Note that most of the GList functions expect to be passed a pointer to the first element in the list. The functions which insert elements return the new start of the list, which may have changed.

There is no function to create a GList. NULL is considered to be the empty list so you simply set a GList* to NULL.

To add elements, use g_list_append(), g_list_prepend(), g_list_insert() and g_list_insert_sorted().
To remove elements, use g_list_remove().
To find elements in the list use g_list_first(), g_list_last(), g_list_next(), g_list_previous(), g_list_nth(), g_list_nth_data(), g_list_nth_prev(), g_list_find() and g_list_find_custom().
To find the index of an element use g_list_position() and g_list_index().
To call a function for each element in the list use g_list_foreach().
To free the entire list, use g_list_free().

Details

GList

typedef struct {
    gpointer data;
    GList *next;
    GList *prev;
} GList;

The GList struct is used for each element in a doubly-linked list.

gpointer data; holds the element’s data, which can be a pointer to any kind of data, or any integer value using the Type Conversion Macros.
GList *next; contains the link to the next element in the list.

GList *prev; contains the link to the previous element in the list.

g_list_append ()

| GList* | g_list_append | (GList* list, gpointer data); |

Adds a new element on to the end of the list.

**NOTE**

The return value is the new start of the list, which may have changed, so make sure you store the new value.

**NOTE**

Note that g_list_append() has to traverse the entire list to find the end, which is inefficient when adding multiple elements. A common idiom to avoid the inefficiency is to prepend the elements and reverse the list when all elements have been added.

```c
/* Notice that these are initialized to the empty list. */
GList* list = NULL, *number_list = NULL;

/* This is a list of strings. */
list = g_list_append (list, "first");
list = g_list_append (list, "second");

/* This is a list of integers. */
number_list = g_list_append (number_list, GINT_TO_POINTER (27));
number_list = g_list_append (number_list, GINT_TO_POINTER (14));
```

**list:** a pointer to a GList

**data:** the data for the new element

**Returns:** the new start of the GList

g_list_prepend ()

| GList* | g_list_prepend | (GList* list, gpointer data); |

Adds a new element on to the start of the list.

**NOTE**

The return value is the new start of the list, which may have changed, so make sure you store the new value.
/* Notice that it is initialized to the empty list. */
GList *list = NULL;
list = g_list_prepend (list, "last");
list = g_list_prepend (list, "first");

list: a pointer to a GList
data: the data for the new element
Returns: the new start of the GList

g_list_insert ()

<table>
<thead>
<tr>
<th>GList*</th>
<th>g_list_insert</th>
</tr>
</thead>
<tbody>
<tr>
<td>{GList *list, gpointer data, gint position};</td>
<td></td>
</tr>
</tbody>
</table>

Inserts a new element into the list at the given position.

list: a pointer to a GList
data: the data for the new element
position: the position to insert the element. If this is negative, or is larger than the number of elements in the list, the new element is added on to the end of the list.
Returns: the new start of the GList

g_list_insert_before ()

<table>
<thead>
<tr>
<th>GList*</th>
<th>g_list_insert_before</th>
</tr>
</thead>
<tbody>
<tr>
<td>{GList *list, GList *sibling, gpointer data};</td>
<td></td>
</tr>
</tbody>
</table>

Inserts a new element into the list before the given position.

list: a pointer to a GList
Sibling: the list element before which the new element is inserted or NULL to insert at the end of the list
data: the data for the new element
Returns: the new start of the GList

g_list_insert_sorted ()

<table>
<thead>
<tr>
<th>GList*</th>
<th>g_list_insert_sorted</th>
</tr>
</thead>
<tbody>
<tr>
<td>{GList *list, gpointer data, GCompareFunc func};</td>
<td></td>
</tr>
</tbody>
</table>

Inserts a new element into the list, using the given comparison function to determine its position.

list: a pointer to a GList
data: the data for the new element
func: the function to compare elements in the list. It should return a number > 0 if the first parameter comes after the second parameter in the sort order.
Returns: the new start of the GList
CHAPTER 5. GLIB DATA TYPES

5.3. DOUBLY-LINKED LISTS

**g_list_remove()**

GList* g_list_remove (GList *list, gconstpointer data);

Removes an element from a GList. If two elements contain the same data, only the first is removed. If none of the elements contain the data, the GList is unchanged.

*list*: a GList  
*data*: the data of the element to remove  

*Returns*: the new start of the GList

**g_list_remove_link()**

GList* g_list_remove_link (GList *list, GList *llink);

Removes an element from a GList, without freeing the element. The removed element’s prev and next links are set to NULL, so that it becomes a self-contained list with one element.

*list*: a GList  
*llink*: an element in the GList  

*Returns*: the new start of the GList, without the element

**g_list_delete_link()**

GList* g_list_delete_link (GList *list, GList *link_);

Removes the node link_ from the list and frees it. Compare this to g_list_remove_link() which removes the node without freeing it.

*list*: a GList  
*link_*: node to delete from list  

*Returns*: the new head of list

**g_list_remove_all()**

GList* g_list_remove_all (GList *list, gconstpointer data);

Removes all list nodes with data equal to data. Returns the new head of the list. Contrast with g_list_remove() which removes only the first node matching the given data.

*list*: a GList  
*data*: data to remove  

*Returns*: new head of list
CHAPTER 5. GLIB DATA TYPES

5.3. DOUBLY-LINKED LISTS

**g_list_free()**

```c
void g_list_free (GList *list);
```

Frees all of the memory used by a GList. The freed elements are returned to the slice allocator.

**Note**

If list elements contain dynamically-allocated memory, they should be freed first.

**list**: a GList

**g_list_free_1()**

```c
void g_list_free_1 (GList *list);
```

Frees one GList element. It is usually used after `g_list_remove_link()`.

**list**: a GList element

**g_list_free1**

```c
#define g_list_free1
```

Another name for `g_list_free_1()`.

**g_list_length()**

```c
guint g_list_length (GList *list);
```

Gets the number of elements in a GList.

**Note**

This function iterates over the whole list to count its elements.

**list**: a GList

**Returns**: the number of elements in the GList
CHAPTER 5. GLIB DATA TYPES

5.3. DOUBLY-LINKED LISTS

\textbf{g_list_copy 0}

\begin{verbatim}
GList* g_list_copy (GList *list);
\end{verbatim}

Copies a \texttt{GList}.

\textbf{NOTE}

Note that this is a "shallow" copy. If the list elements consist of pointers to data, the
pointers are copied but the actual data is not.

\textit{list}: a \texttt{GList}

\textbf{Returns}: a copy of \textit{list}

\textbf{g_list_reverse 0}

\begin{verbatim}
GList* g_list_reverse (GList *list);
\end{verbatim}

Reverses a \texttt{GList}. It simply switches the next and prev pointers of each element.

\textit{list}: a \texttt{GList}

\textbf{Returns}: the start of the reversed \texttt{GList}

\textbf{g_list_sort 0}

\begin{verbatim}
GList* g_list_sort (GList *list,
    GCompareFunc compare_func);
\end{verbatim}

Sorts a \texttt{GList} using the given comparison function.

\textit{list}: a \texttt{GList}

\textit{compare_func}: the comparison function used to sort the \texttt{GList}. This function is passed the data from
2 elements of the \texttt{GList} and should return 0 if they are equal, a negative value if the first element
comes before the second, or a positive value if the first element comes after the second.

\textbf{Returns}: the start of the sorted \texttt{GList}

\textbf{GCompareFunc 0}

\begin{verbatim}
gint (*GCompareFunc) (gconstpointer a,
    gconstpointer b);
\end{verbatim}

Specifies the type of a comparison function used to compare two values. The function should return
a negative integer if the first value comes before the second, 0 if they are equal, or a positive integer if
the first value comes after the second.

\textit{a}: a value.

\textit{b}: a value to compare with.

\textbf{Returns}: negative value if \textit{a} < \textit{b}; zero if \textit{a} = \textit{b}; positive value if \textit{a} > \textit{b}. 

473
### g_list_insert_sorted_with_data ()

**Signature:**

```c
GList* g_list_insert_sorted_with_data (GList *list, gpointer data, GCompareDataFunc func, gpointer user_data);
```

**Purpose:** Inserts a new element into the list, using the given comparison function to determine its position.

- **list:** a pointer to a GList
- **data:** the data for the new element
- **func:** the function to compare elements in the list. It should return a number > 0 if the first parameter comes after the second parameter in the sort order.
- **user_data:** user data to pass to comparison function.

**Returns:** the new start of the GList

**Since 2.10**

### g_list_sort_with_data ()

**Signature:**

```c
GList* g_list_sort_with_data (GList *list, GCompareDataFunc compare_func, gpointer user_data);
```

**Purpose:** Like g_list_sort(), but the comparison function accepts a user data argument.

- **list:** a GList
- **compare_func:** comparison function
- **user_data:** user data to pass to comparison function

**Returns:** the new head of list

### GCompareDataFunc ()

**Signature:**

```c
gint (*GCompareDataFunc) (gconstpointer a, gconstpointer b, gpointer user_data);
```

**Purpose:** Specifies the type of a comparison function used to compare two values. The function should return a negative integer if the first value comes before the second, 0 if they are equal, or a positive integer if the first value comes after the second.

- **a:** a value.
- **b:** a value to compare with.
- **user_data:** user data to pass to comparison function.

**Returns:** negative value if \( a < b \); zero if \( a = b \); positive value if \( a > b \).

### g_list_concat ()

**Signature:**

```c
GList* g_list_concat (GList *list1, GList *list2);
```

**Purpose:** Adds the second GList onto the end of the first GList. Note that the elements of the second GList are not copied. They are used directly.

- **list1:** a GList
- **list2:** the GList to add to the end of the first GList

**Returns:** the start of the new GList
### CHAPTER 5. GLIB DATA TYPES

#### 5.3. DOUBLY-LINKED LISTS

**g_list_foreach()**

```c
void g_list_foreach (GList *list, GFunc func, gpointer user_data);
```

Calls a function for each element of a GList.

- **list**: a GList
- **func**: the function to call with each element’s data
- **user_data**: user data to pass to the function

**GFunc()**

```c
void (GFunc) (gpointer data, gpointer user_data);
```

Specifies the type of functions passed to `g_list_foreach()` and `g_slist_foreach()`.

- **data**: the element’s data.
- **user_data**: user data passed to `g_list_foreach()` or `g_slist_foreach()`.

**g_list_first()**

```c
GList* g_list_first (GList *list);
```

Gets the first element in a GList.

- **list**: a GList

**Returns**: the first element in the GList, or NULL if the GList has no elements

**g_list_last()**

```c
GList* g_list_last (GList *list);
```

Gets the last element in a GList.

- **list**: a GList

**Returns**: the last element in the GList, or NULL if the GList has no elements

**g_list_previous()**

```c
#define g_list_previous(list)
```

A convenience macro to get the previous element in a GList.

- **list**: an element in a GList

**Returns**: the previous element, or NULL if there are no previous elements.

**g_list_next()**

```c
#define g_list_next(list)
```

A convenience macro to get the next element in a GList.

- **list**: an element in a GList

**Returns**: the next element, or NULL if there are no more elements.
g_list_nth 0

GList* g_list_nth (GList *list, guint n);

Gets the element at the given position in a GList.

list: a GList

n: the position of the element, counting from 0

Returns: the element, or NULL if the position is off the end of the GList

g_list_nth_data 0

gpointer g_list_nth_data (GList *list, guint n);

Gets the data of the element at the given position.

list: a GList

n: the position of the element

Returns: the element’s data, or NULL if the position is off the end of the GList

g_list_nth_prev 0

GList* g_list_nth_prev (GList *list, guint n);

Gets the element n places before list.

list: a GList

n: the position of the element, counting from 0

Returns: the element, or NULL if the position is off the end of the GList

g_list_find 0

GList* g_list_find (GList *list, gconstpointer data);

Finds the element in a GList which contains the given data.

list: a GList

data: the element data to find

Returns: the found GList element, or NULL if it is not found

g_list_find_custom 0

GList* g_list_find_custom (GList *list, gconstpointer data, GCompareFunc func);

Finds an element in a GList, using a supplied function to find the desired element. It iterates over the list, calling the given function which should return 0 when the desired element is found. The function takes two gconstpointer arguments, the GList element’s data as the first argument and the given user data.

list: a GList
**data**: user data passed to the function

**func**: the function to call for each element. It should return 0 when the desired element is found

**Returns**: the found GList element, or NULL if it is not found

### g_list_position()

```c
#include <glib/glist.h>

gint g_list_position (GList *list, gpointer llink);
```

Gets the position of the given element in the GList (starting from 0).

**list**: a GList

**llink**: an element in the GList

**Returns**: the position of the element in the GList, or -1 if the element is not found

### g_list_index()

```c
#include <glib/glist.h>

gint g_list_index (GList *list, gconstpointer data);
```

Gets the position of the element containing the given data (starting from 0).

**list**: a GList

**data**: the data to find

**Returns**: the index of the element containing the data, or -1 if the data is not found

### g_list_push_allocator()

```c
#include <glib/glist.h>

void g_list_push_allocator (gpointer allocator);
```

**Warning**: g_list_push_allocator has been deprecated since version 2.10 and should not be used in newly-written code. It does nothing, since GList has been converted to the slice allocator

Sets the allocator to use to allocate GList elements. Use g_list_pop_allocator() to restore the previous allocator.

Note that this function is not available if GLib has been compiled with --disable-mem-pools

**allocator**: the GAllocator to use when allocating GList elements.

### g_list_pop_allocator()

```c
#include <glib/glist.h>

void g_list_pop_allocator (void);
```

**Warning**: g_list_pop_allocator has been deprecated since version 2.10 and should not be used in newly-written code. It does nothing, since GList has been converted to the slice allocator
5.4 Singly-Linked Lists

Name
Singly-Linked Lists – linked lists containing integer values or pointers to data, limited to iterating over the list in one direction

Synopsis

```c
#include <glib.h>

GSList;

GSList* g_slist_alloc;
GSList* g_slist_append (GList *list, gpointer data);
GSList* g_slist_prepend (GList *list, gpointer data);
GSList* g_slist_insert (GList *list, gpointer data, gint position);
GSList* g_slist_insert_before (GList *slist, GSList *sibling, gpointer data);
GSList* g_slist_insert_sorted (GList *list, gpointer data, GCompareFunc func);
GSList* g_slist_remove (GList *list, gconstpointer data);
GSList* g_slist_remove_link (GList *list, GSList *link_);
GSList* g_slist_delete_link (GList *list, GSList *link_);
GSList* g_slist_remove_all (GList *list, gconstpointer data);
void g_slist_free (GList *list);
void g_slist_free_1 (GList *list);
#define g_slist_free1

 guint g_slist_length (GList *list);
GSList* g_slist_copy (GList *list);
GSList* g_slist_reverse (GList *list);
GSList* g_slist_insert_sorted_with_data (GList *list, gpointer data, GCompareDataFunc func, gpointer user_data);
GSList* g_slist_sort (GList *list, GCompareFunc compare_func);
GSList* g_slist_sort_with_data (GList *list, GCompareDataFunc compare_func, gpointer user_data);
GSList* g_slist_concat (GList *list1, GList *list2);
void g_slist_foreach (GList *list, GListForEachFunc f);
```

Restores the previous GAllocator, used when allocating GList elements. Note that this function is not available if GLib has been compiled with `--disable-mem-pools`
CHAPTER 5. GLIB DATA TYPES  5.4. SINGLY-LINKED LISTS

GFunc func, gpointer user_data);

GList* g_slist_last (GList *list);

#define g_slist_next (slist)

GList* g_slist_nth (GList *list, guint n);
gpointer g_slist_nth_data (GList *list, guint n);

GList* g_slist_find (GList *list, gconstpointer data);

GList* g_slist_find_custom (GList *list, gconstpointer data, GCompareFunc func);

gint g_slist_position (GList *list, GSList *llink);
gint g_slist_index (GList *list, gconstpointer data);

void g_slist_push_allocator (gpointer dummy);
void g_slist_pop_allocator (void);

Description

The GList structure and its associated functions provide a standard singly-linked list data structure.

Each element in the list contains a piece of data, together with a pointer which links to the next element in the list. Using this pointer it is possible to move through the list in one direction only (unlike the Doubly-Linked Lists which allow movement in both directions).

The data contained in each element can be either integer values, by using one of the Type Conversion Macros, or simply pointers to any type of data.

List elements are allocated from the slice allocator, which is more efficient than allocating elements individually.

Note that most of the GList functions expect to be passed a pointer to the first element in the list. The functions which insert elements return the new start of the list, which may have changed.

There is no function to create a GList. NULL is considered to be the empty list so you simply set a GList* to NULL.

To add elements, use g_slist_append(), g_slist_prepend(), g_slist_insert() and g_slist_insert_sorted().
To remove elements, use g_slist_remove().
To find elements in the list use g_slist_last(), g_slist_next(), g_slist_nth(), g_slist_nth_data(), g_slist_find() and g_slist_find_custom().
To find the index of an element use g_slist_position() and g_slist_index().
To call a function for each element in the list use g_slist_foreach().
To free the entire list, use g_slist_free().

Details

GList

typedef struct {
    gpointer data;
    GList *next;
} GList;

The GList struct is used for each element in the singly-linked list.

gpointer data; holds the element’s data, which can be a pointer to any kind of data, or any integer value using the Type Conversion Macros.

GList *next; contains the link to the next element in the list.
5.4. SINGLY-LINKED LISTS

**g_slist_alloc()**

```c
GList* g_slist_alloc (void);
```

Allocates space for one GList element. It is called by the `g_slist_append()`, `g_slist_prepend()`, `g_slist_insert()` and `g_slist_insert_sorted()` functions and so is rarely used on its own.

**Returns**: a pointer to the newly-allocated GList element.

**g_slist_append()**

```c
GList* g_slist_append (GList *list, gpointer data);
```

Adds a new element on to the end of the list.

**Note**

The return value is the new start of the list, which may have changed, so make sure you store the new value.

**Note**

Note that `g_slist_append()` has to traverse the entire list to find the end, which is inefficient when adding multiple elements. A common idiom to avoid the inefficiency is to prepend the elements and reverse the list when all elements have been added.

```c
/* Notice that these are initialized to the empty list. */
GList *list = NULL, *number_list = NULL;

/* This is a list of strings. */
list = g_slist_append (list, "first");
list = g_slist_append (list, "second");

/* This is a list of integers. */
number_list = g_slist_append (number_list, GINT_TO_POINTER (27));
number_list = g_slist_append (number_list, GINT_TO_POINTER (14));
```

**list**: a GList

**data**: the data for the new element

**Returns**: the new start of the GList

**g_slist_prepend()**

```c
GList* g_slist_prepend (GList *list, gpointer data);
```

Adds a new element on to the start of the list.
CHAPTER 5. GLIB DATA TYPES  5.4. SINGLY-LINKED LISTS

NOTE

The return value is the new start of the list, which may have changed, so make sure you store the new value.

/* Notice that it is initialized to the empty list. */
GSList *list = NULL;
list = g_slist_prepend (list, "last");
list = g_slist_prepend (list, "first");

list: a GSList
data: the data for the new element

Returns: the new start of the GSList

g_slist_insert ()

GSLList* g_slist_insert (GSList *list, gpointer data, gint position);

Inserts a new element into the list at the given position.

list: a GSList
data: the data for the new element

position: the position to insert the element. If this is negative, or is larger than the number of elements in the list, the new element is added on to the end of the list.

Returns: the new start of the GSList

g_slist_insert_before ()

GSLList* g_slist_insert_before (GSList *slist, GSList *sibling, gpointer data);

Inserts a node before sibling containing data.

slist: a GSList
sibling: node to insert data before
data: data to put in the newly-inserted node

Returns: the new head of the list.

g_slist_insert_sorted ()

GSLList* g_slist_insert_sorted (GSList *list, gpointer data, GCompareFunc func);

Inserts a new element into the list, using the given comparison function to determine its position.

list: a GSList
**data**: the data for the new element

**func**: the function to compare elements in the list. It should return a number > 0 if the first parameter comes after the second parameter in the sort order.

**Returns**: the new start of the GSList

---

**g_slist_remove()**

```c
GList* g_slist_remove (GList *list, gconstpointer data);
```

Removes an element from a GSLIST. If two elements contain the same data, only the first is removed. If none of the elements contain the data, the GSLIST is unchanged.

**list**: a GSLIST

**data**: the data of the element to remove

**Returns**: the new start of the GSLIST

---

**g_slist_remove_link()**

```c
GList* g_slist_remove_link (GList *list, GSList *link_);
```

Removes an element from a GSLIST, without freeing the element. The removed element’s next link is set to NULL, so that it becomes a self-contained list with one element.

**list**: a GSLIST

**link_**: an element in the GSLIST

**Returns**: the new start of the GSLIST, without the element

---

**g_slist_delete_link()**

```c
GList* g_slist_delete_link (GList *list, GSList *link_);
```

Removes the node link_ from the list and frees it. Compare this to g_slist_remove_link() which removes the node without freeing it.

**list**: a GSLIST

**link_**: node to delete

**Returns**: the new head of list

---

**g_slist_remove_all()**

```c
GList* g_slist_remove_all (GList *list, gconstpointer data);
```

Removes all list nodes with data equal to data. Returns the new head of the list. Contrast with g_slist_remove() which removes only the first node matching the given data.

**list**: a GSLIST

**data**: data to remove

**Returns**: new head of list
**g_slist_free**

```c
void g_slist_free (GSList *list);
```

Frees all of the memory used by a GSList. The freed elements are returned to the slice allocator.

**list**: a GSList

**g_slist_free_1**

```c
void g_slist_free_1 (GSList *list);
```

Frees one GSList element. It is usually used after `g_slist_remove_link()`.

**list**: a GSList element

**g_slist_free1**

```c
#define g_slist_free1
```

A macro which does the same as `g_slist_free_1()`. Since 2.10

**g_slist_length**

```c
guint g_slist_length (GSList *list);
```

Gets the number of elements in a GSList.

**NOTE**

This function iterates over the whole list to count its elements.

**list**: a GSList

**Returns**: the number of elements in the GSList

**g_slist_copy**

```c
GSList* g_slist_copy (GSList *list);
```

Copies a GSList.

**NOTE**

Note that this is a "shallow" copy. If the list elements consist of pointers to data, the pointers are copied but the actual data isn’t.

**list**: a GSList

**Returns**: a copy of `list`
CHAPTER 5. GLIB DATA TYPES

5.4. SINGLY-LINKED LISTS

**g_slist_reverse()**

```c
GList* g_slist_reverse (GList *list);
```

Reverses a GList.

*list*: a GList

**Returns**: the start of the reversed GList

**g_slist_insert_sorted_with_data()**

```c
GList* g_slist_insert_sorted_with_data (GList *list,
                                           gpointer data,
                                           GCompareDataFunc func,
                                           gpointer user_data);
```

Inserts a new element into the list, using the given comparison function to determine its position.

*list*: a GList

*data*: the data for the new element

*func*: the function to compare elements in the list. It should return a number > 0 if the first parameter comes after the second parameter in the sort order.

*user_data*: data to pass to comparison function

**Returns**: the new start of the GList

Since 2.10

**g_slist_sort()**

```c
GList* g_slist_sort (GList *list,
                      GCompareFunc compare_func);
```

Sorts a GList using the given comparison function.

*list*: a GList

*compare_func*: the comparison function used to sort the GList. This function is passed the data from 2 elements of the GList and should return 0 if they are equal, a negative value if the first element comes before the second, or a positive value if the first element comes after the second.

**Returns**: the start of the sorted GList

**g_slist_sort_with_data()**

```c
GList* g_slist_sort_with_data (GList *list,
                               GCompareDataFunc func,
                               gpointer user_data);
```

Like `g_slist_sort()`, but the sort function accepts a user data argument.

*list*: a GList

*compare_func*: comparison function

*user_data*: data to pass to comparison function

**Returns**: new head of the list
g_slist_concat ()

```c
GList* g_slist_concat (GList *list1,
                        GList *list2);
```

Adds the second GList onto the end of the first GList. Note that the elements of the second GList are not copied. They are used directly.

- **list1**: a GList
- **list2**: the GList to add to the end of the first GList

**Returns**: the start of the new GList

---

**g_slist_foreach ()**

```c
void g_slist_foreach (GList *list,
                      GFunc func,
                      gpointer user_data);
```

Calls a function for each element of a GList.

- **list**: a GList
- **func**: the function to call with each element’s data
- **user_data**: user data to pass to the function

---

**g_slist_last ()**

```c
GList* g_slist_last (GList *list);
```

Gets the last element in a GList.

**NOTE**

This function iterates over the whole list.

- **list**: a GList

**Returns**: the last element in the GList, or NULL if the GList has no elements

---

**g_slist_next()**

```c
#define g_slist_next(slist)
```

A convenience macro to get the next element in a GList.

- **slist**: an element in a GList

**Returns**: the next element, or NULL if there are no more elements.

---

**g_slist_nth ()**

```c
GList* g_slist_nth (GList *list,
                    guint n);
```

Gets the element at the given position in a GList.

- **list**: a GList
- **n**: the position of the element, counting from 0

**Returns**: the element, or NULL if the position is off the end of the GList
CHAPTER 5. GLIB DATA TYPES

5.4. SINGLY-LINKED LISTS

**g_slist_nth_data ()**

```c
gpointer g_slist_nth_data (GSList *list, guint n);
```

Gets the data of the element at the given position.

- **list**: a GSList
- **n**: the position of the element

**Returns**: the element’s data, or NULL if the position is off the end of the GSList

**g_slist_find ()**

```c
GSList* g_slist_find (GSList *list, gconstpointer data);
```

Finds the element in a GSList which contains the given data.

- **list**: a GSList
- **data**: the element data to find

**Returns**: the found GSList element, or NULL if it is not found

**g_slist_find_custom ()**

```c
GSList* g_slist_find_custom (GSList *list, gconstpointer data, GCompareFunc func);
```

Finds an element in a GSList, using a supplied function to find the desired element. It iterates over the list, calling the given function which should return 0 when the desired element is found. The function takes two gconstpointer arguments, the GSList element’s data as the first argument and the given user data.

- **list**: a GSList
- **data**: user data passed to the function
- **func**: the function to call for each element. It should return 0 when the desired element is found

**Returns**: the found GSList element, or NULL if it is not found

**g_slist_position ()**

```c
gint g_slist_position (GSList *list, GSList *llink);
```

Gets the position of the given element in the GSList (starting from 0).

- **list**: a GSList
- **llink**: an element in the GSList

**Returns**: the position of the element in the GSList, or -1 if the element is not found
CHAPTER 5. GLIB DATA TYPES

5.5. DOUBLE-ENDED QUEUES

**g_slist_index()**

```
gint g_slist_index (GSList *list, gconstpointer data);
```

Gets the position of the element containing the given data (starting from 0).

- **list**: a GSList
- **data**: the data to find
- **Returns**: the index of the element containing the data, or -1 if the data is not found

**g_slist_push_allocator()**

```
void g_slist_push_allocator (gpointer dummy);
```

**WARNING**

*has been deprecated since version 2.10 and should not be used in newly-written code. It does nothing, since GSList has been converted to the slice allocator*

Sets the allocator to use to allocate GSList elements. Use `g_slist_pop_allocator()` to restore the previous allocator.

- **dummy**: the GAllocator to use when allocating GSList elements.

**g_slist_pop_allocator()**

```
void g_slist_pop_allocator (void);
```

**WARNING**

*has been deprecated since version 2.10 and should not be used in newly-written code. It does nothing, since GSList has been converted to the slice allocator*

Restores the previous GAllocator, used when allocating GSList elements.

- **Note**: this function is not available if GLib has been compiled with `--disable-mem-pools`

5.5 Double-ended Queues

**Name**

Double-ended Queues – double-ended queue data structure
Synopsis

#include <glib.h>

GQueue;
GQueue* g_queue_new (void);
void g_queue_free (GQueue *queue);
#define G_QUEUE_INIT
void g_queue_init (GQueue *queue);
void g_queue_clear (GQueue *queue);
gboolean g_queue_is_empty (GQueue *queue);
uint g_queue_get_length (GQueue *queue);
void g_queue_reverse (GQueue *queue);
GQueue * g_queue_copy (GQueue *queue);
void g_queue_foreach (GQueue *queue, GFunc func, gpointer user_data);
GList * g_queue_find (GQueue *queue, gconstpointer data);
GList * g_queue_find_custom (GQueue *queue, gconstpointer data, GCompareFunc func);
void g_queue_sort (GQueue *queue, GCompareDataFunc compare_func, gpointer user_data);
void g_queue_push_head (GQueue *queue, gpointer data);
void g_queue_push_tail (GQueue *queue, gpointer data);
void g_queue_push_nth (GQueue *queue, gpointer data, guint n);
gpointer g_queue_pop_head (GQueue *queue);
gpointer g_queue_pop_tail (GQueue *queue);
gpointer g_queue_pop_nth (GQueue *queue, guint n);
gpointer g_queue_peek_head (GQueue *queue);
gpointer g_queue_peek_tail (GQueue *queue);
gpointer g_queue_peek_nth (GQueue *queue, guint n);
guint g_queue_index (GQueue *queue, gconstpointer data);
void g_queue_remove (GQueue *queue, gconstpointer data);
void g_queue_remove_all (GQueue *queue, gconstpointer data);
void g_queue_insert_before (GQueue *queue, GList *sibling, gpointer data);
void g_queue_insert_after (GQueue *queue, GList *sibling, gpointer data);
void g_queue_insert_sorted (GQueue *queue, gpointer data, GCompareDataFunc func, gpointer user_data);
void g_queue_push_head_link (GQueue *queue, GList *link_);
CHAPTER 5. GLIB DATA TYPES

5.5. DOUBLE-ENDED QUEUES

void g_queue_push_tail_link (GQueue *queue,
   GList *link_);

void g_queue_push_nth_link (GQueue *queue,
   gint n,
   GList *link_);

GList* g_queue_pop_head_link (GQueue *queue);

GList* g_queue_pop_tail_link (GQueue *queue);

GList* g_queue_pop_nth_link (GQueue *queue,
   guint n);

GList* g_queue_peek_head_link (GQueue *queue);

GList* g_queue_peek_tail_link (GQueue *queue);

GList* g_queue_peek_nth_link (GQueue *queue,
   guint n);

gint g_queue_link_index (GQueue *queue,
   GList *link_);

void g_queue_unlink (GQueue *queue,
   GList *link_);

void g_queue_delete_link (GQueue *queue,
   GList *link_);

Description

The GQueue structure and its associated functions provide a standard queue data structure. Internally,
GQueue uses the same data structure as GList to store elements.

The data contained in each element can be either integer values, by using one of the Type Conversion
Macros, or simply pointers to any type of data.

To create a new GQueue, use g_queue_new().

To initialize a statically-allocated GQueue, use G_QUEUE_INIT or g_queue_init().

To add elements, use g_queue_push_head(), g_queue_push_head_link(), g_queue_push_tail() and
g_queue_push_tail_link().

To remove elements, use g_queue_pop_head() and g_queue_pop_tail().

To free the entire queue, use g_queue_free().

Details

GQueue
typedef struct {
   GList *head;
   GList *tail;
   guint length;
} GQueue;

Contains the public fields of a Queue.

GList *head; a pointer to the first element of the queue.

GList *tail; a pointer to the last element of the queue.

guint length; the number of elements in the queue.

g_queue_new()

GQueue* g_queue_new (void);

Creates a new GQueue.

Returns: a new GQueue.
CHAPTER 5. GLIB DATA TYPES

5.5. DOUBLE-ENDED QUEUES

**g_queue_free()**

```c
void g_queue_free (GQueue *queue);
```

Frees the memory allocated for the GQueue. Only call this function if `queue` was created with `g_queue_new()`. If queue elements contain dynamically-allocated memory, they should be freed first.

- **queue**: a GQueue.

