Iniciando un proyecto GNOME con libglade y autotools

Claudio Saavedra

csaavedra en alumnos utalca cl

Historial de revisiones

Revisión 0.1 10 ene 2004
1ª versión de revisión
Revisión 0.11 13 ene 2004
Corregidos algunos errores menores
Revisión 0.12 02 abr 2005
Actualizado correo electrónico, se cambia el nombre del directorio raíz del proyecto a "hola". Se sugiere la insta
Revisión 0.13 28 ene 2006
Corregidos problemas de codificación de la versión html. Corrección de errores de traducción y ortografía. Agre

En este documento se explica mediante un ejemplo como iniciar un proyecto GNOME utilizando una interfaz creada en Glade-2, la biblioteca libglade, y las herramientas GNU automake y autoconf.

1. Introducción

Como una gran ayuda al desarrollo de aplicaciones GTK y GNOME nació la aplicación Glade, que permite mediante WYSIWYG crear interfaces fácilmente y almacenar éstas en un formato XML de fácil comprensión. Una de las características de Glade es que no solo permite definir la interfaz, sino que además permite crear automáticamente los archivos fuente del proyecto y los archivos `configure` y `Makefile` de modo de que al construir el proyecto la interfaz forma parte de los binarios. Si bien esto es una gran ayuda, se puede transformar en una limitante, ya que cada vez que se necesite modificar algo de la interfaz se debe compilar nuevamente el proyecto.

Para evitar esto y brindar mayor libertad en el uso de los archivos XML de Glade, existe la biblioteca `libglade`. Esta contiene funciones que permiten crear las interfaces accediendo a los archivos XML en tiempo de ejecución, y accediendo solo a los componentes que se indique en el código del programa. Esto da una mayor flexibilidad a la hora de reutilizar Widgets, facilita la edición de la interfaz en la etapa de desarrollo y permite el uso de múltiples interfaces sin necesidad de compilar la aplicación para intercambiar entre éstas.

Explicaremos en este documento como utilizar esta biblioteca para crear una interfaz simple que permita al lector familiarizarse con las funciones más elementales de ésta. No se pretende dar una cátedra avanzada, y se sugiere la lectura del Manual de Referencia de Libglade (http://developer.gnome.org/doc/API/libglade/libglade.html) para mayores detalles.

Nuestro proyecto se llamará `hellolibglade`, y consistirá en una ventana con la frase Hello libGlade! (emulando al popular 'Hello World!') y un botón para finalizar el programa.

Para facilitar la construcción de nuestro proyecto, y adaptarnos con todas las de la ley al desarrollo GNOME, utilizaremos las herramientas GNU automake y autoconf que automatizan la creación de los scripts necesarios para compilar, instalar, distribuir y desinstalar el proyecto, entre otras funcionalidades. Daremos una pequeña introducción al uso de estas herramientas, pero solo de un modo explicativo. Mayor teoría debe ser investigada por el desarrollador.

2. Interfaz del proyecto

Comenzaremos por definir la interfaz del proyecto. Esta ha sido creada en Glade-2.

