The GLib/GTK+ Development Platform

A Getting Started Guide

Version 0.7

Sébastien Wilmet

April 21, 2017

Contents

1 Introduction **3**
- 1.1 License . . . 3
- 1.2 Financial Support . . . 3
- 1.3 What is GLib and GTK+? . . . 4
- 1.4 The GNOME Desktop . . . 5
- 1.5 Prerequisites . . . 6
- 1.6 Why and When Using the C Language? . . . 6
  - 1.6.1 Separate the Backend from the Frontend . . . 7
  - 1.6.2 Other Aspects to Keep in Mind . . . 7
- 1.7 Learning Path . . . 9
- 1.8 The Development Environment . . . 9
- 1.9 Acknowledgments . . . 10

I GLib, the Core Library **11**

2 GLib, the Core Library **12**
- 2.1 Basics . . . 13
  - 2.1.1 Type Definitions . . . 13
  - 2.1.2 Frequently Used Macros . . . 13
  - 2.1.3 Debugging Macros . . . 14
  - 2.1.4 Memory . . . 16
  - 2.1.5 String Handling . . . 18
- 2.2 Data Structures . . . 20
  - 2.2.1 Lists . . . 20
  - 2.2.2 Trees . . . 24
  - 2.2.3 Hash Tables . . . 29
- 2.3 The Main Event Loop . . . 31
- 2.4 Other Features . . . 33

II Object-Oriented Programming in C **35**

3 Semi-Object-Oriented Programming in C **37**
- 3.1 Header Example . . . 37
  - 3.1.1 Project Namespace . . . 37
  - 3.1.2 Class Namespace . . . 39
  - 3.1.3 Lowercase, Uppercase or CamelCase? . . . 39
  - 3.1.4 Include Guard . . . 39
  - 3.1.5 C++ Support . . . 39
  - 3.1.6 #include . . . 39

3.1.7 Type Definition . . . 40
3.1.8 Object Constructor . . . 40
3.1.9 Object Destructor . . . 40
3.1.10 Other Public Functions . . . 41
3.2 The Corresponding *.c File . . . 41
3.2.1 Order of #include's . . . 43
3.2.2 GTK-Doc Comments . . . 43
3.2.3 GObject Introspection Annotations . . . 44
3.2.4 Static vs Non-Static Functions . . . 44
3.2.5 Defensive Programming . . . 45
3.2.6 Coding Style . . . 45

4 A Gentle Introduction to GObject 47
4.1 Inheritance . . . 47
4.2 GObject Macros . . . 48
4.3 Interfaces . . . 49
4.4 Reference Counting . . . 49
4.4.1 Avoiding Reference Cycles with Weak References . . . 49
4.4.2 Floating References . . . 50
4.5 Signals and Properties . . . 50
4.5.1 Connecting a Callback Function to a Signal . . . 51
4.5.2 Disconnecting Signal Handlers . . . 53
4.5.3 Properties . . . 56

III Further Reading 59

5 Further Reading 60
5.1 GTK+ and GIO . . . 60
5.2 Writing Your Own GObject Classes . . . 61
5.3 Build System . . . 61
5.3.1 The Autotools . . . 61
5.3.2 Meson . . . 61
5.4 Programming Best-Practices . . . 62

Bibliography 63

Chapter 1

Introduction

This text is a guide to get started with the GLib/GTK+ development platform, with the C language. It is sometimes assumed that the reader uses a Unix-like system.

Warning: this "book" is far from finished, you're reading the version 0.7. If you have any comments, don't hesitate to contact me at `swilmet@gnome.org`, thanks!

The sources of this book are available at:
`https://github.com/swilmet/glib-gtk-book`

1.1 License

Some sections are based on the book *GTK+/Gnome Application Development*, by Havoc Pennington, written in 1999, edited by the New Riders Publishing, and licensed under the Open Publication License. The latest version of the Open Publication License can be found at:
`http://www.opencontent.org/openpub/`

1.2 Financial Support

If you like this guide, you can financially support it!

The guide is released as a Free/*Libre* document and comes free of charge. But it does not materialize out of empty space; it takes time to write. By donating, you demonstrate your appreciation of this work and help its future development.

A donate button is available at:
`https://people.gnome.org/~swilmet/glib-gtk-book/`

Thanks!

1.3 What is GLib and GTK+?

Roughly speaking, GLib is a set of libraries: GLib core, GObject and GIO. Those three libraries are developed in the same Git repository called *glib*, so when referring to "GLib", it can either mean "GLib core" or the broader set including also GObject and GIO.

GLib core provides data structure handling for C (linked lists, trees, hash tables, ...), portability wrappers, an event loop, threads, dynamic loading of modules and lots of utility functions.

GObject – which depends on GLib core – simplifies the object-oriented programming and event-driven programming paradigms in C. Event-driven programming is not only useful for graphical user interfaces (with user events such as key presses and mouse clicks), but also for daemons that respond to hardware changes (a USB stick inserted, a second monitor connected, a printer low on paper), or software that listen to network connections or messages from other processes, and so on.

GIO – which depends on GLib core and GObject – provides high-level APIs for input/output: reading a local file, a remote file, a network stream, inter-process communication with D-Bus, and many more.

The GLib libraries can be used to write operating system services, libraries, command line utilities and such. GLib offers higher-level APIs than the POSIX standard; it is therefore more comfortable to write a C program with GLib.

GTK+ is a widget toolkit based on GLib that can be used to develop applications with a graphical user interface (GUI). The first version of GTK+, or the GIMP Tool Kit[1], was mainly written by Peter Mattis in 1996 for the GIMP (GNU Image Manipulation Program), but has quickly become a general-purpose library. The "+" has been added later to distinguish between the original version and a new version that added object oriented features. GLib started as part of GTK+, but is now a standalone library.

The GLib and GTK+ APIs are documented with GTK-Doc. Special comments are written in the source code, and GTK-Doc extracts those comments to generate HTML pages.

Although GLib and GTK+ are written in C, language bindings are available for JavaScript, Python, Perl and many other programming languages. At the beginning, manual bindings were created, which needed to be updated each time the API of the library changed. Nowadays the language bindings are generic and are thus automatically updated when, for example, new functions are added. This is thanks to GObject Introspection. Special annotations are added to the GTK-Doc comments, to expose more information than what the C syntax can provide, for example about ownership transfer of dynamically-allocated content[2]. Any C library with GObject Introspection support is thus available from many programming languages. In addition, the annotations are also useful to the C programmer because it's a good and succint way to document certain recurrent API aspects.

[1] The name "The GIMP Tool Kit" is now rarely used, today it is more commonly known as GTK+ for short.

[2] For example, whether you need to free the return value.

GLib and GTK+ are part of the GNU Project, whose overall goal is developing a free operating system (named GNU) plus applications to go with it. GNU stands for "GNU's Not Unix", a humorous way of saying that the GNU operating system is Unix-compatible. You can learn more about GNU at `https://www.gnu.org`.

The GLib/GTK+ web site is: `http://www.gtk.org`

1.4 The GNOME Desktop

An important project for GLib and GTK+ is GNOME. GNOME, which is also part of GNU, is a free/libre desktop environment started in 1997 by Miguel de Icaza and Federico Mena-Quintero. GNOME makes extensive use of GTK+, and the latter is now developed primarily by GNOME developers.

"GNOME" is actually an acronym: GNU Network Object Model Environment[3]. Originally, the project was intended to create a framework for application objects, similar to Microsoft's OLE and COM technologies. However, the scope of the project rapidly expanded; it became clear that substantial groundwork was required before the "network object" part of the name could become reality. Old versions of GNOME included an object embedding architecture called Bonobo, and GNOME 1.0 included a fast, light CORBA ORB called ORBit. Bonobo has been replaced by D-Bus, an inter-process communication system.

GNOME has two important faces. From the user's perspective, it is an integrated desktop environment and application suite. From the programmer's perspective, it is an application development framework (made up of numerous useful libraries that are based on GLib and GTK+). Applications written with the GNOME libraries run fine even if the user isn't running the desktop environment, but they integrate nicely with the GNOME desktop if it's available.

The desktop environment includes a "shell" for task switching and launching programs, a "control center" for configuration, lots of applications such as a file manager, a web browser, a movie player, etc. These programs hide the traditional Unix command line behind an easy-to-use graphical interface.

GNOME's development framework makes it possible to write consistent, easy-to-use, interoperable applications. The designers of windowing systems such as X11 or Wayland made a deliberate decision not to impose any user interface policy on developers; GNOME adds a "policy layer," creating a consistent look-and-feel. Finished GNOME applications work well with the GNOME desktop, but can also be used "standalone" – users only need to install GNOME's shared libraries. A GNOME application isn't tied to a specific windowing system, GTK+ provides backends for the X Window System, Wayland, Mac OS X, Windows, and even for a web browser.

At the time of writing, there are new GLib, GTK+ and GNOME stable releases every six months, around March and September. A version number has the form "`major.minor.micro`", where "`minor`" is even for stable versions and is odd for unstable versions. For example, the 3.18.* versions are stable, but the 3.19.* versions are unstable. A new micro stable version (e.g. 3.18.0 → 3.18.1) doesn't add new features, only translation updates, bug fixes and performance

[3] As for GTK+, the complete name of GNOME is rarely used and doesn't reflect today's reality.

improvements. GNOME components are meant to be installed with the same versions, alongside the version of GTK+ and GLib released at the same time; it's for instance a bad idea to run a GNOME daemon in version 3.18 with the control center in version 3.16. At the time of writing, the latest stable versions are: GLib 2.46, GTK+ 3.18 and GNOME 3.18, all released at the same time in September 2015. For a library, a new major version number usually means there has been an API break, but thankfully previous major versions are parallel-installable with the new version. During a development cycle (e.g. 3.19), there is no API stability guarantees for *new* features; but by being an early adopter your comments are useful to discover more quickly design flaws and bugs.

More information about GNOME: `https://www.gnome.org/`

1.5 Prerequisites

This book assumes that you already have some programming practice. Here is a list of recommended prerequisites, with book references.

- This text supposes that you already know the C language. The reference book is *The C Programming Language*, by Brian Kernighan and Dennis Ritchie [1].
- Object-oriented programming (OOP) is also required for learning GObject. You should be familiar with concepts like inheritance, an interface, a virtual method or polymorphism. A good book, with more than sixty guidelines, is *Object-Oriented Design Heuristics*, by Arthur Riel [2].
- Having read a book on data structures and algorithms is useful, but you can learn that in parallel. A recommended book is *The Algorithm Design Manual*, by Steven Skiena [4].
- If you want to develop your software on a Unix-like system, another prerequisite is to know how Unix works, and be familiar with the command line, a bit of shell scripting and how to write a Makefile. A possible book is *UNIX for the Impatient*, by Paul Abrahams [5].
- Not strictly required, but highly recommended is to use a version control system like Git. A good book is *Pro Git*, by Scott Chacon [6].

1.6 Why and When Using the C Language?

The GLib and GTK+ libraries can be used by other programming languages than C. Thanks to GObject Introspection, automatic bindings are available for a variety of languages for all libraries based on GObject. Official GNOME bindings are available for the following languages: C++, JavaScript, Perl, Python and Vala[4]. Vala is a new programming language based on GObject which integrates the peculiarities of GObject directly in its C#-like syntax. Beyond the official GNOME bindings, GLib and GTK+ can be used in more than a dozen other programming languages, with a varying level of support. So why and when choosing the C language? For writing a daemon on a Unix system, C is the *de*

[4] `https://wiki.gnome.org/Projects/Vala`

facto language. But it is less obvious for an application. To answer the question, let's first look at how to structure an application codebase.

1.6.1 Separate the Backend from the Frontend

A good practice is to separate the graphical user interface from the rest of the application. For a variety of reasons, an application's graphical interface tends to be an exceptionally volatile and ever-changing piece of software. It's the focus of most user requests for change. It is difficult to plan and execute well the first time around – often you will discover that some aspect of it is unpleasant to use only after you have written it. Having several different user interfaces is sometimes desirable, for example a command-line version, or a web-based interface.

In practical terms, this means that any large application should have a radical separation between its various *frontends*, or interfaces, and the *backend*. The backend should contain all the "hard parts": your algorithms and data structures, the real work done by the application. Think of it as an abstract "model" being displayed to and manipulated by the user.

Each frontend should be a "view" and a "controller." As a "view," the frontend must note any changes in the backend, and change the display accordingly. As a "controller," the frontend must allow the user to relay requests for change to the backend (it defines how manipulations of the frontend translate into changes in the model).

There are many ways to discipline yourself to keep your application separated. A couple of useful ideas:

- Write the backend as a library. At the beginning the library can be internal to the application and statically linked, without API/ABI stability guarantees. When the project grows up, and if the code is useful for other programs, you can turn easily your backend into a shared library.
- Write at least two frontends from the start; one or both can be ugly prototypes, you just want to get an idea how to structure the backend. Remember, frontends should be easy; the backend has the hard parts.

The C language is a good choice for the backend part of an application. By using GObject and GObject Introspection, your library will be available for other projects written in various programming languages. On the other hand, a Python or JavaScript library cannot be used in other languages. For the frontend(s), a higher-level language may be more convenient, depending on what languages you already are proficient with.

1.6.2 Other Aspects to Keep in Mind

If you're hesitant about the language to choose, here are other aspects to keep in mind. Note that this text is a little biased since the C language has been chosen.