**G_QUEUE_INIT**

```c
#define G_QUEUE_INIT { NULL, NULL, 0 }
```

A statically-allocated GQueue must be initialized with this macro before it can be used. This macro can be used to initialize a variable, but it cannot be assigned to a variable. In that case you have to use `g_queue_init()`.

```c
GQueue my_queue = G_QUEUE_INIT;
```

Since 2.14

**g_queue_init()**

```c
void g_queue_init (GQueue *queue);
```

A statically-allocated GQueue must be initialized with this function before it can be used. Alternatively you can initialize it with `G_QUEUE_INIT`. It is not necessary to initialize queues created with `g_queue_new()`.

- **queue**: an uninitialized GQueue

Since 2.14

**g_queue_clear()**

```c
void g_queue_clear (GQueue *queue);
```

Removes all the elements in `queue`. If queue elements contain dynamically-allocated memory, they should be freed first.

- **queue**: a GQueue

Since 2.14

**g_queue_is_empty()**

```c
gboolean g_queue_is_empty (GQueue *queue);
```

Returns `TRUE` if the queue is empty.

- **queue**: a GQueue.

Returns: `TRUE` if the queue is empty.

**g_queue_get_length()**

```c
guint g_queue_get_length (GQueue *queue);
```

Returns the number of items in `queue`.

- **queue**: a GQueue

Returns: The number of items in `queue`.

Since 2.4
g_queue_reverse ()

```c
void g_queue_reverse (GQueue *queue);
```

Reverses the order of the items in queue.

**queue**: a GQueue

Since 2.4

g_queue_copy ()

```c
GQueue * g_queue_copy (GQueue *queue);
```

Copies a queue. Note that is a shallow copy. If the elements in the queue consist of pointers to data, the pointers are copied, but the actual data is not.

**queue**: a GQueue

**Returns**: A copy of queue

Since 2.4

g_queue_foreach ()

```c
void g_queue_foreach (GQueue *queue, GFunc func, gpointer user_data);
```

Calls func for each element in the queue passing user_data to the function.

**queue**: a GQueue

**func**: the function to call for each element’s data

**user_data**: user data to pass to func

Since 2.4

g_queue_find ()

```c
GList * g_queue_find (GQueue *queue, gconstpointer data);
```

Finds the first link in queue which contains data.

**queue**: a GQueue

**data**: data to find

**Returns**: The first link in queue which contains data.

Since 2.4

g_queue_find_custom ()

```c
GList * g_queue_find_custom (GQueue *queue, gconstpointer data, GCompareFunc func);
```

Finds an element in a GQueue, using a supplied function to find the desired element. It iterates over the queue, calling the given function which should return 0 when the desired element is found. The function takes two gconstpointer arguments, the GQueue element’s data as the first argument and the given user data as the second argument.
**queue**: a GQueue

**data**: user data passed to *func*

**func**: a GCompareFunc to call for each element. It should return 0 when the desired element is found

**Returns**: The found link, or NULL if it wasn’t found

Since 2.4

### g_queue_sort()

```c
void g_queue_sort (GQueue *queue,
                  GCompareDataFunc compare_func,
                  gpointer user_data);
```

Sorts *queue* using *compare_func*.

**queue**: a GQueue

**compare_func**: the GCompareDataFunc used to sort *queue*. This function is passed two elements of the *queue* and should return 0 if they are equal, a negative value if the first comes before the second, and a positive value if the second comes before the first.

**user_data**: user data passed to *compare_func*

Since 2.4

### g_queue_push_head()

```c
void g_queue_push_head (GQueue *queue,
                        gpointer data);
```

Adds a new element at the head of the queue.

**queue**: a GQueue.

**data**: the data for the new element.

### g_queue_push_tail()

```c
void g_queue_push_tail (GQueue *queue,
                        gpointer data);
```

Adds a new element at the tail of the queue.

**queue**: a GQueue.

**data**: the data for the new element.

### g_queue_push_nth()

```c
void g_queue_push_nth (GQueue *queue,
                       gpointer data,
                       gint n);
```

Inserts a new element into *queue* at the given position

**queue**: a GQueue

**data**: the data for the new element

**n**: the position to insert the new element. If *n* is negative or larger than the number of elements in the *queue*, the element is added to the end of the queue.

Since 2.4
5.5. DOUBLE-ENDED QUEUES

`g_queue_pop_head ()`

```c
gpointer g_queue_pop_head (GQueue *queue);
```

Removes the first element of the queue.

*queue*: a `GQueue`.

*Returns*: the data of the first element in the queue, or `NULL` if the queue is empty.

`g_queue_pop_tail ()`

```c
gpointer g_queue_pop_tail (GQueue *queue);
```

Removes the last element of the queue.

*queue*: a `GQueue`.

*Returns*: the data of the last element in the queue, or `NULL` if the queue is empty.

`g_queue_pop_nth ()`

```c
gpointer g_queue_pop_nth (GQueue *queue, guint n);
```

Removes the *n*’th element of *queue*.

*queue*: a `GQueue`  

*n*: the position of the element.

*Returns*: the element’s data, or `NULL` if *n* is off the end of *queue*.

Since 2.4

`g_queue_peek_head ()`

```c
gpointer g_queue_peek_head (GQueue *queue);
```

Returns the first element of the queue.

*queue*: a `GQueue`.

*Returns*: the data of the first element in the queue, or `NULL` if the queue is empty.

`g_queue_peek_tail ()`

```c
gpointer g_queue_peek_tail (GQueue *queue);
```

Returns the last element of the queue.

*queue*: a `GQueue`.

*Returns*: the data of the last element in the queue, or `NULL` if the queue is empty.

`g_queue_peek_nth ()`

```c
gpointer g_queue_peek_nth (GQueue *queue, guint n);
```

Returns the *n*’th element of *queue*.

*queue*: a `GQueue`  

*n*: the position of the element.

*Returns*: The data for the *n*’th element of *queue*, or `NULL` if *n* is off the end of *queue*.

Since 2.4
g_queue_index()

```c
gint g_queue_index (GQueue *queue, gconstpointer data);
```

Returns the position of the first element in `queue` which contains `data`.

**queue**: a GQueue

**data**: the data to find.

**Returns**: The position of the first element in `queue` which contains `data`, or -1 if no element in `queue` contains `data`.

Since 2.4

---

g_queue_remove()

```c
void g_queue_remove (GQueue *queue, gconstpointer data);
```

Removes the first element in `queue` that contains `data`.

**queue**: a GQueue

**data**: data to remove.

Since 2.4

---

g_queue_remove_all()

```c
void g_queue_remove_all (GQueue *queue, gconstpointer data);
```

Remove all elements in `queue` which contains `data`.

**queue**: a GQueue

**data**: data to remove

Since 2.4

---

g_queue_insert_before()

```c
void g_queue_insert_before (GQueue *queue, GList *sibling, gpointer data);
```

Inserts `data` into `queue` before `sibling`. `sibling` must be part of `queue`.

**queue**: a GQueue

**sibling**: a GList link that must be part of `queue`

**data**: the data to insert

Since 2.4
g_queue_insert_after ()

```c
void g_queue_insert_after (GQueue *queue,
                           GList *sibling,
                           gpointer data);
```

Inserts `data` into `queue` after `sibling`. `sibling` must be part of `queue`.

- **queue**: a GQueue
- **sibling**: a GList link that must be part of `queue`
- **data**: the data to insert

Since 2.4

---

g_queue_insert_sorted ()

```c
void g_queue_insert_sorted (GQueue *queue,
                           gpointer data,
                           GCompareDataFunc func,
                           gpointer user_data);
```

Inserts `data` into `queue` using `func` to determine the new position.

- **queue**: a GQueue
- **data**: the data to insert
- **func**: the GCompareDataFunc used to compare elements in the queue. It is called with two elements of the `queue` and `user_data`. It should return 0 if the elements are equal, a negative value if the first element comes before the second, and a positive value if the second element comes before the first.
- **user_data**: user data passed to `func`.

Since 2.4

---

g_queue_push_head_link ()

```c
void g_queue_push_head_link (GQueue *queue,
                             GList *link_);
```

Adds a new element at the head of the queue.

- **queue**: a GQueue.
- **link_**: a single GList element, not a list with more than one element.

---

g_queue_push_tail_link ()

```c
void g_queue_push_tail_link (GQueue *queue,
                            GList *link_);
```

Adds a new element at the tail of the queue.

- **queue**: a GQueue.
- **link_**: a single GList element, not a list with more than one element.
g_queue_push_nth_link ()

```c
void g_queue_push_nth_link (GQueue *queue, gint n, GList *link_);
```

Inserts link into queue at the given position.

**queue**: a GQueue

**n**: the position to insert the link. If this is negative or larger than the number of elements in queue, the link is added to the end of queue.

**link_**: the link to add to queue

Since 2.4

---

g_queue_pop_head_link ()

```c
GList* g_queue_pop_head_link (GQueue *queue);
```

Removes the first element of the queue.

**queue**: a GQueue.

**Returns**: the GList element at the head of the queue, or NULL if the queue is empty.

---

g_queue_pop_tail_link ()

```c
GList* g_queue_pop_tail_link (GQueue *queue);
```

Removes the last element of the queue.

**queue**: a GQueue.

**Returns**: the GList element at the tail of the queue, or NULL if the queue is empty.

---

g_queue_pop_nth_link ()

```c
GList* g_queue_pop_nth_link (GQueue *queue, guint n);
```

Removes and returns the link at the given position.

**queue**: a GQueue

**n**: the link’s position

**Returns**: The n’th link, or NULL if n is off the end of queue.

Since 2.4

---

g_queue.peek_head_link ()

```c
GList* g_queue.peek_head_link (GQueue *queue);
```

Returns the first link in queue

**queue**: a GQueue

**Returns**: the first link in queue, or NULL if queue is empty

Since 2.4
5.5. DOUBLE-ENDED QUEUES

**g_queue_peek_tail_link**

```c
GList* g_queue_peek_tail_link (GQueue *queue);
```

Returns the last link in `queue`.

- **queue**: a GQueue
- **Returns**: the last link in `queue`, or NULL if `queue` is empty

Since 2.4

**g_queue_peek_nth_link**

```c
GList* g_queue_peek_nth_link (GQueue *queue, guint n);
```

Returns the link at the given position

- **queue**: a GQueue
- **n**: the position of the link
- **Returns**: The link at the `n`'th position, or NULL if `n` is off the end of the list

Since 2.4

**g_queue_link_index**

```c
gint g_queue_link_index (GQueue *queue, GList *link_);
```

Returns the position of `link_` in `queue`.

- **queue**: a GQueue
- **link_**: A GList link
- **Returns**: The position of `link_`, or -1 if the link is not part of `queue`

Since 2.4

**g_queue_unlink**

```c
void g_queue_unlink (GQueue *queue, GList *link_);
```

Unlinks `link_` so that it will no longer be part of `queue`. The link is not freed.

- **queue**: a GQueue
- **link_**: a GList link that must be part of `queue`

Since 2.4

**g_queue_delete_link**

```c
void g_queue_delete_link (GQueue *queue, GList *link_);
```

Removes `link_` from `queue` and frees it.

- **queue**: a GQueue
- **link_**: a GList link that must be part of `queue`

Since 2.4
5.6 Sequences

Name
Sequences – scalable lists

Synopsis

#include <glib.h>

typedef GSequence;

typedef GSequenceIter;

gint (*GSequenceIterCompareFunc) (GSequenceIter *a,
               GSequenceIter *b,
               gpointer data);

GSequence * g_sequence_new (GDestroyNotify data_destroy);
void g_sequence_free (GSequence *seq);
 gint g_sequence_get_length (GSequence *seq);
void g_sequence_foreach (GSequence *seq,
               GFunc func,
               gpointer user_data);
void g_sequence_foreach_range (GSequenceIter *begin,
               GSequenceIter *end,
               GFunc func,
               gpointer user_data);
void g_sequence_sort (GSequence *seq,
               GCompareDataFunc cmp_func,
               gpointer cmp_data);
void g_sequence_sort_iter (GSequence *seq,
               GSequenceIterCompareFunc cmp_func,
               gpointer cmp_data);

GSequenceIter * g_sequence_get_begin_iter (GSequence *seq);
GSequenceIter * g_sequence_get_end_iter (GSequence *seq);
GSequenceIter * g_sequence_get_iter_at_pos (GSequence *seq,
               gint pos);
GSequenceIter * g_sequence_append (GSequence *seq,
               gpointer data);
GSequenceIter * g_sequence_prepend (GSequence *seq,
               gpointer data);
GSequenceIter * g_sequence_insert_before (GSequenceIter *iter,
               gpointer data);
void g_sequence_move (GSequenceIter *src,
               GSequenceIter *dest);
void g_sequence_swap (GSequenceIter *a,
               GSequenceIter *b);
GSequenceIter * g_sequence_insert_sorted (GSequence *seq,
               gpointer data,
               GCompareDataFunc cmp_func,
               gpointer cmp_data);
GSequenceIter * g_sequence_insert_sorted_iter (GSequence *seq,
               gpointer data,
               GSequenceIterCompareFunc iter_cmp,
               gpointer cmp_data);

void g_sequence_sort_changed (GSequenceIter *iter,
               GCompareDataFunc cmp_func,
               gpointer cmp_data);

### Description

The **GSequence** data structure has the API of a list, but is implemented internally with a balanced binary tree. This means that it is possible to maintain a sorted list of n elements in time O(n \log n). The data contained in each element can be either integer values, by using of the Type Conversion Macros, or simply pointers to any type of data.

A **GSequence** is accessed through **iterators**, represented by a **GSequenceIter**. An iterator represents a position between two elements of the sequence. For example, the `begin` iterator represents the gap immediately before the first element of the sequence, and the `end` iterator represents the gap immediately after the last element. In an empty sequence, the `begin` and `end` iterators are the same.

Some methods on **GSequence** operate on ranges of items. For example `g_sequence_foreach_range()` will call a user-specified function on each element with the given range. The range is delimited by the gaps represented by the passed-in iterators, so if you pass in the `begin` and `end` iterators, the range in question is the entire sequence.

The function `g_sequence_get()` is used with an iterator to access the element immediately following the gap that the iterator represents. The iterator is said to **point** to that element.

Iterators are stable across most operations on a **GSequence**. For example an iterator pointing to some element of a sequence will continue to point to that element even after the sequence is sorted. Even moving an element to another sequence using for example `g_sequence_move_range()` will not invalidate the iterators pointing to it. The only operation that will invalidate an iterator is when the element it points to is removed from any sequence.
Details

GSequence

typedef struct _GSequence GSequence;

The GSequence struct is an opaque data type representing a Sequence data type.

GSequenceIter

typedef struct _GSequenceNode GSequenceIter;

The GSequenceIter struct is an opaque data type representing an iterator pointing into a GSequence.

GSequenceIterCompareFunc()

gint (*GSequenceIterCompareFunc)(GSequenceIter *a, GSequenceIter *b, gpointer data);

A GSequenceIterCompareFunc is a function used to compare iterators. It must return zero if the iterators compare equal, a negative value if a comes before b, and a positive value if b comes before a.

a: a GSequenceIter
b: a GSequenceIter
data: user data

Returns: zero if the iterators are equal, a negative value if a comes before b, and a positive value if b comes before a.

g_sequence_new()

GSequence * g_sequence_new (GDestroyNotify data_destroy);

Creates a new GSequence. The data_destroy function, if non-NULL will be called on all items when the sequence is destroyed and on items that are removed from the sequence.

data_destroy: a GDestroyNotify function, or NULL

Returns: a new GSequence

Since 2.14

g_sequence_free()

void g_sequence_free (GSequence *seq);

Frees the memory allocated for seq. If seq has a data destroy function associated with it, that function is called on all items in seq.

seq: a GSequence

Since 2.14

g_sequence_get_length()

gint g_sequence_get_length (GSequence *seq);

Returns the length of seq

seq: a GSequence

Returns: the length of seq

Since 2.14
CHAPTER 5. GLIB DATA TYPES

5.6. SEQUENCES

### g_sequence_foreach ()

```c
void g_sequence_foreach (GSequence *seq, GFunc func, gpointer user_data);
```

Calls `func` for each item in the sequence passing `user_data` to the function.

- **seq**: a `GSequence`
- **func**: the function to call for each item in `seq`
- **user_data**: user data passed to `func`

Since 2.14

### g_sequence_foreach_range ()

```c
void g_sequence_foreach_range (GSequenceIter *begin, GSequenceIter *end, GFunc func, gpointer user_data);
```

Calls `func` for each item in the range `(begin, end)` passing `user_data` to the function.

- **begin**: a `GSequenceIter`
- **end**: a `GSequenceIter`
- **func**: a `GFunc`
- **user_data**: user data passed to `func`

Since 2.14

### g_sequence_sort ()

```c
void g_sequence_sort (GSequence *seq, GCompareDataFunc cmp_func, gpointer cmp_data);
```

Sorts `seq` using `cmp_func`.

- **seq**: a `GSequence`
- **cmp_func**: the `GCompareDataFunc` used to sort `seq`. This function is passed two items of `seq` and should return 0 if they are equal, a negative value if the first comes before the second, and a positive value if the second comes before the first.
- **cmp_data**: user data passed to `cmp_func`

Since 2.14

### g_sequence_sort_iter ()

```c
void g_sequence_sort_iter (GSequence *seq, GSequenceIterCompareFunc cmp_func, gpointer cmp_data);
```

Like `g_sequence_sort()`, but uses a `GSequenceIterCompareFunc` instead of a `GCompareDataFunc` as the compare function.

- **seq**: a `GSequence`
**cmp_func**: the GSequenceItercompare used to compare iterators in the sequence. It is called with two iterators pointing into `seq`. It should return 0 if the iterators are equal, a negative value if the first iterator comes before the second, and a positive value if the second iterator comes before the first.

**cmp_data**: user data passed to `cmp_func`  
Since 2.14

---

**g_sequence_get_begin_iter ()**

```c
GSequenceIter * g_sequence_get_begin_iter (GSequence *seq);
```

Returns the begin iterator for `seq`.

**seq**: a GSequence

**Returns**: the begin iterator for `seq`  
Since 2.14

---

**g_sequence_get_end_iter ()**

```c
GSequenceIter * g_sequence_get_end_iter (GSequence *seq);
```

Returns the end iterator for `seq`.

**seq**: a GSequence

**Returns**: the end iterator for `seq`  
Since 2.14

---

**g_sequence_get_iter_at_pos ()**

```c
GSequenceIter * g_sequence_get_iter_at_pos (GSequence *seq, gint pos);
```

Returns the iterator at position `pos`. If `pos` is negative or larger than the number of items in `seq`, the end iterator is returned.

**seq**: a GSequence

**pos**: a position in `seq`, or -1 for the end.

**Returns**: The GSequenceIter at position `pos`  
Since 2.14

---

**g_sequence_append ()**

```c
GSequenceIter * g_sequence_append (GSequence *seq, gpointer data);
```

Adds a new item to the end of `seq`.

**seq**: a GSequencePointer

**data**: the data for the new item

**Returns**: an iterator pointing to the new item  
Since 2.14
**g_sequence_prepend ()**

GSequenceIter * g_sequence_prepend (GSequence *seq, gpointer data);

Adds a new item to the front of `seq`:

- **seq**: a GSequence
- **data**: the data for the new item
- **Returns**: an iterator pointing to the new item

Since 2.14

**g_sequence_insert_before ()**

GSequenceIter * g_sequence_insert_before (GSequenceIter *iter, gpointer data);

Inserts a new item just before the item pointed to by `iter`:

- **iter**: a GSequenceIter
- **data**: the data for the new item
- **Returns**: an iterator pointing to the new item

Since 2.14

**g_sequence_move ()**

void g_sequence_move (GSequenceIter *src, GSequenceIter *dest);

Moves the item pointed to by `src` to the position indicated by `dest`. After calling this function `dest` will point to the position immediately after `src`. It is allowed for `src` and `dest` to point into different sequences:

- **src**: a GSequenceIter pointing to the item to move
- **dest**: a GSequenceIter pointing to the position to which the item is moved.

Since 2.14

**g_sequence_swap ()**

void g_sequence_swap (GSequenceIter *a, GSequenceIter *b);

Swaps the items pointed to by `a` and `b`. It is allowed for `a` and `b` to point into difference sequences:

- **a**: a GSequenceIter
- **b**: a GSequenceIter

Since 2.14
g_sequence_insert_sorted ()

Inserts data into sequence using func to determine the new position. The sequence must already be sorted according to cmp_func; otherwise the new position of data is undefined.

seq: a GSequence
data: the data to insert
cmp_func: the GCompareDataFunc used to compare items in the sequence. It is called with two items of the seq and user_data. It should return 0 if the items are equal, a negative value if the first item comes before the second, and a positive value if the second item comes before the first.
cmp_data: user data passed to cmp_func.

Returns: a GSequenceIter pointing to the new item.

Since 2.14

g_sequence_insert_sorted_iter ()

Like g_sequence_insert_sorted(), but uses a GSequenceIterCompareFunc instead of a GCompareDataFunc as the compare function.

seq: a GSequence
data: data for the new item
iter_cmp: the GSequenceIterCompare used to compare iterators in the sequence. It is called with two iterators pointing into seq. It should return 0 if the iterators are equal, a negative value if the first iterator comes before the second, and a positive value if the second iterator comes before the first.
cmp_data: user data passed to cmp_func

Returns: a GSequenceIter pointing to the new item

Since 2.14

g_sequence_sort_changed ()

Moves the data pointed to a new position as indicated by cmp_func. This function should be called for items in a sequence already sorted according to cmp_func whenever some aspect of an item changes so that cmp_func may return different values for that item.

iter: A GSequenceIter
cmp_func: the GCompareDataFunc used to compare items in the sequence. It is called with two items of the seq and user_data. It should return 0 if the items are equal, a negative value if the first item comes before the second, and a positive value if the second item comes before the first.
cmp_data: user data passed to cmp_func.

Since 2.14
CHAPTER 5. GLIB DATA TYPES 5.6. SEQUENCES

**g_sequence_sort_changed_iter()**

```c
void g_sequence_sort_changed_iter (GSequenceIter *iter,
                                   GSequenceIterCompareFunc iter_cmp,
                                   gpointer cmp_data);
```

Like `g_sequence_sort_changed()`, but uses a `GSequenceIterCompareFunc` instead of a `GCompareDataFunc` as the compare function.

**iter**: a `GSequenceIter`

**iter_cmp**: the `GSequenceItercompare` used to compare iterators in the sequence. It is called with two iterators pointing into `seq`. It should return 0 if the iterators are equal, a negative value if the first iterator comes before the second, and a positive value if the second iterator comes before the first.

**cmp_data**: user data passed to `cmp_func`

Since 2.14

**g_sequence_remove()**

```c
void g_sequence_remove (GSequenceIter *iter);
```

Removes the item pointed to by `iter`. It is an error to pass the end iterator to this function.

If the sequence has a data destroy function associated with it, this function is called on the data for the removed item.

**iter**: a `GSequenceIter`

Since 2.14

**g_sequence_remove_range()**

```c
void g_sequence_remove_range (GSequenceIter *begin,
                              GSequenceIter *end);
```

Removes all items in the `(begin, end)` range.

If the sequence has a data destroy function associated with it, this function is called on the data for the removed items.

**begin**: a `GSequenceIter`

**end**: a `GSequenceIter`

Since 2.14

**g_sequence_move_range()**

```c
void g_sequence_move_range (GSequenceIter *dest,
                           GSequenceIter *begin,
                           GSequenceIter *end);
```

Inserts the `(begin, end)` range at the destination pointed to by `ptr`. The `begin` and `end` iters must point into the same sequence. It is allowed for `dest` to point to a different sequence than the one pointed into by `begin` and `end`.

If `dest` is NULL, the range indicated by `begin` and `end` is removed from the sequence. If `dest` iters points to a place within the `(begin, end)` range, the range does not move.

**dest**: a `GSequenceIter`

**begin**: a `GSequenceIter`

**end**: a `GSequenceIter`

Since 2.14
5.6. SEQUENCES

**g_sequence_search()**

```c
GSequenceIter * g_sequence_search (GSequence *seq, gpointer data, GCompareDataFunc cmp_func, gpointer cmp_data);
```

Returns an iterator pointing to the position where `data` would be inserted according to `cmp_func` and `cmp_data`.

- **seq**: a `GSequence`
- **data**: data for the new item
- **cmp_func**: the `GCompareDataFunc` used to compare items in the sequence. It is called with two items of the `seq` and `user_data`. It should return 0 if the items are equal, a negative value if the first item comes before the second, and a positive value if the second item comes before the first.
- **cmp_data**: user data passed to `cmp_func`.

**Returns**: an `GSequenceIter` pointing to the position where `data` would have been inserted according to `cmp_func` and `cmp_data`.

Since 2.14

**g_sequence_search_iter()**

```c
GSequenceIter * g_sequence_search_iter (GSequence *seq, gpointer data, GSequenceIterCompareFunc iter_cmp, gpointer cmp_data);
```

Like `g_sequence_search()`, but uses a `GSequenceIterCompareFunc` instead of a `GCompareDataFunc` as the compare function.

- **seq**: a `GSequence`
- **data**: data for the new item
- **iter_cmp**: the `GSequenceIterCompare` function used to compare iterators in the sequence. It is called with two iterators pointing into `seq`. It should return 0 if the iterators are equal, a negative value if the first iterator comes before the second, and a positive value if the second iterator comes before the first.
- **cmp_data**: user data passed to `iter_cmp`.

**Returns**: a `GSequenceIter` pointing to the position in `seq` where `data` would have been inserted according to `iter_cmp` and `cmp_data`.

Since 2.14

**g_sequence_get()**

```c
gpointer g_sequence_get (GSequenceIter *iter);
```

Returns the data that `iter` points to.

- **iter**: a `GSequenceIter`.

**Returns**: the data that `iter` points to.

Since 2.14
CHAPTER 5. GLIB DATA TYPES

5.6. SEQUENCES

`g_sequence_set()`

```c
void g_sequence_set (GSequenceIter *iter, gpointer data);
```

Changes the data for the item pointed to by `iter` to be `data`. If the sequence has a data destroy function associated with it, that function is called on the existing data that `iter` pointed to.

`iter`: a GSequenceIter

`data`: new data for the item

Since 2.14

`g_sequence_iter_is_begin()`

```c
gboolean g_sequence_iter_is_begin (GSequenceIter *iter);
```

Returns whether `iter` is the begin iterator

`iter`: a GSequenceIter

Returns: whether `iter` is the begin iterator

Since 2.14

`g_sequence_iter_is_end()`

```c
gboolean g_sequence_iter_is_end (GSequenceIter *iter);
```

Returns whether `iter` is the end iterator

`iter`: a GSequenceIter

Returns: Whether `iter` is the end iterator.

Since 2.14

`g_sequence_iter_next()`

```c
GSequenceIter * g_sequence_iter_next (GSequenceIter *iter);
```

Returns an iterator pointing to the next position after `iter`. If `iter` is the end iterator, the end iterator is returned.

`iter`: a GSequenceIter

Returns: a GSequenceIter pointing to the next position after `iter`.

Since 2.14

`g_sequence_iter_prev()`

```c
GSequenceIter * g_sequence_iter_prev (GSequenceIter *iter);
```

Returns an iterator pointing to the previous position before `iter`. If `iter` is the begin iterator, the begin iterator is returned.

`iter`: a GSequenceIter

Returns: a GSequenceIter pointing to the previous position before `iter`.

Since 2.14
g_sequence_iter_get_position()

```c
gint g_sequence_iter_get_position (GSequenceIter *iter);
```

Returns the position of `iter`

**iter**: a GSequenceIter

**Returns**: the position of `iter`

Since 2.14

---

g_sequence_iter_move()

```c
GSequenceIter * g_sequence_iter_move (GSequenceIter *iter, gint delta);
```

Returns the GSequenceIter which is `delta` positions away from `iter`. If `iter` is closer than `-delta` positions to the beginning of the sequence, the begin iterator is returned. If `iter` is closer than `delta` positions to the end of the sequence, the end iterator is returned.

**iter**: a GSequenceIter

**delta**: A positive or negative number indicating how many positions away from `iter` the returned GSequenceIter will be.

**Returns**: a GSequenceIter which is `delta` positions away from `iter`.

Since 2.14

---

g_sequence_iter_get_sequence()

```c
GSequence * g_sequence_iter_get_sequence (GSequenceIter *iter);
```

Returns the GSequence that `iter` points into.

**iter**: a GSequenceIter

**Returns**: the GSequence that `iter` points into.

Since 2.14

---

g_sequence_iter_compare()

```c
gint g_sequence_iter_compare (GSequenceIter *a, GSequenceIter *b);
```

Returns a negative number if `a` comes before `b`, 0 if they are equal, and a positive number if `a` comes after `b`.

The `a` and `b` iterators must point into the same sequence.

**a**: a GSequenceIter

**b**: a GSequenceIter

**Returns**: A negative number if `a` comes before `b`, 0 if they are equal, and a positive number if `a` comes after `b`.

Since 2.14
5.7 Trash Stacks

Name
Trash Stacks – maintain a stack of unused allocated memory chunks

Synopsis

```c
#include <glib.h>

GTrashStack;

void g_trash_stack_push (GTrashStack **stack_p, gpointer data_p);
gpointer g_trash_stack_pop (GTrashStack **stack_p);
gpointer g_trash_stack_peek (GTrashStack **stack_p);
guint g_trash_stack_height (GTrashStack **stack_p);
```

Description
A GTrashStack is an efficient way to keep a stack of unused allocated memory chunks. Each memory chunk is required to be large enough to hold a gpointer. This allows the stack to be maintained without any space overhead, since the stack pointers can be stored inside the memory chunks.

There is no function to create a GTrashStack. A NULL GTrashStack* is a perfectly valid empty stack.

Details
GTrashStack

```c
typedef struct {
    GTrashStack *next;
} GTrashStack;
```

Each piece of memory that is pushed onto the stack is cast to a GTrashStack*.

**GTrashStack** *next:* pointer to the previous element of the stack, gets stored in the first `sizeof (gpointer)` bytes of the element.
5.8 Hash Tables

Name
Hash Tables – associations between keys and values so that given a key the value can be found quickly

Synopsis

```c
#include <glib.h>

GHashTable;
GHashTable* g_hash_table_new (GHashFunc hash_func, GEqualFunc key_equal_func);
GHashTable* g_hash_table_new_full (GHashFunc hash_func, GEqualFunc key_equal_func, GDestroyNotify key_destroy_func, GDestroyNotify value_destroy_func);

guint (*GHashFunc) (gconstpointer key);
```

`g_trash_stack_push()`

```c
void g_trash_stack_push (GTrashStack **stack_p, gpointer data_p);
```

Pushes a piece of memory onto a GTrashStack.

`stack_p`: a pointer to a GTrashStack.
`data_p`: the piece of memory to push on the stack.

`g_trash_stack_pop()`

```c
gpointer g_trash_stack_pop (GTrashStack **stack_p);
```

Pops a piece of memory off a GTrashStack.

`stack_p`: a pointer to a GTrashStack.

`Returns`: the element at the top of the stack.

`g_trash_stack_peek()`

```c
gpointer g_trash_stack_peek (GTrashStack **stack_p);
```

Returns the element at the top of a GTrashStack which may be NULL.

`stack_p`: a pointer to a GTrashStack.

`Returns`: the element at the top of the stack.

`g_trash_stack_height()`

```c
guint g_trash_stack_height (GTrashStack **stack_p);
```

Returns the height of a GTrashStack. Note that execution of this function is of O(N) complexity where N denotes the number of items on the stack.

`stack_p`: a pointer to a GTrashStack.