Figura 1. Interfaz del proyecto

El código del archivo `hello.glade` generado es el siguiente:

```
<?xml version="1.0" standalone="no"?> <!--*- mode: xml -*-->
<!DOCTYPE glade-interface SYSTEM "http://glade.gnome.org/glade-2.0.dtd">

<glade-interface>
<requires lib="gnome"/>

<widget class="GtkWindow" id="main">
  <property name="visible">True</property>
```

```
  <property name="title" translatable="yes">Hello</property>
  <property name="type">GTK_WINDOW_TOPLEVEL</property>
  <property name="window_position">GTK_WIN_POS_CENTER</property>
  <property name="modal">False</property>
  <property name="resizable">False</property>
  <property name="destroy_with_parent">False</property>

  <child>
    <widget class="GtkVBox" id="vbox1">
      <property name="border_width">15</property>
      <property name="visible">True</property>
      <property name="homogeneous">True</property>
      <property name="spacing">0</property>

      <child>
<widget class="GtkLabel" id="label1">
  <property name="visible">True</property>
  <property name="label" translatable="yes">Hello libGlade!</property>
  <property name="use_underline">False</property>
  <property name="use_markup">False</property>
  <property name="justify">GTK_JUSTIFY_LEFT</property>
  <property name="wrap">False</property>
  <property name="selectable">False</property>
  <property name="xalign">0.5</property>
  <property name="yalign">0.5</property>
  <property name="xpad">0</property>
  <property name="ypad">0</property>
</widget>
<packing>
  <property name="padding">0</property>
  <property name="expand">False</property>
  <property name="fill">False</property>
</packing>
      </child>

      <child>
<widget class="GtkButton" id="button_close">
  <property name="visible">True</property>
  <property name="can_default">True</property>
  <property name="has_default">True</property>
  <property name="can_focus">True</property>
  <property name="label">gtk-close</property>
  <property name="use_stock">True</property>
  <property name="relief">GTK_RELIEF_NORMAL</property>
</widget>
<packing>
  <property name="padding">0</property>
  <property name="expand">False</property>
  <property name="fill">False</property>
</packing>
      </child>
    </widget>
  </child>
```

```
</widget>

</glade-interface>
```

La estructura que utilizaremos es la que se sugiere por lo general en un proyecto GNOME y por eso la ocuparemos para ejemplificar.

Los archivos fuente los mantendremos en el directorio `hola/src` y el archivo glade en `hola/src/glade`. Lo que pretendemos con esto es mantener organizados nuestros archivos de modo tal que podamos asociar los archivos glade con las fuentes que se encuentran en el directorio padre de su ubicación. En caso de que necesitemos incluir más fuentes y sus respectivos archivos interfases glade podemos organizarlas de modo tal que incluyamos dichas fuentes en distintos subdirectorios de `hola/src` y en subdirectorios `hola/src/.../glade` las interfaces que corresponden a cada uno, de modo que evitemos que las interfaces se confundan entre si.

Una vez que tenemos nuestro archivo `hello.glade`, procedemos a copiarlo a `hola/src/glade`.

3. Archivos Fuente

Utilizaremos tres archivos para organizar nuestras fuentes. Estos serán `main.c`, `hello.c` y `hello.h`. A continuación explicaremos lo que incluiremos en cada uno de estos archivos.

`main.c`

: En este archivo incluiremos nuestra función main que dará inicio a la ejecución del programa.

`hello.c`

: Contendrá las funcionalidades de nuestro proyecto, que en este caso solo serán las que correspondan al botón, además de la función que crea la ventana de acuerdo a lo definido en la interfaz.

`hello.h`

: Corresponde a la cabecera de nuestro proyecto, donde definiremos las funciones que utilizaremos.

3.1. `hello.c`

Comenzaremos escribiendo nuestro archivo `hello.c`. Lo primero que debemos hacer es incluir las bibliotecas que vamos a necesitar en el proyecto.

```
#include <gtk/gtk.h>
#include <glade/glade.h>
#include "hello.h"
```

Luego definiremos la función que finalizará el loop principal de programa. Esta contiene solo una llamada a `gtk_main_quit()`. Eventualmente, para proyectos de mayor envergadura, en esta función se pueden realizar todas las operaciones necesarias antes de finalizar el programa. Es por eso que creamos esta función y no conectamos directamente el botón close con `gtk_main_quit ()`.

```
void
hello_quit_program (void)
{
gtk_main_quit ();
}
```

Ahora analizaremos la función que toma la ventana definida en el archivo `hello.glade` y la conecta con los objetos widgets que manejaremos utilizando libglade.

```
GtkWidget *
hello_create_main (void)
{
        GtkWidget *window_main;
        GtkWidget *button_close;
         GladeXML *xml_window_main;

        gchar *file_glade;

        file_glade = g_strdup_printf ("%s%c", HELLOLIBGLADE_DATA_DIR,
                        G_DIR_SEPARATOR);

        file_glade = g_strconcat (file_glade, "hello.glade", NULL);

        xml_window_main = glade_xml_new (file_glade, "main", NULL);

        g_free (file_glade);

        glade_xml_signal_autoconnect (xml_window_main);

        window_main = glade_xml_get_widget (xml_window_main, "main");
        button_close = glade_xml_get_widget (xml_window_main, "button_close");

        g_signal_connect (G_OBJECT (window_main), "delete_event",
                          G_CALLBACK (hello_quit_program), NULL);

        g_signal_connect (G_OBJECT (button_close), "clicked",
                          G_CALLBACK (hello_quit_program), NULL);

        return window_main;
}
```