C is a static-typed language: the variable types and function prototypes in a program are all known at compilation time. Lots of trivial errors are discovered by the compiler, such as a typo in a function name. The compiler is also a great help when doing code refactorings, which is essential for the long-term maintainability of a program. For example when you split a class in two, if the

code using the initial class is not updated correctly, the compiler will kindly tell you so[5]. With test-driven development (TDD), and by writing unit tests for *everything*, writing a huge codebase in a dynamic-typed language like Python is also feasible. With a very good code coverage, the unit tests will also detect errors when refactoring the code. But unit tests can be much slower to execute than compiling the code, since it tests also the program behavior. So it's maybe not convenient to run all unit tests when doing code refactorings. Of course writing unit tests is also a good practice for a C codebase! However for the GUI part of the code, writing unit tests is often not a high-priority task if the application is well tested by its developers.

C is an explicit-typed language: the variable types are visible in the code. It is a way to self-document the code; you usually don't need to add comments to explain what the variables contain. Knowing the type of a variable is important to understand the code, to know what the variable represents and what functions can be called on it. On a related matter, the *self* object is passed explicitly as a function argument. Thus when an attribute is accessed through the *self* pointer, you know where the attribute comes from. Some object-oriented languages have the *this* keyword for that purpose, but it is sometimes optional like in C++ or Java. In this latter case, a useful text editor feature is to highlight differently attributes, so even when the *this* keyword is not used, you know that it's an attribute and not a local variable. With the *self* object passed as an argument, there is no possible confusions.

The C language has a very good *toolchain*: stable compilers (GCC, Clang, ...), text editors (Vim, Emacs, ...), debuggers (GDB, Valgrind, ...), static-analysis tools, ...

For some programs, a garbage collector is not appropriate because it pauses the program regularly to release unused memory. For critical code sections, such as real-time animations, it is not desirable to pause the program (a garbage collector can sometimes run during several seconds). In this case, manual memory management like in C is a solution.

Less important, but still useful; the verbosity of C in combination with the GLib/GTK+ conventions has an advantage: the code can be searched easily with a command like `grep`. For example the function `gtk_widget_show()` contains the namespace (`gtk`), the class (`widget`) and the method (`show`). With an object-oriented language, the syntax is generally `object.show()`. If "show" is searched in the code, there will most probably be more false positives, so a smarter tool is needed. Another advantage is that knowing the namespace and the class of a method can be useful when reading the code, it is another form of self-documentation.

More importantly, the GLib/GTK+ API documentation is primarily written for the C language. Reading the C documentation while programming in another language is not convenient. Some tools are currently in development to generate the API documentation for other target languages, so hopefully in the future it won't be a problem anymore.

GLib/GTK+ are themselves written in C. So when programming in C, there is no extra layer. An extra layer is potentially a source of additional bugs and mainte-

[5]Well, *kindly* is perhaps not the best description, spewing out loads of errors is closer to reality.

nance burdens. Moreover, using the C language is probably better for pedagogical purposes. A higher-level language can hide some details about GLib/GTK+. So the code is shorter, but when you have a problem you need to understand not only how the library feature works, but also how the language binding works.

That said, if (1) you're not comfortable in C, (2) you're already proficient with a higher-level language with GObject Introspection support, (3) you plan to write just a small application or plugin, then choosing the higher-level language makes perfect sense.

1.7 Learning Path

Normally this section should be named "Structure of the Book", but as you can see the book is far from finished, so instead the section is named "Learning Path".

The logical learning path is:

1. The basics of GLib core;
2. Object-Oriented Programming in C and the basics of GObject;
3. GTK+ and GIO in parallel.

Since GTK+ is based on GLib and GObject, it is better to understand the basics of those two libraries first. Some tutorials dive directly into GTK+, so after a short period of time you can display a window with some text and three buttons; it's fun, but knowing GLib and GObject is not a luxury if you want to understand what you're doing, and a realistic GTK+ application extensively uses the GLib libraries. GTK+ and GIO can be learned in parallel — once you start using GTK+, you will see that some non-GUI parts are based on GIO.

So this book begins with the GLib core library (part I p. 11), then introduces Object-Oriented Programming in C (part II p. 35) followed by a Further Reading chapter (p. 59).

1.8 The Development Environment

This section describes the development environment typically used when programming with GLib and GTK+ on a Unix system.

On a GNU/Linux distribution, a single package or group can often be installed to get a full C development environment, including, but not limited to:

- a C89-compatible compiler, GCC for instance;
- the GNU debugger GDB;
- GNU Make;
- the Autotools (Autoconf, Automake and Libtool);
- the man-pages of: the Linux kernel and the glibc[6].

For using GLib and GTK+ as a developer, there are several solutions:

[6]Do not confuse the GNU C Library (glibc) with GLib. The former is lower-level.

- The headers and the documentation can be installed with the package manager. The name of the packages end typically with one of the following suffixes: `-devel`, `-dev` or `-doc`. For example `glib2-devel` and `glib2-doc` on Fedora.
- The latest versions of GLib and GTK+ can be installed with Jhbuild: `https://wiki.gnome.org/Projects/Jhbuild`

To read the API documentation of GLib and GTK+, Devhelp is a handy application, if you have installed the `-dev` or `-doc` package. For the text editor or IDE, there are many choices (and a source of many trolls): Vim, Emacs, gedit, Anjuta, MonoDevelop/Xamarin Studio, Geany, ... A promising specialized IDE for GNOME is Builder, currently in development. For creating a GUI with GTK+, you can either write directly the code to do it, or you can use Glade to design the GUI graphically. Finally, GTK-Doc is used for writing API documentation and add the GObject Introspection annotations.

When using GLib or GTK+, pay attention to not use deprecated APIs for newly-written code. Be sure that you read the latest documentations. They are also available online at:
`https://developer.gnome.org/`

1.9 Acknowledgments

Thanks to: Christian Stadelmann, Errol van de l'Isle, Andrew Colin Kissa and Link Dupont.

Part I

GLib, the Core Library

Chapter 2

GLib, the Core Library

GLib is the low-level core library that forms the basis for projects such as GTK+ and GNOME. It provides data structures, utility functions, portability wrappers, and other essential functionality such as an event loop and threads. GLib is available on most Unix-like systems and Windows.

This chapter covers some of the most commonly-used features. GLib is simple, and the concepts are familiar; so we'll move quickly. For more complete coverage of GLib, see the latest API documentation that comes with the library (for the development environment, see section 1.8 on p. 9). By the way: if you have very specific questions about the implementation, don't be afraid to look at the source code. Normally the documentation contains enough information, but if you come across a missing detail, please file a bug (of course, the best would be with a provided patch).

GLib's various facilities are intended to have a consistent interface; the coding style is semi-object-oriented, and identifiers are prefixed with "g" to create a kind of namespace.

GLib has a few toplevel headers:

- `glib.h`, the main header;
- `gmodule.h` for dynamic loading of modules;
- `glib-unix.h` for Unix-specific APIs;
- `glib/gi18n.h` and `glib/gi18n-lib.h` for internationalization;
- `glib/gprintf.h` and `glib/gstdio.h` to avoid pulling in all of stdio.

Note: instead of reinventing the wheel, this chapter is heavily based on the corresponding chapter in the book *GTK+/Gnome Application Development* by Havoc Pennington, licensed under the Open Publication License (see section 1.1 p. 3). GLib has a very stable API. Despite the fact that Havoc Pennington's book was written in 1999 (for GLib 1.2), only a few updates were required to fit the latest GLib versions (version 2.42 at the time of writing).

```
#include <glib.h>

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

Listing 2.1: Familiar C Macros

2.1 Basics

GLib provides substitutes for many standard and commonly-used C language constructs. This section describes GLib's fundamental type definitions, macros, memory allocation routines, and string utility functions.

2.1.1 Type Definitions

Rather than using C's standard types (`int`, `long`, etc.) GLib defines its own. These serve a variety of purposes. For example, `gint32` is guaranteed to be 32 bits wide, something no standard C89 type can ensure. `guint` is simply easier to type than `unsigned`. A few of the typedefs exist only for consistency; for example, `gchar` is always equivalent to the standard `char`.

The most important primitive types defined by GLib:

- `gint8`, `guint8`, `gint16`, `guint16`, `gint32`, `guint32`, `gint64`, `guint64` — these give you integers of a guaranteed size. (If it isn't obvious, the `guint` types are unsigned, the `gint` types are signed.)
- `gboolean` is useful to make your code more readable, since C89 has no `bool` type.
- `gchar`, `gshort`, `glong`, `gint`, `gfloat`, `gdouble` are purely cosmetic.
- `gpointer` may be more convenient to type than `void *`. `gconstpointer` gives you `const void *`. (`const gpointer` will *not* do what you typically mean it to; spend some time with a good book on C if you don't see why.)
- `gsize` is an unsigned integer type that can hold the result of the `sizeof` operator.

2.1.2 Frequently Used Macros

GLib defines a number of familiar macros used in many C programs, shown in Listing 2.1. All of these should be self-explanatory. `MIN()`/`MAX()` return the smaller or larger of their arguments. `ABS()` returns the absolute value of its argument. `CLAMP(x, low, high)` means `x`, unless `x` is outside the range [`low`, `high`]; if `x` is below the range, `low` is returned; if `x` is above the range, `high` is returned. In addition to the macros shown in Listing 2.1, `TRUE`/`FALSE`/`NULL` are defined as the usual `1`/`0`/`((void *)0)`.

There are also many macros unique to GLib, such as the portable `gpointer`-to-`gint` and `gpointer`-to-`guint` conversions shown in Listing 2.2.

```
#include <glib.h>

GINT_TO_POINTER (p);
GPOINTER_TO_INT (p);
GUINT_TO_POINTER (p);
GPOINTER_TO_UINT (p);
```

Listing 2.2: Macros for storing integers in pointers

```
#include <glib.h>

g_return_if_fail (condition);
g_return_val_if_fail (condition, return_value);
```

Listing 2.3: Precondition Checks

Most of GLib's data structures are designed to store a `gpointer`. If you want to store pointers to dynamically allocated objects, this is the right thing. However, sometimes you want to store a simple list of integers without having to dynamically allocate them. Though the C standard does not strictly guarantee it, it is possible to store a `gint` or `guint` in a `gpointer` variable on the wide range of platforms GLib has been ported to; in some cases, an intermediate cast is required. The macros in Listing 2.2 abstract the presence of the cast.

Here's an example:

```
gint my_int;
gpointer my_pointer;

my_int = 5;
my_pointer = GINT_TO_POINTER (my_int);
printf ("We are storing %d\n", GPOINTER_TO_INT (my_pointer));
```

Be careful, though; these macros allow you to store an integer in a pointer, but storing a pointer in an integer will *not* work. To do that portably, you must store the pointer in a `long`. (It's undoubtedly a bad idea to do so, however.)

2.1.3 Debugging Macros

GLib has a nice set of macros you can use to enforce invariants and preconditions in your code. GTK+ uses these liberally – one of the reasons it's so stable and easy to use. They all disappear when you define `G_DISABLE_CHECKS` or `G_DISABLE_ASSERT`, so there's no performance penalty in production code. Using these liberally is a very, very good idea. You'll find bugs much faster if you do. You can even add assertions and checks whenever you find a bug to be sure the bug doesn't reappear in future versions – this complements a regression suite. Checks are especially useful when the code you're writing will be used as a black box by other programmers; users will immediately know when and how they've misused your code.

Of course you should be very careful to ensure your code isn't subtly dependent on debug-only statements to function correctly. Statements that will disappear in production code should *never* have side effects.