`Returns`: the height of the stack.
CHAPTER 5. GLIB DATA TYPES

5.8. HASH TABLES

gboolean (*GEqualFunc)(gconstpointer a, gconstpointer b);
void g_hash_table_insert(GHashTable *hash_table, gpointer key, gpointer value);
void g_hash_table_replace(GHashTable *hash_table, gpointer key, gpointer value);
guint g_hash_table_size(GHashTable *hash_table);
gpointer g_hash_table_lookup(GHashTable *hash_table, gconstpointer key);
gboolean g_hash_table_lookup_extended(GHashTable *hash_table, gconstpointer lookup_key, gpointer *orig_key, gpointer *value);
void g_hash_table_foreach(GHashTable *hash_table, GHFunc func, gpointer user_data);
gpointer g_hash_table_find(GHashTable *hash_table, GHRFunc predicate, gpointer user_data);
void (*GHFunc)(gpointer key, gpointer value, gpointer user_data);
gboolean g_hash_table_remove(GHashTable *hash_table, gconstpointer key);
gboolean g_hash_table_steal(GHashTable *hash_table, gconstpointer key);
guint g_hash_table_foreach_remove(GHashTable *hash_table, GHRFunc func, gpointer user_data);
guint g_hash_table_foreach_steal(GHashTable *hash_table, GHRFunc func, gpointer user_data);
void g_hash_table_remove_all(GHashTable *hash_table);
void g_hash_table_steal_all(GHashTable *hash_table);
GList * g_hash_table_get_keys(GHashTable *hash_table);
GList * g_hash_table_get_values(GHashTable *hash_table);
gboolean (*GHRFunc)(gpointer key, gpointer value, gpointer user_data);
#define g_hash_table_freeze(hash_table)
#define g_hash_table_thaw(hash_table)
void g_hash_table_destroy(GHashTable *hash_table);
GHashTable* g_hash_table_ref(GHashTable *hash_table);
void g_hash_table_unref(GHashTable *hash_table);
GHashTableIter;
void g_hash_table_iter_init(GHashTableIter *iter, GHashTable *hash_table);
gboolean g_hash_table_iter_next(GHashTableIter *iter, gpointer *key, gpointer *value);
GHashTableIter * g_hash_table_iter_get_hash_table(GHashTableIter *iter);
void g_hash_table_iter_remove(GHashTableIter *iter);
gboolean g_hash_table_iter_steal(GHashTableIter *iter);
GHashTable* g_hash_table_iter_get_hash_table(GHashTableIter *iter);
void g_hash_table_iter_remove(GHashTableIter *iter);
void g_hash_table_iter_steal(GHashTableIter *iter);
gboolean g_direct_equal(gconstpointer v1, gconstpointer v2);
guint g_direct_hash(gconstpointer v);
gboolean g_int_equal (gconstpointer v1, gconstpointer v2);
guint g_int_hash (gconstpointer v);
gboolean g_int64_equal (gconstpointer v1, gconstpointer v2);
guint g_int64_hash (gconstpointer v);
gboolean g_double_equal (gconstpointer v1, gconstpointer v2);
guint g_double_hash (gconstpointer v);
 gboolean g_str_equal (gconstpointer v1, gconstpointer v2);
guint g_str_hash (gconstpointer v);

Description

A GHashTable provides associations between keys and values which is optimized so that given a key, the associated value can be found very quickly. 

Note that neither keys nor values are copied when inserted into the GHashTable, so they must exist for the lifetime of the GHashTable. This means that the use of static strings is OK, but temporary strings (i.e. those created in buffers and those returned by GTK+ widgets) should be copied with g_strdup() before being inserted.

If keys or values are dynamically allocated, you must be careful to ensure that they are freed when they are removed from the GHashTable, and also when they are overwritten by new insertions into the GHashTable. It is also not advisable to mix static strings and dynamically-allocated strings in a GHashTable, because it then becomes difficult to determine whether the string should be freed.

To create a GHashTable, use g_hash_table_new().

To insert a key and value into a GHashTable, use g_hash_table_insert().

To lookup a value corresponding to a given key, use g_hash_table_lookup() and g_hash_table_lookup_extended().

To remove a key and value, use g_hash_table_remove().

To call a function for each key and value pair use g_hash_table_foreach() or use a iterator to iterate over the key/value pairs in the hash table, see GHashTableIter.

To destroy a GHashTable use g_hash_table_destroy().

Details

GHashTable

typedef struct _GHashTable GHashTable;

The GHashTable struct is an opaque data structure to represent a Hash Table. It should only be accessed via the following functions.

g_hash_table_new ()

GHashTable* g_hash_table_new (GHashFunc hash_func, GEqualFunc key_equal_func);

Creates a new GHashTable with a reference count of 1.

hash_func: a function to create a hash value from a key. Hash values are used to determine where keys are stored within the GHashTable data structure. The g_direct_hash(), g_int_hash(), g_int64_hash(), g_double_hash() and g_str_hash() functions are provided for some common types of keys. If hash_func is NULL, g_direct_hash() is used.

key_equal_func: a function to check two keys for equality. This is used when looking up keys in the GHashTable. The g_direct_equal(), g_int_equal(), g_int64_equal(), g_double_equal() and g_str_equal() functions are provided for the most common types of keys. If key_equal_func is NULL, keys are compared directly in a similar fashion to g_direct_equal(), but without the overhead of a function call.
CHAPTER 5. GLIB DATA TYPES

5.8. HASH TABLES

Returns: a new GHashTable.

g_hash_table_new_full()

GHashTable* g_hash_table_new_full (GHashFunc hash_func, GEqualFunc key_equal_func, GDestroyNotify key_destroy_func, GDestroyNotify value_destroy_func);

Creates a new GHashTable like g_hash_table_new() with a reference count of 1 and allows to specify functions to free the memory allocated for the key and value that get called when removing the entry from the GHashTable.

hash_func: a function to create a hash value from a key.

key_equal_func: a function to check two keys for equality.

key_destroy_func: a function to free the memory allocated for the key used when removing the entry from the GHashTable or NULL if you don’t want to supply such a function.

value_destroy_func: a function to free the memory allocated for the value used when removing the entry from the GHashTable or NULL if you don’t want to supply such a function.

Returns: a new GHashTable.

GHashFunc()

 guint (*GHashFunc) (gconstpointer key);

Specifies the type of the hash function which is passed to g_hash_table_new() when a GHashTable is created.

The function is passed a key and should return a guint hash value. The functions g_direct_hash(), g_int_hash() and g_str_hash() provide hash functions which can be used when the key is a gpointer, gint, and gchar* respectively.

The hash values should be evenly distributed over a fairly large range? The modulus is taken with the hash table size (a prime number) to find the ‘bucket’ to place each key into. The function should also be very fast, since it is called for each key lookup.

key: a key.

Returns: the hash value corresponding to the key.

GEqualFunc()

gboolean (*GEqualFunc) (gconstpointer a, gconstpointer b);

Specifies the type of a function used to test two values for equality. The function should return TRUE if both values are equal and FALSE otherwise.

a: a value.

b: a value to compare with.

Returns: %TRUE if \(a = b\); FALSE otherwise.
CHAPTER 5. GLIB DATA TYPES

5.8. HASH TABLES

**g_hash_table_insert ()**

```c
void g_hash_table_insert (GHashTable *hash_table, gpointer key, gpointer value);
```

Inserts a new key and value into a GHashTable.

If the key already exists in the GHashTable its current value is replaced with the new value. If you supplied a `value_destroy_func` when creating the GHashTable, the old value is freed using that function. If you supplied a `key_destroy_func` when creating the GHashTable, the passed key is freed using that function.

**hash_table**: a GHashTable.

**key**: a key to insert.

**value**: the value to associate with the key.

**g_hash_table_replace ()**

```c
void g_hash_table_replace (GHashTable *hash_table, gpointer key, gpointer value);
```

Inserts a new key and value into a GHashTable similar to `g_hash_table_insert()`. The difference is that if the key already exists in the GHashTable, it gets replaced by the new key. If you supplied a `value_destroy_func` when creating the GHashTable, the old value is freed using that function. If you supplied a `key_destroy_func` when creating the GHashTable, the old key is freed using that function.

**hash_table**: a GHashTable.

**key**: a key to insert.

**value**: the value to associate with the key.

**g_hash_table_size ()**

```c
guint g_hash_table_size (GHashTable *hash_table);
```

Returns the number of elements contained in the GHashTable.

**hash_table**: a GHashTable.

**Returns**: the number of key/value pairs in the GHashTable.

**g_hash_table_lookup ()**

```c
gpointer g_hash_table_lookup (GHashTable *hash_table, gconstpointer key);
```

Looks up a key in a GHashTable. Note that this function cannot distinguish between a key that is not present and one which is present and has the value NULL. If you need this distinction, use `g_hash_table_lookup_extended()`.

**hash_table**: a GHashTable.

**key**: the key to look up.

**Returns**: the associated value, or NULL if the key is not found.
g_hash_table_lookup_extended ()

gboolean g_hash_table_lookup_extended (GHashTable *hash_table, 
    gconstpointer lookup_key, 
    gpointer *orig_key, 
    gpointer *value);

Looks up a key in the GHashTable, returning the original key and the associated value and a gboolean which is TRUE if the key was found. This is useful if you need to free the memory allocated for the original key, for example before calling g_hash_table_remove().

You can actually pass NULL for lookup_key to test whether the NULL key exists.

**hash_table**: a GHashTable

**lookup_key**: the key to look up

**orig_key**: return location for the original key, or NULL

**value**: return location for the value associated with the key, or NULL

**Returns**: TRUE if the key was found in the GHashTable.

g_hash_table_foreach ()

void g_hash_table_foreach (GHashTable *hash_table, 
    GHFunc func, 
    gpointer user_data);

Calls the given function for each of the key/value pairs in the GHashTable. The function is passed the key and value of each pair, and the given user_data parameter. The hash table may not be modified while iterating over it (you can’t add/remove items). To remove all items matching a predicate, use g_hash_table_foreach_remove().

See g_hash_table_find() for performance caveats for linear order searches in contrast to g_hash_table_lookup().

**hash_table**: a GHashTable.

**func**: the function to call for each key/value pair.

**user_data**: user data to pass to the function.

g_hash_table_find ()

gpointer g_hash_table_find (GHashTable *hash_table, 
    GHRFunc predicate, 
    gpointer user_data);

Calls the given function for key/value pairs in the GHashTable until predicate returns TRUE. The function is passed the key and value of each pair, and the given user_data parameter. The hash table may not be modified while iterating over it (you can’t add/remove items).

Note, that hash tables are really only optimized for forward lookups, i.e. g_hash_table_lookup(). So code that frequently issues g_hash_table_find() or g_hash_table_foreach() (e.g. in the order of once per every entry in a hash table) should probably be reworked to use additional or different data structures for reverse lookups (keep in mind that an O(n) find/foreach operation issued for all n values in a hash table ends up needing O(n*n) operations).

**hash_table**: a GHashTable.

**predicate**: function to test the key/value pairs for a certain property.

**user_data**: user data to pass to the function.

**Returns**: The value of the first key/value pair is returned, for which func evaluates to TRUE. If no pair with the requested property is found, NULL is returned.

Since 2.4
GHFunc ()

void (*GHFunc) (gpointer key, 
gpointer value, 
gpointer user_data); 

Specifies the type of the function passed to g_hash_table_foreach(). It is called with each key/value pair, together with the user_data parameter which is passed to g_hash_table_foreach().

key: a key.
value: the value corresponding to the key.
user_data: user data passed to g_hash_table_foreach().

GHFunc ()

gboolean g_hash_table_remove (GHashTable *hash_table, 
gconstpointer key); 

Removes a key and its associated value from a GHashTable.
If the GHashTable was created using g_hash_table_new_full(), the key and value are freed using the supplied destroy functions, otherwise you have to make sure that any dynamically allocated values are freed yourself.
hash_table: a GHashTable.
key: the key to remove.
Returns: TRUE if the key was found and removed from the GHashTable.

 gboolean g_hash_table_steal (GHashTable *hash_table, 
gconstpointer key); 

Removes a key and its associated value from a GHashTable without calling the key and value destroy functions.
hash_table: a GHashTable.
key: the key to remove.
Returns: TRUE if the key was found and removed from the GHashTable.

 guint g_hash_table_foreach_remove (GHashTable *hash_table, 
GHRFunc func, 
gpointer user_data); 

Calls the given function for each key/value pair in the GHashTable. If the function returns TRUE, then the key/value pair is removed from the GHashTable. If you supplied key or value destroy functions when creating the GHashTable, they are used to free the memory allocated for the removed keys and values.
See GHashTableIter for an alternative way to loop over the key/value pairs in the hash table.
hash_table: a GHashTable.
func: the function to call for each key/value pair.
user_data: user data to pass to the function.
Returns: the number of key/value pairs removed.
g_hash_table_foreach_steal()

```c
uint g_hash_table_foreach_steal (GHashTable *hash_table,
                                             GHRFunc func,
                                             gpointer user_data);
```

Calls the given function for each key/value pair in the GHashTable. If the function returns TRUE, then the key/value pair is removed from the GHashTable, but no key or value destroy functions are called.

See GHashTableIter for an alternative way to loop over the key/value pairs in the hash table.

**hash_table**: a GHashTable.

**func**: the function to call for each key/value pair.

**user_data**: user data to pass to the function.

**Returns**: the number of key/value pairs removed.

---

g_hash_table_remove_all()

```c
void g_hash_table_remove_all (GHashTable *hash_table);
```

Removes all keys and their associated values from a GHashTable.
If the GHashTable was created using g_hash_table_new_full(), the keys and values are freed using the supplied destroy functions, otherwise you have to make sure that any dynamically allocated values are freed yourself.

**hash_table**: a GHashTable

Since 2.12

---

g_hash_table_steal_all()

```c
void g_hash_table_steal_all (GHashTable *hash_table);
```

Removes all keys and their associated values from a GHashTable without calling the key and value destroy functions.

**hash_table**: a GHashTable.

Since 2.12

---

g_hash_table_get_keys()

```c
GList * g_hash_table_get_keys (GHashTable *hash_table);
```

Retrieves every key inside *hash_table*. The returned data is valid until *hash_table* is modified.

**hash_table**: a GHashTable

**Returns**: a GList containing all the keys inside the hash table. The content of the list is owned by the hash table and should not be modified or freed. Use g_list_free() when done using the list.

Since 2.14

---

g_hash_table_get_values()

```c
GList * g_hash_table_get_values (GHashTable *hash_table);
```

Retrieves every value inside *hash_table*. The returned data is valid until *hash_table* is modified.

**hash_table**: a GHashTable

**Returns**: a GList containing all the values inside the hash table. The content of the list is owned by the hash table and should not be modified or freed. Use g_list_free() when done using the list.

Since 2.14
GHRFunc ()

```c
gboolean ( *GHRFunc) (gpointer key, gpointer value, gpointer user_data);
```

Specifies the type of the function passed to `g_hash_table_foreach_remove()`. It is called with each key/value pair, together with the `user_data` parameter passed to `g_hash_table_foreach_remove()`. It should return `TRUE` if the key/value pair should be removed from the GHashTable.

**key**: a key.

**value**: the value associated with the key.

**user_data**: user data passed to `g_hash_table_remove()`.

**Returns**: `TRUE` if the key/value pair should be removed from the GHashTable.

---

**g_hash_table_freeze()**

```c
#define g_hash_table_freeze(hash_table)
```

**WARNING**

`g_hash_table_freeze` is deprecated and should not be used in newly-written code.

This function is deprecated and will be removed in the next major release of GLib. It does nothing.

**hash_table**: a GHashTable

---

**g_hash_table_thaw()**

```c
#define g_hash_table_thaw(hash_table)
```

**WARNING**

`g_hash_table_thaw` is deprecated and should not be used in newly-written code.

This function is deprecated and will be removed in the next major release of GLib. It does nothing.

**hash_table**: a GHashTable

---

**g_hash_table_destroy ()**

```c
void g_hash_table_destroy (GHashTable *hash_table);
```

Destroys all keys and values in the GHashTable and decrements its reference count by 1. If keys and/or values are dynamically allocated, you should either free them first or create the GHashTable with destroy notifiers using `g_hash_table_new_full()`. In the latter case the destroy functions you supplied will be called on all keys and values during the destruction phase.

**hash_table**: a GHashTable.
CHAPTER 5. GLIB DATA TYPES  5.8. HASH TABLES

**g_hash_table_ref ()**

```c
GHashTable* g_hash_table_ref (GHashTable *hash_table);
```

Atomically increments the reference count of *hash_table* by one. This function is MT-safe and may be called from any thread.

*hash_table*: a valid GHashTable.

**Returns**: the passed in GHashTable.

Since 2.10

**g_hash_table_unref ()**

```c
void g_hash_table_unref (GHashTable *hash_table);
```

Atomically decrements the reference count of *hash_table* by one. If the reference count drops to 0, all keys and values will be destroyed, and all memory allocated by the hash table is released. This function is MT-safe and may be called from any thread.

*hash_table*: a valid GHashTable.

Since 2.10

**GHashTableIter**

```c
typedef struct {
} GHashTableIter;
```

A GHashTableIter structure represents an iterator that can be used to iterate over the elements of a GHashTable. GHashTableIter structures are typically allocated on the stack and then initialized with `g_hash_table_iter_init()`.

**g_hash_table_iter_init ()**

```c
void g_hash_table_iter_init (GHashTableIter *iter, GHashTable *hash_table);
```

Initializes a key/value pair iterator and associates it with *hash_table*. Modifying the hash table after calling this function invalidates the returned iterator.

```c
GHashTableIter iter;
gpointer key, value;

g_hash_table_iter_init (&iter, hash_table);
while (g_hash_table_iter_next (&iter, &key, &value))
{
    /* do something with key and value */
}
```

*iter*: an uninitialized GHashTableIter.

*hash_table*: a GHashTable.

Since 2.16
g_hash_table_iter_next ()

gboolean g_hash_table_iter_next (GHashTableIter *iter,
   gpointer *key,
   gpointer *value);

Advances _iter_ and retrieves the key and/or value that are now pointed to as a result of this advancement. If FALSE is returned, _key_ and _value_ are not set, and the iterator becomes invalid.

_iter_: an initialized GHashTableIter.
_key_: a location to store the key, or NULL.
_value_: a location to store the value, or NULL.

_Returns_: FALSE if the end of the GHashTable has been reached.

Since 2.16

g_hash_table_iter_get_hash_table ()

GHashTable* g_hash_table_iter_get_hash_table (GHashTableIter *iter);

Returns the GHashTable associated with _iter_.

_iter_: an initialized GHashTableIter.

_Returns_: the GHashTable associated with _iter_.

Since 2.16

g_hash_table_iter_remove ()

void g_hash_table_iter_remove (GHashTableIter *iter);

Removes the key/value pair currently pointed to by the iterator from its associated GHashTable. Can only be called after _g_hash_table_iter_next_() returned TRUE, and cannot be called more than once for the same key/value pair.

If the GHashTable was created using _g_hash_table_new_full_(), the key and value are freed using the supplied destroy functions, otherwise you have to make sure that any dynamically allocated values are freed yourself.

_iter_: an initialized GHashTableIter.

Since 2.16

g_hash_table_iter_steal ()

void g_hash_table_iter_steal (GHashTableIter *iter);

Removes the key/value pair currently pointed to by the iterator from its associated GHashTable, without calling the key and value destroy functions. Can only be called after _g_hash_table_iter_next_() returned TRUE, and cannot be called more than once for the same key/value pair.

_iter_: an initialized GHashTableIter.

Since 2.16
CHAPTER 5. GLIB DATA TYPES

5.8. HASH TABLES

g_direct_equal ()

gboolean g_direct_equal (gconstpointer v1, gconstpointer v2);

Compares two gpointer arguments and returns TRUE if they are equal. It can be passed to g_hash_table_new() as the key_equal_func parameter, when using pointers as keys in a GHashTable.

v1 : a key.
v2 : a key to compare with v1.

Returns: TRUE if the two keys match.

g_direct_hash ()

guint g_direct_hash (gconstpointer v);

Converts a gpointer to a hash value. It can be passed to g_hash_table_new() as the hash_func parameter, when using pointers as keys in a GHashTable.

v : a gpointer key

Returns: a hash value corresponding to the key.

g_int_equal ()

gboolean g_int_equal (gconstpointer v1, gconstpointer v2);

Compares the two gint values being pointed to and returns TRUE if they are equal. It can be passed to g_hash_table_new() as the key_equal_func parameter, when using pointers to integers as keys in a GHashTable.

v1 : a pointer to a gint key.
v2 : a pointer to a gint key to compare with v1.

Returns: TRUE if the two keys match.

g_int_hash ()

guint g_int_hash (gconstpointer v);

Converts a pointer to a gint to a hash value. It can be passed to g_hash_table_new() as the hash_func parameter, when using pointers to integers values as keys in a GHashTable.

v : a pointer to a gint key

Returns: a hash value corresponding to the key.

g_int64_equal ()

gboolean g_int64_equal (gconstpointer v1, gconstpointer v2);

Compares the two gint64 values being pointed to and returns TRUE if they are equal. It can be passed to g_hash_table_new() as the key_equal_func parameter, when using pointers to 64-bit integers as keys in a GHashTable.

v1 : a pointer to a gint64 key.
v2 : a pointer to a gint64 key to compare with v1.

Returns: TRUE if the two keys match.

Since 2.22
g_int64_hash()

```c
uint g_int64_hash (gconstpointer v);
```

Converts a pointer to a gint64 to a hash value. It can be passed to `g_hash_table_new()` as the `hash_func` parameter, when using pointers to 64-bit integers values as keys in a GHashTable.

**v**: a pointer to a gint64 key

**Returns**: a hash value corresponding to the key.

Since 2.22

---

g_double_equal()

```c
gboolean g_double_equal (gconstpointer v1, gconstpointer v2);
```

Compares the two gdouble values being pointed to and returns TRUE if they are equal. It can be passed to `g_hash_table_new()` as the `key_equal_func` parameter, when using pointers to doubles as keys in a GHashTable.

**v1**: a pointer to a gdouble key.

**v2**: a pointer to a gdouble key to compare with `v1`.

**Returns**: TRUE if the two keys match.

Since 2.22

---

g_double_hash()

```c
uint g_double_hash (gconstpointer v);
```

Converts a pointer to a gdouble to a hash value. It can be passed to `g_hash_table_new()` as the `hash_func` parameter, when using pointers to doubles as keys in a GHashTable.

**v**: a pointer to a gdouble key

**Returns**: a hash value corresponding to the key.

Since 2.22

---

g_str_equal()

```c
gboolean g_str_equal (gconstpointer v1, gconstpointer v2);
```

Compares two strings for byte-by-byte equality and returns TRUE if they are equal. It can be passed to `g_hash_table_new()` as the `key_equal_func` parameter, when using strings as keys in a GHashTable.

**v1**: a key

**v2**: a key to compare with `v1`.

**Returns**: TRUE if the two keys match

---

g_str_hash()

```c
uint g_str_hash (gconstpointer v);
```

Converts a string to a hash value. It can be passed to `g_hash_table_new()` as the `hash_func` parameter, when using strings as keys in a GHashTable.

**v**: a string key

**Returns**: a hash value corresponding to the key
5.9 Strings

Name

Strings – text buffers which grow automatically as text is added

Synopsis

```c
#include <glib.h>

GString;  
GString* g_string_new (const gchar *init);
GString* g_string_new_len (const gchar *init, gssize len);
GString* g_string_sized_new (gsize dfl_size);
GString* g_string_assign (GString *string, const gchar *rval);
#define g_string_sprintf
#define g_string_sprintfa
void g_string_vprintf (GString *string, const gchar *format, va_list args);
void g_string_append_vprintf (GString *string, const gchar *format, va_list args);
void g_string_printf (GString *string, const gchar *format, ...);
void g_string_append_printf (GString *string, const gchar *format, ...);
GString* g_string_append (GString *string, const gchar *val);
GString* g_string_append_c (GString *string, gchar c);
GString* g_string_append_unichar (GString *string, gunichar wc);
GString* g_string_append_len (GString *string, const gchar *val, gssize len);
GString* g_string_append_uri_escaped (GString *string, const char *unescaped, const char *reserved_chars_allowed, gboolean allow_utf8);
GString* g_string_prepend (GString *string, const gchar *val);
GString* g_string_prepend_c (GString *string, gchar c);
GString* g_string_prepend_unichar (GString *string, gunichar wc);
GString* g_string_prepend_len (GString *string, const gchar *val, gssize len);
GString* g_string_insert (GString *string, gssize pos, const gchar *val);
GString* g_string_insert_c (GString *string, gssize pos,
```

523
CHAPTER 5. GLIB DATA TYPES

5.9. STRINGS

GString*  g_string_insert_unichar (GString *string, gssize pos, gunichar wc);
GString*  g_string_insert_len (GString *string, gssize pos, const gchar *val, gssize len);
GString*  g_string_overwrite (GString *string, gsize pos, const gchar *val);
GString*  g_string_overwrite_len (GString *string, gsize pos, const gchar *val, gssize len);
GString*  g_string_erase (GString *string, gssize pos, gssize len);
GString*  g_string_truncate (GString *string, gsize len);
GString*  g_string_set_size (GString *string, gsize len);
gchar*    g_string_free (GString *string, gboolean free_segment);
GString*  g_string_up (GString *string);
GString*  g_string_down (GString *string);
guint     g_string_hash (const GString *str);
gboolean  g_string_equal (const GString *v, const GString *v2);

Description

A GString is similar to a standard C string, except that it grows automatically as text is appended or inserted. Also, it stores the length of the string, so can be used for binary data with embedded null bytes.

Details

GString

typedef struct {
  gchar *str;
  gsize len;
  gsize allocated_len;
} GString;

The GString struct contains the public fields of a GString.

gchar *str; points to the character data. It may move as text is added. The str field is null-terminated and so can be used as an ordinary C string.

gsize len; contains the length of the string, not including the terminating null byte.

gsize allocated_len; the number of bytes that can be stored in the string before it needs to be reallocated. May be larger than len.

g_string_new ()

GString*  g_string_new (const gchar *init);
CHAPTER 5. GLIB DATA TYPES 5.9. STRINGS

Creates a new GString, initialized with the given string.

*init*: the initial text to copy into the string

*Returns*: the new GString

**g_string_new_len ()**

```c
GString* g_string_new_len (const gchar *init, gssize len);
```

Creates a new GString with *len* bytes of the *init* buffer. Because a length is provided, *init* need not be nul-terminated, and can contain embedded nul bytes.

Since this function does not stop at nul bytes, it is the caller’s responsibility to ensure that *init* has at least *len* addressable bytes.

*init*: initial contents of the string

*len*: length of *init* to use

*Returns*: a new GString

**g_string_sized_new ()**

```c
GString* g_string_sized_new (gsize dfl_size);
```

Creates a new GString, with enough space for *dfl_size* bytes. This is useful if you are going to add a lot of text to the string and don’t want it to be reallocated too often.

*dfl_size*: the default size of the space allocated to hold the string

*Returns*: the new GString

**g_string_assign ()**

```c
GString* g_string_assign (GString *string, const gchar *rval);
```

Copies the bytes from a string into a GString, destroying any previous contents. It is rather like the standard strcpy() function, except that you do not have to worry about having enough space to copy the string.

*string*: the destination GString. Its current contents are destroyed.

*rval*: the string to copy into *string*

*Returns*: *string*

**g_string_sprintf**

```c
#define g_string_sprintf
```

*WARNING*

*g_string_sprintf* is deprecated and should not be used in newly-written code. This function has been renamed to *g_string_printf(*).

Writes a formatted string into a GString. This is similar to the standard sprintf() function, except that the GString buffer automatically expands to contain the results. The previous contents of the GString are destroyed.
string: a GString

format: the string format. See the sprintf() documentation

...: the parameters to insert into the format string

**g_string_sprintfa**

```c
#define g_string_sprintfa
```

**WARNING**

`g_string_sprintfa` is deprecated and should not be used in newly-written code. This function has been renamed to `g_string_append_printf()`.

Appends a formatted string onto the end of a GString. This function is similar to `g_string_sprintf()` except that the text is appended to the GString.

string: a GString

format: the string format. See the sprintf() documentation

...: the parameters to insert into the format string

**g_string_vprintf()**

```c
void g_string_vprintf (GString *string, const gchar *format, va_list args);
```

Writes a formatted string into a GString. This function is similar to `g_string_printf()` except that the arguments to the format string are passed as a va_list.

string: a GString

format: the string format. See the printf() documentation

args: the parameters to insert into the format string

Since 2.14

**g_string_append_vprintf()**

```c
void g_string_append_vprintf (GString *string, const gchar *format, va_list args);
```

Appends a formatted string onto the end of a GString. This function is similar to `g_string_append_printf()` except that the arguments to the format string are passed as a va_list.

string: a GString

format: the string format. See the printf() documentation

args: the list of arguments to insert in the output

Since 2.14
g_string_printf ()

```c
void g_string_printf (GString *string,
                 const gchar *format,
                 ...);
```

Writes a formatted string into a GString. This is similar to the standard `sprintf()` function, except that the GString buffer automatically expands to contain the results. The previous contents of the GString are destroyed.

**string**: a GString

**format**: the string format. See the `printf()` documentation

...: the parameters to insert into the format string

---

g_string_append_printf ()

```c
void g_string_append_printf (GString *string,
                         const gchar *format,
                         ...);
```

Appends a formatted string onto the end of a GString. This function is similar to `g_string_printf()` except that the text is appended to the GString.

**string**: a GString

**format**: the string format. See the `printf()` documentation

...: the parameters to insert into the format string

---

**g_string_append ()**

```c
GString* g_string_append (GString *string,
                        const gchar *val);
```

Adds a string onto the end of a GString, expanding it if necessary.

**string**: a GString

**val**: the string to append onto the end of **string**

**Returns**: string

---

**g_string_append_c ()**

```c
GString* g_string_append_c (GString *string,
                        gchar c);
```

Adds a byte onto the end of a GString, expanding it if necessary.

**string**: a GString

**c**: the byte to append onto the end of **string**

**Returns**: string

---

**g_string_append_unichar ()**

```c
GString* g_string_append_unichar (GString *string,
                          gunichar wc);
```

Converts a Unicode character into UTF-8, and appends it to the string.

**string**: a GString

**wc**: a Unicode character

**Returns**: string
**g_string_append_len()**

Appends *len* bytes of *val* to *string*. Because *len* is provided, *val* may contain embedded nulls and need not be null-terminated.

Since this function does not stop at null bytes, it is the caller’s responsibility to ensure that *val* has at least *len* addressable bytes.

*string*: a GString
*val*: bytes to append
*len*: number of bytes of *val* to use

**Returns**: *string*

**g_string_append_uri_escaped()**

Appends *unescaped* to *string*, escaped any characters that are reserved in URIs using URI-style escape sequences.

*string*: a GString
*unescaped*: a string
*reserved_chars_allowed*: a string of reserved characters allowed to be used
*allow_utf8*: set TRUE if the escaped string may include UTF8 characters

**Returns**: *string*

Since 2.16

**g_string_prepend()**

Adds a string on to the start of a GString, expanding it if necessary.

*string*: a GString
*val*: the string to prepend on the start of *string*

**Returns**: *string*

**g_string_prepend_c()**

Adds a byte onto the start of a GString, expanding it if necessary.

*string*: a GString
*c*: the byte to prepend on the start of the GString

**Returns**: *string*
g_string_prepend_unichar ()

GString* g_string_prepend_unichar (GString *string, gunichar wc);

Converts a Unicode character into UTF-8, and prepends it to the string.

string: a GString

wc: a Unicode character

Returns: string

g_string_prepend_len ()

GString* g_string_prepend_len (GString *string, const gchar *val, gssize len);

Prepends len bytes of val to string. Because len is provided, val may contain embedded nuls and need not be nul-terminated.

Since this function does not stop at nul bytes, it is the caller’s responsibility to ensure that val has at least len addressable bytes.

string: a GString

val: bytes to prepend

len: number of bytes in val to prepend

Returns: string

g_string_insert ()

GString* g_string_insert (GString *string, gssize pos, const gchar *val);

Inserts a copy of a string into a GString, expanding it if necessary.

string: a GString

pos: the position to insert the copy of the string

val: the string to insert

Returns: string

g_string_insert_c ()

GString* g_string_insert_c (GString *string, gssize pos, gchar c);

Inserts a byte into a GString, expanding it if necessary.

string: a GString

pos: the position to insert the byte

c: the byte to insert

Returns: string
g_string_insert_unichar()

Converting a Unicode character into UTF-8, and insert it into the string at the given position.