Este proyecto tiene dos widgets que son de nuestro interés. La ventana principal, y el botón close. Es por eso que definimos dos objetos para manejar cada uno de estos. En particular, `window_main` nos permitirá vincular la ventana main del archivo `hello.glade` con el programa, y `button_close` será utilizado para conectar el botón close con la función `hello_quit_program ()` que hemos definido anteriormente.

Para poder conectar el objeto `window_main` con la ventana main definida en el archivo `hello.glade` debemos crear un objeto GladeXML y vincularlo con el respectivo archivo. Para eso usamos la función

```
GladeXML* glade_xml_new(const char *fname, const char *root, const char
*domain);
```

que se encarga de abrir el archivo `fname` y en particular la ventana `root`, si es que se especifica una, y retorna un puntero al nuevo objeto GladeXML (o NULL en caso de error) vinculado a dicha ventana.

En nuestra función, vinculamos el objeto `xml_window_main` con la ventana main en el archivo `hola/src/glade/hello.glade`. Con lo que ya tenemos acceso directo a los componentes de dicha ventana.

Para aumentar la flexibilidad del programa, utilizamos una macro `HELLOLIBGLADE_DATA_DIR` para referirnos a la ubicación del archivo `hello.glade` (esta macro está definida en el archivo `Makefile.am`), junto con la macro `G_DIR_SEPARATOR`, que representa un '/' en sistemas basados en Unix, y un '\' en sistemas Windows. El uso de estas macros es fundamental en la portabilidad del proyecto a distintas plataformas.

la función `xml_signal_autoconnect()` busca coincidencias entre los manejadores de señales definidos en la interfaz y los símbolos en la aplicación, para luego conectar dichas señales con estos símbolos.

Luego conectamos mediante `glade_xml_get_widget()` los objetos GtkWidget del botón y de la ventana con los Widgets definidos en la interfaz. Aquí utilizamos la referencia a la ventana main definida en la interfaz mediante el objeto `xml_window_main ()`.

Cuando una ventana es cerrada mediante una combinación de teclas o presionando el botón X en la parte superior derecha de ésta, se activa su señal 'delete_event'. Para especificar que queremos finalizar la ejecución del programa cuando esto suceda, conectamos esta señal con la función `hello_quit_program ()`.

Luego conectamos la señal 'clicked' del botón `button_close` con la función `hello_quit_program ()`.

De este modo nuestra ventana principal es completamente funcional y está lista para ser llamada. Ahora retornamos la ventana ya lista para ser levantada desde la función `main`, que a continuación explicaremos.

3.2. `main.c`

Este archivo fuente contendrá la función `main` de nuestro proyecto, que será la que dará inicio a la ejecución del programa.

Necesitaremos los siguientes bibliotecas:

```
#include <gtk/gtk.h>
#include <gnome.h>
#include <glade/glade.h>
#include "hello.c"

int
main (int argc, char* argv[])
{
        GtkWidget *window_main;

        gtk_init(&argc, &argv);

        gnome_program_init ("hellolibglade", VERSION, LIBGNOMEUI_MODULE,
                            argc, argv, GNOME_PARAM_APP_DATADIR,
                            HELLOLIBGLADE_DATA_DIR, NULL);

         window_main = hello_create_main ();

        gtk_widget_show (window_main);

        gtk_main ();

        return 0;

}
```

Antes que nada, llamamos a la función `gtk_init ()` que se encarga de inicializar todo lo necesario para utilizar las herramientas de GTK, y recibe algunos argumentos estándares (como es común, en `argc` y `argv[]`) que pudieran venir de la ejecución desde la linea de comandos, por ejemplo, de nuestro programa.

Básicamente, lo que esta función hace es mostrar la ventana `window_main` que es construida por la función `hello_create_main ()`.