```
#include <glib.h>

g_assert (condition);
g_assert_not_reached ();
```

Listing 2.4: Assertions

Listing 2.3 shows GLib's precondition checks. `g_return_if_fail()` prints a warning and immediately returns from the current function if `condition` is `FALSE`. `g_return_val_if_fail()` is similar but allows you to return some `return_value`. These macros are incredibly useful – if you use them liberally, especially in combination with GObject's runtime type checking, you'll halve the time you spend looking for bad pointers and type errors.

Using these functions is simple; here's an example from the GLib hash table implementation:

```
void
g_hash_table_foreach (GHashTable *hash_table,
                      GHFunc      func,
                      gpointer    user_data)
{
  gint i;

  g_return_if_fail (hash_table != NULL);
  g_return_if_fail (func != NULL);

  for (i = 0; i < hash_table->size; i++)
    {
      guint node_hash = hash_table->hashes[i];
      gpointer node_key = hash_table->keys[i];
      gpointer node_value = hash_table->values[i];

      if (HASH_IS_REAL (node_hash))
        (* func) (node_key, node_value, user_data);
    }
}
```

Without the checks, passing `NULL` as a parameter to this function would result in a mysterious segmentation fault. The person using the library would have to figure out where the error occurred with a debugger, and maybe even dig in to the GLib code to see what was wrong. With the checks, they'll get a nice error message telling them that `NULL` arguments are not allowed.

GLib also has more traditional assertion macros, shown in Listing 2.4. `g_assert()` is basically identical to `assert()`, but responds to `G_DISABLE_ASSERT` and behaves consistently across all platforms. `g_assert_not_reached()` is also provided; this is an assertion which always fails. Assertions call `abort()` to exit the program and (if your environment supports it) dump a core file for debugging purposes.

Fatal assertions should be used to check *internal consistency* of a function or library, while `g_return_if_fail()` is intended to ensure sane values are passed to the public interfaces of a program module. That is, if an assertion fails, you typically look for a bug in the module containing the assertion; if a `g_return_if_fail()` check fails, you typically look for the bug in the code which invokes the module.

This code from GLib's calendrical calculations module shows the difference:

```
GDate *
g_date_new_dmy (GDateDay   day,
                GDateMonth month,
                GDateYear  year)
{
  GDate *date;
  g_return_val_if_fail (g_date_valid_dmy (day, month, year), NULL);

  date = g_new (GDate, 1);

  date->julian = FALSE;
  date->dmy = TRUE;

  date->month = month;
  date->day = day;
  date->year = year;

  g_assert (g_date_valid (date));

  return date;
}
```

The precondition check at the beginning ensures the user passes in reasonable values for the day, month and year; the assertion at the end ensures that GLib constructed a sane object, given sane values.

`g_assert_not_reached()` should be used to mark "impossible" situations; a common use is to detect switch statements that don't handle all possible values of an enumeration:

```
switch (value)
  {
  case FOO_ONE:
    break;

  case FOO_TWO:
    break;

  default:
    g_assert_not_reached ();
  }
```

All of the debugging macros print a warning using GLib's `g_log()` facility, which means the warning includes the name of the originating application or library, and you can optionally install a replacement warning-printing routine. For example, you might send all warnings to a dialog box or log file instead of printing them on the console.

2.1.4 Memory

GLib wraps the standard `malloc()` and `free()` with its own `g_` variants, `g_malloc()` and `g_free()`, shown in Listing 2.5. These are nice in several small ways:

```
#include <glib.h>

gpointer g_malloc (gsize n_bytes);
void g_free (gpointer mem);
gpointer g_realloc (gpointer mem, gsize n_bytes);
gpointer g_memdup (gconstpointer mem, guint n_bytes);
```

Listing 2.5: GLib memory allocation

```
#include <glib.h>

g_new (type, count);
g_new0 (type, count);
g_renew (type, mem, count);
```

Listing 2.6: Allocation macros

- `g_malloc()` always returns a `gpointer`, never a `char *`, so there's no need to cast the return value[1].
- `g_malloc()` aborts the program if the underlying `malloc()` fails, so you don't have to check for a `NULL` return value.
- `g_malloc()` gracefully handles a `size` of `0`, by returning `NULL`.
- `g_free()` will ignore any `NULL` pointers you pass to it.

It's important to match `g_malloc()` with `g_free()`, plain `malloc()` with `free()`, and (if you're using C++) `new` with `delete`. Otherwise bad things can happen, since these allocators may use different memory pools (and `new`/`delete` call constructors and destructors).

Of course there's a `g_realloc()` equivalent to `realloc()`. There's also a convenient `g_malloc0()` which fills allocated memory with 0s, and `g_memdup()` which returns a copy of `n_bytes` bytes starting at `mem`. `g_realloc()` and `g_malloc0()` will both accept a size of 0, for consistency with `g_malloc()`. However, `g_memdup()` will not.

If it isn't obvious: `g_malloc0()` fills raw memory with unset bits, not the value 0 for whatever type you intend to put there. Occasionally someone expects to get an array of floating point numbers initialized to 0.0; this is *not* guaranteed to work portably.

Finally, there are type-aware allocation macros, shown in Listing 2.6. The `type` argument to each of these is the name of a type, and the `count` argument is the number of `type`-size blocks to allocate. These macros save you some typing and multiplication, and are thus less error-prone. They automatically cast to the target pointer type, so attempting to assign the allocated memory to the wrong kind of pointer should trigger a compiler warning. (If you have warnings turned on, as a responsible programmer should!)

[1]Before the ANSI/ISO C standard, the `void *` generic pointer type didn't exist, and `malloc()` returned a `char *` value. Nowadays `malloc()` returns a `void *` type — which is the same as `gpointer` — and `void *` allows implicit pointer conversions in C. Casting the return value of `malloc()` is needed if: the developer wants to support old compilers; or if the developer thinks that an explicit conversion makes the code clearer; or if a C++ compiler is used, because in C++ a cast from the `void *` type is required.

```
gint g_snprintf (gchar *string, gulong n, gchar const *format, ...);
```

Listing 2.7: Portability Wrapper

```
#include <glib.h>

gchar * g_strdup (const gchar *str);
gchar * g_strndup (const gchar *str, gsize n);
gchar * g_strdup_printf (const gchar *format, ...);
gchar * g_strdup_vprintf (const gchar *format, va_list args);
gchar * g_strnfill (gsize length, gchar fill_char);
```

Listing 2.8: Allocating Strings

2.1.5 String Handling

GLib provides a number of functions for string handling; some are unique to GLib, and some solve portability concerns. They all interoperate nicely with the GLib memory allocation routines.

For those interested in a better string than `gchar *`, there's also a `GString` type. It isn't covered in this book, see the API documentation for further information.

Listing 2.7 shows a substitute GLib provides for the `snprintf()` function. `g_snprintf()` wraps native `snprintf()` on platforms that have it, and provides an implementation on those that don't.

Pay attention to not use the crash-causing, security-hole-creating, generally evil `sprintf()` function. By using the relatively safe `g_snprintf()` or `g_strdup_printf()` (see below), you can say goodbye to `sprintf()` forever.

Listing 2.8 shows GLib's rich array of functions for allocating strings. Unsurprisingly, `g_strdup()` and `g_strndup()` produce an allocated copy of `str` or the first `n` characters of `str`. For consistency with the GLib memory allocation functions, they return `NULL` if passed a `NULL` pointer. The `printf()` variants return a formatted string. `g_strnfill()` returns a string of size `length` filled with `fill_char`.

`g_strdup_printf()` deserves a special mention; it is a simpler way to handle this common piece of code:

```
gchar *str = g_malloc (256);
g_snprintf (str, 256, "%d printf-style %s", num, string);
```

Instead you could say this, and avoid having to figure out the proper length of the buffer to boot:

```
gchar *str = g_strdup_printf ("%d printf-style %s", num, string);
```

```
#include <glib.h>

gchar * g_strchug (gchar *string);
gchar * g_strchomp (gchar *string);
gchar * g_strstrip (gchar *string);
```

Listing 2.9: In-place string modifications

```
#include <glib.h>

gdouble g_strtod (const gchar *nptr, gchar **endptr);
const gchar * g_strerror (gint errnum);
const gchar * g_strsignal (gint signum);
```

Listing 2.10: String Conversions

```
#include <glib.h>

gchar * g_strconcat (const gchar *string1, ...);
gchar * g_strjoin (const gchar *separator, ...);
```

Listing 2.11: Concatenating Strings

The functions in Listing 2.9 modify a string in-place: `g_strchug()` and `g_strchomp()` "chug" the string (remove leading spaces), or "chomp" it (remove trailing spaces). Those two functions return the string, in addition to modifying it in-place; in some cases it may be convenient to use the return value. There is a macro, `g_strstrip()`, which combines both functions to remove both leading and trailing spaces.

Listing 2.10 shows a few more semi-standard functions GLib wraps. `g_strtod` is like `strtod()` – it converts string `nptr` to a double – with the exception that it will also attempt to convert the double in the `"C"` locale if it fails to convert it in the user's default locale. `*endptr` is set to the first unconverted character, i.e. any text after the number representation. If conversion fails, `*endptr` is set to `nptr`. `endptr` may be `NULL`, causing it to be ignored.

`g_strerror()` and `g_strsignal()` are like their non-`g_` equivalents, but portable. (They return a string representation for an `errno` or a signal number.)

GLib provides some convenient functions for concatenating strings, shown in Listing 2.11. `g_strconcat()` returns a newly-allocated string created by concatenating each of the strings in the argument list. The last argument must be `NULL`, so `g_strconcat()` knows when to stop. `g_strjoin()` is similar, but `separator` is inserted between each string. If `separator` is `NULL`, no separator is used.

Finally, Listing 2.12 summarizes a few routines which manipulate `NULL`-terminated arrays of strings. `g_strsplit()` breaks `string` at each `delimiter`, returning a newly-allocated array. `g_strjoinv()` concatenates each string in the array with an optional `separator`, returning an allocated string. `g_strfreev()` frees each string in the array and then the array itself.

```
#include <glib.h>

gchar ** g_strsplit (const gchar *string,
                     const gchar *delimiter,
                     gint max_tokens);
gchar * g_strjoinv (const gchar *separator, gchar **str_array);
void g_strfreev (gchar **str_array);
```

Listing 2.12: Manipulating `NULL`-terminated string vectors

```
typedef struct _GSList GSList;

struct _GSList
{
  gpointer data;
  GSList *next;
};
```

Listing 2.13: `GSList` cell

```
#include <glib.h>

GSList * g_slist_append (GSList *list, gpointer data);
GSList * g_slist_prepend (GSList *list, gpointer data);
GSList * g_slist_insert (GSList *list, gpointer data, gint position);
GSList * g_slist_remove (GSList *list, gconstpointer data);
```

Listing 2.14: Changing linked list contents

2.2 Data Structures

GLib implements many common data structures, so you don't have to reinvent the wheel every time you want a linked list. This section covers GLib's implementation of linked lists, sorted binary trees, N-ary trees, and hash tables.

2.2.1 Lists

GLib provides generic single and doubly linked lists, `GSList` and `GList`, respectively. These are implemented as lists of `gpointer`; you can use them to hold integers with the `GINT_TO_POINTER` and `GPOINTER_TO_INT` macros. `GSList` and `GList` have almost the same API's, except that there is a `g_list_previous()` function and no `g_slist_previous()`. This section will discuss `GSList` but everything also applies to the doubly linked list.

A `GSList` cell is a self-explanatory structure shown in Listing 2.13. The structure fields are public, so you can use them directly to access the data or to traverse the list.

In the GLib implementation, the empty list is simply a `NULL` pointer. It's always safe to pass `NULL` to list functions since it's a valid list of length 0. Code to create a list and add one element might look like this:

```
GSList *list = NULL;
gchar *element = g_strdup ("a string");
list = g_slist_append (list, element);
```

GLib lists have a noticeable Lisp influence; the empty list is a special "nil" value for that reason. `g_slist_prepend()` works much like `cons` – it's a constant-time operation ($O(1)$) that adds a new cell to the front of the list.

Listing 2.14 shows the basic functions for changing `GSList` contents. For all of these, you must assign the return value to your list pointer in case the head of the list changes. Note that GLib does *not* store a pointer to the tail of the list,

```
#include <glib.h>

typedef void (* GDestroyNotify) (gpointer data);

void g_slist_free (GSList *list);
void g_slist_free_full (GSList *list, GDestroyNotify free_func);
```

Listing 2.15: Freeing entire linked lists

so the append, insert, and remove functions run in $O(n)$ time, with n the length of the list.

GLib will handle memory issues, deallocating and allocating list cells as needed. For example, the following code would remove the above-added element and empty the list:

```
list = g_slist_remove (list, element);
```

`list` is now `NULL`. You still have to free `element` yourself, of course.

To access a list element, you refer to the `GSList` struct directly:

```
gchar *my_data = list->data;
```

To iterate over the list, you might write code like this:

```
GSList *l;

for (l = list; l != NULL; l = l->next)
  {
    gchar *str = l->data;
    g_print ("Element: %s\n", str);
  }
```

Listing 2.15 shows functions to clear an entire list. `g_slist_free()` removes all the links in one fell swoop. `g_slist_free()` has no return value because it would always be `NULL`, and you can simply assign that value to your list if you like. Obviously, `g_slist_free()` frees only the list cells; it has no way of knowing what to do with the list contents. The smarter function `g_slist_free_full()` takes a second argument with a destroy function pointer that is called on each element's data. To free the list containing dynamically-allocated strings, you can write:

```
g_slist_free_full (list, g_free);

/* If list may be used later: */
list = NULL;
```

This is equivalent of writing:

```
GSList *l;

for (l = list; l != NULL; l = l->next)
  g_free (l->data);

g_slist_free (list);
list = NULL;
```

```
#include <glib.h>

typedef void (* GFunc) (gpointer data, gpointer user_data);

GSList * g_slist_find (GSList *list, gconstpointer data);
GSList * g_slist_nth (GSList *list, guint n);
gpointer g_slist_nth_data (GSList *list, guint n);
GSList * g_slist_last (GSList *list);
gint g_slist_index (GSList *list, gconstpointer data);
void g_slist_foreach (GSList *list, GFunc func, gpointer user_data);
```

Listing 2.16: Accessing data in a linked list

Constructing a list using `g_slist_append()` is a *terrible* idea; use `g_slist_prepend()` and then call `g_slist_reverse()` if you need items in a particular order. If you anticipate frequently appending to a list, you can also keep a pointer to the last element. The following code can be used to perform efficient appends[2]:

```
void
efficient_append (GSList   **list,
                  GSList   **list_end,
                  gpointer   data)
{
  g_return_if_fail (list != NULL);
  g_return_if_fail (list_end != NULL);

  if (*list == NULL)
    {
      g_assert (*list_end == NULL);

      *list = g_slist_append (*list, data);
      *list_end = *list;
    }
  else
    {
      *list_end = g_slist_append (*list_end, data)->next;
    }
}
```

To use this function, you would store the list and its end somewhere, and pass their address to `efficient_append()`:

```
GSList* list = NULL;
GSList* list_end = NULL;

efficient_append (&list, &list_end, g_strdup ("Foo"));
efficient_append (&list, &list_end, g_strdup ("Bar"));
efficient_append (&list, &list_end, g_strdup ("Baz"));
```

Of course you have to be careful not to use any list functions that might change the end of the list without updating `list_end`.

For accessing list elements, the functions in Listing 2.16 are provided. None of these change the list's structure. `g_slist_foreach()` applies a `GFunc` to each element of the list.

[2] A more convenient way is to use the `GQueue` data type: a double-ended queue that keeps a pointer to the head, a pointer to the tail, and the length of the doubly linked list.

Used in `g_slist_foreach()`, your `GFunc` will be called on each `list->data` in `list`, passing the `user_data` you provided to `g_slist_foreach()`. `g_slist_foreach()` is comparable to Scheme's "map" function.

For example, you might have a list of strings, and you might want to be able to create a parallel list with some transformation applied to the strings. Here is some code, using the `efficient_append()` function from an earlier example:

```
typedef struct _AppendContext AppendContext;
struct _AppendContext
{
  GSList *list;
  GSList *list_end;
  const gchar *append;
};

static void
append_foreach (gpointer data,
                gpointer user_data)
{
  gchar *oldstring = data;
  AppendContext *context = user_data;

  efficient_append (&context->list,
                    &context->list_end,
                    g_strconcat (oldstring, context->append, NULL));
}

GSList *
copy_with_append (GSList      *list_of_strings,
                  const gchar *append)
{
  AppendContext context;

  context.list = NULL;
  context.list_end = NULL;
  context.append = append;

  g_slist_foreach (list_of_strings, append_foreach, &context);

  return context.list;
}
```

GLib and GTK+ use the "function pointer and user data" idiom heavily. If you have functional programming experience, this is much like using lambda expressions to create a *closure*. (A closure combines a function with an *environment* – a set of name-value bindings. In this case the "environment" is the user data you pass to `append_foreach()`, and the "closure" is the combination of the function pointer and the user data.)

There are some handy list-manipulation routines, listed in Listing 2.17. With the exception of `g_slist_copy()`, all of these affect the lists in-place. Which means you must assign the return value and forget about the passed-in pointer, just as you do when adding or removing list elements. `g_slist_copy()` returns a newly-allocated list, so you can continue to use both lists and must free both lists eventually.

Finally, there are some provisions for sorted lists, shown in Listing 2.18. To use

```
#include <glib.h>

guint g_slist_length (GSList *list);
GSList * g_slist_concat (GSList *list1, GSList *list2);
GSList * g_slist_reverse (GSList *list);
GSList * g_slist_copy (GSList *list);
```

Listing 2.17: Manipulating a linked list

```
#include <glib.h>

typedef gint (* GCompareFunc) (gconstpointer a, gconstpointer b);

GSList * g_slist_insert_sorted (GSList *list, gpointer data, GCompareFunc func);
GSList * g_slist_sort (GSList *list, GCompareFunc compare_func);
GSList * g_slist_find_custom (GSList *list, gconstpointer data, GCompareFunc func);
```

Listing 2.18: Sorted lists

these, you must write a `GCompareFunc`, which is just like the comparison function in the standard C `qsort()`.

If `a < b`, the `GCompareFunc` should return a negative value; if `a > b` a positive value; if `a == b` it should return 0.

Once you have a comparison function, you can insert an element into an already-sorted list, or sort an entire list. Lists are sorted in ascending order. You can even recycle your `GCompareFunc` to find list elements, using `g_slist_find_custom()`.

Be careful with sorted lists; misusing them can rapidly become very inefficient. For example, `g_slist_insert_sorted()` is an $O(n)$ operation, but if you use it in a loop to insert multiple elements the loop runs in quadratic time ($O(n^2)$). It's better to simply prepend all your elements, and then call `g_slist_sort()`. `g_slist_sort()` runs in $O(n \log n)$.

You can also use the `GSequence` data structure for sorted data. `GSequence` has an API of a list, but is implemented internally with a balanced binary tree.

2.2.2 Trees

There are two different kinds of tree in GLib; `GTree` is your basic balanced binary tree, useful to store key-value pairs sorted by key; `GNode` stores arbitrary tree-structured data, such as a parse tree or taxonomy.

GTree

To create and destroy a `GTree`, use a constructor and destructor displayed in Listing 2.19. `GCompareFunc` is the same `qsort()`-style comparison function described for `GSList`; in this case it's used to compare keys in the tree. `g_tree_new_full()` is useful to ease memory management for dynamically-allocated keys and values.

The `GTree` struct is an opaque data type. Its content is accessed and modified only with public functions.

```
#include <glib.h>

typedef gint (* GCompareFunc) (gconstpointer a, gconstpointer b);
typedef gint (* GCompareDataFunc) (gconstpointer a,
                                   gconstpointer b,
                                   gpointer user_data);

GTree * g_tree_new (GCompareFunc key_compare_func);

GTree * g_tree_new_full (GCompareDataFunc key_compare_func,
                         gpointer key_compare_data,
                         GDestroyNotify key_destroy_func,
                         GDestroyNotify value_destroy_func);

void g_tree_destroy (GTree *tree);
```

Listing 2.19: Creating and destroying balanced binary trees

```
#include <glib.h>

void g_tree_insert (GTree *tree, gpointer key, gpointer value);
gboolean g_tree_remove (GTree *tree, gconstpointer key);
gpointer g_tree_lookup (GTree *tree, gconstpointer key);
```

Listing 2.20: Manipulating `GTree` contents

Functions for manipulating the contents of the tree are shown in Listing 2.20. All very straightforward; `g_tree_insert()` overwrites any existing value, so if you don't use `g_tree_new_full()`, be careful if the existing value is your only pointer to a chunk of allocated memory. If `g_tree_lookup()` fails to find the key, it returns `NULL`, otherwise it returns the associated value. Both keys and values have type `gpointer` or `gconstpointer`, but the `GPOINTER_TO_INT()` and `GPOINTER_TO_UINT()` macros allow you to use integers instead.

There are two functions which give you an idea how large the tree is, shown in Listing 2.21.

Using `g_tree_foreach()` (Listing 2.22) you can walk the entire tree. To use it, you provide a `GTraverseFunc`, which is passed each key-value pair and a `data` argument you give to `g_tree_foreach()`. Traversal continues as long as the `GTraverseFunc` returns `FALSE`; if it ever returns `TRUE` then traversal stops. You can use this to search the tree by *value*.

GNode

A `GNode` is an N-ary tree, implemented as a doubly linked list with parent and

```
#include <glib.h>

gint g_tree_nnodes (GTree *tree);
gint g_tree_height (GTree *tree);
```

Listing 2.21: Determining the size of a `GTree`

```
#include <glib.h>

typedef gboolean (* GTraverseFunc) (gpointer key,
                                    gpointer value,
                                    gpointer data);

void g_tree_foreach (GTree *tree, GTraverseFunc func, gpointer user_data);
```

Listing 2.22: Traversing a `GTree`

```
typedef struct _GNode GNode;

struct _GNode
{
  gpointer data;
  GNode *next;
  GNode *prev;
  GNode *parent;
  GNode *children;
};
```

Listing 2.23: `GNode` cell

child lists. Thus, most list operations have analogues in the `GNode` API. You can walk the tree in various ways. Listing 2.23 shows the declaration for a node.

There are macros to access `GNode` members, shown in Listing 2.24. As with `GList`, the `data` member is intended to be used directly. These macros return the `next`, `prev`, and `children` members respectively; they also check whether their argument is `NULL` before dereferencing it, and return `NULL` if it is.

To create a node, the usual `_new()` function is provided (Listing 2.25). `g_node_new()` creates a childless and parentless node containing `data`. Typically `g_node_new()` is used only to create the root node; convenience macros are provided which automatically create new nodes as needed.

To build a tree the fundamental operations shown in Listing 2.26 are used. Each operation returns the just-added node, for convenience when writing loops or recursing the tree. Unlike `GList`, it is safe to ignore the return value.

The convenience macros shown in Listing 2.27 are implemented in terms of the fundamental operations. `g_node_append()` is analagous to `g_node_prepend()`; the rest take a `data` argument, automatically allocate a node for it, and call the corresponding basic operation.

To remove a node from the tree, there are two functions shown in Listing 2.28. `g_node_destroy()` removes the node from a tree, destroying it and all its children.

```
#include <glib.h>

g_node_prev_sibling (node);
g_node_next_sibling (node);
g_node_first_child (node);
```

Listing 2.24: Accessing `GNode`

```
#include <glib.h>

GNode * g_node_new (gpointer data);
```

Listing 2.25: Creating a `GNode`

```
#include <glib.h>

GNode * g_node_insert (GNode *parent, gint position, GNode *node);
GNode * g_node_insert_before (GNode *parent, GNode *sibling, GNode *node);
GNode * g_node_prepend (GNode *parent, GNode *node);
```

Listing 2.26: Building a `GNode` tree

```
#include <glib.h>

g_node_append (parent, node);
g_node_insert_data (parent, position, data);
g_node_insert_data_before (parent, sibling, data);
g_node_prepend_data (parent, data);
g_node_append_data (parent, data);
```

Listing 2.27: Building a `GNode`

```
#include <glib.h>

void g_node_destroy (GNode *root);
void g_node_unlink (GNode *node);
```

Listing 2.28: Destroying a `GNode`

```
#include <glib.h>

G_NODE_IS_ROOT (node);
G_NODE_IS_LEAF (node);
```

Listing 2.29: Predicates for `GNode`

```
#include <glib.h>

guint g_node_n_nodes (GNode *root, GTraverseFlags flags);
GNode * g_node_get_root (GNode *node);
gboolean g_node_is_ancestor (GNode *node, GNode *descendant);
guint g_node_depth (GNode *node);
GNode * g_node_find (GNode *root,
                     GTraverseType order,
                     GTraverseFlags flags,
                     gpointer data);
```

Listing 2.30: `GNode` properties

`g_node_unlink()` removes a node and makes it into a root node; i.e., it converts a subtree into an independent tree.

There are two macros for detecting the top and bottom of a `GNode` tree, shown in Listing 2.29. A root node is defined as a node with no parent or siblings. A leaf node has no children.

You can ask GLib to report useful information about a `GNode`, including the number of nodes it contains, its root node, its depth, and the node containing a particular data pointer. These functions are shown in Listing 2.30.

`GTraverseType` is an enumeration; there are four possible values. Here are their meanings:

- `G_PRE_ORDER` visits the current node, then recurses each child in turn.
- `G_POST_ORDER` recurses each child in order, then visits the current node.
- `G_IN_ORDER` first recurses the leftmost child of the node, then visits the node itself, then recurses the rest of the node's children. This isn't very useful; mostly it is intended for use with a binary tree.
- `G_LEVEL_ORDER` first visits the node itself; then each of the node's children; then the children of the children; then the children of the children of the children; and so on. That is, it visits each node of depth 0, then each node of depth 1, then each node of depth 2, etc.

`GNode`'s tree-traversal functions have a `GTraverseFlags` argument. This is a bit-field used to change the nature of the traversal. Currently there are only three flags – you can visit only leaf nodes, only non-leaf nodes, or all nodes:

- `G_TRAVERSE_LEAVES` means to traverse only leaf nodes.
- `G_TRAVERSE_NON_LEAVES` means to traverse only non-leaf nodes.
- `G_TRAVERSE_ALL` is simply a shortcut for `(G_TRAVERSE_LEAVES | G_TRAVERSE_NON_LEAVES)`.