**string**: a GString

**pos**: the position at which to insert character, or -1 to append at the end of the string

**wc**: a Unicode character

**Returns**: string

---

g_string_insert_len()

Inserts `len` bytes of `val` into `string` at `pos`. Because `len` is provided, `val` may contain embedded nuls and need not be nul-terminated. If `pos` is -1, bytes are inserted at the end of the string.

Since this function does not stop at nul bytes, it is the caller’s responsibility to ensure that `val` has at least `len` addressable bytes.

**string**: a GString

**pos**: position in `string` where insertion should happen, or -1 for at the end

**val**: bytes to insert

**len**: number of bytes of `val` to insert

**Returns**: string

---

g_string_overwrite()

Overwrites part of a string, lengthening it if necessary.

**string**: a GString

**pos**: the position at which to start overwriting

**val**: the string that will overwrite the `string` starting at `pos`

**Returns**: string

Since 2.14
**5.9. STRINGS**

---

### g_string_overwrite_len()

**Function Signature:**

```c
GString* g_string_overwrite_len (GString *string, gsize pos, const gchar *val, gssize len);
```

**Description:**

Overwrites part of a string, lengthening it if necessary. This function will work with embedded nuls.

**Parameters:**

- `string`: a `GString`
- `pos`: the position at which to start overwriting
- `val`: the string that will overwrite the `string` starting at `pos`
- `len`: the number of bytes to write from `val`

**Returns:**

- `string`

Since 2.14

### g_string_erase()

**Function Signature:**

```c
GString* g_string_erase (GString *string, gssize pos, gssize len);
```

**Description:**

Removes `len` bytes from a `GString`, starting at position `pos`. The rest of the `GString` is shifted down to fill the gap.

**Parameters:**

- `string`: a `GString`
- `pos`: the position of the content to remove
- `len`: the number of bytes to remove, or -1 to remove all following bytes

**Returns:**

- `string`

### g_string_truncate()

**Function Signature:**

```c
GString* g_string_truncate (GString *string, gsize len);
```

**Description:**

Cuts off the end of the `GString`, leaving the first `len` bytes.

**Parameters:**

- `string`: a `GString`
- `len`: the new size of `string`

**Returns:**

- `string`

### g_string_set_size()

**Function Signature:**

```c
GString* g_string_set_size (GString *string, gsize len);
```

**Description:**

Sets the length of a `GString`. If the length is less than the current length, the string will be truncated. If the length is greater than the current length, the contents of the newly added area are undefined. (However, as always, `string->str[string->len]` will be a nul byte.)

**Parameters:**

- `string`: a `GString`
- `len`: the new length

**Returns:**

- `string`
**g_string_free()**

```
qchar* g_string_free (GString *string, gboolean free_segment);
```

Frees the memory allocated for the GString. If `free_segment` is `TRUE` it also frees the character data.

**string**: a GString

**free_segment**: if `TRUE` the actual character data is freed as well

**Returns**: the character data of `string` (i.e. NULL if `free_segment` is `TRUE`)

---

**g_string_up()**

```
GString* g_string_up (GString *string);
```

**WARNING**

`g_string_up` has been deprecated since version 2.2 and should not be used in newly-written code. This function uses the locale-specific `toupper()` function, which is almost never the right thing. Use `g_string_ascii_up()` or `g_utf8_strup()` instead.

Converts a GString to uppercase.

**string**: a GString

**Returns**: `string`

---

**g_string_down()**

```
GString* g_string_down (GString *string);
```

**WARNING**

`g_string_down` has been deprecated since version 2.2 and should not be used in newly-written code. This function uses the locale-specific `tolower()` function, which is almost never the right thing. Use `g_string_ascii_down()` or `g_utf8_strdown()` instead.

Converts a GString to lowercase.

**string**: a GString

**Returns**: the GString.

---

**g_string_hash()**

```
guint g_string_hash (const GString *str);
```

Creates a hash code for `str`; for use with GHashTable.

**str**: a string to hash

**Returns**: hash code for `str`
CHAPTER 5. GLIB DATA TYPES 5.10. STRING CHunks

g_string_equal()

gboolean g_string_equal (const GString *v, const GString *v2);

Compares two strings for equality, returning TRUE if they are equal. For use with GHashTable.

v: a GString

v2: another GString

Returns: TRUE if they strings are the same length and contain the same bytes

5.10 String Chunks

Name

String Chunks – efficient storage of groups of strings

Synopsis

#include <glib.h>

GStringChunk;
gchar* g_string_chunk_new (gsize size);
gchar* g_string_chunk_insert (GStringChunk *chunk, const gchar *string);
gchar* g_string_chunk_insert_const (GStringChunk *chunk, const gchar *string);
gchar* g_string_chunk_insert_len (GStringChunk *chunk, const gchar *string, gssize len);
void g_string_chunk_clear (GStringChunk *chunk);
void g_string_chunk_free (GStringChunk *chunk);

Description

String chunks are used to store groups of strings. Memory is allocated in blocks, and as strings are added to the GStringChunk they are copied into the next free position in a block. When a block is full a new block is allocated.

When storing a large number of strings, string chunks are more efficient than using g_strdup() since fewer calls to malloc() are needed, and less memory is wasted in memory allocation overheads.

By adding strings with g_string_chunk_insert_const() it is also possible to remove duplicates.

To create a new GStringChunk use g_string_chunk_new().

To add strings to a GStringChunk use g_string_chunk_insert().

To add strings to a GStringChunk, but without duplicating strings which are already in the GStringChunk, use g_string_chunk_insert_len().

To free the entire GStringChunk use g_string_chunk_free(). It is not possible to free individual strings.

Details

GStringChunk

typedef struct _GStringChunk GStringChunk;

An opaque data structure representing String Chunks. It should only be accessed by using the following functions.
### 5.10. STRING CHUNKS

#### g_string_chunk_new()

```c
GStringChunk* g_string_chunk_new (gsize size);
```

Creates a new `GStringChunk`.

**size**: the default size of the blocks of memory which are allocated to store the strings. If a particular string is larger than this default size, a larger block of memory will be allocated for it.

**Returns**: a new `GStringChunk`

#### g_string_chunk_insert()

```c
gchar* g_string_chunk_insert (GStringChunk *chunk, const gchar *string);
```

Adds a copy of `string` to the `GStringChunk`. It returns a pointer to the new copy of the string in the `GStringChunk`. The characters in the string can be changed, if necessary, though you should not change anything after the end of the string.

Unlike `g_string_chunk_insert_const()`, this function does not check for duplicates. Also strings added with `g_string_chunk_insert()` will not be searched by `g_string_chunk_insert_const()` when looking for duplicates.

**chunk**: a `GStringChunk`

**string**: the string to add

**Returns**: a pointer to the copy of `string` within the `GStringChunk`

#### g_string_chunk_insert_const()

```c
gchar* g_string_chunk_insert_const (GStringChunk *chunk, const gchar *string);
```

Adds a copy of `string` to the `GStringChunk`, unless the same string has already been added to the `GStringChunk` with `g_string_chunk_insert_const()`.

This function is useful if you need to copy a large number of strings but do not want to waste space storing duplicates. But you must remember that there may be several pointers to the same string, and so any changes made to the strings should be done very carefully.

Note that `g_string_chunk_insert_const()` will not return a pointer to a string added with `g_string_chunk_insert()`, even if they do match.

**chunk**: a `GStringChunk`

**string**: the string to add

**Returns**: a pointer to the new or existing copy of `string` within the `GStringChunk`

#### g_string_chunk_insert_len()

```c
gchar* g_string_chunk_insert_len (GStringChunk *chunk, const gchar *string, gssize len);
```

Adds a copy of the first `len` bytes of `string` to the `GStringChunk`. The copy is nul-terminated.

Since this function does not stop at nul bytes, it is the caller’s responsibility to ensure that `string` has at least `len` addressable bytes.

The characters in the returned string can be changed, if necessary, though you should not change anything after the end of the string.

**chunk**: a `GStringChunk`

**string**: bytes to insert
**5.11 Arrays**

**Name**

Arrays – arrays of arbitrary elements which grow automatically as elements are added

**Synopsis**

```c
#include <glib.h>

GArray;
GArray* g_array_new (gboolean zero_terminated,
                      gboolean clear_,
                      guint element_size);
GArray* g_array_sized_new (gboolean zero_terminated,
                           gboolean clear_,
                           guint element_size,
                           guint reserved_size);
GArray * g_array_ref (GArray *array);
void g_array_unref (GArray *array);
guint g_array_get_element_size (GArray *array);
#define g_array_append_val (a,v)
GArray* g_array_append_vals (GArray *array,
                          gconstpointer data,
                          guint len);
#define g_array_prepend_val (a,v)
GArray* g_array_prepend_vals (GArray *array,
                          gconstpointer data,
                          guint len);
#define g_array_insert_val (a,i,v)
GArray* g_array_insert_vals (GArray *array,
                        guint index_,
                        gconstpointer data,
```

---

**len**: number of bytes of `string` to insert, or -1 to insert a nul-terminated string

**Returns**: a pointer to the copy of `string` within the GStringChunk

Since 2.4

**g_string_chunk_clear ()**

```c
void g_string_chunk_clear (GStringChunk *chunk);
```

Frees all strings contained within the GStringChunk. After calling `g_string_chunk_clear()` it is not safe to access any of the strings which were contained within it.

**chunk**: a GStringChunk

Since 2.14

**g_string_chunk_free ()**

```c
void g_string_chunk_free (GStringChunk *chunk);
```

Frees all memory allocated by the GStringChunk. After calling `g_string_chunk_free()` it is not safe to access any of the strings which were contained within it.

**chunk**: a GStringChunk
Arrays are similar to standard C arrays, except that they grow automatically as elements are added. Array elements can be of any size (though all elements of one array are the same size), and the array can be automatically cleared to '0's and zero-terminated.

To create a new array use `g_array_new()`.
To add elements to an array, use `g_array_append_val()`, `g_array_append_vals()`, `g_array_prepend_val()`, and `g_array_prepend_vals()`.
To access an element of an array, use `g_array_index()`.
To set the size of an array, use `g_array_set_size()`.
To free an array, use `g_array_free()`.

### Example 5.5 Using a GArray to store gint values

```c
GArray *garray;
gint i;
/* We create a new array to store gint values. We don't want it zero-terminated or cleared to 0's. */
garray = g_array_new (FALSE, FALSE, sizeof (gint));
for (i = 0; i < 10000; i++)
    g_array_append_val (garray, i);
for (i = 0; i < 10000; i++)
    if (g_array_index (garray, gint, i) != i)
        g_print ("ERROR: got %d instead of %d\n",
                 g_array_index (garray, gint, i), i);
g_array_free (garray, TRUE);
```

### Details

**GArray**

```c
typedef struct {
    gchar *data;
    guint len;
} GArray;
```

Contains the public fields of an `Array`.

- **gchar *data**: a pointer to the element data. The data may be moved as elements are added to the `GArray`.
- **guint len**: the number of elements in the `GArray`. 

---

536
**g_array_new()**

```c
GArray* g_array_new (gboolean zero_terminated, gboolean clear_, guint element_size);
```

Creates a new `GArray` with a reference count of 1.

- **zero_terminated**: %TRUE if the array should have an extra element at the end which is set to 0.
- **clear_**: %TRUE if `GArray` elements should be automatically cleared to 0 when they are allocated.
- **element_size**: the size of each element in bytes.

*Returns*: the new `GArray`.

**g_array_sized_new()**

```c
GArray* g_array_sized_new (gboolean zero_terminated, gboolean clear_, guint element_size, guint reserved_size);
```

Creates a new `GArray` with `reserved_size` elements preallocated and a reference count of 1. This avoids frequent reallocation, if you are going to add many elements to the array. Note however that the size of the array is still 0.

- **zero_terminated**: %TRUE if the array should have an extra element at the end with all bits cleared.
- **clear_**: %TRUE if all bits in the array should be cleared to 0 on allocation.
- **element_size**: size of each element in the array.
- **reserved_size**: number of elements preallocated.

*Returns*: the new `GArray`.

**g_array_ref()**

```c
GArray* g_array_ref (GArray* array);
```

Atomically increments the reference count of `array` by one. This function is MT-safe and may be called from any thread.

*array*: A `GArray`.

*Returns*: The passed in `GArray`.

Since 2.22

**g_array_unref()**

```c
void g_array_unref (GArray* array);
```

Atomically decrements the reference count of `array` by one. If the reference count drops to 0, all memory allocated by the array is released. This function is MT-safe and may be called from any thread.

*array*: A `GArray`.

Since 2.22
CHAPTER 5. GLIB DATA TYPES

5.11. ARRAYS

**g_array_get_element_size**

```c
uint g_array_get_element_size (GArray *array);
```

Gets the size of the elements in `array`.

- **array**: A `GArray`.
- **Returns**: Size of each element, in bytes.
  - Since 2.22

**g_array_append_val**

```c
#define g_array_append_val(a,v)
```

Adds the value on to the end of the array. The array will grow in size automatically if necessary.

**NOTE**

`g_array_append_val()` is a macro which uses a reference to the value parameter `v`. This means that you cannot use it with literal values such as "27". You must use variables.

- **a**: a `GArray`.
- **v**: the value to append to the `GArray`.
- **Returns**: the `GArray`.

**g_array_append_vals**

```c
GArray* g_array_append_vals (GArray *array, gconstpointer data, guint len);
```

Adds `len` elements onto the end of the array.

- **array**: a `GArray`.
- **data**: a pointer to the elements to append to the end of the array.
- **len**: the number of elements to append.
- **Returns**: the `GArray`.

**g_array_prepend_val**

```c
#define g_array_prepend_val(a,v)
```

Adds the value on to the start of the array. The array will grow in size automatically if necessary. This operation is slower than `g_array_append_val()` since the existing elements in the array have to be moved to make space for the new element.

**NOTE**

`g_array_prepend_val()` is a macro which uses a reference to the value parameter `v`. This means that you cannot use it with literal values such as "27". You must use variables.
5.11. ARRAYS

g_array_prepend_vals()

GArray* g_array_prepend_vals (GArray *array, constpointer data, guint len);

Adds `len` elements onto the start of the array. This operation is slower than `g_array_append_vals()` since the existing elements in the array have to be moved to make space for the new elements.

array: a GArray.
data: a pointer to the elements to prepend to the start of the array.
len: the number of elements to prepend.

Returns: the GArray.

NOTE

`g_array_insert_val()` is a macro which uses a reference to the value parameter `v`. This means that you cannot use it with literal values such as "27". You must use variables.

a: a GArray.
i: the index to place the element at.
v: the value to insert into the array.

Returns: the GArray.

g_array_insert_vals()

GArray* g_array_insert_vals (GArray *array, guint index_, constpointer data, guint len);

Inserts `len` elements into a GArray at the given index.

array: a GArray.
index_: the index to place the elements at.
data: a pointer to the elements to insert.
len: the number of elements to insert.

Returns: the GArray.
g_array_remove_index ()

```c
GArray* g_array_remove_index (GArray *array,
                           guint index_);
```

Removes the element at the given index from a GArray. The following elements are moved down one place.

- `array`: a GArray.
- `index_`: the index of the element to remove.
- **Returns**: the GArray.


g_array_remove_index_fast ()

```c
GArray* g_array_remove_index_fast (GArray *array,
                                    guint index_);
```

Removes the element at the given index from a GArray. The last element in the array is used to fill in the space, so this function does not preserve the order of the GArray. But it is faster than `g_array_remove_index()`.

- `array`: a GArray.
- `index_`: the index of the element to remove.
- **Returns**: the GArray.


g_array_remove_range ()

```c
GArray* g_array_remove_range (GArray *array,
                               guint index_,
                               guint length);
```

Removes the given number of elements starting at the given index from a GArray. The following elements are moved to close the gap.

- `array`: a GArray.
- `index_`: the index of the first element to remove.
- `length`: the number of elements to remove.
- **Returns**: the GArray.

Since 2.4


g_array_sort ()

```c
void g_array_sort (GArray *array,
                    GCompareFunc compare_func);
```

Sorts a GArray using `compare_func` which should be a qsort()-style comparison function (returns less than zero for first arg is less than second arg, zero for equal, greater zero if first arg is greater than second arg). If two array elements compare equal, their order in the sorted array is undefined.

- `array`: a GArray.
- `compare_func`: comparison function.
**g_array_sort_with_data()**

```c
void g_array_sort_with_data (GArray *array,
    GCompareDataFunc compare_func,
    gpointer user_data);
```

Like `g_array_sort()`, but the comparison function receives an extra user data argument.

**array**: a `GArray`.

**compare_func**: comparison function.

**user_data**: data to pass to `compare_func`.

**g_array_index()**

```c
#define g_array_index(a,t,i)...
```

Returns the element of a `GArray` at the given index. The return value is cast to the given type.

### Example 5.6 Getting a pointer to an element in a `GArray`

```c
EDayViewEvent *event;
/* This gets a pointer to the 4th element in the array of EDayViewEvent structs. */
    event = &g_array_index (events, EDayViewEvent, 3);
```

**a**: a `GArray`.

**t**: the type of the elements.

**i**: the index of the element to return.

**Returns**: the element of the `GArray` at the index given by `i`.

**g_array_set_size()**

```c
GArray* g_array_set_size (GArray *array,
    guint length);
```

Sets the size of the array, expanding it if necessary. If the array was created with `clear_set` to `TRUE`, the new elements are set to 0.

**array**: a `GArray`.

**length**: the new size of the `GArray`.

**Returns**: the `GArray`.

**g_array_free()**

```c
gchar* g_array_free (GArray *array,
        gboolean free_segment);
```

Frees the memory allocated for the `GArray`. If `free_segment` is `TRUE` it frees the memory block holding the elements as well and also each element if `array` has an `element_free_func` set. Pass `FALSE` if you want to free the `GArray` wrapper but preserve the underlying array for use elsewhere. If the reference count of `array` is greater than one, the `GArray` wrapper is preserved but the size of `array` will be set to zero.
5.12 Pointer Arrays

### Name

Pointer Arrays – arrays of pointers to any type of data, which grow automatically as new elements are added.

### Synopsis

```c
#include <glib.h>

GPtrArray* g_ptr_array_new (void);
GPtrArray* g_ptr_array_sized_new (guint reserved_size);
GPtrArray* g_ptr_array_new_with_free_func (GDestroyNotify element_free_func);
void g_ptr_array_set_free_func (GPtrArray *array, GDestroyNotify element_free_func);
GPtrArray* g_ptr_array_ref (GPtrArray *array);
void g_ptr_array_unref (GPtrArray *array);
void g_ptr_array_add (GPtrArray *array, gpointer data);
gboolean g_ptr_array_remove (GPtrArray *array, gpointer data);
gboolean g_ptr_array_remove_index (GPtrArray *array, guint index_);
gboolean g_ptr_array_remove_fast (GPtrArray *array, gpointer data);
gpointer g_ptr_array_remove_index_fast (GPtrArray *array, guint index_);
void g_ptr_array_remove_range (GPtrArray *array, guint index_, guint length);
void g_ptr_array_sort (GPtrArray *array, GCompareFunc compare_func);
void g_ptr_array_sort_with_data (GPtrArray *array, GCompareDataFunc compare_func, gpointer user_data);
void g_ptr_array_set_size (GPtrArray *array, gint length);
#define g_ptr_array_index (array, index_)
gpointer* g_ptr_array_free (GPtrArray *array, gboolean free_seg);
```

### Description

*array*: a `GArray`.

*free_segment*: if `TRUE` the actual element data is freed as well.

*Returns*: the element data if `free_segment` is `FALSE`, otherwise `NULL`. The element data should be freed using `g_free()`.

---

**Note**

If array elements contain dynamically-allocated memory, they should be freed separately.
void g_ptr_array_foreach (GPtrArray *array, GFunc func, gpointer user_data);

**Description**

Pointer Arrays are similar to Arrays but are used only for storing pointers.

**NOTE**

If you remove elements from the array, elements at the end of the array are moved into the space previously occupied by the removed element. This means that you should not rely on the index of particular elements remaining the same. You should also be careful when deleting elements while iterating over the array.

To create a pointer array, use `g_ptr_array_new()`.
To add elements to a pointer array, use `g_ptr_array_add()`.
To remove elements from a pointer array, use `g_ptr_array_remove()`, `g_ptr_array_remove_index()` or `g_ptr_array_remove_index_fast()`.
To access an element of a pointer array, use `g_ptr_array_index()`.
To set the size of a pointer array, use `g_ptr_array_set_size()`.
To free a pointer array, use `g_ptr_array_free()`.

**Example 5.7 Using a GPtrArray**

```c
GPtrArray *gparray;
gchar *string1 = "one", *string2 = "two", *string3 = "three";
 gparray = g_ptr_array_new ();
g_ptr_array_add (gparray, (gpointer) string1);
g_ptr_array_add (gparray, (gpointer) string2);
g_ptr_array_add (gparray, (gpointer) string3);
if (g_ptr_array_index (gparray, 0) != (gpointer) string1)
    g_print ("ERROR: got %p instead of %p\n",
             g_ptr_array_index (gparray, 0), string1);
g_ptr_array_free (gparray, TRUE);
```

**Details**

**GPtrArray**

```c
typedef struct {
    gpointer *pdata;
    guint   len;
} GPtrArray;
```

Contains the public fields of a pointer array.

- **gpointer *pdata;** points to the array of pointers, which may be moved when the array grows.
- **guint len;** number of pointers in the array.

**g_ptr_array_new ()**

```c
GPtrArray* g_ptr_array_new (void);
```

Creates a new GPtrArray with a reference count of 1.

**Returns:** the new GPtrArray.
CHAPTER 5. GLIB DATA TYPES

5.12. POINTER ARRAYS

**g_ptr_array_sized_new()**

```c
GPtrArray* g_ptr_array_sized_new (guint reserved_size);
```

Creates a new GPtrArray with `reserved_size` pointers preallocated and a reference count of 1. This avoids frequent reallocation, if you are going to add many pointers to the array. Note however that the size of the array is still 0.

`reserved_size`: number of pointers preallocated.

**Returns**: the new GPtrArray.

**g_ptr_array_new_with_free_func()**

```c
GPtrArray* g_ptr_array_new_with_free_func (GDestroyNotify element_free_func);
```

Creates a new GPtrArray with a reference count of 1 and use `element_free_func` for freeing each element when the array is destroyed either via `g_ptr_array_unref()`, when `g_ptr_array_free()` is called with `free_segment` set to TRUE or when removing elements.

`element_free_func`: A function to free elements with destroy array or NULL.

**Returns**: A new GPtrArray.

Since 2.22

**g_ptr_array_set_free_func()**

```c
void g_ptr_array_set_free_func (GPtrArray *array, GDestroyNotify element_free_func);
```

Sets a function for freeing each element when `array` is destroyed either via `g_ptr_array_unref()`, when `g_ptr_array_free()` is called with `free_segment` set to TRUE or when removing elements.

`array`: A GPtrArray.

`element_free_func`: A function to free elements with destroy array or NULL.

Since 2.22

**g_ptr_array_ref()**

```c
GPtrArray* g_ptr_array_ref (GPtrArray *array);
```

Atomically increments the reference count of `array` by one. This function is MT-safe and may be called from any thread.

`array`: A GArray.

**Returns**: The passed in GPtrArray.

Since 2.22

**g_ptr_array_unref()**

```c
void g_ptr_array_unref (GPtrArray *array);
```

Atomically decrements the reference count of `array` by one. If the reference count drops to 0, the effect is the same as calling `g_ptr_array_free()` with `free_segment` set to TRUE. This function is MT-safe and may be called from any thread.

`array`: A GPtrArray.

Since 2.22
### g_ptr_array_add()

```c
void g_ptr_array_add (GPtrArray *array, gpointer data);
```

Adds a pointer to the end of the pointer array. The array will grow in size automatically if necessary.

**array**: a GPtrArray.

**data**: the pointer to add.

### g_ptr_array_remove()

```c
gboolean g_ptr_array_remove (GPtrArray *array, gpointer data);
```

Removes the first occurrence of the given pointer from the pointer array. The following elements are moved down one place. If **array** has a non-NULL **GDestroyNotify** function it is called for the removed element.

It returns **TRUE** if the pointer was removed, or **FALSE** if the pointer was not found.

**array**: a GPtrArray.

**data**: the pointer to remove.

**Returns**: **TRUE** if the pointer is removed. **FALSE** if the pointer is not found in the array.

### g_ptr_array_remove_index()

```c
gpointer g_ptr_array_remove_index (GPtrArray *array, guint index_);
```

Removes the pointer at the given index from the pointer array. The following elements are moved down one place. If **array** has a non-NULL **GDestroyNotify** function it is called for the removed element.

**array**: a GPtrArray.

**index_**: the index of the pointer to remove.

**Returns**: the pointer which was removed.

### g_ptr_array_remove_fast()

```c
gboolean g_ptr_array_remove_fast (GPtrArray *array, gpointer data);
```

Removes the first occurrence of the given pointer from the pointer array. The last element in the array is used to fill in the space, so this function does not preserve the order of the array. But it is faster than **g_ptr_array_remove()**. If **array** has a non-NULL **GDestroyNotify** function it is called for the removed element.

It returns **TRUE** if the pointer was removed, or **FALSE** if the pointer was not found.

**array**: a GPtrArray.

**data**: the pointer to remove.

**Returns**: **TRUE** if the pointer was found in the array.
CHAPTER 5. GLIB DATA TYPES

5.12. POINTER ARRAYS

**g_ptr_array_remove_index_fast()**

- **Function**: `gpointer g_ptr_array_remove_index_fast (GPtrArray *array, guint index_);`

  Removes the pointer at the given index from the pointer array. The last element in the array is used to fill in the space, so this function does not preserve the order of the array. But it is faster than `g_ptr_array_remove_index()`. If `array` has a non-NULL `GDestroyNotify` function it is called for the removed element.

  - **array**: a `GPtrArray`.
  - **index_**: the index of the pointer to remove.
  - **Returns**: the pointer which was removed.

**g_ptr_array_remove_range()**

- **Function**: `void g_ptr_array_remove_range (GPtrArray *array, guint index_, guint length);`

  Removes the given number of pointers starting at the given index from a `GPtrArray`. The following elements are moved to close the gap. If `array` has a non-NULL `GDestroyNotify` function it is called for the removed elements.

  - **array**: a `GPtrArray`.
  - **index_**: the index of the first pointer to remove.
  - **length**: the number of pointers to remove.

  Since 2.4

**g_ptr_array_sort()**

- **Function**: `void g_ptr_array_sort (GPtrArray *array, GCompareFunc compare_func);`

  Sorts the array, using `compare_func` which should be a `qsort()`-style comparison function (returns less than zero for first arg is less than second arg, zero for equal, greater than zero if first arg is greater than second arg).

  If two array elements compare equal, their order in the sorted array is undefined.

  **Note**

  - The comparison function for `g_ptr_array_sort()` doesn’t take the pointers from the array as arguments, it takes pointers to the pointers in the array.

  - **array**: a `GPtrArray`.
  - **compare_func**: comparison function.
**g_ptr_array_sort_with_data()**

```c
void g_ptr_array_sort_with_data (GPtrArray *array,
    GCompareDataFunc compare_func,
    gpointer user_data);
```

Like `g_ptr_array_sort()`, but the comparison function has an extra user data argument.

**NOTE**

The comparison function for `g_ptr_array_sort_with_data()` doesn’t take the pointers from the array as arguments, it takes pointers to the pointers in the array.

- **array**: a `GPtrArray`.
- **compare_func**: comparison function.
- **user_data**: data to pass to `compare_func`.

**g_ptr_array_set_size()**

```c
void g_ptr_array_set_size (GPtrArray *array,
    gint length);
```

Sets the size of the array, expanding it if necessary. New elements are set to `NULL`.

- **array**: a `GPtrArray`.
- **length**: the new length of the pointer array.

**g_ptr_array_index()**

```c
#define g_ptr_array_index(array,index_)
```

Returns the pointer at the given index of the pointer array.

- **array**: a `GPtrArray`.
- **index_**: the index of the pointer to return.

**Returns**: the pointer at the given index.

**g_ptr_array_free()**

```c
gpointer* g_ptr_array_free (GPtrArray *array,
    gboolean free_seg);
```

Frees the memory allocated for the `GPtrArray`. If `free_seg` is `TRUE` it frees the memory block holding the elements as well. Pass `FALSE` if you want to free the `GPtrArray` wrapper but preserve the underlying array for use elsewhere. If the reference count of `array` is greater than one, the `GPtrArray` wrapper is preserved but the size of `array` will be set to zero.

**NOTE**

If array contents point to dynamically-allocated memory, they should be freed separately if `free_seg` is `TRUE` and no `GDestroyNotify` function has been set for `array`.

547
**array**: a `GPtrArray`.

**free_seg**: if `TRUE` the actual pointer array is freed as well.

**Returns**: the pointer array if `free_seg` is `FALSE`, otherwise `NULL`. The pointer array should be freed using `g_free()`.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>g_ptr_array_foreach()</code></td>
<td>Calls a function for each element of a <code>GPtrArray</code>.</td>
</tr>
<tr>
<td><strong>array</strong>: a <code>GPtrArray</code></td>
<td></td>
</tr>
<tr>
<td><strong>func</strong>: the function to call for each array element</td>
<td></td>
</tr>
<tr>
<td><strong>user_data</strong>: user data to pass to the function</td>
<td></td>
</tr>
<tr>
<td>Since 2.4</td>
<td></td>
</tr>
</tbody>
</table>

## 5.13 Byte Arrays

### Name

Byte Arrays – arrays of bytes, which grow automatically as elements are added

### Synopsis

```c
#include <glib.h>

GByteArray;

GByteArray* g_byte_array_new
            (void);

GByteArray* g_byte_array_sized_new
            (quint reserved_size);

GByteArray* g_byte_array_ref
            (GByteArray *array);

void g_byte_array_unref
            (GByteArray *array);

GByteArray* g_byte_array_append
            (GByteArray *array,
             const guint8 *data,
             quint len);

GByteArray* g_byte_array_prepend
            (GByteArray *array,
             const guint8 *data,
             quint len);

GByteArray* g_byte_array_remove_index
            (GByteArray *array,
             quint index_);

GByteArray* g_byte_array_remove_index_fast
            (GByteArray *array,
             quint index_);

GByteArray* g_byte_array_remove_range
            (GByteArray *array,
             quint index_,
             quint length);

void g_byte_array_sort
            (GByteArray *array,
             GCompareFunc compare_func);

void g_byte_array_sort_with_data
            (GByteArray *array,
             GCompareDataFunc compare_func,
             gpointer user_data);

GByteArray* g_byte_array_set_size
            (GByteArray *array,
             quint length);

guint8* g_byte_array_free
            (GByteArray *array,
             gboolean free_segment);
```
Description

GByteArray is based on GArray, to provide arrays of bytes which grow automatically as elements are added.

To create a new GByteArray use g_byte_array_new().
To add elements to a GByteArray, use g_byte_array_append(), and g_byte_array_prepend().
To set the size of a GByteArray, use g_byte_array_set_size().
To free a GByteArray, use g_byte_array_free().

Example 5.8 Using a GByteArray

```c
GByteArray *gbarray;
gint i;
gbarray = g_byte_array_new();
for (i = 0; i < 10000; i++)
    g_byte_array_append(gbarray, (guint8*) "abcd", 4);
for (i = 0; i < 10000; i++)
{
    g_assert(gbarray->data[4*i] == 'a');
    g_assert(gbarray->data[4*i+1] == 'b');
    g_assert(gbarray->data[4*i+2] == 'c');
    g_assert(gbarray->data[4*i+3] == 'd');
}
g_byte_array_free(gbarray, TRUE);
```

Details

GByteArray

typedef struct {
    guint8 *data;
guint len;
} GByteArray;

The GByteArray struct allows access to the public fields of a GByteArray.