Una vez que esta ventana es mostrada, iniciamos el ciclo `gtk_main ()` que solo se detendrá con una llamada a la función `gtk_main_quit ()`. Vagamente hablando, la ejecución de `main` se mantendrá en `gtk_main ()` hasta que `gtk_main_quit ()` sea llamada, entonces seguirá con la ejecución para encontrarse con el

```
return 0;
```

que finalizará la ejecución de nuestro programa.

3.3. `hello.h`

En este archivo definiremos las funciones que hemos creado en `hello.c`. Obviamente, el contenido de este archivo es

```
GtkWidget *hello_create_main(void);
void hello_quit_program(void);
```

4. Autotools

Los dos archivos elementales con los que debe contar el proyecto en su directorio raíz son `Makefile.am` y `configure.in`. Estos archivos son usados por automake y autoconf para generar los archivos `Makefile` y el script `configure` que permiten automatizar el proceso de compilación, empaquetado e instalación del proyecto. Esto es fundamental, dado que todo proyecto GNOME debe hacer uso de estas herramientas para facilitar la instalación de el proyecto en múltiples plataformas, sin hacer más complicado el proceso para una u otra.

4.1. `configure.in`

Este archivo se encarga de generar el script `configure` que hará todas las pruebas necesarias en el sistema para determinar si están dadas las condiciones necesarias para la instalación del programa. Por ejemplo, se encargará de buscar las bibliotecas que son necesarias para la compilación, un compilador de C, etc.

A continuación vamos a incluir el código de dicho archivo y lo explicaremos en detalle.

```
AC_INIT([hellolibglade], [0.1], [http://baboon.utalca.cl/~csaavedra])

AM_INIT_AUTOMAKE

AC_PROG_CC

pkg_modules="libgnomeui-2.0, libglade-2.0"
PKG_CHECK_MODULES(HELLOLIBGLADE, [$pkg_modules])

AC_SUBST(HELLOLIBGLADE_CFLAGS)
AC_SUBST(HELLOLIBGLADE_LIBS)

AC_OUTPUT([
Makefile
src/Makefile
src/glade/Makefile
])
```

En gral., los archivos `configure.in` deben comenzar con una llamada a la macro `AC_INIT`, que se encarga de realizar varias inicializaciones y verificaciones. El formato de uso es el siguiente:

```
AC_INIT(package, version, [bug-report], [tarname])
```

donde:

package

Indica el nombre del paquete, en nuestro caso, la aplicación.

version

Un número para identificar la versión del paquete.

bug-report

(opcional) Una referencia donde un usuario o desarrollador puede dirigirse para hacer comentarios sobre errores en el programa. Puede ser una dirección de correo electrónico o una URL.

tarname

(opcional) Este parámetro difiere de *package* en que este debe ser un nombre corto y sin espacios, para ser utilizado como nombre del archivo tarball, mientras que el primero puede ser un nombre extenso.

`AM_INIT_AUTOMAKE` es una macro que debe se incluida para indicar a autoconf que se utilizará automake para generar y procesar los archivos `Makefile`.

Para poder chequear la existencia de un compilador de C utilizamos la macro`AC_PROG_CC`. Además esta macro se encarga de definir una variable de entorno que indicará el compilador a usar, ya sea **gcc**, **cc**, etc.

Con la macro `PKG_CHECK_MODULES (MYAPP, modules)` verificamos la existencia en el sistema de las bibliotecas indicadas por *modules*, que en este caso controlamos mediante una variable *$packages*.

`AC_SUBST(variable, [value])` es utilizada para explicitar que la variable *variable* debe ser reemplazada en los archivos de salida con el valor del entorno, o con el valor *value* que puede ser especificado opcionalmente.

automake utiliza la macro `AC_OUTPUT` para determinar los archivos que debe crear. Todos los archivos `Makefile` listados son tratados como tal, y los otros se tratan de un modo diferente. Normalmente, la única diferencia es que los archivos `Makefile` son limpiados al hacer un **make distclean** mientras que el resto se eliminan con **make clean**.

