La plataforma de desarrollo GLib/GTK

Una guía de introducción

Versión 0.8

Sébastien Wilmetn

6 de enero de 2021

Índice general

Capítulo 1

Introducción

Este texto es una guía para comenzar con la plataforma de desarrollo GLib/GTK, haciendo uso del lenguaje C. En ocasiones se asumirá que el lector usa un sistema similar a Unix.

La traducción de este libro fue hecha por Gerson Benavides.

Advertencia: este "libro" está lejos de estar terminado, estás leyendo la versión 0.8. Si tiene algún comentario, no dude en ponerse en contacto con el autor en swilmet@gnome.org o con el traductor en gmbsdeveloper@gmail.com, ¡gracias!

Las fuentes de este libro en el idioma original están disponibles en:
`https://github.com/swilmet/glib-gtk-book`

1.1. Licencia

Algunas secciones están basadas en el libro *GTK+/Gnome Application Development*, escrito en 1999, editado por New Riders Publishing y con licencia de Open Publication License. La última versión de la licencia de publicación abierta se puede encontrar en:
`http://www.opencontent.org/openpub/`

1.2. Soporte financiero

Si te gusta esta guía, ¡puedes apoyarla económicamente!

La guía se publica como un documento Free/*Libre* y es gratuita. Pero no se materializa en un espacio vacío; se necesita tiempo para escribir. Al donar, demuestras tu aprecio por este trabajo y ayudas a su desarrollo futuro.

Hay un botón de donación disponible en
`https://people.gnome.org/~swilmet/glib-gtk-book/`

¡Gracias!

1.3. Lista de tareas para este libro y una actualización rápida de 2019

- Hacer que el capítulo **??** (sobre GTK) sea más fácil de entender y adaptar el resto del documento para integrar mejor el capítulo;
- GTK + ha cambiado de nombre a GTK, el signo más se ha eliminado. Adaptar el texto en consecuencia;
- Ya no se recomienda Jhbuild, en su lugar se recomienda Flatpak[1] y BuildStream[2];
- Hablar sobre las especificaciones de freedesktop.org[3];
- La mayoría de los módulos GNOME activos se han migrado ahora al sistema de compilación Meson (en lugar de Autotools);
- Escribir un capítulo sobre el desarrollo de bibliotecas C/GObject;
- Hablar sobre el lenguaje de programación Rust.

1.4. ¿Qué es GLib y GTK?

En términos generales, GLib es un conjunto de bibliotecas: GLib core, GObject y GIO. Esas tres bibliotecas se desarrollan en el mismo repositorio de Git llamado *glib*, por lo que cuando se hace referencia a "GLib", puede significar "GLib core" o el conjunto más amplio que incluye también GObject y GIO.

GLib core proporciona manejo de estructura de datos para C (listas enlazadas, árboles, tablas hash, ...), envoltorios de portabilidad, un bucle de eventos, hilos, carga dinámica de módulos y muchas funciones de utilidad.

GObject – que depende del núcleo GLib – simplifica la programación orientada a objetos y los paradigmas de programación dirigida por eventos en C. La programación dirigida por eventos no solo es útil para interfaces gráficas de usuario (con eventos de usuario como pulsaciones de teclas y clics del mouse) , pero también para demonios que responden a cambios de hardware (una memoria USB insertada, un segundo monitor conectado, una impresora con poco papel), o software que escucha conexiones de red o mensajes de otros procesos, etc.

GIO – que depende de GLib core y GObject – proporciona API de alto nivel para entrada / salida: lectura de un archivo local, un archivo remoto, un flujo de red, comunicación entre procesos con D-Bus y muchos más.

Las bibliotecas GLib se pueden utilizar para escribir servicios del sistema operativo, bibliotecas, utilidades de línea de comandos y demás. GLib ofrece API de mayor nivel que el estándar POSIX; por lo tanto, es más cómodo escribir un programa en C con GLib.

GTK es un conjunto de herramientas de widgets basado en GLib que se puede utilizar para desarrollar aplicaciones con una interfaz gráfica de usuario (GUI). Un "widget" es un elemento de la GUI, por ejemplo, un botón, un texto, un menú, etc. Y hay algunos tipos especiales de widgets que se denominan "containers",

[1] `https://flatpak.org/`
[2] `https://buildstream.build/`
[3] `https://www.freedesktop.org/`

que pueden contener otros widgets, para ensamblar los elementos en una ventana. GTK proporciona una amplia gama de widgets y contenedores.