**guint8 *data;** a pointer to the element data. The data may be moved as elements are added to the GByteArray.

**guint len;** the number of elements in the GByteArray.

**g_byte_array_new ()**

GByteArray* g_byte_array_new(void);

Creates a new GByteArray with a reference count of 1.

**Returns:** the new GByteArray.

**g_byte_array_sized_new ()**

GByteArray* g_byte_array_sized_new (guint reserved_size);

Creates a new GByteArray with reserved_size bytes preallocated. This avoids frequent reallocation, if you are going to add many bytes to the array. Note however that the size of the array is still 0.

**reserved_size:** number of bytes preallocated.

**Returns:** the new GByteArray.
g_byte_array_ref ()

GByteArray * g_byte_array_ref (GByteArray *array);

Atomically increments the reference count of array by one. This function is MT-safe and may be called from any thread.

array : A GByteArray.

Returns : The passed in GByteArray.

Since 2.22

g_byte_array_unref ()

void g_byte_array_unref (GByteArray *array);

Atomically decrements the reference count of array by one. If the reference count drops to 0, all memory allocated by the array is released. This function is MT-safe and may be called from any thread.

array : A GByteArray.

Since 2.22

g_byte_array_append ()

GByteArray* g_byte_array_append (GByteArray *array, const guint8 *data, guint len);

Adds the given bytes to the end of the GByteArray. The array will grow in size automatically if necessary.

array : a GByteArray.
data : the byte data to be added.
len : the number of bytes to add.

Returns : the GByteArray.

g_byte_array_prepend ()

GByteArray* g_byte_array_prepend (GByteArray *array, const guint8 *data, guint len);

Adds the given data to the start of the GByteArray. The array will grow in size automatically if necessary.

array : a GByteArray.
data : the byte data to be added.
len : the number of bytes to add.

Returns : the GByteArray.
g_byte_array_remove_index ()

```c
GByteArray* g_byte_array_remove_index (GByteArray *array, guint index_);
```

Removes the byte at the given index from a GByteArray. The following bytes are moved down one place.

**array**: a GByteArray.

**index**: the index of the byte to remove.

**Returns**: the GByteArray.

g_byte_array_remove_index_fast ()

```c
GByteArray* g_byte_array_remove_index_fast (GByteArray *array, guint index_);
```

Removes the byte at the given index from a GByteArray. The last element in the array is used to fill in the space, so this function does not preserve the order of the GByteArray. But it is faster than g_byte_array_remove_index().

**array**: a GByteArray.

**index**: the index of the byte to remove.

**Returns**: the GByteArray.

g_byte_array_remove_range ()

```c
GByteArray* g_byte_array_remove_range (GByteArray *array, guint index_, guint length);
```

Removes the given number of bytes starting at the given index from a GByteArray. The following elements are moved to close the gap.

**array**: a GByteArray.

**index**: the index of the first byte to remove.

**length**: the number of bytes to remove.

**Returns**: the GByteArray.

Since 2.4

g_byte_array_sort ()

```c
void g_byte_array_sort (GByteArray *array, GCompareFunc compare_func);
```

Sorts a byte array, using `compare_func` which should be a qsort()-style comparison function (returns less than zero for first arg is less than second arg, zero for equal, greater than zero if first arg is greater than second arg).

If two array elements compare equal, their order in the sorted array is undefined.

**array**: a GByteArray.

**compare_func**: comparison function.
5.14 Balanced Binary Trees

Name
Balanced Binary Trees – a sorted collection of key/value pairs optimized for searching and traversing in order

Synopsis

```c
#include <glib.h>

GTree;
GTree* g_tree_new (GCompareFunc key_compare_func);
GTree* g_tree_new_with_data (GCompareDataFunc key_compare_func, gpointer key_compare_data);
GTree* g_tree_new_full (GCompareDataFunc key_compare_func, gpointer key_compare_data);
```
void g_tree_insert (GTree *tree, gpointer key, gpointer value);
void g_tree_replace (GTree *tree, gpointer key, gpointer value);
gint g_tree_nnodes (GTree *tree);
gint g_tree_height (GTree *tree);
gpointer g_tree_lookup (GTree *tree, gconstpointer key);
gboolean g_tree_lookup_extended (GTree *tree, gconstpointer lookup_key, gpointer *orig_key, gpointer *value);
void g_tree_foreach (GTree *tree, GTraverseFunc func, gpointer user_data);
void g_tree_traverse (GTree *tree, GTraverseFunc traverse_func, GTraverseType traverse_type, gpointer user_data);
(gboolean (*GTraverseFunc) (gpointer key, gpointer value, gpointer data));
enum GTraverseType;
gpointer g_tree_search (GTree *tree, GCompareFunc search_func, gconstpointer user_data);
gboolean g_tree_remove (GTree *tree, gconstpointer key);
gboolean g_tree_steal (GTree *tree, gconstpointer key);
void g_tree_destroy (GTree *tree);

Description

The GTree structure and its associated functions provide a sorted collection of key/value pairs optimized for searching and traversing in order.

To create a new GTree use g_tree_new().
To insert a key/value pair into a GTree use g_tree_insert().
To lookup the value corresponding to a given key, use g_tree_lookup() and g_tree_lookup_extended().
To find out the number of nodes in a GTree, use g_tree_nnodes(). To get the height of a GTree, use g_tree_height().
To traverse a GTree, calling a function for each node visited in the traversal, use g_tree_foreach().
To remove a key/value pair use g_tree_remove().
To destroy a GTree, use g_tree_destroy().

Details

GTree

typedef struct _GTree GTree;

The GTree struct is an opaque data structure representing a Balanced Binary Tree. It should be accessed only by using the following functions.
**g_tree_new ()**

```c
GTree* g_tree_new (GCompareFunc key_compare_func);
```

Creates a new GTree.

**key_compare_func**: the function used to order the nodes in the GTree. It should return values similar to the standard `strcmp()` function - 0 if the two arguments are equal, a negative value if the first argument comes before the second, or a positive value if the first argument comes after the second.

**Returns**: a new GTree.

**g_tree_new_with_data ()**

```c
GTree* g_tree_new_with_data (GCompareDataFunc key_compare_func,
                              gpointer key_compare_data);
```

Creates a new GTree with a comparison function that accepts user data. See `g_tree_new()` for more details.

**key_compare_func**: qsort()-style comparison function.

**key_compare_data**: data to pass to comparison function.

**Returns**: a new GTree.

**g_tree_new_full ()**

```c
GTree* g_tree_new_full (GCompareDataFunc key_compare_func,
                        gpointer key_compare_data,
                        GDestroyNotify key_destroy_func,
                        GDestroyNotify value_destroy_func);
```

Creates a new GTree like `g_tree_new()` and allows to specify functions to free the memory allocated for the key and value that get called when removing the entry from the GTree.

**key_compare_func**: qsort()-style comparison function.

**key_compare_data**: data to pass to comparison function.

**key_destroy_func**: a function to free the memory allocated for the key used when removing the entry from the GTree or NULL if you don’t want to supply such a function.

**value_destroy_func**: a function to free the memory allocated for the value used when removing the entry from the GTree or NULL if you don’t want to supply such a function.

**Returns**: a new GTree.

**g_tree_insert ()**

```c
void g_tree_insert (GTree *tree,
                    gpointer key,
                    gpointer value);
```


Inserts a key/value pair into a GTree. If the given key already exists in the GTree its corresponding value is set to the new value. If you supplied a value_destroy_func when creating the GTree, the old value is freed using that function. If you supplied a key_destroy_func when creating the GTree, the passed key is freed using that function.

The tree is automatically 'balanced' as new key/value pairs are added, so that the distance from the root to every leaf is as small as possible.

**tree**: a GTree.

**key**: the key to insert.

**value**: the value corresponding to the key.

### g_tree_replace()

```c
void g_tree_replace (GTree *tree,
                    gpointer key,
                    gpointer value);
```

Inserts a new key and value into a GTree similar to `g_tree_insert()`. The difference is that if the key already exists in the GTree, it gets replaced by the new key. If you supplied a value_destroy_func when creating the GTree, the old value is freed using that function. If you supplied a key_destroy_func when creating the GTree, the old key is freed using that function.

The tree is automatically 'balanced' as new key/value pairs are added, so that the distance from the root to every leaf is as small as possible.

**tree**: a GTree.

**key**: the key to insert.

**value**: the value corresponding to the key.

### g_tree_nnodes()

```c
gint g_tree_nnodes (GTree *tree);
```

Gets the number of nodes in a GTree.

**tree**: a GTree.

**Returns**: the number of nodes in the GTree.

### g_tree_height()

```c
gint g_tree_height (GTree *tree);
```

Gets the height of a GTree.

If the GTree contains no nodes, the height is 0. If the GTree contains only one root node the height is 1. If the root node has children the height is 2, etc.

**tree**: a GTree.

**Returns**: the height of the GTree.

### g_tree_lookup()

```c
gpointer g_tree_lookup (GTree *tree,
                        gconstpointer key);
```

Gets the value corresponding to the given key. Since a GTree is automatically balanced as key/value pairs are added, key lookup is very fast.

**tree**: a GTree.

**key**: the key to look up.

**Returns**: the value corresponding to the key, or NULL if the key was not found.
CHAPTER 5. GLIB DATA TYPES

5.14. BALANCED BINARY TREES

g_tree_lookup_extended ()

```c
gboolean g_tree_lookup_extended (GTree *tree,
    gconstpointer lookup_key,
    gpointer *orig_key,
    gpointer *value);
```

Looks up a key in the GTree, returning the original key and the associated value and a gboolean which is TRUE if the key was found. This is useful if you need to free the memory allocated for the original key, for example before calling g_tree_remove().

- **tree**: a GTree.
- **lookup_key**: the key to look up.
- **orig_key**: returns the original key.
- **value**: returns the value associated with the key.

**Returns**: TRUE if the key was found in the GTree.

---

g_tree_foreach ()

```c
void g_tree_foreach (GTree *tree,
    GTraverseFunc func,
    gpointer user_data);
```

Calls the given function for each of the key/value pairs in the GTree. The function is passed the key and value of each pair, and the given data parameter. The tree is traversed in sorted order.

The tree may not be modified while iterating over it (you can't add/remove items). To remove all items matching a predicate, you need to add each item to a list in your GTraverseFunc as you walk over the tree, then walk the list and remove each item.

- **tree**: a GTree.
- **func**: the function to call for each node visited. If this function returns TRUE, the traversal is stopped.
- **user_data**: user data to pass to the function.

---

g_tree_traverse ()

```c
void g_tree_traverse (GTree *tree,
    GTraverseFunc *traverse_func,
    GTraverseType *traverse_type,
    gpointer user_data);
```

**WARNING**

* g_tree_traverse has been deprecated since version 2.2 and should not be used in newly-written code. The order of a balanced tree is somewhat arbitrary. If you just want to visit all nodes in sorted order, use g_tree_foreach() instead. If you really need to visit nodes in a different order, consider using an N-ary Tree.

Calls the given function for each node in the GTree.

- **tree**: a GTree.
**traverse_func:** the function to call for each node visited. If this function returns **TRUE**, the traversal is stopped.

**traverse_type:** the order in which nodes are visited, one of **G_IN_ORDER**, **G_PRE_ORDER** and **G_POST_ORDER**.

**user_data:** user data to pass to the function.

### GTraverseFunc ()

```c
gboolean (*GTraverseFunc)(gpointer key, gpointer value, gpointer data);
```

Specifies the type of function passed to `g_tree_traverse()`. It is passed the key and value of each node, together with the `user_data` parameter passed to `g_tree_traverse()`. If the function returns **TRUE**, the traversal is stopped.

**key:** a key of a **GTree** node.

**value:** the value corresponding to the key.

**data:** user data passed to `g_tree_traverse()`.

**Returns:** **%TRUE** to stop the traversal.

### enum GTraverseType

```c
typedef enum
{
    G_IN_ORDER,
    G_PRE_ORDER,
    G_POST_ORDER,
    G_LEVEL_ORDER
} GTraverseType;
```

Specifies the type of traversal performed by `g_tree_traverse()`, `g_node_traverse()` and `g_node_find()`.

**G_IN_ORDER** visits a node’s left child first, then the node itself, then its right child. This is the one to use if you want the output sorted according to the compare function.

**G_PRE_ORDER** visits a node, then its children.

**G_POST_ORDER** visits the node’s children, then the node itself.

**G_LEVEL_ORDER** is not implemented for **Balanced Binary Trees**. For **N-ary Trees**, it visits the root node first, then its children, then its grandchildren, and so on. Note that this is less efficient than the other orders.

### g_tree_search ()

```c
gpointer g_tree_search (GTree *tree, GCompareFunc search_func ← , gconstpointer user_data);
```

Searches a **GTree** using `search_func`. The `search_func` is called with a pointer to the key of a key/value pair in the tree, and the passed in `user_data`. If `search_func` returns 0 for a key/value pair, then `g_tree_search_func()` will return the value of that pair. If `search_func` returns -1, searching will proceed among the key/value pairs that have a smaller key; if `search_func` returns 1, searching will proceed among the key/value pairs that have a larger key.
**tree**: a **GTree**.

**search_func**: a function used to search the **GTree**.

**user_data**: the data passed as the second argument to the **search_func** function.

**Returns**: the value corresponding to the found key, or **NULL** if the key was not found.

### `g_tree_remove ()`

```c
gboolean g_tree_remove (GTree *tree, gconstpointer key);
```

Removes a key/value pair from a **GTree**.

If the **GTree** was created using **g_tree_new_full()**, the key and value are freed using the supplied destroy functions, otherwise you have to make sure that any dynamically allocated values are freed yourself. If the key does not exist in the **GTree**, the function does nothing.

**tree**: a **GTree**.

**key**: the key to remove.

**Returns**: **TRUE** if the key was found (prior to 2.8, this function returned nothing)

### `g_tree_steal ()`

```c
gboolean g_tree_steal (GTree *tree, gconstpointer key);
```

Removes a key and its associated value from a **GTree** without calling the key and value destroy functions.

If the key does not exist in the **GTree**, the function does nothing.

**tree**: a **GTree**.

**key**: the key to remove.

**Returns**: **TRUE** if the key was found (prior to 2.8, this function returned nothing)

### `g_tree_destroy ()`

```c
void g_tree_destroy (GTree *tree);
```

Destroys the **GTree**. If keys and/or values are dynamically allocated, you should either free them first or create the **GTree** using **g_tree_new_full()**. In the latter case the destroy functions you supplied will be called on all keys and values before destroying the **GTree**.

**tree**: a **GTree**.

## 5.15 N-ary Trees

**Name**

N-ary Trees – trees of data with any number of branches
**Synopsis**

```c
#include <glib.h>

GNode;

GNode* g_node_new (gpointer data);
GNode* g_node_copy (GNode *node);
gpointer (*GCopyFunc) (gconstpointer src,
                       gpointer data);
GNode* g_node_copy_deep (GNode *node,
                       GCopyFunc copy_func,
                       gpointer data);

GNode* g_node_insert (GNode *parent,
                       gint position,
                       GNode *node);
GNode* g_node_insert_before (GNode *parent,
                       GNode *sibling,
                       GNode *node);
GNode* g_node_insert_after (GNode *parent,
                       GNode *sibling,
                       GNode *node);

#define g_node_append (parent, node)
GNode* g_node_prepend (GNode *parent,
                       GNode *node);
#define g_node_insert_data (parent, position, data)
#define g_node_insert_data_before (parent, sibling, data)
#define g_node_append_data (parent, data)
#define g_node_prepend_data (parent, data)

void g_node_reverse_children (GNode *node);
void g_node_traverse (GNode *root,
                       GTraverseType order,
                       GTraverseFlags flags,
                       gint max_depth,
                       GNodeTraverseFunc func,
                       gpointer data);

enum GTraverseFlags;
gboolean (*GNodeTraverseFunc) (GNode *node,
                                gpointer data);
void g_node_children_foreach (GNode *node,
                           GTraverseFlags flags,
                           GNodeForeachFunc func,
                           gpointer data);

void (*GNodeForeachFunc) (GNode *node,
                           gpointer data);
GNode* g_node_get_root (GNode *node);
GNode* g_node_find (GNode *root,
                    GTraverseType order,
                    GTraverseFlags flags,
                    gpointer data);
GNode* g_node_find_child (GNode *node,
                       GTraverseFlags flags,
                       gpointer data);
gint g_node_child_index (GNode *node,
                       gpointer data);
```
gint g_node_child_position (GNode *node,
GNode *child);
#define g_node_first_child (node)
GNode* g_node_last_child (GNode *node);
GNode* g_node_nth_child (GNode *node,
guint n);
GNode* g_node_first_sibling (GNode *node);
#define g_node_next_sibling (node)
#define g_node_prev_sibling (node)
GNode* g_node_last_sibling (GNode *node);
#define G_NODE_IS_LEAF (node)
#define G_NODE_IS_ROOT (node)
guint g_node_depth (GNode *node);
guint g_node_n_nodes (GNode *root,
GTraverseFlags flags);
guint g_node_n_children (GNode *node);
gboolean g_node_is_ancestor (GNode *node,
GNode *descendant);
guint g_node_max_height (GNode *root);
void g_node_unlink (GNode *node);
void g_node_destroy (GNode *root);
void g_node_push_allocator (gpointer dummy);
void g_node_pop_allocator (void);

Description

The GNode struct and its associated functions provide a N-ary tree data structure, where nodes in the
tree can contain arbitrary data.

To create a new tree use g_node_new().
To insert a node into a tree use g_node_insert(), g_node_insert_before(), g_node_append() and g_node_prepend().
To create a new node and insert it into a tree use g_node_insert_data(), g_node_insert_data_before(),
g_node_append_data() and g_node_prepend_data().
To reverse the children of a node use g_node_reverse_children().
To find a node use g_node_get_root(), g_node_find(), g_node_find_child(), g_node_child_index(),
g_node_child_position(), g_node_first_child(), g_node_last_child(), g_node_nth_child(), g_node_first_sibling(),
g_node_prev_sibling(), g_node_next_sibling() or g_node_last_sibling().
To get information about a node or tree use G_NODE_IS_LEAF(), G_NODE_IS_ROOT(), g_node_depth(),
g_node_n_nodes(), g_node_n_children(), g_node_is_ancestor() or g_node_max_height().
To traverse a tree, calling a function for each node visited in the traversal, use g_node_traverse() or
g_node_children_foreach().
To remove a node or subtree from a tree use g_node_unlink() or g_node_destroy().

Details

GNode

typedef struct {
  gpointer data;
  GNode  *next;
  GNode  *prev;
  GNode  *parent;
  GNode  *children;
} GNode;

The GNode struct represents one node in a N-ary Tree. fields
gpointer data; contains the actual data of the node.
GNode *next; points to the node’s next sibling (a sibling is another GNode with the same parent).

GNode *prev; points to the node’s previous sibling.

GNode *parent; points to the parent of the GNode, or is NULL if the GNode is the root of the tree.

GNode *children; The children field points to the first child of the GNode. The other children are accessed by using the next pointer of each child.

g_node_new()

GNode* g_node_new (gpointer data);

Creates a new GNode containing the given data. Used to create the first node in a tree.

data: the data of the new node

Returns: a new GNode

g_node_copy()

GNode* g_node_copy (GNode *node);

Recursively copies a GNode (but does not deep-copy the data inside the nodes, see g_node_copy_deep() if you need that).

node: a GNode

Returns: a new GNode containing the same data pointers

GCopyFunc()

gpointer (*GCopyFunc)(gconstpointer src, gpointer data);

A function of this signature is used to copy the node data when doing a deep-copy of a tree.

src: A pointer to the data which should be copied

data: Additional data

Returns: A pointer to the copy

Since 2.4

g_node_copy_deep()

GNode* g_node_copy_deep (GNode *node,
GCopyFunc copy_func,
gpointer data);

Recursively copies a GNode and its data.

node: a GNode

copy_func: the function which is called to copy the data inside each node, or NULL to use the original data.

data: data to pass to copy_func

Returns: a new GNode containing copies of the data in node.

Since 2.4
g_node_insert()

GNode* g_node_insert (GNode *parent, gint position, GNode *node);

Inserts a GNode beneath the parent at the given position.

parent: the GNode to place node under
position: the position to place node at, with respect to its siblings. If position is -1, node is inserted as the last child of parent.
node: the GNode to insert

Returns: the inserted GNode

g_node_insert_before()

GNode* g_node_insert_before (GNode *parent, GNode *sibling, GNode *node);

Inserts a GNode beneath the parent before the given sibling.

parent: the GNode to place node under
sibling: the sibling GNode to place node before. If sibling is NULL, the node is inserted as the last child of parent.
node: the GNode to insert

Returns: the inserted GNode

g_node_insert_after()

GNode* g_node_insert_after (GNode *parent, GNode *sibling, GNode *node);

Inserts a GNode beneath the parent after the given sibling.

parent: the GNode to place node under
sibling: the sibling GNode to place node after. If sibling is NULL, the node is inserted as the first child of parent.
node: the GNode to insert

Returns: the inserted GNode

g_node_append()

#define g_node_append (parent, node)

Inserts a GNode as the last child of the given parent.

parent: the GNode to place the new GNode under
node: the GNode to insert

Returns: the inserted GNode
g_node_prepend()

```c
GNode* g_node_prepend (GNode *parent, GNode *node);
```

Inserts a *GNode* as the first child of the given parent.

**parent**: the *GNode* to place the new *GNode* under

**node**: the *GNode* to insert

**Returns**: the inserted *GNode*

g_node_insert_data()

```c
#define g_node_insert_data(parent, position, data)
```

Inserts a new *GNode* at the given position.

**parent**: the *GNode* to place the new *GNode* under

**position**: the position to place the new *GNode* at. If position is -1, the new *GNode* is inserted as the last child of *parent*

**data**: the data for the new *GNode*

**Returns**: the new *GNode*

g_node_insert_data_before()

```c
#define g_node_insert_data_before(parent, sibling, data)
```

Inserts a new *GNode* before the given sibling.

**parent**: the *GNode* to place the new *GNode* under

**sibling**: the sibling *GNode* to place the new *GNode* before

**data**: the data for the new *GNode*

**Returns**: the new *GNode*

g_node_append_data()

```c
#define g_node_append_data(parent, data)
```

Inserts a new *GNode* as the last child of the given parent.

**parent**: the *GNode* to place the new *GNode* under

**data**: the data for the new *GNode*

**Returns**: the new *GNode*

g_node_prepend_data()

```c
#define g_node_prepend_data(parent, data)
```

Inserts a new *GNode* as the first child of the given parent.

**parent**: the *GNode* to place the new *GNode* under

**data**: the data for the new *GNode*

**Returns**: the new *GNode*
5.15. N-ARY TREES

**g_node_reverse_children()**

```c
void g_node_reverse_children (GNode *node);
```

Reverses the order of the children of a GNode. (It doesn’t change the order of the grandchildren.)

**node**: a GNode.

**g_node_traverse()**

```c
void g_node_traverse (GNode *root, GTraverseType order, GTraverseFlags flags, gint max_depth, GNodeTraverseFunc func, gpointer data);
```

Traverses a tree starting at the given root GNode. It calls the given function for each node visited. The traversal can be halted at any point by returning TRUE from func.

**root**: the root GNode of the tree to traverse

**order**: the order in which nodes are visited - G_IN_ORDER, G_PRE_ORDER, G_POST_ORDER, or G_LEVEL_ORDER.

**flags**: which types of children are to be visited, one of G_TRAVERSE_ALL, G_TRAVERSE_LEAVES and G_TRAVERSE_NON_LEAVES

**max_depth**: the maximum depth of the traversal. Nodes below this depth will not be visited. If max_depth is -1 all nodes in the tree are visited. If depth is 1, only the root is visited. If depth is 2, the root and its children are visited. And so on.

**func**: the function to call for each visited GNode

**data**: user data to pass to the function

**enum GTraverseFlags**

```c
typedef enum
{
    G_TRAVERSE_LEAVES = 1 << 0,
    G_TRAVERSE_NON_LEAVES = 1 << 1,
    G_TRAVERSE_ALL = G_TRAVERSE_LEAVES | G_TRAVERSE_NON_LEAVES,
    G_TRAVERSE_MASK = 0x03,
    G_TRAVERSE_LEAVES = G_TRAVERSE_LEAVES,
    G_TRAVERSE_NON_LEAVES = G_TRAVERSE_NON_LEAVES
} GTraverseFlags;
```

Specifies which nodes are visited during several of the tree functions, including g_node_traverse() and g_node_find().

**G_TRAVERSE_LEAVES** only leaf nodes should be visited. This name has been introduced in 2.6, for older version use G_TRAVERSE_LEAVES.

**G_TRAVERSE_NON_LEAVES** only non-leaf nodes should be visited. This name has been introduced in 2.6, for older version use G_TRAVERSE_NON_LEAVES.

**G_TRAVERSE_ALL** all nodes should be visited.

**G_TRAVERSE_MASK** a mask of all traverse flags.

**G_TRAVERSE_LEAVES** identical to G_TRAVERSE_LEAVES.

**G_TRAVERSE_NON_LEAVES** identical to G_TRAVERSE_NON_LEAVES.
GNodeTraverseFunc ()

gboolean (*GNodeTraverseFunc) (GNode *node, gpointer data);

Specifies the type of function passed to g_node_traverse(). The function is called with each of the nodes visited, together with the user data passed to g_node_traverse(). If the function returns TRUE, then the traversal is stopped.

**node**: a GNode.

**data**: user data passed to g_node_traverse().

**Returns**: %TRUE to stop the traversal.

GNodeChildrenForeach ()

void g_node_children_foreach (GNode *node, GTraverseFlags flags, GNodeForeachFunc func, gpointer data);

Calls a function for each of the children of a GNode. Note that it doesn’t descend beneath the child nodes.

**node**: a GNode

**flags**: which types of children are to be visited, one of G_TRAVERSE_ALL, G_TRAVERSE_LEAVES and G_TRAVERSE_NON_LEAVES

**func**: the function to call for each visited node

**data**: user data to pass to the function

GNodeForeachFunc ()

void (*GNodeForeachFunc) (GNode *node, gpointer data);

Specifies the type of function passed to g_node_children_foreach(). The function is called with each child node, together with the user data passed to g_node_children_foreach().

**node**: a GNode.

**data**: user data passed to g_node_children_foreach().

g_node_get_root ()

GNode* g_node_get_root (GNode *node);

Gets the root of a tree.

**node**: a GNode

**Returns**: the root of the tree
g_node_find()

GNode* g_node_find (GNode *root, GTraverseType order, GTraverseFlags flags, gpointer data);

Finds a GNode in a tree.

**root**: the root GNode of the tree to search

**order**: the order in which nodes are visited - G_IN_ORDER, G_PRE_ORDER, G_POST_ORDER, or G_LEVEL_ORDER

**flags**: which types of children are to be searched, one of G_TRAVERSE_ALL, G_TRAVERSE_LEAVES and G_TRAVERSE_NON_LEAVES

**data**: the data to find

**Returns**: the found GNode, or NULL if the data is not found

---

g_node_find_child()

GNode* g_node_find_child (GNode *node, GTraverseFlags flags, gpointer data);

Finds the first child of a GNode with the given data.

**node**: a GNode

**flags**: which types of children are to be searched, one of G_TRAVERSE_ALL, G_TRAVERSE_LEAVES and G_TRAVERSE_NON_LEAVES

**data**: the data to find

**Returns**: the found child GNode, or NULL if the data is not found

---

g_node_child_index()

gint g_node_child_index (GNode *node, gpointer data);

Gets the position of the first child of a GNode which contains the given data.

**node**: a GNode

**data**: the data to find

**Returns**: the index of the child of node which contains data, or -1 if the data is not found

---

g_node_child_position()

gint g_node_child_position (GNode *node, GNode *child);

Gets the position of a GNode with respect to its siblings. child must be a child of node. The first child is numbered 0, the second 1, and so on.

**node**: a GNode

**child**: a child of node

**Returns**: the position of child with respect to its siblings
CHAPTER 5. GLIB DATA TYPES

5.15. N-ARY TREES

**g_node_first_child()**

```
define g_node_first_child(node)
```

Gets the first child of a GNode.

**node**: a GNode

**Returns**: the first child of *node*, or NULL if *node* is NULL or has no children

**g_node_last_child()**

```
GNode* g_node_last_child(GNode *node);
```

Gets the last child of a GNode.

**node**: a GNode (must not be NULL)

**Returns**: the last child of *node*, or NULL if *node* has no children

**g_node_nth_child()**

```
GNode* g_node_nth_child(GNode *node, guint n);
```

Gets a child of a GNode, using the given index. The first child is at index 0. If the index is too big, NULL is returned.

**node**: a GNode

**n**: the index of the desired child

**Returns**: the child of *node* at index *n*

**g_node_first_sibling()**

```
GNode* g_node_first_sibling(GNode *node);
```

Gets the first sibling of a GNode. This could possibly be the node itself.

**node**: a GNode

**Returns**: the first sibling of *node*

**g_node_next_sibling()**

```
define g_node_next_sibling(node)
```

Gets the next sibling of a GNode.

**node**: a GNode

**Returns**: the next sibling of *node*, or NULL if *node* is NULL

**g_node_prev_sibling()**

```
define g_node_prev_sibling(node)
```

Gets the previous sibling of a GNode.

**node**: a GNode

**Returns**: the previous sibling of *node*, or NULL if *node* is NULL
CHAPTER 5. GLIB DATA TYPES

5.15. N-ARY TREES

**g_node_last_sibling()**

```c
GNode* g_node_last_sibling (GNode *node);
```

Gets the last sibling of a GNode. This could possibly be the node itself.

*node*: a GNode

*Returns*: the last sibling of *node*

**G_NODE_IS_LEAF()**

```c
#define G_NODE_IS_LEAF(node) (((GNode *) (node))->children == NULL)
```

Returns TRUE if a GNode is a leaf node.

*node*: a GNode

*Returns*: TRUE if the GNode is a leaf node (i.e. it has no children)

**G_NODE_IS_ROOT()**

```c
#define G_NODE_IS_ROOT(node)
```

Returns TRUE if a GNode is the root of a tree.

*node*: a GNode

*Returns*: TRUE if the GNode is the root of a tree (i.e. it has no parent or siblings)

**g_node_depth()**

```c
guint g_node_depth (GNode *node);
```

Gets the depth of a GNode.

If *node* is NULL the depth is 0. The root node has a depth of 1. For the children of the root node the depth is 2. And so on.

*node*: a GNode

*Returns*: the depth of the GNode

**g_node_n_nodes()**

```c
guint g_node_n_nodes (GNode *root, GTraverseFlags flags);
```

Gets the number of nodes in a tree.

*root*: a GNode

*flags*: which types of children are to be counted, one of G_TRAVERSE_ALL, G_TRAVERSE_LEAVES and G_TRAVERSE_NON_LEAVES

*Returns*: the number of nodes in the tree

**g_node_n_children()**

```c
guint g_node_n_children (GNode *node);
```

Gets the number of children of a GNode.

*node*: a GNode

*Returns*: the number of children of *node*
CHAPTER 5. GLIB DATA TYPES

5.15. N-ARY TREES

**g_node_is_ancestor()**

```c
gboolean g_node_is_ancestor (GNode *node, GNode *descendant);
```

Returns **TRUE** if `node` is an ancestor of `descendant`. This is true if `node` is the parent of `descendant`, or if `node` is the grandparent of `descendant` etc.

- **node**: a GNode
- **descendant**: a GNode
- **Returns**: **TRUE** if `node` is an ancestor of `descendant`

**g_node_max_height()**

```c
guint g_node_max_height (GNode *root);
```

Gets the maximum height of all branches beneath a GNode. This is the maximum distance from the GNode to all leaf nodes.