```
#include <glib.h>

typedef gboolean (* GNodeTraverseFunc) (GNode *node, gpointer data);
typedef void (* GNodeForeachFunc) (GNode *node, gpointer data);

void g_node_traverse (GNode *root,
                      GTraverseType order,
                      GTraverseFlags flags,
                      gint max_depth,
                      GNodeTraverseFunc func,
                      gpointer data);

void g_node_children_foreach (GNode *node,
                              GTraverseFlags flags,
                              GNodeForeachFunc func,
                              gpointer data);

guint g_node_max_height (GNode *root);
void g_node_reverse_children (GNode *node);
guint g_node_n_children (GNode *node);
gint g_node_child_position (GNode *node, GNode *child);
GNode * g_node_nth_child (GNode *node, guint n);
GNode * g_node_last_child (GNode *node);
```

Listing 2.31: Accessing a `GNode`

Listing 2.31 shows some of the remaining `GNode` functions. They are straightforward; most of them are simply operations on the node's list of children. There are two function typedefs unique to `GNode`: `GNodeTraverseFunc` and `GNodeForeachFunc`. These are called with a pointer to the node being visited, and the user data you provide. A `GNodeTraverseFunc` can return `TRUE` to stop whatever traversal is in progress; thus you can use `g_node_traverse()` to search the tree by value.

2.2.3 Hash Tables

`GHashTable` is a simple hash table implementation, providing an associative array with constant-time lookups. To create and destroy a `GHashTable`, use a constructor and destructor listed in Listing 2.32. You must provide a `GHashFunc`, which should return a positive integer when passed a hash key. Each returned `guint` (modulus the size of the table) corresponds to a "slot" or "bucket" in the hash; `GHashTable` handles collisions by storing a linked list of key-value pairs in each slot. Thus, the `guint` values returned by your `GHashFunc` must be fairly evenly distributed over the set of possible `guint` values, or the hash table will degenerate into a linked list. Your `GHashFunc` must also be fast, since it is used for every lookup.

In addition to `GHashFunc`, a `GEqualFunc` is required to test keys for equality. It's used to find the correct key-value pair when hash collisions result in more than one pair in the same hash slot.

If you use the basic constructor `g_hash_table_new()`, remember that GLib has no way of knowing how to destroy the data contained in your hash table; it only destroys the table itself. If the keys and values need to be freed, use `g_hash_table_new_full()`, the destroy functions will be called on each keys and

```
#include <glib.h>

typedef guint (* GHashFunc) (gconstpointer key);
typedef gboolean (* GEqualFunc) (gconstpointer a, gconstpointer b);
typedef void (* GDestroyNotify) (gpointer data);

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);

void g_hash_table_destroy (GHashTable *hash_table);
```

Listing 2.32: `GHashTable` constructors and destructor

```
#include <glib.h>

guint g_int_hash (gconstpointer key);
gboolean g_int_equal (gconstpointer key1, gconstpointer key2);
guint g_direct_hash (gconstpointer key);
gboolean g_direct_equal (gconstpointer key1, gconstpointer key2);
guint g_str_hash (gconstpointer key);
gboolean g_str_equal (gconstpointer key1, gconstpointer key2);
```

Listing 2.33: Pre-written hashes/comparisons

values before destroying the hash table.

Ready-to-use hash and comparison functions are provided for common keys: integers, pointers, strings, and other GLib types. The most common are listed in Listing 2.33. The functions for integers accept a pointer to a `gint`, rather than the `gint` itself. If you pass `NULL` as the hash function argument to `g_hash_table_new()`, `g_direct_hash()` is used by default. If you pass `NULL` as the key equality function, then simple pointer comparison is used (equivalent to `g_direct_equal()`, but without a function call).

Manipulating the hash table is simple. The routines are summarized in Listing 2.34. Insertions do *not* copy the key or value; these are entered into the table exactly as you provide them, replacing any pre-existing key-value pair with the same key ("same" is defined by your hash and equality functions, remember). If this is a problem, you must do a lookup or remove before you insert. Be especially careful if you dynamically allocate keys or values. If you have provided `GDestroyNotify` functions, those will be called automatically on the old key-value pair before replacing it.

The simple `g_hash_table_lookup()` returns the value it finds associated with `key`, or `NULL` if there is no value. Sometimes this won't do. For example, `NULL` may be a valid value in itself. If you're using strings as keys, especially dynamically allocated strings, knowing that a key is in the table might not be enough; you might want to retrieve the exact `gchar *` the hash table is using to represent key `"foo"`. A second lookup function is provided for cases like these. `g_hash_table_lookup_extended()` returns `TRUE` if the lookup succeeded; if it returns `TRUE`, it places the key and value it found in the locations it's given.

```
#include <glib.h>

gboolean g_hash_table_insert (GHashTable *hash_table, gpointer key, gpointer value);
gboolean g_hash_table_remove (GHashTable *hash_table, gconstpointer key);
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);
```

Listing 2.34: Manipulating a `GHashTable`

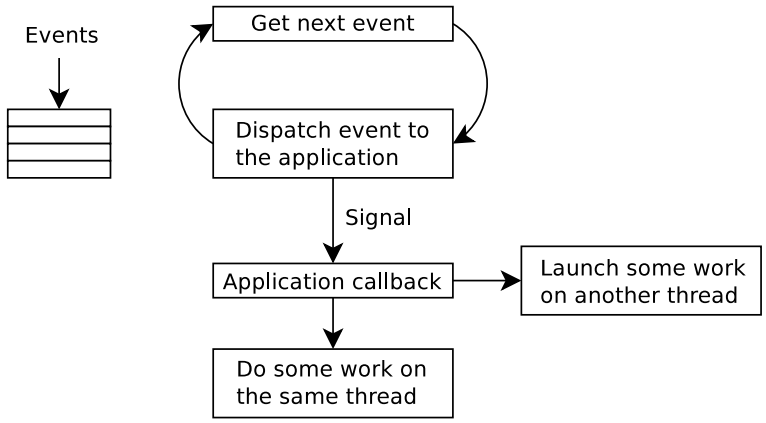

Figure 2.1: Structure of an event-driven application, with a main event loop

2.3 The Main Event Loop

Today's applications are often event-driven. For GUI applications, there are many sources of events: a key press, a mouse click, a touch gesture, a message from another application, a change on the file-system, being connected or disconnected from the network, and so on. An application needs to react to those events. For example when a key is pressed when a text entry has the focus, the character should be inserted and displayed on the screen.

But event-driven programming applies also to daemons. Inter-process communication also happens between daemons and applications. A daemon could receive an event when a packet arrives on a network interface. A printer daemon could receive events when a printer is connected, disconnected, is low-on-paper, etc. A mount daemon can listen to USB sticks inserted. Another daemon can listen to external monitors connections to reconfigure the screens, and so on.

An event-driven program is not structured the same as a batch program. The work to do by a batch program is determined at the start. It then analyzes the input, do some computations on it, and outputs a report. For instance, most Unix commands and scripts are batch programs.

So, how to structure an application that needs to respond to various events that can arrive at any time? As the title of this section suggests... with a *main event loop* of course! That's another important part of GLib; it provides core support of event-driven programming, with a main event loop abstraction, a portable implementation of threads and asynchronous communication between threads. An event loop listens to some sources of events. A priority is associated with

each source of events. When an event arrives, the event loop dispatches it to the application. The event can then be taken into account, either in the same thread or another thread. The Figure 2.1 shows a high-level view of what is a main event loop.

The `main()` function of an event-driven application looks like:

```
gint
main (gint   argc,
      gchar *argv[])
{
  /* Create main window and attach signal callbacks. */

  /* Run the main event loop. */
  gtk_main ();

  return 0;
}
```

The GTK+ event loop is slightly higher level than the GLib event loop abstraction. `gtk_main()` runs the main event loop until `gtk_main_quit()` is called. `gtk_main_quit()` is typically called in the function callback when the close button is clicked or the Quit menu action is activated.

A callback is a function that is called when a signal is sent. The signal system is implemented by the GObject library. Listening to a signal is achieved with the `g_signal_connect()` function:

```
static void
button_clicked_cb (GtkButton *button,
                   gpointer   user_data)
{
  MyObject *self = user_data;

  /* Do something */
}

static void
create_button (MyObject *self)
{
  GtkButton *button;

  /* Create button */

  /* Attach signal callback */
  g_signal_connect (button,
                    "clicked",
                    G_CALLBACK (button_clicked_cb),
                    self);
}
```

When a callback runs, it blocks the main loop. So to not freeze the user interface, there are two solutions:

1. Long operations (especially I/O) can be launched in another thread.
2. Long operations can be split into smaller chunks, and each chunk is run in a separate main loop iteration.

```
#include <glib.h>

typedef gboolean (* GSourceFunc) (gpointer user_data);

guint g_idle_add (GSourceFunc function, gpointer data);
guint g_timeout_add (guint interval, GSourceFunc function, gpointer data);

gboolean g_source_remove (guint source_id);
```

Listing 2.35: Idles and timeouts

For the second solution, GLib provides the `g_idle_add()` and `g_timeout_add()` functions (see Listing 2.35). An idle function will be called when the main loop is idle, that is, when the main loop has nothing else to do. A timeout function is called at regular intervals. The boolean return value of a `GSourceFunc` permits to continue or stop the function. If it continues, the function will be called again by the main loop, at the next idle time or timeout. You can manually remove the `GSourceFunc` by calling `g_source_remove()`, which takes as the parameter the source ID as returned by `g_idle_add()` or `g_timeout_add()`. You must pay attention to remove a `GSourceFunc` when the object on which you do the computation is destroyed. So you can store the source ID in an object attribute, and call `g_source_remove()` in the destructor if the source ID is different from `0`. (See the GObject library to create your own classes in C.)

2.4 Other Features

There simply isn't space to cover all of GLib's features in this book. It's worth looking at GLib whenever you find yourself thinking, "There really *should* be a function that...". This section lists other features GLib provides, but is *not* exhaustive.

Some core application support not already mentioned:

- `GError` – an error reporting system, similar to exceptions in other languages.
- The `g_log()` facility, allows you to print warnings, messages, etc. with configurable log levels and pluggable print routines.

Utilities:

- A commandline option parser.
- A unit-test framework.
- A timer facility.
- Calendrical/date-arithmetic functions.
- Filename manipulation, such as `g_path_get_basename()` and `g_path_is_absolute()`.
- A simple XML parser.
- Perl-compatible regular expressions.

A selection of smaller utilities:

- `G_MAXFLOAT`, etc. equivalents for many numeric types.

- Byte-order conversions.
- `G_DIR_SEPARATOR` handles Windows/Unix differences.
- Convenience/portability routines to get the user's home directory, get the name of a `/tmp` directory, and similar tasks.
- `G_VA_COPY` copies a `va_list` in a portable way.
- Numerous macros to permit the use of compiler extensions (especially GCC extensions) in a portable way.
- Bitfield manipulation.
- Portable `g_htonl()` and other host-to-network conversions.

And last, but not least, other interesting data types:

- Enhanced string and array classes. Pointer and byte arrays.
- `GQuark` – two way mapping from strings to integer identifiers.
- `GVariant` – a generic data type that stores a value along with information about the type of that value.

Part II

Object-Oriented Programming in C

Introduction to Part II

Now that you're familiar with the GLib core library, what is the next step? As the Learning Path section explained (section 1.7 p. 9), the logical follow-up is Object-Oriented Programming (OOP) in C and the basics of GObject.

Every GTK+ widget is a subclass of the GObject base class. So knowing the basic concepts of GObject is important for *using* a GTK+ widget or another GObject-based utility, but also for *creating* your own GObject classes.

It is important to note that although the C language is not object-oriented, it is possible to write "semi-object-oriented" C code easily, without GObject. For learning purposes, that's what this part begins with. GObject is then easier to learn. What GObject adds is more features such as reference counting, inheritance, virtual functions, interfaces, signals and more.

But why following an object-oriented style in the first place? An object-oriented code permits to avoid global variables. And if you have read any sort of programming best-practices guide, you know that you *should*, if possible, avoid global variables[1]. Because using global data makes the code harder to manage and understand, especially when a program becomes larger. It also makes the code more difficult to re-use. It is instead better to break a program into smaller, self-contained pieces, so that you can focus on only one part of the code at a time.

This part of the book comprises two chapters:

- Chapter 3, which explains how to write your own semi-OOP classes;
- Chapter 4, which explains the basics of GObject.

[1] A global variable in C can be a **`static`** variable declared at the top of a *.c file, that can thus be accessed from any function in that *.c file. This is sometimes useful, but should be avoided if possible. There is another kind of global variable in C: an **`extern`** variable that can be accessed from any *.c file. The latter is much worse than the former.

Chapter 3

Semi-Object-Oriented Programming in C

It was explained in the previous chapter that the GLib core library uses a semi-object-oriented coding style. This section explains what it means, and how to write your own code with this coding style.

One of the main ideas of OOP is to *keep related data and behavior in one place*[1]. In C, the data is stored in a `struct`, and the behavior is implemented with functions. To keep those in one place, we put them in the same *.c file, with the public functions present in the corresponding *.h file (the header).

Another important idea of OOP is to *hide all data within its class.* In C, it means that the `struct` full declaration should be present only in the *.c file, while the header contains only a `typedef`. How the data is stored within the class and which data structures are used should remain an implementation detail. A user of the class should not be aware of the implementation details, it should instead rely only on the external interface, that is, what is present in the header and the public documentation. That way, the implementation of the class can change without affecting the users of the class, as long as the API doesn't change.

3.1 Header Example

The Listing 3.1 p. 38 shows an example of a header providing a simple spell checker. This is a fictitious code; if you need a spell checker in your GTK+ application you would nowadays probably use the gspell library[2].

3.1.1 Project Namespace

The first thing to note is the use of the namespace "Myapp". Each symbol in the header is prefixed by the project namespace.

It is a good practice to choose a namespace for your code, to avoid symbol conflicts at link time. It is especially important to have a namespace for a library, but it is also better to have one for an application. Of course the namespace

[1] This is one of the guidelines discussed in *Object-Oriented Design Heuristics* [2].

[2] `https://wiki.gnome.org/Projects/gspell`

```
#ifndef MYAPP_SPELL_CHECKER_H
#define MYAPP_SPELL_CHECKER_H

#include <glib.h>

G_BEGIN_DECLS

typedef struct _MyappSpellChecker MyappSpellChecker;

MyappSpellChecker *
  myapp_spell_checker_new             (const gchar *language_code);

void
  myapp_spell_checker_free            (MyappSpellChecker *checker);

gboolean
  myapp_spell_checker_check_word      (MyappSpellChecker *checker,
                                       const gchar       *word,
                                       gssize             word_length);

GSList *
  myapp_spell_checker_get_suggestions (MyappSpellChecker *checker,
                                       const gchar       *word,
                                       gssize             word_length);