La primera versión de GTK +, o GIMP Tool Kit[4], fue escrita principalmente por Peter Mattis en 1996 para el GIMP (Programa de manipulación de imágenes GNU), pero se ha convertido rápidamente en una biblioteca de uso general. El "+" se ha agregado más tarde para distinguir entre la versión original y una nueva versión que agregó características orientadas a objetos. GLib comenzó como parte de GTK +, pero ahora es una biblioteca independiente.

Las API GLib y GTK están documentadas con GTK-Doc. Los comentarios especiales están escritos en el código fuente y GTK-Doc extrae esos comentarios para generar páginas HTML.

Aunque GLib y GTK están escritos en C, los enlaces de lenguaje están disponibles para JavaScript, Python, Perl y muchos otros lenguajes de programación. Al principio, se crearon enlaces manuales, que debían actualizarse cada vez que cambiaba la API de la biblioteca. Hoy en día, los enlaces de lenguaje son genéricos y, por lo tanto, se actualizan automáticamente cuando, por ejemplo, se agregan nuevas funciones, esto es gracias a GObject Introspection . Se agregan anotaciones especiales a los comentarios de GTK-Doc, para exponer más información de la que puede proporcionar la sintaxis de C, por ejemplo, sobre la transferencia de propiedad de contenido asignado dinámicamente[5]. Además, las anotaciones también son útiles para el programador en C porque es una forma buena y concisa de documentar ciertos aspectos recurrentes de la API.

GLib y GTK son parte del Proyecto GNU, cuyo objetivo general es desarrollar un sistema operativo libre (llamado GNU) más aplicaciones que lo acompañen. GNU significa "GNU's Not Unix", una forma divertida de decir que el sistema operativo GNU es compatible con Unix. Puede obtener más información sobre GNU en `https://www.gnu.org`.

El sitio web de GLib/GTK es: `http://www.gtk.org`

1.5. El escritorio GNOME

Un proyecto importante para GLib y GTK es GNOME. GNOME, que también forma parte de GNU, es un entorno de escritorio libre iniciado en 1997 por Miguel de Icaza y Federico Mena-Quintero. GNOME hace un uso extensivo de GTK, y el último ahora es desarrollado principalmente por desarrolladores de GNOME.

"GNOME" es en realidad un acrónimo: GNU Network Object Model Environment[6]. Originalmente, el proyecto tenía la intención de crear un marco para objetos de aplicación, similar a las tecnologías OLE y COM de Microsoft. Sin embargo, el alcance del proyecto se expandió rápidamente; quedó claro se requería un trabajo preliminar sustancial antes de que la parte del nombre de "network object" pudiera convertirse en realidad. Las versiones antiguas de GNOME incluían una arquitectura de incrustación de objetos llamada Bonobo, y GNOME

[4]El nombre " The GIMP Tool Kit " ahora rara vez se usa, hoy se conoce más comúnmente como GTK para abreviar.

[5]Por ejemplo, si necesita liberar el valor de retorno.

[6]En cuanto a GTK, el nombre completo de GNOME rara vez se usa y no refleja la realidad actual.

1.0 incluía un ORB CORBA rápido y ligero llamado ORBit. Bonobo ha sido reemplazado por D-Bus un sistema de comunicación entre procesos.

GNOME tiene dos caras importantes. Desde la perspectiva del usuario, es un entorno de escritorio integrado y una suite de aplicaciones. Desde la perspectiva del programador, es un marco de desarrollo de aplicaciones (compuesto por numerosas bibliotecas útiles que se basan en GLib y GTK). Las aplicaciones escritas con las bibliotecas de GNOME funcionan bien incluso si el usuario no está ejecutando el entorno de escritorio, pero se integran bien con el escritorio de GNOME si está disponible.

El entorno de escritorio incluye un "shell" para cambiar de tareas y ejecutar programas, un "centro de control" para la configuración, muchas aplicaciones como un administrador de archivos, un navegador web, un reproductor de películas, etc. línea de comando tradicional de Unix detrás de una interfaz gráfica fácil de usar.

El marco de desarrollo de GNOME permite escribir aplicaciones interoperables, coherentes y fáciles de usar. Los diseñadores de sistemas de ventanas como X11 o Wayland tomaron la decisión deliberada de no imponer ninguna política de interfaz de usuario a los desarrolladores; GNOME agrega una "capa de política", creando una apariencia coherente. Las aplicaciones GNOME terminadas funcionan bien con el escritorio GNOME, pero también se pueden usar de forma "independiente" – los usuarios solo necesitan instalar las bibliotecas compartidas de GNOME. Una aplicación GNOME no está vinculada a un sistema de ventanas específico, GTK proporciona backends para X Window System, Wayland, Mac OS X, Windows e incluso para un navegador web.