- **root**: a GNode
- **Returns**: the maximum height of the tree beneath `root`

**g_node_unlink()**

```c
void g_node_unlink (GNode *node);
```

Unlinks a GNode from a tree, resulting in two separate trees.

- **node**: the GNode to unlink, which becomes the root of a new tree

**g_node_destroy()**

```c
void g_node_destroy (GNode *root);
```

Removes `root` and its children from the tree, freeing any memory allocated.

- **root**: the root of the tree/subtree to destroy

**g_node_push_allocator()**

```c
void g_node_push_allocator (gpointer dummy);
```

**WARNING**

- **g_node_push_allocator** has been deprecated since version 2.10 and should not be used in newly-written code. It does nothing, since GNode has been converted to the slice allocator.

- **dummy**: the GAllocator to use when allocating GNode elements.

Sets the allocator to use to allocate GNode elements. Use `g_node_pop_allocator()` to restore the previous allocator.

Note that this function is not available if GLib has been compiled with `--disable-mem-pools`.
g_node_pop_allocator()

```c
void g_node_pop_allocator (void);
```

Warning

```
g_node_pop_allocator has been deprecated since version 2.10 and should not be used in newly-written code. It does nothing, since GNode has been converted to the slice allocator.
```

Restores the previous GAllocator, used when allocating GNode elements.
Note that this function is not available if GLib has been compiled with `--disable-mem-pools`.

5.16 Quarks

Name

Quarks – a 2-way association between a string and a unique integer identifier

Synopsis

```c
#include <glib.h>

typedef GQuark;
GQuark g_quark_from_string (const gchar *string);
GQuark g_quark_from_static_string (const gchar *string);
const gchar* g_quark_to_string (GQuark quark);
GQuark g_quark_try_string (const gchar *string);
const gchar* g_intern_string (const gchar *string);
const gchar* g_intern_static_string (const gchar *string);
```

Description

Quarks are associations between strings and integer identifiers. Given either the string or the GQuark identifier it is possible to retrieve the other.

Quarks are used for both Datasets and Keyed Data Lists.
To create a new quark from a string, use `g_quark_from_string()` or `g_quark_from_static_string()`.
To find the string corresponding to a given GQuark, use `g_quark_to_string()`.
To find the GQuark corresponding to a given string, use `g_quark_try_string()`.

Another use for the string pool maintained for the quark functions is string interning, using `g_intern_string()` or `g_intern_static_string()`. An interned string is a canonical representation for a string. One important advantage of interned strings is that they can be compared for equality by a simple pointer comparison, rather than using `strcmp()`.

Details

GQuark

```c
typedef guint32 GQuark;
```

A GQuark is a non-zero integer which uniquely identifies a particular string. A GQuark value of zero is associated to `NULL`.

570
CHAPTER 5. GLIB DATA TYPES

5.16. QUARKS

**g_quark_from_string ()**

GQuark g_quark_from_string (const gchar *string);

Gets the GQuark identifying the given string. If the string does not currently have an associated GQuark, a new GQuark is created, using a copy of the string.

*string*: a string.

**Returns**: the GQuark identifying the string, or 0 if *string* is NULL.

**g_quark_from_static_string ()**

GQuark g_quark_from_static_string (const gchar *string);

Gets the GQuark identifying the given (static) string. If the string does not currently have an associated GQuark, a new GQuark is created, linked to the given string.

Note that this function is identical to g_quark_from_string() except that if a new GQuark is created the string itself is used rather than a copy. This saves memory, but can only be used if the string will always exist. It can be used with statically allocated strings in the main program, but not with statically allocated memory in dynamically loaded modules, if you expect to ever unload the module again (e.g. do not use this function in GTK+ theme engines).

*string*: a string.

**Returns**: the GQuark identifying the string, or 0 if *string* is NULL.

**g_quark_to_string ()**

const gchar* g_quark_to_string (GQuark quark);

Gets the string associated with the given GQuark.

*quark*: a GQuark.

**Returns**: the string associated with the GQuark.

**g_quark_try_string ()**

GQuark g_quark_try_string (const gchar *string);

Gets the GQuark associated with the given string, or 0 if string is NULL or it has no associated GQuark.

If you want the GQuark to be created if it doesn’t already exist, use g_quark_from_string() or g_quark_from_static_string().

*string*: a string.

**Returns**: the GQuark associated with the string, or 0 if *string* is NULL or there is no GQuark associated with it.

**g_intern_string ()**

const gchar* g_intern_string (const gchar *string);

Returns a canonical representation for *string*. Interned strings can be compared for equality by comparing the pointers, instead of using strcmp().

*string*: a string.

**Returns**: a canonical representation for the string

Since 2.10

571
CHAPTER 5. GLIB DATA TYPES

5.17. KEYED DATA LISTS

\[ \text{g_intern_static_string}() \]

\[
\begin{array}{ll}
\text{const gchar*} & \text{g_intern_static_string} (\text{const gchar *string});
\end{array}
\]

Returns a canonical representation for \texttt{string}. Interned strings can be compared for equality by comparing the pointers, instead of using \texttt{strcmp()}. \texttt{g_intern_static_string()} does not copy the string, therefore \texttt{string} must not be freed or modified.

\texttt{string} : a static string

\texttt{Returns} : a canonical representation for the string

Since 2.10

5.17 Keyed Data Lists

\textbf{Name}

Keyed Data Lists – lists of data elements which are accessible by a string or GQuark identifier

\textbf{Synopsis}

\#include <glib.h>

```
void g_datalist_init (GData **datalist);
#define g_datalist_id_set_data (dl, q, d)
void g_datalist_id_set_data_full (GData **datalist, GQuark key_id, gpointer data, GDestroyNotify destroy_func);
gpointer g_datalist_id_get_data (GData **datalist, GQuark key_id);
#define g_datalist_id_remove_data (dl, q)
gpointer g_datalist_id_remove_no_notify (GData **datalist, GQuark key_id);
#define g_datalist_set_data (dl, k, d)
#define g_datalist_set_data_full (dl, k, d, f)
#define g_datalist_set_data_full2 (dl, k, d, f)
#define g_datalist_get_data (dl, k)
#define g_datalist_remove_data (dl, k)
#define g_datalist_remove_no_notify (dl, k)
void g_datalist_foreach (GData **datalist, GDataForeachFunc func, gpointer user_data);
void g_datalist_clear (GData **datalist);
void g_datalist_set_flags (GData **datalist, guint flags);
void g_datalist_unset_flags (GData **datalist, guint flags);
guint g_datalist_get_flags (GData **datalist);
#define G_DATALIST_FLAGS_MASK
```

\textbf{Description}

Keyed data lists provide lists of arbitrary data elements which can be accessed either with a string or with a GQuark corresponding to the string.
The \textit{GQuark} methods are quicker, since the strings have to be converted to \textit{GQuarks} anyway. Data lists are used for associating arbitrary data with \textit{GObjects}, using \texttt{g\_object\_set\_data} and related functions.

To create a datalist, use \texttt{g\_datalist\_init}.

To add data elements to a datalist use \texttt{g\_datalist\_id\_set\_data}, \texttt{g\_datalist\_id\_set\_data\_full}, \texttt{g\_datalist\_set\_data} and \texttt{g\_datalist\_set\_data\_full}.

To get data elements from a datalist use \texttt{g\_datalist\_id\_get\_data} and \texttt{g\_datalist\_get\_data}.

To iterate over all data elements in a datalist use \texttt{g\_datalist\_foreach} (not thread-safe).

To remove data elements from a datalist use \texttt{g\_datalist\_id\_remove\_data} and \texttt{g\_datalist\_remove\_data}.

To remove all data elements from a datalist, use \texttt{g\_datalist\_clear}.

\section*{Details}

\textbf{GData}

\begin{verbatim}
typedef struct _GData GData;

The \textit{GData} struct is an opaque data structure to represent a \textit{Keyed Data List}. It should only be accessed via the following functions.

\begin{itemize}
\item \texttt{g\_datalist\_init}()
\end{itemize}

\begin{verbatim}
void g_datalist_init (GData **datalist);
\end{verbatim}

Resets the datalist to \texttt{NULL}. It does not free any memory or call any destroy functions.

\texttt{datalist}: a pointer to a pointer to a datalist.

\begin{itemize}
\item \texttt{g\_datalist\_id\_set\_data}()
\end{itemize}

\begin{verbatim}
#define g_datalist_id_set_data(dl, q, d)

Sets the data corresponding to the given \textit{GQuark} id. Any previous data with the same key is removed, and its destroy function is called.

\texttt{dl}: a datalist.
\texttt{q}: the \textit{GQuark} to identify the data element.
\texttt{d}: the data element, or \texttt{NULL} to remove any previous element corresponding to \texttt{q}.

\begin{itemize}
\item \texttt{g\_datalist\_id\_set\_data\_full}()
\end{itemize}

\begin{verbatim}
void g_datalist_id_set_data_full (GData **datalist,
    GQuark key_id,
    gpointer data,
    GDestroyNotify destroy_func);
\end{verbatim}

Sets the data corresponding to the given \textit{GQuark} id, and the function to be called when the element is removed from the datalist. Any previous data with the same key is removed, and its destroy function is called.

\texttt{datalist}: a datalist.

\texttt{key\_id}: the \textit{GQuark} to identify the data element.

\texttt{data}: the data element or \texttt{NULL} to remove any previous element corresponding to \texttt{key\_id}.

\texttt{destroy\_func}: the function to call when the data element is removed. This function will be called with the data element and can be used to free any memory allocated for it. If \texttt{data} is \texttt{NULL}, then \texttt{destroy\_func} must also be \texttt{NULL}.
g_datalist_id_get_data ()

```
gpointer g_datalist_id_get_data (GData **datalist, GQuark key_id);
```

Retrieves the data element corresponding to `key_id`.

datalist: a datalist.

key_id: the GQuark identifying a data element.

Returns: the data element, or NULL if it is not found.

g_datalist_id_remove_data()

```
#define g_datalist_id_remove_data(dl, q)
```

Removes an element, using its GQuark identifier.

dl: a datalist.

q: the GQuark identifying the data element.

g_datalist_id_remove_no_notify ()

```
gpointer g_datalist_id_remove_no_notify (GData **datalist, GQuark key_id);
```

Removes an element, without calling its destroy notification function.

datalist: a datalist.

key_id: the GQuark identifying a data element.

Returns: the data previously stored at `key_id`, or NULL if none.

g_datalist_set_data()

```
#define g_datalist_set_data(dl, k, d)
```

Sets the data element corresponding to the given string identifier.

dl: a datalist.

k: the string to identify the data element.

d: the data element, or NULL to remove any previous element corresponding to `k`.

g_datalist_set_data_full()

```
#define g_datalist_set_data_full(dl, k, d, f)
```

Sets the data element corresponding to the given string identifier, and the function to be called when the data element is removed.

dl: a datalist.

k: the string to identify the data element.

d: the data element, or NULL to remove any previous element corresponding to `k`.

f: the function to call when the data element is removed. This function will be called with the data element and can be used to free any memory allocated for it. If `d` is NULL, then `f` must also be NULL.
CHAPTER 5. GLIB DATA TYPES

5.17. KEYED DATA LISTS

**g_datalist_get_data()**

```c
#define g_datalist_get_data(dl, k)
```

Gets a data element, using its string identifier. This is slower than **g_datalist_id_get_data()** because the string is first converted to a GQuark.

- **dl**: a datalist.
- **k**: the string identifying a data element.
- **Returns**: the data element, or **NULL** if it is not found.

**g_datalist_remove_data()**

```c
#define g_datalist_remove_data(dl, k)
```

Removes an element using its string identifier. The data element’s destroy function is called if it has been set.

- **dl**: a datalist.
- **k**: the string identifying the data element.

**g_datalist_remove_no_notify()**

```c
#define g_datalist_remove_no_notify(dl, k)
```

Removes an element, without calling its destroy notifier.

- **dl**: a datalist.
- **k**: the string identifying the data element.

**g_datalist_foreach ()**

```c
void g_datalist_foreach (GData **datalist, GDataForeachFunc func, gpointer user_data);
```

Calls the given function for each data element of the datalist. The function is called with each data element’s GQuark id and data, together with the given **user_data** parameter. Note that this function is NOT thread-safe. So unless **datalist** can be protected from any modifications during invocation of this function, it should not be called.

- **datalist**: a datalist.
- **func**: the function to call for each data element.
- **user_data**: user data to pass to the function.

**g_datalist_clear ()**

```c
void g_datalist_clear (GData **datalist);
```

Frees all the data elements of the datalist. The data elements’ destroy functions are called if they have been set.

- **datalist**: a datalist.
**g_datalist_set_flags ()**

```c
void g_datalist_set_flags (GData **datalist, guint flags);
```

Turns on flag values for a data list. This function is used to keep a small number of boolean flags in an object with a data list without using any additional space. It is not generally useful except in circumstances where space is very tight. (It is used in the base GObject type, for example.)

- **datalist**: pointer to the location that holds a list
- **flags**: the flags to turn on. The values of the flags are restricted by `G_DATALIST_FLAGS_MASK` (currently 3; giving two possible boolean flags). A value for `flags` that doesn’t fit within the mask is an error.

Since 2.8

**g_datalist_unset_flags ()**

```c
void g_datalist_unset_flags (GData **datalist, guint flags);
```

Turns off flag values for a data list. See `g_datalist_unset_flags()`

- **datalist**: pointer to the location that holds a list
- **flags**: the flags to turn off. The values of the flags are restricted by `G_DATALIST_FLAGS_MASK` (currently 3; giving two possible boolean flags). A value for `flags` that doesn’t fit within the mask is an error.

Since 2.8

**g_datalist_get_flags ()**

```c
guint g_datalist_get_flags (GData **datalist);
```

Gets flags values packed in together with the datalist. See `g_datalist_set_flags()`.

- **datalist**: pointer to the location that holds a list

  **Returns**: the flags of the datalist

Since 2.8

**G_DATALIST_FLAGS_MASK**

```c
#define G_DATALIST_FLAGS_MASK 0x3
```

A bitmask that restricts the possible flags passed to `g_datalist_set_flags()`. Passing a flags value where flags & ~G_DATALIST_FLAGS_MASK != 0 is an error.

### 5.18 Datasets

**Name**

Datasets – associate groups of data elements with particular memory locations
5.18. DATASETS

Synopsis

```c
#include <glib.h>

#define g_dataset_id_set_data(l, k, d) (l, k, d)

void g_dataset_id_set_data_full (gconstpointer dataset_location, GQuark key_id, gpointer data, GDestroyNotify destroy_func);

void (*GDestroyNotify) (gpointer data);

gpointer g_dataset_id_get_data (gconstpointer dataset_location, GQuark key_id);

#define g_dataset_id_remove_data (l, k) (l, k)

gpointer g_dataset_id_remove_no_notify (gconstpointer dataset_location, GQuark key_id);

#define g_dataset_set_data (l, k, d) (l, k, d)

#define g_dataset_set_data_full (l, k, d, f) (l, k, d, f)

#define g_dataset_get_data (l, k) (l, k)

#define g_dataset_remove_data (l, k) (l, k)

#define g_dataset_remove_no_notify (l, k) (l, k)

void g_dataset_foreach (gconstpointer dataset_location, GDataForeachFunc func, gpointer user_data);

void (*GDataForeachFunc) (GQuark key_id, gpointer data, gpointer user_data);

void g_dataset_destroy (gconstpointer dataset_location);
```

Description

Datasets associate groups of data elements with particular memory locations. These are useful if you need to associate data with a structure returned from an external library. Since you cannot modify the structure, you use its location in memory as the key into a dataset, where you can associate any number of data elements with it.

There are two forms of most of the dataset functions. The first form uses strings to identify the data elements associated with a location. The second form uses GQuark identifiers, which are created with a call to g_quark_from_string() or g_quark_from_static_string(). The second form is quicker, since it does not require looking up the string in the hash table of GQuark identifiers.

There is no function to create a dataset. It is automatically created as soon as you add elements to it.

To add data elements to a dataset use g_dataset_id_set_data(), g_dataset_id_set_data_full(), g_dataset_set_data() and g_dataset_set_data_full().

To get data elements from a dataset use g_dataset_id_get_data() and g_dataset_get_data().

To iterate over all data elements in a dataset use g_dataset_foreach() (not thread-safe).

To remove data elements from a dataset use g_dataset_id_remove_data() and g_dataset_remove_data().

To destroy a dataset, use g_dataset_destroy().

Details

**g_dataset_id_set_data()**

```c
#define g_dataset_id_set_data(l, k, d) (l, k, d)
```

Sets the data element associated with the given GQuark id. Any previous data with the same key is removed, and its destroy function is called.

*l*: the location identifying the dataset.
**k**: the GQuark id to identify the data element.

**d**: the data element.

### g_dataset_id_set_data_full()

```c
void g_dataset_id_set_data_full (gconstpointer dataset_location,
                                GQuark key_id,
                                gpointer data,
                                GDestroyNotify destroy_func);
```

Sets the data element associated with the given GQuark id, and also the function to call when the data element is destroyed. Any previous data with the same key is removed, and its destroy function is called.

**dataset_location**: the location identifying the dataset.

**key_id**: the GQuark id to identify the data element.

**data**: the data element.

**destroy_func**: the function to call when the data element is removed. This function will be called with the data element and can be used to free any memory allocated for it.

### GDestroyNotify()

```c
void (*GDestroyNotify) (gpointer data);
```

Specifies the type of function which is called when a data element is destroyed. It is passed the pointer to the data element and should free any memory and resources allocated for it.

**data**: the data element.

### g_dataset_id_get_data()

```c
gpointer g_dataset_id_get_data (gconstpointer dataset_location,
                                GQuark key_id);
```

Gets the data element corresponding to a GQuark.

**dataset_location**: the location identifying the dataset.

**key_id**: the GQuark id to identify the data element.

**Returns**: the data element corresponding to the GQuark, or NULL if it is not found.

### g_dataset_id_remove_data()

```c
#define g_dataset_id_remove_data(l, k)
```

Removes a data element from a dataset. The data element’s destroy function is called if it has been set.

**l**: the location identifying the dataset.

**k**: the GQuark id identifying the data element.
**g_dataset_id_remove_no_notify()**

```c
void g_dataset_id_remove_no_notify (gconstpointer dataset_location, GQuark key_id);
```

Removes an element, without calling its destroy notification function.

- **dataset_location**: the location identifying the dataset.
- **key_id**: the GQuark ID identifying the data element.

**Returns**: the data previously stored at `key_id`, or NULL if none.

**g_dataset_set_data()**

```c
#define g_dataset_set_data(l, k, d)
```

Sets the data corresponding to the given string identifier.

- **l**: the location identifying the dataset.
- **k**: the string to identify the data element.
- **d**: the data element.

**g_dataset_set_data_full()**

```c
#define g_dataset_set_data_full(l, k, d, f)
```

Sets the data corresponding to the given string identifier, and the function to call when the data element is destroyed.

- **l**: the location identifying the dataset.
- **k**: the string to identify the data element.
- **d**: the data element.
- **f**: the function to call when the data element is removed. This function will be called with the data element and can be used to free any memory allocated for it.

**g_dataset_get_data()**

```c
#define g_dataset_get_data(l, k)
```

Gets the data element corresponding to a string.

- **l**: the location identifying the dataset.
- **k**: the string identifying the data element.

**Returns**: the data element corresponding to the string, or NULL if it is not found.

**g_dataset_remove_data()**

```c
#define g_dataset_remove_data(l, k)
```

Removes a data element corresponding to a string. Its destroy function is called if it has been set.

- **l**: the location identifying the dataset.
- **k**: the string identifying the data element.
5.19 Relations and Tuples

Name

Relations and Tuples – tables of data which can be indexed on any number of fields
Synopsis

#include <glib.h>

GRelation;
GRelation* g_relation_new (gint fields);
void g_relation_index (GRelation *relation, gint field,
GHashFunc hash_func, GEqualFunc key_equal_func);
void g_relation_insert (GRelation *relation, ...);
gboolean g_relation_exists (GRelation *relation, ...);
gint g_relation_count (GRelation *relation, gconstpointer key,
   gint field);
GTuples* g_relation_select (GRelation *relation, gconstpointer key,
   gint field);
gint g_relation_delete (GRelation *relation, gconstpointer key,
   gint field);
void g_relation_destroy (GRelation *relation);
void g_relation_print (GRelation *relation);

GTuples;
void g_tuples_destroy (GTuples *tuples);
gpointer g_tuples_index (GTuples *tuples, gint index_,
   gint field);

Description

A GRelation is a table of data which can be indexed on any number of fields, rather like simple database
tables. A GRelation contains a number of records, called tuples. Each record contains a number of fields.
Records are not ordered, so it is not possible to find the record at a particular index.

Note that GRelation tables are currently limited to 2 fields.
To create a GRelation, use g_relation_new().
To specify which fields should be indexed, use g_relation_index(). Note that this must be called
before any tuples are added to the GRelation.
To add records to a GRelation use g_relation_insert().
To determine if a given record appears in a GRelation, use g_relation_exists(). Note that fields are
compared directly, so pointers must point to the exact same position (i.e. different copies of the same
string will not match.)
To count the number of records which have a particular value in a given field, use g_relation_count().
To get all the records which have a particular value in a given field, use g_relation_select(). To access
fields of the resulting records, use g_tuples_index(). To free the resulting records use g_tuples_destroy().
To delete all records which have a particular value in a given field, use g_relation_delete().
To destroy the GRelation, use g_relation_destroy().
To help debug GRelation objects, use g_relation_print().

Details

GRelation

typedef struct _GRelation GRelation;
The GRelation struct is an opaque data structure to represent a Relation. It should only be accessed via the following functions.

**g_relation_new ()**

```c
GRelation* g_relation_new (gint fields);
```

Creates a new GRelation with the given number of fields. Note that currently the number of fields must be 2.

**fields**: the number of fields.

**Returns**: a new GRelation.

**g_relation_index ()**

```c
void g_relation_index (GRelation *relation,
                       gint field,
                       GHashFunc hash_func,
                       GEqualFunc key_equal_func);
```

Creates an index on the given field. Note that this must be called before any records are added to the GRelation.

**relation**: a GRelation.

**field**: the field to index, counting from 0.

**hash_func**: a function to produce a hash value from the field data.

**key_equal_func**: a function to compare two values of the given field.

**g_relation_insert ()**

```c
void g_relation_insert (GRelation *relation,
                        ...);
```

Inserts a record into a GRelation.

**relation**: a GRelation.

**...**: the fields of the record to add. These must match the number of fields in the GRelation, and of type gpointer or gconstpointer.

**g_relation_exists ()**

```c
gboolean g_relation_exists (GRelation *relation,
                           ...);
```

Returns TRUE if a record with the given values exists in a GRelation. Note that the values are compared directly, so that, for example, two copies of the same string will not match.

**relation**: a GRelation.

**...**: the fields of the record to compare. The number must match the number of fields in the GRelation.

**Returns**: %TRUE if a record matches.
g_relation_count()

```c
gboolean g_relation_count (GRelation *relation,
                          gconstpointer key,
                          gint field);
```

Returns the number of tuples in a GRelation that have the given value in the given field.

**relation**: a GRelation.

**key**: the value to compare with.

**field**: the field of each record to match.

**Returns**: the number of matches.

g_relation_select()

```c
Gtuples* g_relation_select (GRelation *relation,
                             gconstpointer key,
                             gint field);
```

Returns all of the tuples which have the given key in the given field. Use g_tuples_index() to access the returned records. The returned records should be freed with g_tuples_destroy().

**relation**: a GRelation.

**key**: the value to compare with.

**field**: the field of each record to match.

**Returns**: the records (tuples) that matched.

g_relation_delete()

```c
gboolean g_relation_delete (GRelation *relation,
                           gconstpointer key,
                           gint field);
```

Deletes any records from a GRelation that have the given key value in the given field.

**relation**: a GRelation.

**key**: the value to compare with.

**field**: the field of each record to match.

**Returns**: the number of records deleted.

g_relation_destroy()

```c
void g_relation_destroy (GRelation *relation);
```

Destroys the GRelation, freeing all memory allocated. However, it does not free memory allocated for the tuple data, so you should free that first if appropriate.

**relation**: a GRelation.

g_relation_print()

```c
void g_relation_print (GRelation *relation);
```

Outputs information about all records in a GRelation, as well as the indexes. It is for debugging.

**relation**: a GRelation.
GTuples

typedef struct {
    guint len;
} GTuples;

The GTuples struct is used to return records (or tuples) from the GRelation by g_relation_select(). It
only contains one public member - the number of records that matched. To access the matched records,
you must use g_tuples_index().

    guint len; the number of records that matched.

g_tuples_destroy()

void g_tuples_destroy (GTuples *tuples);

Frees the records which were returned by g_relation_select(). This should always be called after
g_relation_select() when you are finished with the records. The records are not removed from the GRe-
lation.

tuples: the tuple data to free.

g_tuples_index()

 gpointer g_tuples_index (GTuples *tuples,
                        gint index_,
                        gint field);

Gets a field from the records returned by g_relation_select(). It returns the given field of the record
at the given index. The returned value should not be changed.

tuples: the tuple data, returned by g_relation_select().

index_: the index of the record.

field: the field to return.

Returns: the field of the record.

5.20 Caches

Name

Caches – caches allow sharing of complex data structures to save resources

Synopsis

#include <glib.h>

 GCache;
 GCACHE* g_cache_new
     (GCACHENewFunc value_new_func,
      GCACHEDestroyFunc value_destroy_func,
      GCDUPFunc keyDupFunc,
      GCACHEDestroyFunc key_destroy_func,
      GHashFunc hash_key_func,
      GHashFunc hash_value_func,
      GEQUALFunc key_equal_func);

gpointer g_cache_insert
     (GCACHE* cache,
      gpointer key);

void g_cache_remove
     (GCACHE* cache,
void g_cache_destroy (GCache *cache);
void g_cache_key_foreach (GCache *cache, GHFunc func, gpointer user_data);
void g_cache_value_foreach (GCache *cache, GHFunc func, gpointer user_data);

void (*GCacheDestroyFunc) (gpointer value);
gpointer (*GCacheDupFunc) (gpointer value);
gpointer (*GCacheNewFunc) (gpointer key);

Description

A GCache allows sharing of complex data structures, in order to save system resources. GTK+ uses caches for GtkStyles and GdkGCs. These consume a lot of resources, so a GCache is used to see if a GtkStyle or GdkGC with the required properties already exists. If it does, then the existing object is used instead of creating a new one.

GCache uses keys and values. A GCache key describes the properties of a particular resource. A GCache value is the actual resource.

Details

GCache

typedef struct _GCache GCache;

The GCache struct is an opaque data structure containing information about a GCache. It should only be accessed via the following functions.

g_cache_new ()

GCache* g_cache_new (GCacheNewFunc value_new_func,
                    GCacheDestroyFunc value_destroy_func,
                    GCacheDupFunc key_dup_func,
                    GCacheDestroyFunc key_destroy_func,
                    GHashFunc hash_key_func,
                    GHashFunc hash_value_func,
                    GEqualFunc key_equal_func);

Creates a new GCache.

value_new_func: a function to create a new object given a key. This is called by g_cache_insert() if an object with the given key does not already exist.

value_destroy_func: a function to destroy an object. It is called by g_cache_remove() when the object is no longer needed (i.e. its reference count drops to 0).

key_dup_func: a function to copy a key. It is called by g_cache_insert() if the key does not already exist in the GCache.

key_destroy_func: a function to destroy a key. It is called by g_cache_remove() when the object is no longer needed (i.e. its reference count drops to 0).
**hash_key_func**: a function to create a hash value from a key.

**hash_value_func**: a function to create a hash value from a value.

**key_equal_func**: a function to compare two keys. It should return `TRUE` if the two keys are equivalent.

*Returns*: a new `GCache`.

### g_cache_insert ()

```c
gboolean g_cache_insert (GCache *cache, gpointer key);
```

Gets the value corresponding to the given key, creating it if necessary. It first checks if the value already exists in the `GCache`, by using the `key_equal_func` function passed to `g_cache_new()`. If it does already exist it is returned, and its reference count is increased by one. If the value does not currently exist, if is created by calling the `value_new_func`. The key is duplicated by calling `key_dup_func` and the duplicated key and value are inserted into the `GCache`.

*cache*: a `GCache`.

*key*: a key describing a `GCache` object.

*Returns*: a pointer to a `GCache` value.

### g_cache_remove ()

```c
void g_cache_remove (GCache *cache, gconstpointer value);
```

Decreases the reference count of the given value. If it drops to 0 then the value and its corresponding key are destroyed, using the `value_destroy_func` and `key_destroy_func` passed to `g_cache_new()`.

*cache*: a `GCache`.

*value*: the value to remove.

### g_cache_destroy ()

```c
void g_cache_destroy (GCache *cache);
```

Frees the memory allocated for the `GCache`. Note that it does not destroy the keys and values which were contained in the `GCache`.

*cache*: a `GCache`.

### g_cache_key_foreach ()

```c
void g_cache_key_foreach (GCache *cache, GHFunc func, gpointer user_data);
```

Calls the given function for each of the keys in the `GCache`.

**NOTE**

`func` is passed three parameters, the value and key of a cache entry and the `user_data`. The order of value and key is different from the order in which `g_hash_table_foreach()` passes key-value pairs to its callback function!
**cache**: a `GCache`.

**func**: the function to call with each `GCache` key.

**user_data**: user data to pass to the function.

### g_cache_value_foreach()

```c
void g_cache_value_foreach (GCache *cache, GFunc func, gpointer user_data);
```

**Calls the given function for each of the values in the GCache.**

**cache**: a `GCache`.

**func**: the function to call with each `GCache` value.

**user_data**: user data to pass to the function.

### GCacheDestroyFunc()

```c
void (*GCacheDestroyFunc)(gpointer value);
```

**Specifies the type of the** `value_destroy_func` **and** `key_destroy_func` **functions passed to** `g_cache_new()`. **The functions are passed a pointer to the GCache key or GCache value and should free any memory and other resources associated with it.**

**value**: the `GCache` value to destroy.

### GCacheDupFunc()

```c
gpointer (*GCacheDupFunc)(gpointer value);
```

**Specifies the type of the** `key_dup_func` **function passed to** `g_cache_new()`. **The function is passed a key (**not** a value as the prototype implies) and should return a duplicate of the key.**

**value**: the `GCache` key to destroy (**not** a `GCache` value as it seems).

**Returns**: a copy of the `GCache` key.

### GCacheNewFunc()

```c
gboolean (*GCacheNewFunc)(gpointer key);
```

**Specifies the type of the** `value_new_func` **function passed to** `g_cache_new()`. **It is passed a GCache key and should create the value corresponding to the key.**

**key**: a `GCache` key.

**Returns**: a new `GCache` value corresponding to the key.
5.21 Memory Allocators

Name

Memory Allocators – deprecated way to allocate chunks of memory for GList, GSList and GNode

Synopsis

```c
#include <glib.h>

GAllocator;
GAllocator* g_allocator_new (const gchar *name,
                                 guint n_preallocs);

void g_allocator_free (GAllocator *allocator);
```

Description

Prior to 2.10, GAllocator was used as an efficient way to allocate small pieces of memory for use with the GList, GSList and GNode data structures. Since 2.10, it has been completely replaced by the slice allocator and deprecated.

Details

GAllocator

```c
typedef struct _GAllocator GAllocator;
```

**WARNING**

GAllocator is deprecated and should not be used in newly-written code.

The GAllocator struct contains private data and should only be accessed using the following functions.