G_END_DECLS

#endif /* MYAPP_SPELL_CHECKER_H */
```

Listing 3.1: myapp-spell-checker.h

needs to be unique for each codebase; for instance you must *not* re-use the "G" or "Gtk" namespaces for your application or library!

3.1.2 Class Namespace

Additionally, there is a second namespace with the class name, here "SpellChecker". If you follow that convention consistently, the name of a symbol is more predictable, which makes the API easier to work with. The name of a symbol will always be "project namespace" + "class name" + "symbol name".

3.1.3 Lowercase, Uppercase or CamelCase?

Depending on the symbol type, its name is either in uppercase, lowercase or CamelCase. The convention in the GLib world is:

- Uppercase letters for a constant, either for a `#define` or an `enum` value;
- CamelCase for a `struct` or `enum` type;
- Lowercase for functions, variables, `struct` fields, ...

3.1.4 Include Guard

The header contains the typical include guard:

```
#ifndef MYAPP_SPELL_CHECKER_H
#define MYAPP_SPELL_CHECKER_H

/* ... */

#endif /* MYAPP_SPELL_CHECKER_H */
```

It protects the header from being included several times in the same *.c file.

3.1.5 C++ Support

The `G_BEGIN_DECLS`/`G_END_DECLS` pair permits the header to be included from C++ code. It is more important for a library, but it is also a good practice to add those macros in application code too, even if the application doesn't use C++. That way an application class could be moved to a library easily (it can be hard to notice that the `G_BEGIN_DECLS` and `G_END_DECLS` macros are missing). And if the desire arises, the application could be ported to C++ incrementally.

3.1.6 #include

There are several ways to organize the `#include`'s in a C codebase. The convention in GLib/GTK+ is that including a header in a *.c file should not require including another header beforehand[3].

[3]That's the general rule of thumb, but exceptions exist: for example including `glib/gi18n-lib.h` requires that `GETTEXT_PACKAGE` is defined, the latter being usually present in the `config.h` header.

`myapp-spell-checker.h` contains the following `#include`:

```
#include <glib.h>
```

Because `glib.h` is needed for the `G_BEGIN_DECLS` and `G_END_DECLS` macros, for the GLib basic type definitions (`gchar`, `gboolean`, etc) and `GSList`.

If the `#include` in `myapp-spell-checker.h` is removed, each *.c file that includes `myapp-spell-checker.h` would also need to include `glib.h` beforehand, otherwise the compiler would not be able to compile that *.c file. But we don't want to add such requirement for the users of the class.

3.1.7 Type Definition

The `MyappSpellChecker` type is defined as follows:

```
typedef struct _MyappSpellChecker MyappSpellChecker;
```

The `struct _MyappSpellChecker` will be declared in the `myapp-spell-checker.c` file. When you use `MyappSpellChecker` in another file, you should not need to know what the `struct` contains, you should use the public functions of the class instead. The exception to that OOP best practice is when calling a function would be a performance problem, for example for low-level data structures used for computer graphics (coordinates, rectangles, ...). But remember: *premature optimization is the root of all evil* (Donald Knuth).

3.1.8 Object Constructor

`myapp_spell_checker_new()` is the constructor of the class. It takes a language code parameter — for example `"en_US"` — and returns an *instance* of the class, also called an *object*. What the function does is simply to allocate dynamically the `struct` and return the pointer. That return value is then used as the first parameter of the remaining functions of the class. In some languages like Python, that first parameter is called the *self* parameter, since it references "itself", i.e. its own class. Other object-oriented languages such as Java and C++ have the *this* keyword to access the object data.

Note that `myapp_spell_checker_new()` can be called several times to create different objects. Each object holds its own data. That's the fundamental difference with global variables: if the data is stored in global variables, there can be at most one instance of it[4].

3.1.9 Object Destructor

`myapp_spell_checker_free()` is the destructor of the class. It destroys an object by freeing its memory and releasing other allocated resources.

[4]Unless the global variable stores the "objects" in an aggregated data structure like an array, a linked list or a hash table, with an identifier as the "*self*" parameter (e.g. an integer or a string) to access a specific element. But this is really ugly code, please don't do that!

3.1.10 Other Public Functions

The `myapp_spell_checker_check_word()` and `myapp_spell_checker_get_suggestions()` functions are the available features of the class. It checks whether a word is correctly spelled, and get a list of suggestions to fix a misspelled word.

The `word_length` parameter type is `gssize`, which is a GLib integer type that can hold — for instance — the result of `strlen()`, and can also hold a negative value since — contrary to `gsize` — `gssize` is a *signed* integer type. The `word_length` parameter can be `-1` if the string is nul-terminated, that is, if the string is terminated by the special character `'\0'`. The purpose of the `word_length` parameter is to be able to pass a pointer to a word that belongs to a larger string, without the need to call for example `g_strndup()`.

3.2 The Corresponding *.c File

Let's now look at the `myapp-spell-checker.c` file:

```
#include "myapp-spell-checker.h"
#include <string.h>

struct _MyappSpellChecker
{
  gchar *language_code;

  /* Put here other data structures used to implement
   * the spell checking.
   */
};

static void
load_dictionary (MyappSpellChecker *checker)
{
  /* ... */
};

/**
 * myapp_spell_checker_new:
 * @language_code: the language code to use.
 *
 * Returns: a new #MyappSpellChecker object. Free with
 * myapp_spell_checker_free().
 */
MyappSpellChecker *
myapp_spell_checker_new (const gchar *language_code)
{
  MyappSpellChecker *checker;

  g_return_val_if_fail (language_code != NULL, NULL);

  checker = g_new0 (MyappSpellChecker, 1);
  checker->language_code = g_strdup (language_code);

  load_dictionary (checker);
```

```
  return checker;
}

/**
 * myapp_spell_checker_free:
 * @checker: a #MyappSpellChecker.
 *
 * Frees @checker.
 */
void
myapp_spell_checker_free (MyappSpellChecker *checker)
{
  if (checker == NULL)
    return;

  g_free (checker->language_code);
  g_free (checker);
}

/**
 * myapp_spell_checker_check_word:
 * @checker: a #MyappSpellChecker.
 * @word: the word to check.
 * @word_length: the byte length of @word, or -1 if @word is nul-terminated.
 *
 * Returns: %TRUE if @word is correctly spelled, %FALSE otherwise.
 */
gboolean
myapp_spell_checker_check_word (MyappSpellChecker *checker,
                                const gchar       *word,
                                gssize             word_length)
{
  g_return_val_if_fail (checker != NULL, FALSE);
  g_return_val_if_fail (word != NULL, FALSE);
  g_return_val_if_fail (word_length >= -1, FALSE);

  /* ... Check if the word is present in a dictionary. */

  return TRUE;
}

/**
 * myapp_spell_checker_get_suggestions:
 * @checker: a #MyappSpellChecker.
 * @word: a misspelled word.
 * @word_length: the byte length of @word, or -1 if @word is nul-terminated.
 *
 * Gets the suggestions for @word. Free the return value with
 * g_slist_free_full(suggestions, g_free).
 *
 * Returns: (transfer full) (element-type utf8): the list of suggestions.
 */
GSList *
myapp_spell_checker_get_suggestions (MyappSpellChecker *checker,
                                     const gchar       *word,
                                     gssize             word_length)
{
```

```
  GSList *suggestions = NULL;

  g_return_val_if_fail (checker != NULL, NULL);
  g_return_val_if_fail (word != NULL, NULL);
  g_return_val_if_fail (word_length >= -1, NULL);

  if (word_length == -1)
    word_length = strlen (word);

  if (strncmp (word, "punchness", word_length) == 0)
    suggestions = g_slist_prepend (suggestions,
                                   g_strdup ("punchiness"));

  return suggestions;
}
```

Listing 3.2: myapp-spell-checker.c

3.2.1 Order of #include's

At the top of the file, there is the usual list of `#include`'s. A small but noteworthy detail is that the include order was not chosen at random. In a certain *.c file, it is better to include first its corresponding *.h file, and then the other headers[5]. By including first `myapp-spell-checker.h`, if an `#include` is missing in `myapp-spell-checker.h`, the compiler will report an error. As explained in section 3.1.6 p. 39, a header should always have the minimum required `#include`'s for that header to be included in turn.

Also, since `glib.h` is already included in `myapp-spell-checker.h`, there is no need to include it a second time in `myapp-spell-checker.c`.

3.2.2 GTK-Doc Comments

The public API is documented with GTK-Doc comments. A GTK-Doc comment begins with `/**`, with the name of the symbol to document on the next line. When we refer to a symbol, there is a special syntax to use depending on the symbol type:

- A function parameter is prefixed by `@`.
- The *name* of a `struct` or `enum` is prefixed by `#`.
- A constant — for example an `enum` *value* — is prefixed by `%`.
- A function is suffixed by `()`.

GTK-Doc can parse those special comments and generate HTML pages that can then be easily navigated by an API browser like Devhelp. But the specially-formatted comments in the code are not the only thing that GTK-Doc needs, it also needs integration to the build system of your project (for example the Autotools), alongside some other files to list the different pages, describe the

[5] Except if you have a `config.h` file, in that case you should *first* include `config.h`, *then* the corresponding *.h, and then the other headers.

general structure with the list of symbols and optionally provide additional content written in the DocBook XML format. Those files are usually present in the `docs/reference/` directory.

Describing in detail how to integrate GTK-Doc support in your code is beyond the scope of this book. For that, you should refer to the GTK-Doc manual [9].

Every GLib-based library should have a GTK-Doc documentation. But it is also useful to write a GTK-Doc documentation for the internal code of an application. As explained in section 1.6.1 p. 7, it is a good practice to separate the backend of an application from its frontend(s), and write the backend as an internal library, or, later, a shared library. As such, it is recommended to document the public API of the backend with GTK-Doc, even if it is still an internal, statically-linked library. Because when the codebase becomes larger, it is a great help – especially for newcomers – to have an overview of the available classes, and to know how to use a class without the need to understand its implementation.

3.2.3 GObject Introspection Annotations

The GTK-Doc comment for the `myapp_spell_checker_get_suggestions()` function contains GObject Introspection annotations for the return value:

```
/**
 * ...
 * Returns: (transfer full) (element-type utf8): the list of suggestions.
 */
```

The function return type is `GSList *`, which is not sufficient for language bindings to know what the list contains, and whether and how to free it. `(transfer full)` means that the return value must be fully freed by the caller: the list itself *and* its elements. `(element-type utf8)` means that the list contains UTF-8 strings.

For a return value, if the transfer annotation is `(transfer container)`, the caller needs to free only the data structure, not its elements. And if the transfer annotation is `(transfer none)`, the ownership of the variable content is not transferred, and thus the caller must not free the return value.

There are many other GObject Introspection (GI) annotations, to name just a couple others:

- `(nullable)`: the value can be `NULL`;
- `(out)`: an "out" function parameter, i.e. a parameter that returns a value.

As you can see, the GI annotations are not just useful for language bindings, they also gather useful information to the C programmer.

For a comprehensive list of GI annotations, see the GObject Introspection wiki [10].

3.2.4 Static vs Non-Static Functions

In the example code, you can see that the `load_dictionary()` function has been marked as `static`. It is in fact a good practice in C to mark internal functions as `static`. A `static` function can be used only in the same *.c file. On the other hand, a public function should be non-static and have a prototype in a header.

There is the `-Wmissing-prototypes` GCC warning option to ensure that a piece of code follows this convention[6].

Also, contrarily to a public function, a `static` function doesn't require to be prefixed by the project and class namespaces (here, `myapp_spell_checker`).

3.2.5 Defensive Programming

Each public function checks its preconditions with the `g_return_if_fail()` or `g_return_val_if_fail()` debugging macros, as described in section 2.1.3 p. 14. The *self* parameter is also checked, to see if it is not `NULL`, except for the destructor, `myapp_spell_checker_free()`, to be consistent with `free()` and `g_free()` which accept a `NULL` parameter for convenience.

Note that the `g_return` macros should be used only for the entry points of a class, that is, in its public functions. You can usually assume that the parameters passed to a `static` function are valid, especially the *self* parameter. However, it is sometimes useful to check an argument of a `static` function with `g_assert()`, to make the code more robust and self-documented.

3.2.6 Coding Style

Finally, it is worth explaining a few things about the coding style. You are encouraged to use early on the same coding style for your project, because it can be something difficult to change afterwards. If every program has different code conventions, it's a nightmare for someone willing to contribute.

Here is an example of a function definition:

```
gboolean
myapp_spell_checker_check_word (MyappSpellChecker *checker,
                                const gchar       *word,
                                gssize             word_length)
{
  /* ... */
}
```

See how the parameters are aligned: there is one parameter per line, with the type aligned on the opening parenthesis, and with the names aligned on the same column. Some text editors can be configured to do that automatically.

For a function *call*, if putting all the parameters on the same line results in a too long line, the parameters should also be aligned on the parenthesis, to make the code easier to read:

```
  function_call (one_param,
                 another_param,
                 yet_another_param);
```

Unlike other projects like the Linux kernel, there is not really a limit on the line length. A GObject-based code can have quite lengthy lines, even if the parameters of a function are aligned on the parenthesis. Of course if a line ends

[6] When using the Autotools, the `AX_COMPILER_FLAGS` Autoconf macro enables, among other things, that GCC flag.

at, say, column 120, it might mean that there are too many indentation levels and that you should extract the code into intermediate functions.

The return type of a function *definition* is on the previous line than the function name, so that the function name is at the beginning of the line. When writing a function *prototype*, the function name should never be at the beginning of a line, it should ideally be on the same line as the return type[7]. That way, you can do a regular expression search to find the implementation of a function, for example with the `grep` shell command:

```
$ grep -n -E "^function_name" *.c
```

Or, if the code is inside a Git repository, it's a little more convenient to use `git grep`:

```
$ git grep -n -E "^function_name"
```

Likewise, there is an easy way to find the declaration of a public `struct`. The convention is to prefix the type name by an underscore when declaring the `struct`. For example:

```
/* In the header: */
typedef struct _MyappSpellChecker MyappSpellChecker;

/* In the *.c file: */
struct _MyappSpellChecker
{
  /* ... */
};
```

As a result, to find the full declaration of the `MyappSpellChecker` type, you can search "`_MyappSpellChecker`":

```
$ git grep -n _MyappSpellChecker
```

In GLib/GTK+, this underscore-prefix convention is normally applied to every `struct` that has a separate `typedef` line. The convention is not thoroughly followed when a `struct` is used in only one *.c file. And the convention is usually not followed for `enum` types. However, for your project, nothing prevents you from applying consistently the underscore-prefix convention to all types.

Note that there exists more sophisticated tools than `grep` to browse a C codebase, for example Cscope[8].

To learn more about the coding style used in GLib, GTK+ and GNOME, read the GNOME Programming Guidelines [11].

[7] In `myapp-spell-checker.h`, the function names are indented with two spaces instead of being on the same lines as the return types, because there is not enough horizontal space on the page.

[8] http://cscope.sourceforge.net/

Chapter 4

A Gentle Introduction to GObject

In the previous chapter we have learned how to write semi-object-oriented code in C. This is how the classes in GLib core are written. GObject goes several steps further into Object-Oriented Programming, with inheritance, interfaces, virtual functions, etc. GObject also simplifies the event-driven programming paradigm, with signals and properties.

It is recommended to create your own GObject classes for writing a GLib/GTK+ application. Unfortunately the code is a little verbose, because the C language is not object-oriented. Boilerplate code is needed for some features, but don't be afraid, there are tools and scripts to generate the boilerplate.

However this chapter takes a step back from the previous chapter, it is just a small introduction to GObject; it will explain the essential things to know how to *use* an existing GObject class (like all GTK+ widgets and classes in GIO). It will not explain how to *create* your own GObject classes, because it is already well covered in the GObject reference manual, and the goal of this book is not to duplicate the whole content of the reference manuals, the goal is more to serve as a getting started guide.

So for more in-depth information on GObject and to know how to create sub-classes, the GObject reference documentation contains introductory chapters: "*Concepts*" and "*Tutorial*", available at:

`https://developer.gnome.org/gobject/stable/`

To explain certain concepts, some examples are taken from GTK+ or GIO. When reading this chapter, you are encouraged to open in parallel Devhelp, to look at the API reference and see for yourself how a GObject-based library is documented. The goal is that you become autonomous and be able to learn any new GObject class, be it in GIO, GTK+ or any other library.

4.1 Inheritance

An important concept of OOP is inheritance. A class can be a sub-class of a parent class. The sub-class inherits the features of the parent class, extending or overriding its behavior.

The GObject library provides the `GObject` base class. Every class in GIO and GTK+ inherit — directly or indirectly — from the `GObject` base class. When looking at a GObject-based class, the documentation (if written with GTK-Doc) always contains an *Object Hierarchy*. For instance, the `GtkApplication` has the following object hierarchy:

```
GObject
└── GApplication
    └── GtkApplication
```

It means that when you create a `GtkApplication` object, you also have access to the functions, signals and properties of `GApplication` (implemented in GIO) and `GObject`. Of course, the `g_application_*` functions take as first argument a variable of type "`GApplication *`", not "`GtkApplication *`". To cast the variable to the good type, the recommended way is to use the `G_APPLICATION()` macro. For example:

```
GtkApplication *app;

g_application_mark_busy (G_APPLICATION (app));
```

4.2 GObject Macros

Each GObject class provides a set of standard macros. The `G_APPLICATION()` macro as demonstrated in the previous section is one of the standard macros provided by the `GApplication` class.

Not all the standard GObject macros will be explained here, just the macros useful for *using* a GObject in a basic way. The other macros are more advanced and are usually useful only when sub-classing a GObject class, when creating a property or a signal, or when overriding a virtual function.

Each GObject class defines a macro of the form `NAMESPACE_CLASSNAME(object)`, which casts the variable to the type "`NamespaceClassname *`" and checks at runtime if the variable correctly contains a `NamespaceClassname` object or a sub-class of it. If the variable is `NULL` or contains an incompatible object, the macro prints a critical warning message to the console and returns NULL.

A standard cast works too, but is most of the time not recommended because there are no runtime checks:

```
GtkApplication *app;

/* Not recommended */
g_application_mark_busy ((GApplication *) app);
```

Another macro useful when using a GObject is `NAMESPACE_IS_CLASSNAME(object)`, which returns `TRUE` if the variable is a `NamespaceClassname` object or a sub-class of it.

4.3 Interfaces

With GObject it is possible to create interfaces. An interface is just an API, it doesn't contain the implementation. A GObject class can implement one or several interfaces. If a GObject class is documented with GTK-Doc, the documentation will contain a section *Implemented Interfaces*.

For example GTK+ contains the `GtkOrientable` interface that is implemented by many widgets and permits to set the orientation: horizontal or vertical.

The two macros explained in the previous section work for interfaces too. An example with `GtkGrid`:

```
GtkWidget *vgrid;

vgrid = gtk_grid_new ();
gtk_orientable_set_orientation (GTK_ORIENTABLE (vgrid),
                                GTK_ORIENTATION_VERTICAL);
```

So when you search a certain feature in the API for a certain GObject class, the feature can be located at three different places:

- In the GObject class itself;
- In one of the parent classes in the *Object Hierarchy*;
- Or in one of the *Implemented Interfaces*.

4.4 Reference Counting

The memory management of GObject classes rely on *reference counting*. A GObject class has a counter:

- When the object is created the counter is equal to one;
- `g_object_ref()` increments the counter;
- `g_object_unref()` decrements the counter;
- If the counter reaches zero, the object is freed.

It permits to store the GObject at several places without the need to coordinate when to free the object.

4.4.1 Avoiding Reference Cycles with Weak References

If object A references object B and object B references object A, there is a reference cycle and the two objects will never be freed. To avoid that problem, there is the concept of "weak" references. When calling `g_object_ref()`, it's a "strong" reference. So in one direction there is a strong reference, and in the other direction there must be a weak reference (or no references at all).

In Figure 4.1 we can see that object A has a strong reference to object B, and object B has a weak reference to object A.

A weak reference can be created with `g_object_add_weak_pointer()` or `g_object_weak_ref()`. As with strong references, it is important to release the reference when no longer

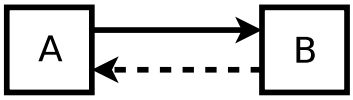

Figure 4.1: Using a weak reference to break the reference cycle between A and B.

needed, usually in the class destructor. A weak reference must be removed with `g_object_remove_weak_pointer()` or `g_object_weak_unref()`. So in Figure 4.1, the destructor of class B must remove the weak reference if it is not already done.

4.4.2 Floating References

When a GObject class inherits from `GInitiallyUnowned` (which is the case of `GtkWidget`), the object initially has a *floating* reference. `g_object_ref_sink()` must be called to convert that floating reference into a normal, strong reference.

When a GObject inherits from `GInitiallyUnowned`, it means that that GObject is meant to be included in some kind of container. The container then assumes ownership of the floating reference, calling `g_object_ref_sink()`. It permits to simplify the code, to remove the need to call `g_object_unref()` after including the object into the container.

The Listing 4.1 p. 52 shows how memory management is handled with a normal GObject. Compare this to the Listing 4.2, which shows how memory management is handled with a GObject deriving from `GInitiallyUnowned`. The difference is that `g_object_unref()` is not called in the latter Listing, so it shortens the code.

So, it's important to know whether a GObject inherits from `GInitiallyUnowned` or not. For that you need to look at the *Object Hierarchy*, for example `GtkEntry` has the following hierarchy:

```
GObject
└── GInitiallyUnowned
    └── GtkWidget
        └── GtkEntry
```

4.5 Signals and Properties

A GObject class can emit signals. With the GLib main event loop (previously explained in section 2.3 p. 31), this is the foundation for event-driven programming. An example of a signal is when the user clicks on a button. The application connects a callback function to the signal to perform the desired action when the event occurs.

Another concept of GObject are *properties*, which is related to signals. A property is basically an instance variable surmounted with a `"notify"` signal that is emitted when its value changes. A good example of a property is the state of a check button, i.e. a boolean value describing whether the button is currently checked or not. When the state changes, the `"notify"` signal is sent.

To create your own signals or properties, a GObject sub-class must be created. As explained in the introduction of this chapter, this is beyond the scope of this book, but you should be aware that creating your own signals or properties is of course possible, and recommended. In fact, creating a GObject signal or property is a nice way to implement the Observer design pattern [3]; that is, one or several objects *observing* state changes of another object, by connecting function callbacks. The object *emitting* the signal is not aware of which objects *receive* the signal. GObject just keeps track of the list of callbacks to call. So adding a signal permits to decouple classes.

4.5.1 Connecting a Callback Function to a Signal

To make things more concrete, if you look at the `GtkButton` documentation, you'll see that it provides the `"clicked"` signal. To perform the desired action when the signal is emitted, one or more callback function(s) must be connected beforehand.

To connect a callback to a signal, the `g_signal_connect()` function can be used, or one of the other `g_signal_connect_*()` functions:

- `g_signal_connect()`
- `g_signal_connect_after()`
- `g_signal_connect_swapped()`
- `g_signal_connect_data()`
- `g_signal_connect_object()`
- And a few more advanced ones.

The Listing 4.3 p. 52 shows the prototype of the `GtkButton::clicked` signal[1].

When using `g_signal_connect()`, the callback function must have the same prototype as the signal prototype. A lot of signals have more arguments, and some signals return a value. If the callback has an incompatible prototype, bad things will happen, there will be random bugs or crashes.

The Listing 4.4 p. 53 shows an example of how to use `g_signal_connect()`.

The `G_CALLBACK()` macro is necessary because `g_signal_connect()` is generic: it can be used to connect to any signal of any GObject class, so the function pointer needs to be casted.

There are two main conventions to name callback functions:

- End the function name with "`cb`", shortcut for "callback". For example: `button_clicked_cb()` as in the above code sample.
- Start the function name with "`on`". For example: `on_button_clicked()`.

With one of those naming conventions — and with the `gpointer user_data` parameter, which is always the last parameter — it is easy to recognize that a function is a callback.

[1]The convention when referring to a GObject signal is "`ClassName::signal-name`". That's how it is documented with GTK-Doc comments.

```
/* Normal GObject */

a_normal_gobject = normal_gobject_new ();
/* a_normal_gobject has now a reference count of 1. */

container_add (container, a_normal_gobject);
/* a_normal_gobject has now a reference count of 2. */

/* We no longer need a_normal_gobject, so we unref it. */
g_object_unref (a_normal_gobject);
/* a_normal_gobject has now a reference count of 1. */
```

Listing 4.1: Memory management of normal GObjects.

```
/* GInitiallyUnowned object, e.g. a GtkWidget */

widget = gtk_entry_new ();
/* widget has now just a floating reference. */

gtk_container_add (container, widget);
/* The container has called g_object_ref_sink(), taking
 * ownership of the floating reference. The code is
 * simplified because we must not call g_object_unref().
 */
```

Listing 4.2: Memory management of GObjects deriving from `GInitiallyUnowned`.

```
void
user_function (GtkButton *button,
               gpointer   user_data);
```

Listing 4.3: The prototype of the `GtkButton::clicked` signal.

```
static void
button_clicked_cb (GtkButton *button,
                   gpointer   user_data)
{
  MyClass *my_class = MY_CLASS (user_data);

  g_message ("Button clicked!");
}

static void
create_button (MyClass *my_class)
{
  GtkButton *button;

  /* Create the button */
  /* ... */

  /* Connect the callback function */
  g_signal_connect (button,
                    "clicked",
                    G_CALLBACK (button_clicked_cb),
                    my_class);
}
```

Listing 4.4: How to connect to a signal

The C language permits to write a different — but compatible — callback function signature, although it is not universally seen as a good thing to do:

- One or more of the *last* function argument(s) can be omitted if they are not used. But as explained above the `gpointer user_data` argument permits to easily recognize that the function is effectively a callback[2].
- The types of the arguments can be modified to a compatible type: e.g. another class in the inheritance hierarchy, or in the above example, replacing "`gpointer`" by "`MyClass *`" (but doing that makes the code a bit less robust because the `MY_CLASS()` macro is not called).

4.5.2 Disconnecting Signal Handlers

In Listing 4.4, `button_clicked_cb()` is called each time that the `button` object emits the `"clicked"` signal. If the `button` object is still alive after `my_class` has been freed, when the signal will be emitted again there will be a little problem... So in the destructor of `MyClass`, the signal handler (i.e. the callback) must be disconnected. How to do that?

The `g_signal_connect*()` functions actually return an ID of the signal handler, as a `gulong` integer always greater than 0 (for successful connections). By storing that ID, it is possible to disconnect that specific signal handler with the `g_signal_handler_disconnect()` function.

Sometimes we also want to disconnect the signal handler simply because we are no longer interested by the event.

[2] As with natural languages, redundancy permits to better and more quickly understand what we read or listen to.

The Listing 4.5 shows a complete example of how to disconnect a signal handler when its `user_data` argument is freed. We come back to an example with a spell checker, because the example with the GTK+ button doesn't fit well the situation[3].

The `user_data` callback argument is a `MyTextView` instance, with `MyTextView` implemented with a semi-OOP style. Since the spell checker object can live longer than the `MyTextView` instance, the signal needs to be disconnected in the `MyTextView` destructor.