En el momento de escribir este artículo, hay nuevas versiones estables de GLib, GTK y GNOME cada seis meses, alrededor de marzo y septiembre. Un número de versión tiene la forma "`major.minor.micro`", donde "`minor`" es incluso para versiones estables e isodd para versiones inestables. Por ejemplo, las versiones 3.18. * Son estables, pero las versiones 3.19. * Son inestables. Una nueva versión micro estable (por ejemplo, 3.18.0 → 3.18.1) no agrega nuevas funciones, solo actualizaciones de traducción, corrección de errores y mejoras de rendimiento. Los componentes de GNOME deben instalarse con las mismas versiones, junto con la versión de GTK y GLib lanzada al mismo tiempo; por ejemplo, es una mala idea ejecutar un demonio GNOME en la versión 3.18 con el centro de control en la versión 3.16. En el momento de escribir este artículo, las últimas versiones estables son: GLib 2.46, GTK 3.18 y GNOME 3.18, todas lanzadas al mismo tiempo en septiembre de 2015. Para una biblioteca, un nuevo número de versión principal generalmente significa que ha habido una interrupción de la API, pero afortunadamente la versión principal anterior. Las versiones se pueden instalar en paralelo con la nueva versión. Durante un ciclo de desarrollo (por ejemplo, 3.19), no hay garantías de estabilidad API para *nuevas* funciones; pero al ser uno de los primeros en adoptarlo, sus comentarios son útiles para descubrir más rápidamente fallas y errores de diseño.

Más información sobre GNOME: `https://www.gnome.org/`

1.6. Prerrequisitos

Este libro asume que ya tiene algo de práctica en programación. A continuación, se muestra una lista de requisitos previos recomendados, con referencias de libros.

- Este texto supone que ya conoces el lenguaje C. El libro de referencia es *El lenguaje de programación C*, de Brian Kernighan y Den-nis Ritchie [**?**].
- La programación orientada a objetos (OOP) también es necesaria para aprender GObject. Debe estar familiarizado con conceptos como herencia, una interfaz, un método virtual o polimorfismo. Un buen libro, con más de sesenta pautas, es *Heurística de diseño orientado a objetos*, de Arthur Riel [**?**].
- Es útil haber leído un libro sobre estructuras de datos y algoritmos, pero puede aprenderlo en paralelo. Un libro recomendado es *The Algorithm Design Manual*, de Steven Skiena [**?**].
- Si desea desarrollar su software en un sistema similar a Unix, otro requisito previo es saber cómo funciona Unix y estar familiarizado con la línea de comandos, un poco de scripts de shell y cómo escribir un Makefile. Un posible libro es *UNIX for the Impatient*, de Paul Abrahams [**?**].
- No es estrictamente necesario, pero se recomienda encarecidamente utilizar un sistema de control de versiones como Git. Un buen libro es *Pro Git*, de Scott Chacon [**?**].

1.7. ¿Por qué y cuándo se usa el lenguaje C?

Las bibliotecas GLib y GTK pueden ser utilizadas por otros lenguajes de programación además de C. Gracias a GObject Introspection, los enlaces automáticos están disponibles para una variedad de idiomas para todas las bibliotecas basadas en GObject. Los enlaces oficiales de GNOME están disponibles para los siguientes lenguajes: C ++, JavaScript, Perl, Python y Vala[7]. Vala es un nuevo lenguaje de programación basado en GObject que integra las peculiaridades de GObject directamente en su sintaxis similar a C#. Más allá de los enlaces oficiales de GNOME, GLib y GTK se pueden usar en más de una docena de lenguajes de programación, con un nivel de soporte variable. Entonces, ¿por qué y cuándo elegir el lenguaje C? Para escribir un demonio en un sistema Unix, C es el idioma *predeterminado*. Pero es menos obvio para una aplicación. Para responder a la pregunta, veamos primero cómo estructurar el código base de una aplicación.

1.8. Separación de backend del frontend

Una buena práctica es separar la interfaz gráfica de usuario del resto de la aplicación. Por diversas razones, la interfaz gráfica de una aplicación tiende a ser una pieza de software excepcionalmente volátil y en constante cambio. Es el foco de la mayoría de las solicitudes de cambio de los usuarios. Es difícil planificar y ejecutar bien la primera vez; a menudo descubrirá que algún aspecto es desagradable de usar solo después de haberlo escrito. A veces es deseable tener varias interfaces de usuario diferentes, por ejemplo, una versión de línea de comandos o una interfaz basada en web.

En términos prácticos, esto significa que cualquier aplicación grande debe tener una separación radical entre sus diversos *frontends* o interfaces y el *backend*. El backend debe contener todas las "partes duras": sus algoritmos y estructuras de

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

datos, el trabajo real realizado por la aplicación. Piense en ello como un "modelo" abstracto que se muestra y manipula el usuario.