4.2. `Makefile.am`

Este archivo incluye información elemental sobre la estructura del proyecto, y es utilizado para generar los complejos archivos `Makefile.in` de modo automático gracias a automake.

Cada subdirectorio del proyecto que debe ser procesado debe contener un archivo `Makefile.am`, indicando los archivos a compilar de ese subdirectorio, los subdirectorios que deben ser procesados recursivamente, entre otros datos.

A continuación incluimos los archivos `Makefile.am` para los directorios `hola`, `hola/src/` y `hola/src/glade` de nuestro proyecto.

4.2.1. `hola/Makefile.am`

```
SUBDIRS = \
src
```

Cuando nuestro proyecto consta de mas de un directorio que debe ser analizado recursivamente, utilizamos la macro `SUBDIRS` para indicar los subdirectorios directos que deben ser analizados. Si se desea analizar un subdirectorio de niveles inferiores, no pueden incluirse directamente en este archivo, sino que debe crearse un `Makefile.am` en el directorio padre de éste que incluya la macro `SUBDIRS`.

Como en nuestro directorio raíz del proyecto no hay nada que construir, este archivo solo servirá de guía para que el proceso continúe en el subdirectorio `hola/src`.

4.2.2. `hola/src/Makefile.am`

```
SUBDIRS = \
        glade

gladedir = $(datadir)/hellolibglade/glade

INCLUDES = \
        -DHELLOLIBGLADE_DATA_DIR=\""$(gladedir)"\" \
        @HELLOLIBGLADE_CFLAGS@

bin_PROGRAMS = hellolibglade

hellolibglade_LDFLAGS = -export-dynamic

hellolibglade_SOURCES = \
        main.c

hellolibglade_LDADD = @HELLOLIBGLADE_LIBS@
```

Además de incluir el subdirectorio glade para que este sea considerado como parte del proyecto que necesita ser procesado, en este subdirectorio realizamos varias tareas.

`INCLUDES` se utiliza para indicar los flags que deben ser pasados al compilador. En nuestro caso, le pasamos el flag *`-DMACRO=macro_value`*, que asigna a la macro `MACRO` el valor especificado por `macro_value` en la compilación de nuestro proyecto. `@HELLOLIBGLADE_CFLAGS@` es simplemente una variable de entorno definida por configure, que contiene los flags que el proyecto necesita.

`bin_PROGRAMS` indica el nombre del archivo binario que se construirá en esta etapa. `hellolibglade_LDFLAGS` indica los flags necesarios para enlazar el proyecto. Utilizamos el flag *`-export-dynamic`* dado que este permite auto conectar los manejadores definidos en el ejecutable principal.

`hellolibglade_SOURCES` indica los archivos fuentes que van a ser compilados.

`hellolibglade_LDADD` especifica las librearías que deben ser pasadas al enlazador para construir el objeto. Estas bibliotecas son las especificadas por la variable `HELLOLIBGLADE_LIBS` definida en `configure`.

4.2.3. `hola/src/glade/Makefile.am`

```
gladedir = $(datadir)/hellolibglade/glade

glade_DATA = hello.glade

EXTRA_DIST = $(glade_DATA)
```

El archivo de interfaz `hello.glade` no es parte del proyecto, en el sentido de que no es necesario para construir el binario. Sin embargo, al momento de ejecutar el proyecto este debe estar presente para poder construir la interfaz. Archivos de este tipo, conocidos como misceláneos, pueden ser incluidos en la distribución mediante el uso de las variables `DATA`. `glade_DATA` es una macro que indica los archivos que son datos del proyecto, referentes a glade. En este caso, solo el archivo `hello.glade`. En caso de que hayan mas interfaces, estas deben ser incluidas aquí.

Mediante `EXTRA_DIST` podemos indicar que archivos extras deben ser distribuidos con la aplicación. En este caso indicamos que `hello.glade` debe ser distribuido, y que debe ser almacenado en el subdirectorio `.../hellolibglade/glade` de la aplicación.

5. Construyendo el proyecto

Una vez que tenemos todos los archivos necesarios del proyecto, debemos utilizar autotools para generar

el script `configure` y los `Makefile`s necesarios para poder distribuir el programa.