**g_allocator_new**

```c
GAllocator* g_allocator_new (const gchar *name,
                                guint n_preallocs);
```

**WARNING**

g_allocator_new has been deprecated since version 2.10 and should not be used in newly-written code. Use the slice allocator instead

Creates a new GAllocator.

name: the name of the GAllocator. This name is used to set the name of the GMemChunk used by the GAllocator, and is only used for debugging.

n_preallocs: the number of elements in each block of memory allocated. Larger blocks mean less calls to g_malloc(), but some memory may be wasted. (GLib uses 128 elements per block by default.) The value must be between 1 and 65535.
Returns: a new GAllocator.

g_allocator_free ()

void g_allocator_free (GAllocator *allocator);

WARNING

! g_allocator_free has been deprecated since version 2.10 and should not be used in newly-written code. Use the slice allocator instead

Frees all of the memory allocated by the GAllocator.

allocator: a GAllocator.
Chapter 6

GLib Tools

6.1 glib-gettextize

Name

glib-gettextize – gettext internationalization utility

Synopsis

glib-gettextize [option...] [directory]

Description

glib-gettextize helps to prepare a source package for being internationalized through gettext. It is a
variant of the gettextize that ships with gettext.

glib-gettextize differs from gettextize in that it doesn’t create an int1/ subdirectory and doesn’t
modify po/ChangeLog (note that newer versions of gettextize behave like this when called with the
--no-changelog option).

Options

--help print help and exit

--version print version information and exit

-c, --copy copy files instead of making symlinks

-f, --force force writing of new files even if old ones exist

See also

ggettextize(1)

6.2 gtester

Name

gtester – test running utility

Synopsis

gtester [option...] [testprogram]
Description

gtester is a utility to run unit tests that have been written using the GLib test framework.
When called with the -o option, gtester writes an XML report of the test results, which can be converted into HTML using the gtester-report utility.

Options

-h, --help  print help and exit
-v, --version print version information and exit
--g-fatal-warnings  make warnings fatal
-k, --keep-going  continue running after tests failed
-l  list paths of available test cases
-m=MODE  run test cases in MODE, which can be perf, slow, thorough or quick. The default mode is quick.
-p=TESTPATH  only run test cases matching TESTPATH
--seed=SEEDSTRING  run all test cases with random number seed SEEDSTRING
-o=LOGFILE  write the test log to LOGFILE
-q, --quiet  suppress per test binary output
--verbose  report success per testcase

See also

gtester-report(1)

6.3  gtester-report

Name

gtester-report – test report formatting utility

Synopsis

gtester-report [option...] [gtester-log]

Description

gtester-report is a script which converts the XML output generated by gtester into HTML.

Options

-h, --help  print help and exit
-v, --version print version information and exit

See also

gtester(1)
Index

A
ABS, 46
g_access, 342
G_ALLOC_AND_FREE, 461
G_ALLOC_ONLY, 461
galloca, 151
GAlloc, 588
GAllocator, 588
g_allocator_free, 589
g_allocator_new, 588
GArray, 536
g_array_append_val, 538
g_array_append_vals, 538
g_array_free, 541
g_array_get_element_size, 538
g_array_index, 541
g_array_insert_val, 539
g_array_insert_vals, 539
g_array_new, 537
g_array_prepend_val, 538
g_array_prepend_vals, 539
g_array_ref, 537
g_array_remove_index, 540
g_array_remove_index_fast, 540
g_array_remove_range, 540
g_array_set_size, 541
g_array_sort, 541
g_array_sort_with_data, 541
g_array_unref, 537
g_ascii_digit_value, 202
g_ascii_dtostr, 208
G_ASCII_DTOSTR_BUF_SIZE, 207
g_ascii_formatd, 208
g_ascii_isalnum, 199
g_ascii_isalpha, 200
g_ascii_iscntrl, 200
g_asciiisdigit, 200
g_ascii_isgraph, 200
g_ascii_islower, 201
g_ascii_isprint, 201
g_ascii_ispunct, 201
g_ascii_isspace, 201
g_ascii_isupper, 201
g_ascii_xdigit_value, 202
G_ASCII_DTOSTR_BUF_SIZE, 207
g_ascii_formatd, 208
G_ASCII_DTOSTR_BUF_SIZE, 207
g_ascii_formatd, 208
g_ascii_isalnum, 199
g_ascii_isalpha, 200
g_ascii_iscntrl, 200
g_asciiisdigit, 200
g_ascii_isgraph, 200
g_ascii_islower, 201
g_ascii_isprint, 201
g_ascii_ispunct, 201
g_ascii_isspace, 201
g_ascii_isupper, 201
g_ascii_xdigit_value, 202
g_ascii_dtostr, 208
G_ASCII_DTOSTR_BUF_SIZE, 207
g_ascii_formatd, 208
G_ASCII_DTOSTR_BUF_SIZE, 207
g_ascii_formatd, 208
g_ascii_isalnum, 199
g_ascii_isalpha, 200
g_ascii_iscntrl, 200
g_asciiisdigit, 200
g_ascii_isgraph, 200
g_ascii_islower, 201
g_ascii_isprint, 201
g_ascii_ispunct, 201
g_ascii_isspace, 201
g_ascii_isupper, 201
g_ascii_xdigit_value, 202
g_ascii_dtostr, 208
G_ASCII_DTOSTR_BUF_SIZE, 207
g_ascii_formatd, 208
G_ASCII_DTOSTR_BUF_SIZE, 207
g_ascii_formatd, 208
g_ascii_isalnum, 199
g_ascii_isalpha, 200
g_ascii_iscntrl, 200
g_asciiisdigit, 200
g_ascii_isgraph, 200
g_ascii_islower, 201
g_ascii_isprint, 201
g_ascii_ispunct, 201
g_ascii_isspace, 201
g_ascii_isupper, 201
g_ascii_xdigit_value, 202
g_ascii_dtostr, 208
G_ASCII_DTOSTR_BUF_SIZE, 207
g_ascii_formatd, 208
G_ASCII_DTOSTR_BUF_SIZE, 207
g_ascii_formatd, 208
g_ascii_isalnum, 199
g_ascii_isalpha, 200
g_ascii_iscntrl, 200
g_asciiisdigit, 200
g_ascii_isgraph, 200
g_ascii_islower, 201
g_ascii_isprint, 201
g_ascii_ispunct, 201
g_ascii_isspace, 201
g_ascii_isupper, 201
g_ascii_xdigit_value, 202
g_ascii_dtostr, 208
G_ASCII_DTOSTR_BUF_SIZE, 207
g_ascii_formatd, 208
G_ASCII_DTOSTR_BUF_SIZE, 207
g_ascii_formatd, 208
g_ascii_isalnum, 199
g_ascii_isalpha, 200
g_ascii_iscntrl, 200
g_asciiisdigit, 200
g_ascii_isgraph, 200
g_ascii_islower, 201
g_ascii_isprint, 201
g_ascii_ispunct, 201
g_ascii_isspace, 201
g_ascii_isupper, 201
g_ascii_xdigit_value, 202

B
g_base64_decode, 251
g_base64_decode_inplace, 251
g_base64_decode_step, 250
g_base64_encode, 250
g_base64_encode_close, 250
g_base64_encode_step, 249
g_basename, 298
G_BEGIN_DECLS, 65
G_BIG_ENDIAN, 52
g_bit_nth_lsf, 301
g_bit_nth_msf, 302
g_bit_storage, 302
g_base64_decode, 251
g_base64_decode_inplace, 251
g_base64_decode_step, 250
g_base64_encode, 250
g_base64_encode_close, 250
g_base64_encode_step, 249
g_basename, 298
G_BEGIN_DECLS, 65
G_BIG_ENDIAN, 52
g_bit_nth_lsf, 301
g_bit_nth_msf, 302
g_bit_storage, 302
g_base64_decode, 251
g_base64_decode_inplace, 251
g_base64_decode_step, 250
g_base64_encode, 250
g_base64_encode_close, 250
g_base64_encode_step, 249
g_basename, 298
G_BEGIN_DECLS, 65
G_BIG_ENDIAN, 52

g_bookmark_file_add_group, 434
G_BOOKMARK_FILE_ERROR, 422
g_bookmark_file_free, 423
g_bookmark_file_get_added, 428
g_bookmark_file_get_app_info, 430
g_bookmark_file_get_applications, 429
g_bookmark_file_get_description, 427
g_bookmark_file_get_groups, 429
g_bookmark_file_get_icon, 428
g_bookmark_file_get_is_private, 427
g_bookmark_file_get_mime_type, 427
g_bookmark_file_get_modified, 428
g_bookmark_file_get_size, 426
g_bookmark_file_get_title, 426
g_bookmark_file_get_visited, 426
GBookmarkFile, 422
GBookmarkFileError, 423
g_byte_array_unref, 550
G_BYTE_ORDER, 51
GByteArray, 549
C
C_, 256
GCache, 585
g_cache_destroy, 586
g_cache_insert, 586
g_cache_key_foreach, 586
g_cache_new, 585
g_cache_remove, 586
g_cache_value_foreach, 587
GCacheDestroyFunc, 587
GCacheDupFunc, 587
GCacheNewFunc, 587
gchar, 36
g_chdir, 343
GChecksum, 252
g_checksum_copy, 253
g_checksum_free, 253
g_checksum_get_digest, 254
g_checksum_get_string, 254
g_checksum_new, 253
g_checksum_reset, 253
g_checksum_type_get_length, 252
g_checksum_update, 254
GChecksumType, 252
g_child_watch_add, 96
g_child_watch_add_full, 96
g_child_watch_source_new, 95
GChildWatchFunc, 95
g_chmod, 342
g_chunk_free, 465
g_chunk_new, 464
g_chunk_new0, 464
CLAMP, 47
g_clear_error, 178
GCompareDataFunc, 474
GCompareFunc, 473
GCompletion, 317
g_completion_add_items, 318
g_completion_clear_items, 319
g_completion_complete, 319
g_completion_complete_utf8, 319
g_completion_free, 320
g_completion_new, 318
g_completion_remove_items, 318
g_completion_set_compare, 319
GCompletionFunc, 318
GCompletionStrncmpFunc, 320
g_compute_checksum_for_data, 254
g_compute_checksum_for_string, 255
GCond, 124
g_cond_broadcast, 125
g_cond_free, 126
g_cond_new, 124
g_cond_signal, 125
g_cond_timed_wait, 125

524
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>g_cond_wait</td>
<td>125</td>
</tr>
<tr>
<td>G_COND国内外 RETURN</td>
<td>48</td>
</tr>
<tr>
<td>gconstpointer</td>
<td>36</td>
</tr>
<tr>
<td>g_convert</td>
<td>215</td>
</tr>
<tr>
<td>G_CONVERT内外ERROR</td>
<td>217</td>
</tr>
<tr>
<td>g_convert_with_fallback</td>
<td>216</td>
</tr>
<tr>
<td>g_convert_with_iclconv</td>
<td>217</td>
</tr>
<tr>
<td>GConvertError</td>
<td>222</td>
</tr>
<tr>
<td>GCopyFunc</td>
<td>561</td>
</tr>
<tr>
<td>g_creat</td>
<td>342</td>
</tr>
<tr>
<td>g_critical</td>
<td>185</td>
</tr>
<tr>
<td>G_CSET_A_2_Z</td>
<td>315</td>
</tr>
<tr>
<td>G_CSET_a_2 z</td>
<td>315</td>
</tr>
<tr>
<td>G_CSET_DIGITS</td>
<td>315</td>
</tr>
<tr>
<td>G_CSET_LATINC</td>
<td>315</td>
</tr>
<tr>
<td>G_CSET_LATINS</td>
<td>315</td>
</tr>
</tbody>
</table>

**D**

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>GData</td>
<td>573</td>
</tr>
<tr>
<td>GDataForeachFunc</td>
<td>580</td>
</tr>
<tr>
<td>g_datalist_clear</td>
<td>575</td>
</tr>
<tr>
<td>G_DATALIST_FLAGS_MASK</td>
<td>576</td>
</tr>
<tr>
<td>g_datalist_foreach</td>
<td>575</td>
</tr>
<tr>
<td>g_datalist_get_data</td>
<td>575</td>
</tr>
<tr>
<td>g_datalist_get_flags</td>
<td>576</td>
</tr>
<tr>
<td>g_datalist_id_get_data</td>
<td>574</td>
</tr>
<tr>
<td>g_datalist_id_remove_data</td>
<td>574</td>
</tr>
<tr>
<td>g_datalist_id_remove_no_notify</td>
<td>574</td>
</tr>
<tr>
<td>g_datalist_id_set_data</td>
<td>573</td>
</tr>
<tr>
<td>g_datalist_id_set_data_full</td>
<td>573</td>
</tr>
<tr>
<td>g_datalist_init</td>
<td>573</td>
</tr>
<tr>
<td>g_datalist_remove_data</td>
<td>575</td>
</tr>
<tr>
<td>g_datalist_remove_no_notify</td>
<td>575</td>
</tr>
<tr>
<td>g_datalist_set_data</td>
<td>574</td>
</tr>
<tr>
<td>g_datalist_set_data_full</td>
<td>574</td>
</tr>
<tr>
<td>g_datalist_set_flags</td>
<td>576</td>
</tr>
<tr>
<td>g_datalist_unset_flags</td>
<td>576</td>
</tr>
<tr>
<td>g_dataset_destroy</td>
<td>580</td>
</tr>
<tr>
<td>g_dataset_foreach</td>
<td>580</td>
</tr>
<tr>
<td>g_dataset_get_data</td>
<td>579</td>
</tr>
<tr>
<td>g_dataset_id_get_data</td>
<td>579</td>
</tr>
<tr>
<td>g_dataset_id_remove_data</td>
<td>578</td>
</tr>
<tr>
<td>g_dataset_id_remove_no_notify</td>
<td>579</td>
</tr>
<tr>
<td>g_dataset_id_set_data</td>
<td>577</td>
</tr>
<tr>
<td>g_dataset_id_set_data_full</td>
<td>577</td>
</tr>
<tr>
<td>g_dataset_remove_data</td>
<td>578</td>
</tr>
<tr>
<td>g_dataset_remove_no_notify</td>
<td>579</td>
</tr>
<tr>
<td>g_dataset_remove_data</td>
<td>579</td>
</tr>
<tr>
<td>g_dataset_set_data</td>
<td>579</td>
</tr>
<tr>
<td>g_dataset_set_data_full</td>
<td>579</td>
</tr>
</tbody>
</table>

**GDate**

<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>g_date_add_days</td>
<td>270</td>
</tr>
<tr>
<td>g_date_add_months</td>
<td>270</td>
</tr>
<tr>
<td>g_date_add_years</td>
<td>270</td>
</tr>
<tr>
<td>G_DATE_BAD_DAY</td>
<td>266</td>
</tr>
<tr>
<td>G_DATE_BAD_JULIAN</td>
<td>266</td>
</tr>
<tr>
<td>G_DATE_BAD_YEAR</td>
<td>266</td>
</tr>
<tr>
<td>g_date_clamp</td>
<td>271</td>
</tr>
<tr>
<td>g_date_clear</td>
<td>267</td>
</tr>
<tr>
<td>g_date_compare</td>
<td>271</td>
</tr>
<tr>
<td>g_date_days_between</td>
<td>271</td>
</tr>
<tr>
<td>g_date_free</td>
<td>267</td>
</tr>
<tr>
<td>g_date_get_day</td>
<td>272</td>
</tr>
<tr>
<td>g_date_get_day_of_year</td>
<td>272</td>
</tr>
<tr>
<td>g_date_get_days_in_month</td>
<td>273</td>
</tr>
<tr>
<td>g_date_get_iso8601_week_of_year</td>
<td>274</td>
</tr>
<tr>
<td>g_date_get_julian</td>
<td>272</td>
</tr>
<tr>
<td>g_date_get_monday_week_of_year</td>
<td>273</td>
</tr>
<tr>
<td>g_date_get_monday_weeks_in_year</td>
<td>274</td>
</tr>
<tr>
<td>g_date_get_month</td>
<td>272</td>
</tr>
<tr>
<td>g_date_get_sunday_week_of_year</td>
<td>274</td>
</tr>
<tr>
<td>g_date_get_sunday_weeks_in_year</td>
<td>274</td>
</tr>
<tr>
<td>g_date_get_weekday</td>
<td>272</td>
</tr>
<tr>
<td>g_date_get_year</td>
<td>272</td>
</tr>
<tr>
<td>g_date_is_first_of_month</td>
<td>273</td>
</tr>
<tr>
<td>g_date_is_last_of_month</td>
<td>273</td>
</tr>
<tr>
<td>g_date_is_leap_year</td>
<td>273</td>
</tr>
<tr>
<td>g_date_new</td>
<td>267</td>
</tr>
<tr>
<td>g_date_new_dmy</td>
<td>267</td>
</tr>
<tr>
<td>g_date_new_julian</td>
<td>267</td>
</tr>
<tr>
<td>g_date_order</td>
<td>271</td>
</tr>
<tr>
<td>g_date_set_day</td>
<td>268</td>
</tr>
<tr>
<td>g_date_set_dmy</td>
<td>268</td>
</tr>
<tr>
<td>g_date_set_julian</td>
<td>268</td>
</tr>
<tr>
<td>g_date_set_month</td>
<td>268</td>
</tr>
<tr>
<td>g_date_set_parse</td>
<td>269</td>
</tr>
<tr>
<td>g_date_set_time</td>
<td>269</td>
</tr>
<tr>
<td>g_date_set_time_t</td>
<td>269</td>
</tr>
<tr>
<td>g_date_set_time_val</td>
<td>269</td>
</tr>
<tr>
<td>g_date_strftime</td>
<td>274</td>
</tr>
<tr>
<td>g_date_subtract_days</td>
<td>270</td>
</tr>
<tr>
<td>g_date_subtract_months</td>
<td>270</td>
</tr>
<tr>
<td>g_date_subtract_years</td>
<td>271</td>
</tr>
<tr>
<td>g_date_to_struct_tm</td>
<td>275</td>
</tr>
<tr>
<td>g_date_valid</td>
<td>275</td>
</tr>
<tr>
<td>g_date_valid_d_day</td>
<td>275</td>
</tr>
<tr>
<td>g_date_valid_dmy</td>
<td>276</td>
</tr>
<tr>
<td>g_date_valid_julian</td>
<td>276</td>
</tr>
<tr>
<td>g_date_valid_month</td>
<td>275</td>
</tr>
<tr>
<td>g_date_valid_weekday</td>
<td>276</td>
</tr>
<tr>
<td>g_date_valid_year</td>
<td>275</td>
</tr>
<tr>
<td>GDateDay</td>
<td>265</td>
</tr>
<tr>
<td>GDateDMY</td>
<td>265</td>
</tr>
<tr>
<td>GDateMonth</td>
<td>265</td>
</tr>
<tr>
<td>GDateWeekday</td>
<td>266</td>
</tr>
<tr>
<td>GDateYear</td>
<td>266</td>
</tr>
<tr>
<td>g_debug</td>
<td>186</td>
</tr>
<tr>
<td>GDebugKey</td>
<td>303</td>
</tr>
<tr>
<td>GDestroyNotify</td>
<td>578</td>
</tr>
<tr>
<td>g_dgettext</td>
<td>258</td>
</tr>
<tr>
<td>GDir</td>
<td>337</td>
</tr>
<tr>
<td>g_dir_close</td>
<td>337</td>
</tr>
<tr>
<td>g_dir_open</td>
<td>337</td>
</tr>
<tr>
<td>g_dir_read_name</td>
<td>337</td>
</tr>
<tr>
<td>g_dir_rewind</td>
<td>337</td>
</tr>
<tr>
<td>G_DIR_SEPARATOR</td>
<td>45</td>
</tr>
<tr>
<td>G_DIR_SEPARATOR_S</td>
<td>45</td>
</tr>
<tr>
<td>g_direct_equal</td>
<td>521</td>
</tr>
</tbody>
</table>
INDEX

INDEX

g_direct_hash, 521

g_dirname, 299

g_dngettext, 258

gdouble, 39

g_double_equal, 522

g_double_hash, 522

GDoubleIEEE754, 62

g_dpgettext, 259

g_dpgettext2, 259

E

G_E, 62

G_END_DECLS, 65

GEqualFunc, 513

GError, 175

g_error, 186

g_error_copy, 176

g_error_free, 176

g_error_matches, 177

g_error_new, 176

g_error_new_literal, 176

GErrorType, 316

F

FALSE, 46

G_FILE_ERROR, 333

g_file_error_from_errno, 334

g_file_get_contents, 334

g_file_open_tmp, 336

g_file_read_link, 336

g_file_set_contents, 334

g_file_test, 335

GFileError, 331

g_filename_display_basename, 221

g_filename_display_name, 221

g_filename_from_uri, 220

g_filename_from_utf8, 219

g_filename_to_uri, 220

g_filename_to_utf8, 219

GFileTest, 333

g_find_program_in_path, 301

gfloat, 39

GFloatIEEE754, 61

g_fopen, 341

g_format_size_for_display, 301

g_fprintf, 197

g_free, 151

GFreeFunc, 303

g_freopen, 341

GFunc, 475

G

g_module_check_init, 147

g_module_unload, 147

g_trap_free_size, 8

g_trap_malloc_size, 8

g_trap_realloc_size, 8

g_get_application_name, 293

g_get_charset, 222

g_get_current_dir, 298

g_get_current_time, 263

g_get_filename_charsets, 220

g_get_home_dir, 298

g_get_host_name, 297

g_get_language_names, 260

g_get_prgname, 293

g_get_real_name, 295

g_get_system_config_dirs, 297

g_get_system_data_dirs, 297

g_get_tmp_dir, 298

g_get_user_cache_dir, 295

g_get_user_config_dir, 296

g_get_user_data_dir, 295

g_get_user_name, 295

g_get_user_special_dir, 296

g_getenv, 294

G_GINT16_FORMAT, 71

G_GINT16_MODIFIER, 71

G_GINT32_FORMAT, 71

G_GINT32_MODIFIER, 71

G_GINT64_CONSTANT, 39

G_GINT64_FORMAT, 72

G_GINT64_MODIFIER, 72

G_GNUC_ALLOC_SIZE, 67

G_GNUC_ALLOC_SIZE2, 67

G_GNUC_CONST, 66

G_GNUC_DEPRECATED, 67

G_GNUC_EXTENSION, 66

G_GNUC_FUNCTION, 69

G_GNUC_INTERNAL, 69

G_GNUC_MALLOC, 67

G_GNUC_MAY_ALIAS, 70

G_GNUC_NO_INSTRUMENT, 69

G_GNUC_NORETURN, 67

G_GNUC_NULL_TERMINATED, 68

G_GNUC_PRETTY_FUNCTION, 69

G_GNUC_PRINTF, 68

G_GNUC_PURE, 66

G_GNUC_SCANF, 68

G_GNUC_UNUSED, 67

G_GNUC_WARN_UNUSED_RESULT, 68

G_GOFFSET_CONSTANT, 40

G_GOFFSET_FORMAT, 73

G_GOFFSET_MODIFIER, 73

G_GSIZE_FORMAT, 73

G_GSIZE_MODIFIER, 72

G_GSIZE_FORMAT, 73

G_GUINT16_FORMAT, 71

G_GUINT32_FORMAT, 71

G_GUINT64_CONSTANT, 39

G_GUINT64_FORMAT, 72

G_GUINT64_MODIFIER, 72

G_GNUC_ALLOC_SIZE, 67

G_GNUC_ALLOC_SIZE2, 67

G_GNUC_COMPATIBILITY, 66

G_GNUC_DEPRECATED, 67

G_GNUC_EXTENSION, 66

G_GNUC_FUNCTION, 69

G_GNUC_INTERNAL, 69

G_GNUC_MALLOC, 67

G_GNUC_MAY_ALIAS, 70

G_GNUC_NO_INSTRUMENT, 69

G_GNUC_NORETURN, 67

G_GNUC_NULL_TERMINATED, 68

G_GNUC_PRETTY_FUNCTION, 69

G_GNUC_PRINTF, 68

G_GNUC_PURE, 66

G_GNUC_SCANF, 68

G_GNUC_UNUSED, 67

G_GNUC_WARN_UNUSED_RESULT, 68

G_GOFFSET_CONSTANT, 40

G_GOFFSET_FORMAT, 73

G_GOFFSET_MODIFIER, 73

G_GSIZE_FORMAT, 73

G_GSIZE_MODIFIER, 72

G_GSIZE_FORMAT, 73

G_GUINT16_FORMAT, 71

G_GUINT32_FORMAT, 71

G_GUINT64_CONSTANT, 39

G_GUINT64_FORMAT, 72

H

g_hash_table_destroy, 518

h_hash_table_find, 515

h_hash_table_foreach, 515

h_hash_table_foreach_remove, 516
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>g_hash_table_foreach_steal</td>
<td>517</td>
</tr>
<tr>
<td>g_hash_table_freeze</td>
<td>518</td>
</tr>
<tr>
<td>g_hash_table_get_keys</td>
<td>517</td>
</tr>
<tr>
<td>g_hash_table_get_values</td>
<td>517</td>
</tr>
<tr>
<td>g_hash_table_insert</td>
<td>514</td>
</tr>
<tr>
<td>g_hash_table_iter_get_hash_table</td>
<td>520</td>
</tr>
<tr>
<td>g_hash_table_iter_init</td>
<td>519</td>
</tr>
<tr>
<td>g_hash_table_iter_next</td>
<td>520</td>
</tr>
<tr>
<td>g_hash_table_iter_remove</td>
<td>520</td>
</tr>
<tr>
<td>g_hash_table_iter_steal</td>
<td>520</td>
</tr>
<tr>
<td>g_hash_table_lookup</td>
<td>514</td>
</tr>
<tr>
<td>g_hash_table_lookup_extended</td>
<td>515</td>
</tr>
<tr>
<td>g_hash_table_new</td>
<td>512</td>
</tr>
<tr>
<td>g_hash_table_new_full</td>
<td>513</td>
</tr>
<tr>
<td>g_hash_table_ref</td>
<td>519</td>
</tr>
<tr>
<td>g_hash_table_remove</td>
<td>516</td>
</tr>
<tr>
<td>g_hash_table_remove_all</td>
<td>517</td>
</tr>
<tr>
<td>g_hash_table_replace</td>
<td>514</td>
</tr>
<tr>
<td>g_hash_table_steal</td>
<td>516</td>
</tr>
<tr>
<td>g_hash_table_steal_all</td>
<td>517</td>
</tr>
<tr>
<td>g_hash_table_thaw</td>
<td>518</td>
</tr>
<tr>
<td>g_hash_table_unref</td>
<td>519</td>
</tr>
<tr>
<td>GHashFunc</td>
<td>513</td>
</tr>
<tr>
<td>GHashTable</td>
<td>512</td>
</tr>
<tr>
<td>GHashTableIter</td>
<td>519</td>
</tr>
<tr>
<td>G_HAVE_GINT64</td>
<td>38</td>
</tr>
<tr>
<td>G_HAVE_GNUC_VISIBILITY</td>
<td>69</td>
</tr>
<tr>
<td>GHashFunc</td>
<td>516</td>
</tr>
<tr>
<td>G_HOOK</td>
<td>290</td>
</tr>
<tr>
<td>GHook</td>
<td>283</td>
</tr>
<tr>
<td>G_HOOK_ACTIVE</td>
<td>290</td>
</tr>
<tr>
<td>g_hook_alloc</td>
<td>286</td>
</tr>
<tr>
<td>g_hook_append</td>
<td>286</td>
</tr>
<tr>
<td>g_hook_compare_ids</td>
<td>287</td>
</tr>
<tr>
<td>g_hook_destroy</td>
<td>291</td>
</tr>
<tr>
<td>g_hook_destroy_link</td>
<td>291</td>
</tr>
<tr>
<td>g_hook_find</td>
<td>287</td>
</tr>
<tr>
<td>g_hook_find_data</td>
<td>288</td>
</tr>
<tr>
<td>g_hook_find_func</td>
<td>288</td>
</tr>
<tr>
<td>g_hook_find_func_data</td>
<td>288</td>
</tr>
<tr>
<td>g_hook_first_valid</td>
<td>289</td>
</tr>
<tr>
<td>G_HOOK_FLAG_USER_SHIFT</td>
<td>290</td>
</tr>
<tr>
<td>G_HOOK_FLAGS</td>
<td>289</td>
</tr>
<tr>
<td>g_hook_get</td>
<td>287</td>
</tr>
<tr>
<td>G_HOOK_IN_CALL</td>
<td>290</td>
</tr>
<tr>
<td>g_hook_insert_before</td>
<td>286</td>
</tr>
<tr>
<td>g_hook_insert_sorted</td>
<td>286</td>
</tr>
<tr>
<td>G_HOOK_IS_UNLINKED</td>
<td>290</td>
</tr>
<tr>
<td>G_HOOK_IS_VALID</td>
<td>290</td>
</tr>
<tr>
<td>g_hook_list_clear</td>
<td>286</td>
</tr>
<tr>
<td>g_hook_list_init</td>
<td>284</td>
</tr>
<tr>
<td>g_hook_list_invoke</td>
<td>284</td>
</tr>
<tr>
<td>g_hook_list_invoke_check</td>
<td>284</td>
</tr>
<tr>
<td>g_hook_list_marshal</td>
<td>285</td>
</tr>
<tr>
<td>g_hook_list_marshal_check</td>
<td>285</td>
</tr>
<tr>
<td>g_hook_next_valid</td>
<td>289</td>
</tr>
<tr>
<td>g_hook_prepend</td>
<td>286</td>
</tr>
<tr>
<td>g_hook_ref</td>
<td>290</td>
</tr>
<tr>
<td>g_hook_unref</td>
<td>291</td>
</tr>
<tr>
<td>GHookCheckFunc</td>
<td>284</td>
</tr>
<tr>
<td>GHookCheckMarshaller</td>
<td>285</td>
</tr>
<tr>
<td>GHookCompareFunc</td>
<td>287</td>
</tr>
<tr>
<td>GHookFinalizeFunc</td>
<td>283</td>
</tr>
<tr>
<td>GHookFindFunc</td>
<td>288</td>
</tr>
<tr>
<td>GHookFlagMask</td>
<td>289</td>
</tr>
<tr>
<td>GHookFunc</td>
<td>284</td>
</tr>
<tr>
<td>GHookList</td>
<td>283</td>
</tr>
<tr>
<td>GHookMarshaller</td>
<td>285</td>
</tr>
<tr>
<td>GHRFunc</td>
<td>518</td>
</tr>
<tr>
<td>g_hhtonl</td>
<td>52</td>
</tr>
<tr>
<td>g_htons</td>
<td>52</td>
</tr>
</tbody>
</table>