```
#include <glib-object.h>

typedef struct _MyTextView MyTextView;

struct _MyTextView
{
  GspellChecker *spell_checker;
  gulong word_added_to_personal_handler_id;
};

static void
word_added_to_personal_cb (GspellChecker *spell_checker,
                           const gchar   *word,
                           gpointer       user_data)
{
  MyTextView *text_view = user_data;

  g_message ("Word '%s' has been added to the user's personal "
             "dictionary. text_view=%p will be updated accordingly.",
             word,
             text_view);
}

MyTextView *
my_text_view_new (GspellChecker *spell_checker)
{
  MyTextView *text_view;

  g_return_val_if_fail (GSPELL_IS_CHECKER (spell_checker), NULL);

  text_view = g_new0 (MyTextView, 1);

  /* We store the spell_checker GObject in the instance variable, so
   * we increase the reference count to be sure that spell_checker
   * stays alive during the lifetime of text_view.
   *
   * Note that spell_checker is provided externally, so spell_checker
   * can live longer than text_view, hence the need to disconnect the
   * signal in my_text_view_free().
   */
```

[3]Most of the time a GTK+ widget doesn't live longer than the container it is added to, and the object listening to the widget signal is usually the container itself. So if the widget dies at the same time as the container, it is not possible for the widget to send a signal while its container has already been destroyed. In that case, there is thus no point in disconnecting the signal in the container destructor, since at that point the widget is already freed; and it's harder for a dead object to send a signal[4].

[4]When I say that it is harder, it is actually impossible, of course.

```
  text_view->spell_checker = g_object_ref (spell_checker);

  text_view->word_added_to_personal_handler_id =
    g_signal_connect (spell_checker,
                      "word-added-to-personal",
                      G_CALLBACK (word_added_to_personal_cb),
                      text_view);

  return text_view;
}

void
my_text_view_free (MyTextView *text_view)
{
  if (text_view == NULL)
    return;

  if (text_view->spell_checker != NULL &&
      text_view->word_added_to_personal_handler_id != 0)
    {
      g_signal_handler_disconnect (text_view->spell_checker,
                                   text_view->word_added_to_personal_handler_id);

      /* Here resetting the value to 0 is not necessary because
       * text_view will anyway be freed, it is just to have a more
       * complete example.
       */
      text_view->word_added_to_personal_handler_id = 0;
    }

  /* The equivalent of:
   * if (text_view->spell_checker != NULL)
   * {
   *   g_object_unref (text_view->spell_checker);
   *   text_view->spell_checker = NULL;
   * }
   *
   * After decreasing the reference count, spell_checker may still be
   * alive if another part of the program still references the same
   * spell_checker.
   */
  g_clear_object (&text_view->spell_checker);

  g_free (text_view);
}
```

Listing 4.5: Disconnecting a signal handler when its `user_data` argument is freed.

There are actually other `g_signal_handler*()` functions that permit to disconnect signal handlers:

- `g_signal_handlers_disconnect_by_data()`
- `g_signal_handlers_disconnect_by_func()`
- `g_signal_handlers_disconnect_matched()`

It would have been possible to use one of the above functions in Listing 4.5, and it would have avoided the need to store `word_added_to_personal_handler_id`.

The basic `g_signal_handler_disconnect()` function has been used for learning purposes.

Note also that if `MyTextView` was a GObject class, it would have been possible to connect to the spell checker signal with `g_signal_connect_object()`, and it would have removed completely the need to manually disconnecting the signal handler in the `MyTextView` destructor. One more (small) reason to learn how to create GObject sub-classes.

4.5.3 Properties

If you have looked at the GTK+ or GIO API reference, you must have noticed that some GObject classes have one or more *properties*. A property is like an instance variable, also called "attribute", but is different in the GObject context because it has additional interesting properties[5].

As previously said at the beginning of section 4.5 p. 50, a good example of a property is the state of a check button, i.e. a boolean value describing whether the button is currently checked or not. If you already know a little GTK+, you might have found that a check button is available with the `GtkCheckButton` widget. But there was a little trap if you wanted to find the property that I was talking about, because the property is implemented in the parent class `GtkToggleButton`: the `GtkToggleButton:active` property[6].

The `"notify"` Signal

The main attribute[7] of a property — besides representing a value — is that the GObject class emits the `GObject::notify` signal when the value of a property changes.

There is one concept of GObject signals that is not yet explained and is used with the `GObject::notify` signal: when emitting a signal, a *detail* can be provided. In the case of the notify signal, the detail is the name of the property whose value has changed. Since there is only one signal for all properties, thanks to the *detail* it is possible to connect a callback to be notified only when a certain property has changed. If the *detail* is not provided when connecting the callback, the callback would be called when *any* of the object properties change, which is generally not what is wanted.

It will be clearer with an example. The Listing 4.6 p. 57 shows how to connect to the notify signal. Note that instead of connecting to the `"notify::active"` detailed signal, it is actually more convenient to use the `GtkToggleButton::toggled` signal. There are better real-world use-cases where it is needed to connect to the notify signal, but at least the Listing 4.6 is hopefully understandable with only limited GTK+ knowledge (and if you look at the documentation in Devhelp in parallel).

[5] Pun intended.

[6] In the same way as signals are documented with GTK-Doc as "`ClassName::signal-name`", properties are documented as "`ClassName:property-name`".

[7] Pun also intended.

```
/* If you look at the notify signal documentation, the first parameter
 * has the type GObject, not GtkCheckButton. Since GtkCheckButton is a
 * sub-class of GObject, the C language allows to write GtkCheckButton
 * directly.
 */
static void
check_button_notify_cb (GtkCheckButton *check_button,
                        GParamSpec     *pspec,
                        gpointer        user_data)
{
  /* Called each time that any property of check_button changes. */
}

static void
check_button_notify_active_cb (GtkCheckButton *check_button,
                               GParamSpec     *pspec,
                               gpointer        user_data)
{
  MyWindow *window = MY_WINDOW (user_data);
  gboolean active;

  active = gtk_toggle_button_get_active (GTK_TOGGLE_BUTTON (check_button));
  gtk_widget_set_visible (window->side_panel, active);
}

static GtkWidget *
create_check_button (MyWindow *window)
{
  GtkWidget *check_button;

  check_button = gtk_check_button_new_with_label ("Show side panel");

  /* Connect without the detail. */
  g_signal_connect (check_button,
                    "notify",
                    G_CALLBACK (check_button_notify_cb),
                    NULL);

  /* Connect with the detail, to be notified only when
   * the GtkToggleButton:active property changes.
   */
  g_signal_connect (check_button,
                    "notify::active",
                    G_CALLBACK (check_button_notify_active_cb),
                    window);

  return check_button;
}
```

Listing 4.6: Connecting to the notify signal to listen to property changes.

Property Bindings

Another useful aspect of properties is that two properties can easily be bound: when one property changes, the other is updated to have the same value. The same can be accomplished with the `"notify"` signal, but higher-level functions exist.

Two properties can be bound in various ways with one of the `g_object_bind_property*()` functions. The Listing 4.7 shows a simpler implementation of Listing 4.6; the code is equivalent, but uses the `g_object_bind_property()` function.

```
static GtkWidget *
create_check_button (MyWindow *window)
{
  GtkWidget *check_button;

  check_button = gtk_check_button_new_with_label ("Show side panel");

  /* When the GtkToggleButton:active property of check_button changes,
   * the GtkWidget:visible property of window->side_panel is updated to
   * have the same boolean value.
   *
   * It would be useful to add G_BINDING_SYNC_CREATE to the flags, but
   * in that case the code would not be equivalent to the previous
   * code Listing.
   */
  g_object_bind_property (check_button, "active",
                          window->side_panel, "visible",
                          G_BINDING_DEFAULT);

  return check_button;
}
```

Listing 4.7: Binding two properties.

Part III

Further Reading

Chapter 5

Further Reading

At this point you should know the basics of GLib core and GObject. You don't need to know *everything* about GLib core and GObject to continue, but having at least a basic understanding will allow you to more easily learn GTK+ and GIO, or any other GObject-based library for that matter.

5.1 GTK+ and GIO

GTK+ and GIO can be learned in parallel.

You should be able to use any GObject class in GIO, just read the class description and skim through the list of functions to have an overview of what features a class provides. Among other interesting things, GIO includes:

- `GFile` to handle files and directories.
- `GSettings` to store application settings.
- `GDBus` – a high-level API for the D-Bus inter-process communication system.
- `GSubprocess` for launching child processes and communicate with them asynchronously.
- `GCancellable`, `GAsyncResult` and `GTask` for using or implementing asynchronous and cancellable tasks.
- Many other features, like I/O streams, network support or application support.

For building graphical applications with GTK+, don't panic, the reference documentation has a Getting Started guide, available with Devhelp or online at:
`https://developer.gnome.org/gtk3/stable/`

After reading the Getting Started guide, skim through the whole API reference to get familiar with the available widgets, containers and base classes. Some widgets have a quite large API, so a few external tutorials are also available, for example for `GtkTextView` and `GtkTreeView`. See the documentation page on:
`http://www.gtk.org`

There is also a series of small tutorials on various GLib/GTK+ topics:
`https://wiki.gnome.org/HowDoI`

5.2 Writing Your Own GObject Classes

The chapter 4 explained how to *use* an existing GObject class, which is very useful to learn GTK+, but it didn't explain how to *create* your own GObject classes. Writing your own GObject classes permit to have reference counting, you can create your own properties and signals, you can implement interfaces, overriding virtual functions (if the virtual function is not associated to a signal), etc.

As was explained at the beginning of chapter 4, if you want to learn more in-depth information on GObject and to know how to create sub-classes, the GObject reference documentation contains introductory chapters: "*Concepts*" and "*Tutorial*", available as usual in Devhelp or online at:
`https://developer.gnome.org/gobject/stable/`

5.3 Build System

A basic Makefile is generally not sufficient if you want to install your application on different systems. So a more sophisticated solution is needed. For a program based on GLib/GTK+, there are two main alternatives: the Autotools and Meson.

GNOME and GTK+ historically use the Autotools, but as of 2017 some modules (including GTK+) are migrating to Meson. For a new project, Meson can be recommended.

5.3.1 The Autotools

The Autotools comprise three main components: Autoconf, Automake and Libtool. It is based on shell scripts, m4 macros and `make`.

Macros are available for various purposes (the user documentation, code coverage statistics for unit tests, etc.). The most recent book on the subject is *Autotools*, by John Calcote [7].

But the Autotools have the reputation to be difficult to learn.

5.3.2 Meson

Meson is a fairly new build system, is easier to learn than the Autotools, and also results to faster builds. Some GNOME modules already use Meson. See the website for more information:
`http://mesonbuild.com/`

5.4 Programming Best-Practices

It is recommended to follow the GNOME Programming Guidelines [11].

The following list is a bit unrelated to GLib/GTK+ development, but is useful for any programming project. After having some practice, it is interesting to learn more about programming *best*-practices. Writing code of good quality is important for preventing bugs and for maintaining a piece of software in the long-run.

- *The* book on programming best-practices is *Code Complete*, by Steve McConnell [8]. Highly recommended[1].
- For guidelines about OOP specifically, see *Object-Oriented Design Heuristics*, by Arthur Riel [2].
- An excellent source of information is the website of Martin Fowler: refactoring, agile methodology, code design, ...
  `http://martinfowler.com/`

More related to GNOME, Havoc Pennington's articles have good advice worth the reading, including "*Working on Free Software*", "*Free software UI*" and "*Free Software Maintenance: Adding Features*":
`http://ometer.com/writing.html`

[1] Although the editor of *Code Complete* is Microsoft Press, the book is not related to Microsoft or Windows. The author sometimes explains stuff related to open source, UNIX and Linux, but one can regret the total absence of the mention "free/libre software" and all the benefits of freedom, in particular for this kind of book: being able to learn by reading other's code. But if you are here, you hopefully already know all of this.

Bibliography

[1] Brian Kernighan and Dennis Ritchie, *The C Programming Language*, Second Edition, Prentice Hall, 1988.

[2] Arthur Riel, *Object-Oriented Design Heuristics*, Addison-Wesley, 1996.

[3] Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1994.

[4] Steven Skiena, *The Algorithm Design Manual*, Second Edition, Springer, 2008.

[5] Paul Abrahams, *UNIX for the Impatient*, Second Edition, Addison-Wesley, 1995.

[6] Scott Chacon, *Pro Git*,
`https://git-scm.com/book`

[7] John Calcote, *Autotools – A Practitioner's Guide to GNU Autoconf, Automake, and Libtool*, No Starch Press, 2010.

[8] Steve McConnell, *Code Complete – A practical handbook of software construction*, Second Edition, Microsoft Press, 2004.

[9] *GTK-Doc Manual*,
`https://developer.gnome.org/gtk-doc-manual/`

[10] *GObject Introspection*,
`https://wiki.gnome.org/Projects/GObjectIntrospection`

[11] *GNOME Programming Guidelines*,
`https://developer.gnome.org/programming-guidelines/stable/`