Primeramente, utilizaremos la utilidad **aclocal** para generar el archivo `aclocal.m4`. Este archivo contiene macros que son necesarias para **autoconf**. Para generar este archivo, simplemente debemos ejecutar

```
$ aclocal
```

con lo que aclocal.m4 será creado.

Ahora debemos proceder a crear el script de configuración `configure`. Para esto utilizaremos **autoconf**, ejecutándolo así

```
$ autoconf
```

Con esto configure y un directorio autom4te.cache serán creados.

Luego necesitamos crear un conjunto de archivos que son necesarios para cumplir con los estándares de GNU. Estos archivos deben estar en el directorio raíz del proyecto y son los siguientes.

`INSTALL`

: Con información referente al proceso de instalación usando autotools.

`COPYING`

: Con la licencia pública general de la Free Software Foundation. Como bien sabemos, el proyecto GNOME se distribuye bajo esta licencia, por lo que nuestro proyecto también debe hacerlo.

`NEWS`

: Aquí se le indica a los usuarios sobre las características principales del proyecto, y en particular de la distribución que tienen en su poder.

`README`

: En este archivo se debe describir el propósito del proyecto, además de mencionar la documentación restante disponible. En resumen, todo lo que el instalador y usuario deba saber debe mencionarse acá.

`AUTHORS`

: En este archivo se mencionan los autores del proyecto y las personas que contribuyeron directamente. Cualquier otra contribución debe ser incluida en el archivo `THANKS`, que no es requisito, pero se sugiere como forma de agradecer a todos quienes colaboran.

`ChangeLog`

: Es un registro con los cambios que va sufriendo a través del tiempo el proyecto. Deben registrarse cambios detalladamente, y deben servir como referencia para los hackers que están participando en el desarrollo.

De estos debemos crear al menos `NEWS`, `README`, `AUTHORS` y `ChangeLog` (se sugiere el uso de **touch** para esto, pero en un proyecto real el uso adecuado de estos archivos es altamente recomendado). Los restantes pueden ser creados automáticamente por automake al generar los `Makefile`s, si ejecutamos este de la siguiente manera:

```
$ automake --add-missing
```

Con esto los archivos `INSTALL` y `COPYING` serán creados e incluirán instrucciones de instalación y la GPL respectivamente. Desde luego, con esto quedarán creados los archivos `Makefile` necesarios en cada subdirectorio del proyecto. Debe notarse que si **automake** finaliza su ejecución sin dar mensaje alguno, es porque todo ha sido construido sin problemas.

Con esto nuestro paquete está listo para ser distribuido. A continuación procederemos a probarlo como es común, mediante

```
$ ./configure
$ make
```

Debido al uso de libglade, a continuación debemos instalar la aplicación. Para esto, como superusuario debemos ejecutar

```
# make install
```

Con esto los archivos binario y de interfaz serán copiados en su directorio de aplicaciones. Opcionalmente, podemos configurar el proyecto usando el parámetro *`--prefix`* para definir una ubicación alternativa para instalarlo. Por ejemplo

```
$ ./configure --prefix=$HOME
$ make && make install
```

instalará el proyecto en nuestro directorio HOME.

A continuación podemos probar nuestro programa:

```
$ hellolibglade
```

Con lo que podremos ver finalmente a Hello libGlade ejecutarse.

Referencias

Libros

Elephterios Gkioulekas, University of Washington, Departament of Applied Maths, 1998, 1999, *Developing Software with GNU*, http://www.amath.washington.edu/~lf/tutorials/autoconf/ .

Enlaces

Elephterios Gkioulekas, 1998, *Learning the GNU development tools*,
http://www.st-andrews.ac.uk/~iam/docs/tutorial.html .

Germán Poo Caamaño, *Entendiendo autoconf y automake: Parte 1*,
http://cronos.dci.ubiobio.cl/~gpoo/documentos/autotools/ .

Inti, *Building a GNU Autotools Project*, http://inti.sourceforge.net/tutorial/libinti/autotoolsproject.html .

Manuales de Referencia

James Henstridge, 1999, 2002, *Libglade Reference Manual*,
http://developer.gnome.org/doc/API/libglade/libglade.html .