I
GIConv | 217 |
g_iconv | 218 |
g_iconv_close | 218 |
g_iconv_open | 217 |
g_idle_add | 94 |
g_idle_add_full | 94 |
g_idle_remove_by_data | 95 |
g_idle_source_new | 94 |
G_IEEE754_DOUBLE_BIAS | 61 |
G_IEEE754_FLOAT_BIAS | 61 |
G_INLINE_FUNC | 64 |
gint | 37 |
gint16 | 38 |
GINT16_FROM_BE | 55 |
GINT16_FROM_LE | 55 |
GINT16_TO_BE | 56 |
GINT16_TO_LE | 56 |
gint32 | 38 |
GINT32_FROM_BE | 57 |
GINT32_FROM_LE | 57 |
GINT32_TO_BE | 58 |
GINT32_TO_LE | 58 |
gint64 | 38 |
GINT64_EQUAL | 521 |
GINT64_FROM_BE | 58 |
GINT64_FROM_LE | 58 |
gint64_hash | 522 |
GINT64_TO_BE | 59 |
GINT64_TO_LE | 59 |
gint8 | 37 |
G_INT64_EQUAL | 521 |
G_INT_FROM_BE | 53 |
G_INT_FROM_LE | 53 |
g_int_hash | 521 |
G_INT_TO_BE | 53 |
G_INT_TO_LE | 53 |
G_INT_TO_POINTER | 49 |
g_int_equal | 521 |
G_INT_FROM_BE | 53 |
G_INT_FROM_LE | 53 |
g_int_equal | 521 |
G_INT_TO_BE | 53 |
G_INT_TO_LE | 53 |
g_io_add_watch | 164 |
g_io_add_watch_full | 164 |
g_io_channel_close | 171 |
G_IO_CHANNEL_ERROR | 163 |
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>GMutex</td>
<td>114</td>
</tr>
<tr>
<td>g_mutex_free</td>
<td>116</td>
</tr>
<tr>
<td>g_mutex_lock</td>
<td>115</td>
</tr>
<tr>
<td>g_mutex_new</td>
<td>115</td>
</tr>
<tr>
<td>g_mutex_trylock</td>
<td>115</td>
</tr>
<tr>
<td>g_mutex_unlock</td>
<td>116</td>
</tr>
<tr>
<td>N</td>
<td>257</td>
</tr>
<tr>
<td>G_N_ELEMENTS</td>
<td>65</td>
</tr>
<tr>
<td>NC</td>
<td>257</td>
</tr>
<tr>
<td>g_new</td>
<td>149</td>
</tr>
<tr>
<td>g_new0</td>
<td>149</td>
</tr>
<tr>
<td>g_newa</td>
<td>152</td>
</tr>
<tr>
<td>GNode</td>
<td>560</td>
</tr>
<tr>
<td>g_node_append</td>
<td>562</td>
</tr>
<tr>
<td>g_node_append_data</td>
<td>563</td>
</tr>
<tr>
<td>g_node_child_index</td>
<td>566</td>
</tr>
<tr>
<td>g_node_child_position</td>
<td>566</td>
</tr>
<tr>
<td>g_node_children_foreach</td>
<td>565</td>
</tr>
<tr>
<td>g_node_copy</td>
<td>561</td>
</tr>
<tr>
<td>g_node_copy_deep</td>
<td>561</td>
</tr>
<tr>
<td>g_node_depth</td>
<td>568</td>
</tr>
<tr>
<td>g_node_destroy</td>
<td>569</td>
</tr>
<tr>
<td>g_node_find</td>
<td>566</td>
</tr>
<tr>
<td>g_node_find_child</td>
<td>566</td>
</tr>
<tr>
<td>g_node_first_child</td>
<td>567</td>
</tr>
<tr>
<td>g_node_first_sibling</td>
<td>567</td>
</tr>
<tr>
<td>g_node_get_root</td>
<td>565</td>
</tr>
<tr>
<td>g_node_insert</td>
<td>562</td>
</tr>
<tr>
<td>g_node_insert_after</td>
<td>562</td>
</tr>
<tr>
<td>g_node_insert_before</td>
<td>562</td>
</tr>
<tr>
<td>g_node_insert_data</td>
<td>563</td>
</tr>
<tr>
<td>g_node_insert_data_before</td>
<td>563</td>
</tr>
<tr>
<td>g_node_is_ancestor</td>
<td>569</td>
</tr>
<tr>
<td>G_NODE_IS_LEAF</td>
<td>568</td>
</tr>
<tr>
<td>G_NODE_IS_ROOT</td>
<td>568</td>
</tr>
<tr>
<td>g_node_last_child</td>
<td>567</td>
</tr>
<tr>
<td>g_node_last_sibling</td>
<td>568</td>
</tr>
<tr>
<td>g_node_max_height</td>
<td>569</td>
</tr>
<tr>
<td>g_node_n_children</td>
<td>568</td>
</tr>
<tr>
<td>g_node_n_nodes</td>
<td>568</td>
</tr>
<tr>
<td>g_node_new</td>
<td>561</td>
</tr>
<tr>
<td>g_node_next_sibling</td>
<td>567</td>
</tr>
<tr>
<td>g_node_nth_child</td>
<td>567</td>
</tr>
<tr>
<td>g_node_pop_allocator</td>
<td>570</td>
</tr>
<tr>
<td>g_node_prepend</td>
<td>563</td>
</tr>
<tr>
<td>g_node_prepend_data</td>
<td>563</td>
</tr>
<tr>
<td>g_node_prev_sibling</td>
<td>567</td>
</tr>
<tr>
<td>g_node_push_allocator</td>
<td>569</td>
</tr>
<tr>
<td>g_node_reverse_children</td>
<td>564</td>
</tr>
<tr>
<td>g_node_traverse</td>
<td>564</td>
</tr>
<tr>
<td>g_node_unlink</td>
<td>569</td>
</tr>
<tr>
<td>GNodeForeachFunc</td>
<td>565</td>
</tr>
<tr>
<td>GNodeTraverseFunc</td>
<td>565</td>
</tr>
<tr>
<td>GNormalizeMode</td>
<td>244</td>
</tr>
<tr>
<td>g_ntohl</td>
<td>52</td>
</tr>
<tr>
<td>g_ntohs</td>
<td>52</td>
</tr>
<tr>
<td>NULL</td>
<td>46</td>
</tr>
<tr>
<td>g_nullify_pointer</td>
<td>304</td>
</tr>
</tbody>
</table>

| O                            |      |
| goffset                      | 39   |
| g_on_error_query             | 181  |
| g_on_error_stack_trace       | 182  |
| GOnce                        | 129  |
| g_once                       | 130  |
| G_ONCE_INIT                  | 130  |
| g_once_init_enter            | 130  |
| g_once_init_leave            | 131  |
| GOnceStatus                  | 129  |
| g_open                       | 339  |
| g_option_context_add_group   | 358  |
| g_option_context_add_main_entries | 358 |
| g_option_context_free        | 353  |
| g_option_context_get_description | 352 |
| g_option_context_get_help    | 355  |
| g_option_context_get_help_enabled | 354 |
| g_option_context_get_ignore_unknown_options | 355 |
| g_option_context_get_main_group | 359 |
| g_option_context_get_summary | 352  |
| g_option_context_new         | 351  |
| g_option_context_parse       | 354  |
| g_option_context_set_description | 352 |
| g_option_context_set_help_enabled | 354 |
| g_option_context_set_ignore_unknown_options | 355 |
| g_option_context_set_main_group | 359 |
| g_option_context_set_summary | 352  |
| g_option_context_set_translate_func | 353 |
| g_option_context_set_translation_domain | 353 |
| G_OPTION_ERROR               | 351  |
| g_option_group_add_entries   | 360  |
| g_option_group_free          | 359  |
| g_option_group_new           | 359  |
| g_option_group_set_error_hook | 361 |
| g_option_group_set_parse_hooks | 360 |
| g_option_group_set_translate_func | 361 |
| g_option_group_set_translation_domain | 361 |
| G_OPTION_REMAINING           | 357  |
| GOptionArg                   | 355  |
| GOptionArgFunc               | 351  |
| GOptionContext               | 351  |
| GOptionEntry                 | 357  |
| GOptionError                 | 350  |
| GOptionErrorFunc             | 360  |
| GOptionFlags                 | 356  |
| GOptionGroup                 | 358  |
| GOptionParseFunc             | 360  |
| G_OS_BEOS                    | 45   |
| G_OS_UNIX                    | 45   |
| G_OS_WIN32                   | 45   |

| P                            |      |
| g_parse_debug_string         | 303  |
| G_PASTE                      | 65   |
| G_PASTE_ARGS                 | 66   |
| g_path_get_basename          | 299  |
| g_path_get_dirname           | 299  |
g_path_is_absolute, 299
G_PATH_SKIP_ROOT, 299
G_PATTERN_MATCH, 363
G_PATTERN_MATCH_SIMPLE, 364
G_PATTERN_MATCH_STRING, 363
G_PATTERN_SPEC_EQUAL, 363
G_PATTERN_SPEC_FREE, 362
G_PATTERN_SPEC_NEW, 362
GPatternSpec, 362
G_PDP_ENDIAN, 52
GPI, 62
GPi2, 62
GPi4, 62
GPid, 95
gpointer, 36
GPOINTER_TO_INT, 49
GPOINTER_TO_SIZE, 50
GPOINTER_TO_UINT, 49
g_poll, 97
GFollFD, 97
GFollFunc, 89
g_prefix_error, 178
g_print, 179
g_printerr, 180
g_printf, 196
g_printf_string_upper_bound, 199
GPrintFunc, 180
G_PRIORITY_DEFAULT, 84
G_PRIORITY_DEFAULT_IDLE, 84
G_PRIORITY_HIGH, 83
G_PRIORITY_HIGH_IDLE, 84
G_PRIORITY_LOW, 84
GPrivate, 126
g_private_get, 127
g_private_new, 126
G_PRIVATE_SET, 127
g_propagate_error, 177
G_PROPAGATE_ERROR, 177
G_PROPAGATE_PREFERRED_ERROR, 177
g_ptr_array_add, 545
G_PTR_ARRAY_FOREACH, 548
G_PTR_ARRAY_FREE, 547
G_PTR_ARRAY_INDEX, 547
G_PTR_ARRAY_NEW, 543
G_PTR_ARRAY_NEW_WITH_FREE_FUNC, 544
G_PTR_ARRAY_REF, 544
G_PTR_ARRAY_REMOVE, 545
G_PTR_ARRAY_REMOVE_FAST, 545
G_PTR_ARRAY_REMOVE_INDEX, 545
G_PTR_ARRAY_REMOVE_INDEX_FAST, 546
G_PTR_ARRAY_REMOVE_RANGE, 546
G_PTR_ARRAY_SET_FREE_FUNC, 544
G_PTR_ARRAY_SIZE, 547
G_PTR_ARRAY_SIZED_NEW, 544
G_PTR_ARRAY_SORT, 546
G_PTR_ARRAY_SORT_WITH_DATA, 547
G_PTR_ARRAY_UNREF, 544
GPtrArray, 543
Q
Q_, 256
GQSortWithData, 303
GQuark, 570
G_QUARK_FROM_STATIC_STRING, 571
G_quark_from_string, 571
G_quark_to_string, 571
G_quark_try_string, 571
GQueue, 489
G_QUEUE_CLEAR, 490
G_queue_copy, 491
G_queue_delete_link, 497
G_queue_find, 491
G_queue_find_custom, 491
G_queue_foreach, 491
G_queue_free, 490
G_queue_get_length, 490
G_queue_index, 494
G_QUEUE_INIT, 490
G_queue_init, 490
G_queue_insert_after, 495
G_queue_insert_before, 494
G_queue_insert_sorted, 495
G_queue_is_empty, 490
G_queue_link_index, 497
G_queue_new, 498
G_queue_peak_head, 493
G_queue_peak_head_link, 496
G_queue_nth, 493
G_queue_nth_link, 497
G_queue_pop_head, 493
G_queue_pop_head_link, 496
G_queue_pop_nth, 493
G_queue_pop_nth_link, 496
G_queue_pop_tail, 493
G_queue_pop_tail_link, 496
G_queue_push_head, 492
G_queue_push_head_link, 495
G_queue_push_nth, 492
G_queue_push_nth_link, 496
G_queue_push_tail, 492
G_queue_push_tail_link, 495
G_queue_remove, 494
G_queue_remove_all, 494
G_queue_reverse, 491
G_queue_sort, 492
G_queue_unlink, 497
R
GRand, 277
G_rand_boolean, 279
G_rand_copy, 278
G_rand_double, 279
G_rand_double_range, 280
G_rand_free, 278
G_rand_int, 279
G_rand_int_range, 279
G_rand_new, 278
INDEX

INDEX

g_rand_new_with_seed, 277
g_rand_new_with_seed_array, 278
g_rand_set_seed, 278
g_rand_set_seed_array, 279
g_random_boolean, 280
g_random_double, 280
g_random_double_range, 281
g_random_int, 280
g_random_int_range, 280
g_random_set_seed, 280
g_realloc, 150
GRegex, 371
g_regex_check_replacement, 381
G_REEGX_ERROR, 369
g_regex_escape_string, 373
g_regex_get_capture_count, 373
g_regex_get_max_backref, 373
g_regex_get_pattern, 374
g_regex_match, 374
g_regex_match_all, 376
g_regex_match_all_full, 376
g_regex_match_full, 375
g_regex_match_simple, 374
g_regex_new, 372
g_regex_ref, 372
g_regex_replace, 379
g_regex_replace_eval, 380
g_regex_replace_literal, 380
g_regex_split, 378
g_regex_split_full, 378
g_regex_split_simple, 377
g_regex_unref, 372
GRegexCompileFlags, 369
GRegexError, 366
GRegexEvalCallback, 371
GRegexMatchFlags, 370
GRelation, 581
g_relation_count, 583
g_relation_delete, 583
g_relation_destroy, 583
g_relation_exists, 582
g_relation_index, 582
g_relation_insert, 582
g_relation_new, 582
g_relation_print, 583
g_relation_select, 583
g_remove, 340
g_rename, 339
g_renew, 149
g_return_if_fail, 180
g_return_if_reached, 181
g_return_val_if_fail, 180
g_return_val_if_reached, 181
g_rmdir, 341
S
GScanner, 305
g_scanner_add_symbol, 312
g_scanner_cur_line, 310
g_scanner_cur_position, 310
g_scanner_cur_token, 310
g_scanner_cur_value, 310
g_scanner_destroy, 308
g_scanner_eof, 309
g_scanner_error, 314
g_scanner_foreach_symbol, 312
g_scanner_freeze_symbol_table, 313
g_scanner_get_next_token, 309
g_scanner_input_file, 308
g_scanner_input_text, 309
g_scanner_loopup_symbol, 313
g_scanner_new, 308
g_scanner_next_token, 309
g_scanner_remove_symbol, 312
g_scanner_scope_add_symbol, 311
g_scanner_scope_foreach_symbol, 311
g_scanner_scope_lookup_symbol, 311
g_scanner_scope_remove_symbol, 311
g_scanner_set_scope, 310
g_scanner_sync_file_offset, 309
g_scanner_thaw_symbol_table, 313
g_scanner_unexp_token, 314
g_scanner_warn, 313
GScannerConfig, 307
GScannerMsgFunc, 314
G_SEARCHPATH_SEPARATOR, 45
G_SEARCHPATH_SEPARATOR_S, 46
GSeekType, 161
GSequence, 500
g_sequence_append, 502
g_sequence_foreach, 501
g_sequence_foreach_range, 501
g_sequence_free, 500
g_sequence_get, 506
g_sequence_get_begin_iter, 502
g_sequence_get_end_iter, 502
g_sequence_get_iter_at_pos, 502
g_sequence_get_length, 500
g_sequence_insert_before, 502
g_sequence_insert_sorted, 504
g_sequence_insert_sorted_iter, 504
g_sequence_iter_compare, 508
g_sequence_iter_get_position, 507
g_sequence_iter_is_begin, 507
g_sequence_iter_is_end, 507
g_sequence_iter_move, 508
g_sequence_iter_prev, 507
g_sequence_move, 503
g_sequence_move_range, 505
g_sequence_new, 500
g_sequence_prepend, 503
g_sequence_range_get_midpoint, 509
g_sequence_remove, 505
g_sequence_remove_range, 505
g_sequence_search, 506

603
<table>
<thead>
<tr>
<th>Function/Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>g_sequence_search_iter, 506</td>
</tr>
<tr>
<td>g_sequence_set, 507</td>
</tr>
<tr>
<td>g_sequence_sort, 501</td>
</tr>
<tr>
<td>g_sequence_sort_changed, 504</td>
</tr>
<tr>
<td>g_sequence_sort_changed_iter, 505</td>
</tr>
<tr>
<td>g_sequence_sort_iter, 501</td>
</tr>
<tr>
<td>g_sequence_swap, 503</td>
</tr>
<tr>
<td>GSequenceIter, 500</td>
</tr>
<tr>
<td>GSequenceIterCompareFunc, 500</td>
</tr>
<tr>
<td>g_set_application_name, 293</td>
</tr>
<tr>
<td>g_set_error, 177</td>
</tr>
<tr>
<td>g_set_error_literal, 177</td>
</tr>
<tr>
<td>g_set_prgname, 294</td>
</tr>
<tr>
<td>g_set_print_handler, 179</td>
</tr>
<tr>
<td>g_set_printerr_handler, 180</td>
</tr>
<tr>
<td>g_setenv, 294</td>
</tr>
<tr>
<td>G_SHELL_ERROR, 346</td>
</tr>
<tr>
<td>g_shell_parse_argv, 346</td>
</tr>
<tr>
<td>g_shell_quote, 347</td>
</tr>
<tr>
<td>g_shell_unquote, 347</td>
</tr>
<tr>
<td>GShellError, 346</td>
</tr>
<tr>
<td>gshort, 37</td>
</tr>
<tr>
<td>gsize, 39</td>
</tr>
<tr>
<td>GSIZE_TO_POINTER, 49</td>
</tr>
<tr>
<td>g_slice_alloc, 456</td>
</tr>
<tr>
<td>g_slice_alloc0, 456</td>
</tr>
<tr>
<td>g_slice_copy, 457</td>
</tr>
<tr>
<td>g_slice_dup, 458</td>
</tr>
<tr>
<td>g_slice_free, 458</td>
</tr>
<tr>
<td>g_slice_free1, 457</td>
</tr>
<tr>
<td>g_slice_free_chain, 459</td>
</tr>
<tr>
<td>g_slice_free_chain_with_offset, 457</td>
</tr>
<tr>
<td>g_slice_new, 458</td>
</tr>
<tr>
<td>g_slice_new0, 458</td>
</tr>
<tr>
<td>GSList, 479</td>
</tr>
<tr>
<td>g_slist_alloc, 480</td>
</tr>
<tr>
<td>g_slist_append, 480</td>
</tr>
<tr>
<td>g_slist_concat, 485</td>
</tr>
<tr>
<td>g_slist_copy, 483</td>
</tr>
<tr>
<td>g_slist_delete_link, 482</td>
</tr>
<tr>
<td>g_slist_find, 486</td>
</tr>
<tr>
<td>g_slist_find_custom, 486</td>
</tr>
<tr>
<td>g_slist_foreach, 485</td>
</tr>
<tr>
<td>g_slist_free, 483</td>
</tr>
<tr>
<td>g_slist_free1, 483</td>
</tr>
<tr>
<td>g_slist_free1_1, 483</td>
</tr>
<tr>
<td>g_slist_index, 487</td>
</tr>
<tr>
<td>g_slist_insert, 481</td>
</tr>
<tr>
<td>g_slist_insert_before, 481</td>
</tr>
<tr>
<td>g_slist_insert_sorted, 481</td>
</tr>
<tr>
<td>g_slist_insert_sorted_with_data, 484</td>
</tr>
<tr>
<td>g_slist_last, 485</td>
</tr>
<tr>
<td>g_slist_length, 483</td>
</tr>
<tr>
<td>g_slist_next, 485</td>
</tr>
<tr>
<td>g_slist_nth, 485</td>
</tr>
<tr>
<td>g_slist_nth_data, 486</td>
</tr>
<tr>
<td>g_slist_pop_allocator, 487</td>
</tr>
<tr>
<td>g_slist_position, 486</td>
</tr>
<tr>
<td>g_slist_prepend, 480</td>
</tr>
<tr>
<td>g_slist_push_allocator, 487</td>
</tr>
<tr>
<td>g_slist_remove, 482</td>
</tr>
<tr>
<td>g_slist_remove_all, 482</td>
</tr>
<tr>
<td>g_slist_remove_link, 482</td>
</tr>
<tr>
<td>g_slist_reverse, 484</td>
</tr>
<tr>
<td>g_slist_sort, 484</td>
</tr>
<tr>
<td>g_slist_sort_with_data, 484</td>
</tr>
<tr>
<td>g_snprintf, 198</td>
</tr>
<tr>
<td>GSource, 97</td>
</tr>
<tr>
<td>g_source_add_poll, 103</td>
</tr>
<tr>
<td>g_source_attach, 100</td>
</tr>
<tr>
<td>g_source_destroy, 100</td>
</tr>
<tr>
<td>g_source_get_can recurse, 101</td>
</tr>
<tr>
<td>g_source_get_context, 102</td>
</tr>
<tr>
<td>g_source_get_current_time, 103</td>
</tr>
<tr>
<td>g_source_get_id, 102</td>
</tr>
<tr>
<td>g_source_get_priority, 101</td>
</tr>
<tr>
<td>g_source_is_destroyed, 100</td>
</tr>
<tr>
<td>g_source_new, 99</td>
</tr>
<tr>
<td>g_source_ref, 99</td>
</tr>
<tr>
<td>g_source_remove, 103</td>
</tr>
<tr>
<td>g_source_remove_byfuncs_user_data, 104</td>
</tr>
<tr>
<td>g_source_remove_byuser_data, 104</td>
</tr>
<tr>
<td>g_source_remove_poll, 103</td>
</tr>
<tr>
<td>g_source_setcallback, 102</td>
</tr>
<tr>
<td>g_source_setcallbackindirect, 103</td>
</tr>
<tr>
<td>g_source_setcanrecurse, 101</td>
</tr>
<tr>
<td>g_source_setfuncs, 99</td>
</tr>
<tr>
<td>g_source_setpriority, 101</td>
</tr>
<tr>
<td>g_source_unref, 99</td>
</tr>
<tr>
<td>GSourceCallbackFuncs, 99</td>
</tr>
<tr>
<td>GSourceDummyMarshal, 98</td>
</tr>
<tr>
<td>GSourceFunc, 102</td>
</tr>
<tr>
<td>GSourceFuncs, 98</td>
</tr>
<tr>
<td>g_spaced_primes_closest, 302</td>
</tr>
<tr>
<td>g_spawn_async, 327</td>
</tr>
<tr>
<td>g_spawn_async_withpipes, 325</td>
</tr>
<tr>
<td>g_spawn_close_pid, 330</td>
</tr>
<tr>
<td>g_spawn_commandlineasync, 329</td>
</tr>
<tr>
<td>g_spawn_commandlinesync, 329</td>
</tr>
<tr>
<td>G_SPAWN_ERROR, 324</td>
</tr>
<tr>
<td>g_spawn_sync, 328</td>
</tr>
<tr>
<td>GSpawnChildSetupFunc, 325</td>
</tr>
<tr>
<td>GSpawnError, 323</td>
</tr>
<tr>
<td>GSpawnFlags, 324</td>
</tr>
<tr>
<td>g_sprintf, 197</td>
</tr>
<tr>
<td>G_SQRT2, 62</td>
</tr>
<tr>
<td>gssize, 39</td>
</tr>
<tr>
<td>g_stat, 340</td>
</tr>
<tr>
<td>G_STATIC_ASSUME, 66</td>
</tr>
<tr>
<td>g_static_mutex_free, 118</td>
</tr>
<tr>
<td>g_static_mutex_getmutex, 118</td>
</tr>
<tr>
<td>G_STATIC_MUTEX_INIT, 117</td>
</tr>
<tr>
<td>g_static_mutexinit, 117</td>
</tr>
<tr>
<td>g_static_mutexlock, 117</td>
</tr>
<tr>
<td>g_static_mutablelock, 117</td>
</tr>
<tr>
<td>g_static_mutexunlock, 117</td>
</tr>
<tr>
<td>g_static_private_free, 129</td>
</tr>
<tr>
<td>g_static_privateget, 128</td>
</tr>
<tr>
<td>Function/Variable</td>
</tr>
<tr>
<td>-----------------------------------</td>
</tr>
<tr>
<td>g_tree_new_full</td>
</tr>
<tr>
<td>g_tree_new_with_data</td>
</tr>
<tr>
<td>g_tree_nnodes</td>
</tr>
<tr>
<td>g_tree_remove</td>
</tr>
<tr>
<td>g_tree_replace</td>
</tr>
<tr>
<td>g_tree_search</td>
</tr>
<tr>
<td>g_tree_steal</td>
</tr>
<tr>
<td>g_tree_traverse</td>
</tr>
<tr>
<td>TRUE</td>
</tr>
<tr>
<td>g_try_malloc</td>
</tr>
<tr>
<td>g_try_malloc0</td>
</tr>
<tr>
<td>g_try_new</td>
</tr>
<tr>
<td>g_try_new0</td>
</tr>
<tr>
<td>g_try_realloc</td>
</tr>
<tr>
<td>g_try_renew</td>
</tr>
<tr>
<td>G_TRYLOCK</td>
</tr>
<tr>
<td>GTuples</td>
</tr>
<tr>
<td>g_tuples_destroy</td>
</tr>
<tr>
<td>g_tuples_index</td>
</tr>
<tr>
<td>guchar</td>
</tr>
<tr>
<td>g_ucs4_to_utf16</td>
</tr>
<tr>
<td>g_ucs4_to_utf8</td>
</tr>
<tr>
<td>guint</td>
</tr>
<tr>
<td>guint16</td>
</tr>
<tr>
<td>GUINT16_FROM_BE</td>
</tr>
<tr>
<td>GUINT16_FROM_LE</td>
</tr>
<tr>
<td>GUINT16_SWAP_BE_PDP</td>
</tr>
<tr>
<td>GUINT16_SWAP_LE_BE</td>
</tr>
<tr>
<td>GUINT16_SWAP_LE_PDP</td>
</tr>
<tr>
<td>GUINT16_TO_BE</td>
</tr>
<tr>
<td>GUINT16_TO_LE</td>
</tr>
<tr>
<td>guint32</td>
</tr>
<tr>
<td>GUINT32_FROM_BE</td>
</tr>
<tr>
<td>GUINT32_FROM_LE</td>
</tr>
<tr>
<td>GUINT32_SWAP_BE_PDP</td>
</tr>
<tr>
<td>GUINT32_SWAP_LE_BE</td>
</tr>
<tr>
<td>GUINT32_SWAP_LE_PDP</td>
</tr>
<tr>
<td>GUINT32_TO_BE</td>
</tr>
<tr>
<td>GUINT32_TO_LE</td>
</tr>
<tr>
<td>guint64</td>
</tr>
<tr>
<td>GUINT64_FROM_BE</td>
</tr>
<tr>
<td>GUINT64_FROM_LE</td>
</tr>
<tr>
<td>GUINT64_SWAP_BE_PDP</td>
</tr>
<tr>
<td>GUINT64_SWAP_LE_BE</td>
</tr>
<tr>
<td>GUINT64_SWAP_LE_PDP</td>
</tr>
<tr>
<td>guint8</td>
</tr>
<tr>
<td>GUINT8_FROM_BE</td>
</tr>
<tr>
<td>GUINT8_FROM_LE</td>
</tr>
<tr>
<td>GUINT8_TO_BE</td>
</tr>
<tr>
<td>GUINT8_TO_LE</td>
</tr>
<tr>
<td>GUINT8_TO_POINTER</td>
</tr>
<tr>
<td>gulong</td>
</tr>
<tr>
<td>GULONG_FROM_BE</td>
</tr>
<tr>
<td>GULONG_FROM_LE</td>
</tr>
<tr>
<td>GULONG_TO_BE</td>
</tr>
<tr>
<td>GULONG_TO_LE</td>
</tr>
<tr>
<td>gunichar</td>
</tr>
<tr>
<td>gunichar2</td>
</tr>
<tr>
<td>g_unichar_break_type</td>
</tr>
<tr>
<td>g_unichar_combining_class</td>
</tr>
<tr>
<td>g_unichar_digit_value</td>
</tr>
<tr>
<td>g_unichar_get_mirror_char</td>
</tr>
<tr>
<td>g_unichar_get_script</td>
</tr>
<tr>
<td>g_unichar_isalnum</td>
</tr>
<tr>
<td>g_unichar_isalpha</td>
</tr>
<tr>
<td>g_unichar_isctrl</td>
</tr>
<tr>
<td>g_unichar_isdigit</td>
</tr>
<tr>
<td>g_unichar_isgraph</td>
</tr>
<tr>
<td>g_unichar_islower</td>
</tr>
<tr>
<td>g_unichar_ismark</td>
</tr>
<tr>
<td>g_unichar_isprint</td>
</tr>
<tr>
<td>g_unichar_ispunct</td>
</tr>
<tr>
<td>g_unichar_isspace</td>
</tr>
<tr>
<td>g_unichar_isudindex</td>
</tr>
<tr>
<td>g_unichar_isupper</td>
</tr>
<tr>
<td>g_unichar_iswide</td>
</tr>
<tr>
<td>g_unichar_iswide_cjk</td>
</tr>
<tr>
<td>g_unichar_isxdigit</td>
</tr>
<tr>
<td>g_unichar_iszerowidth</td>
</tr>
<tr>
<td>g_unichar_to_utf8</td>
</tr>
<tr>
<td>g_unichar_tolower</td>
</tr>
<tr>
<td>g_unichar_toupper</td>
</tr>
<tr>
<td>g_unichar_type</td>
</tr>
<tr>
<td>g_unichar_validate</td>
</tr>
<tr>
<td>g_unichar_xdigit_value</td>
</tr>
<tr>
<td>g_unicode_canonical_decomposition</td>
</tr>
<tr>
<td>g_unicode_canonical_ordering</td>
</tr>
<tr>
<td>GUnicodeBreakType</td>
</tr>
<tr>
<td>GUnicodeScript</td>
</tr>
<tr>
<td>GUnicodeType</td>
</tr>
<tr>
<td>G_UNLIKELY</td>
</tr>
<tr>
<td>g_unlink</td>
</tr>
<tr>
<td>G_UNLOCK</td>
</tr>
<tr>
<td>g_unsetenv</td>
</tr>
<tr>
<td>g_uri_escape_string</td>
</tr>
<tr>
<td>g_uri_list_extract_uri</td>
</tr>
<tr>
<td>g_uri_parse_schema</td>
</tr>
<tr>
<td>G_URI_RESERVED_CHARS_ALLOWED_IN_PATH</td>
</tr>
<tr>
<td>G_URI_RESERVED_CHARS_ALLOWED_IN_PATH_ELEMENT</td>
</tr>
<tr>
<td>G_URI_RESERVED_CHARS_ALLOWED_IN_USERINFO</td>
</tr>
<tr>
<td>G_URI_RESERVED_CHARS_GENERIC_DELIMITERS</td>
</tr>
<tr>
<td>G_URI_RESERVED_CHARS_SUBCOMPONENT_DELIMITERS</td>
</tr>
<tr>
<td>g_uri_unescape_segment</td>
</tr>
<tr>
<td>g_uri_unescape_string</td>
</tr>
<tr>
<td>G_USEC_PER_SEC</td>
</tr>
<tr>
<td>GUserDirectory</td>
</tr>
<tr>
<td>gushort</td>
</tr>
<tr>
<td>g_usleep</td>
</tr>
<tr>
<td>g_utf16_to_ucs4</td>
</tr>
</tbody>
</table>
g_utf16_to_utf8, 247
g_utf8_casefold, 243
*g_utf8_collate*, 244
* g_utf8_collate_key, 244
* g_utf8_collate_key_for_filename, 245
* g_utf8_find_next_char, 240
* g_utf8_find_prev_char, 240
* g_utf8_get_char, 239
* g_utf8_get_char_validated, 239
* g_utf8_next_char, 239
* g_utf8_normalize, 243
* g_utf8_offset_to_pointer, 239
* g_utf8_pointer_to_offset, 240
* g_utf8_prev_char, 240
* g_utf8_strchr, 241
* g_utf8_strdown, 243
* g_utf8_strlen, 241
* g_utf8_strndup, 241
* g_utf8_strncpy, 241
* g_utf8_strrev, 241
* gUtf8_strreverse, 242
* g_utf8strup, 242
* g_utf8_to_ucs4, 245
* g_utf8_to_ucs4_fast, 246
* g_utf8_to_utf16, 245
* g_utf8_validate, 242
* g_utime, 343

V
G_VA_COPY, 65
* g_vasprintf, 199
* g_vfprintf, 197
* GVoidFunc, 303
* g_vprintf, 197
* g_vsnprintf, 198
* g_vsprintf, 198

W
* g_warn_if_fail, 181
* g_warn_if_reached, 181
* g_warning, 185
G_WIN32_DLLMAIN_FOR_DLL_NAME, 453
* g_win32_error_message, 450
* g_win32_get_package_installation_directory, 451
* g_win32_get_package_installation_directory_of_module, 451
* g_win32_get_package_installation_subdirectory, 452
* g_win32_get_windows_version, 452
* g_win32_getlocale, 450
G_WIN32_HAVE_WIDECHAR_API, 453
G_WIN32_IS_NT_BASED, 454
* g_win32_locale_filename_from_utf8, 453