Cada interfaz debe ser una " vista " y un " controlador ". Como una " vista ", la interfaz debe anotar cualquier cambio en el backend y cambiar la pantalla en consecuencia. Como un " controlador ", la interfaz debe permitir al usuario transmitir solicitudes de cambio al backend (define cómo las manipulaciones de la interfaz se traducen en cambios en el modelo).

Hay muchas formas de disciplinarse para mantener su aplicación separada. Un par de ideas útiles:

- Escriba el backend como una biblioteca. Al principio, la biblioteca puede ser interna a la aplicación y estar vinculada estáticamente, sin garantías de estabilidad API/ABI. Cuando el proyecto crezca, y si el código es útil para otros programas, puede convertir fácilmente su backend en una biblioteca compartida.
- Escriba al menos dos interfaces desde el principio; uno o ambos pueden ser prototipos feos, solo desea tener una idea de cómo estructurar el backend. Recuerde, las interfaces deben ser fáciles; el backend tiene las partes difíciles.

El lenguaje C es una buena opción para la parte de backend de una aplicación. Al utilizar GObject y GObject Introspection, su biblioteca estará disponible para otros proyectos escritos en varios lenguajes de programación. Por otro lado, una biblioteca de Python o JavaScript no se puede utilizar en otros lenguajes. Para las interfaces, un idioma de nivel superior puede ser más conveniente, dependiendo de los idiomas con los que ya domine.

1.8.1. Otros aspectos a tener en cuenta

Si tiene dudas sobre el idioma a elegir, aquí hay otros aspectos a tener en cuenta. Tenga en cuenta que este texto está un poco sesgado ya que se eligió el lenguaje C.

C es un lenguaje de tipo estático: los tipos de variables y los prototipos de funciones en un programa se conocen en el momento de la compilación. El compilador descubre muchos errores triviales, como un error tipográfico en el nombre de una función. El compilador también es de gran ayuda cuando se hacen refactorizaciones de código, lo cual es esencial para el mantenimiento a largo plazo de un programa. Por ejemplo, cuando divide una clase en dos, si el código que usa la clase inicial no se actualiza correctamente, el compilador se lo informará amablemente[8]. Con el desarrollo basado en pruebas (TDD), y escribiendo pruebas unitarias para *todo*, también es factible escribir una enorme base de código en un lenguaje de tipo dinámico como Python. Con una muy buena cobertura de código, las pruebas unitarias también detectarán errores al refactorizar el código. Pero las pruebas unitarias pueden ser mucho más lentas de ejecutar que compilar el código, ya que también prueba el comportamiento del programa. Por lo tanto, puede que no sea conveniente ejecutar todas las pruebas unitarias al realizar refactorizaciones de código. ¡Por supuesto, escribir pruebas unitarias también es una buena práctica para una base de código C! Sin embargo, para la parte GUI

[8] Bueno, *amablemente* quizás no sea la mejor descripción, arrojar un montón de errores está más cerca de la realidad.

del código, escribir pruebas unitarias a menudo no es una tarea de alta prioridad si la aplicación está bien probada por sus desarrolladores.

C es un lenguaje escrito explícitamente: los tipos de variables son visibles en el código. Es una forma de auto-documentar el código; por lo general, no es necesario agregar comentarios para explicar qué contienen las variables. Conocer el tipo de variable es importante para comprender el código, saber qué representa la variable y qué funciones se pueden llamar sobre ella. En un asunto relacionado, el objeto *self* se pasa explícitamente como un argumento de función. Por lo tanto, cuando se accede a un atributo a través del puntero *self*, se sabe de dónde procede el atributo. Algunos lenguajes orientados a objetos tienen *esta* palabra clave para ese propósito, pero a veces es opcional como en C ++ o Java. En este último caso, una función útil del editor de texto es resaltar atributos de manera diferente, por lo que incluso cuando no se usa *esta* palabra clave, usted sabe que es un atributo y no una variable local. Con el objeto *self* pasado como argumento, no hay posibles confusiones.

El lenguaje C tiene una *cadena de herramientas* muy buena: compiladores estables (GCC, Clang,...), Editores de texto (Vim, Emacs,...), Depuradores (GDB, Valgrind,...), Herramientas de análisis estático, ...

Para algunos programas, un recolector de basura no es apropiado porque pausa el programa regularmente para liberar la memoria no utilizada. Para secciones de código críticas, como animaciones en tiempo real, no es conveniente pausar el programa (un recolector de basura a veces puede ejecutarse durante varios segundos). En este caso, la gestión manual de la memoria como en C es una solución.

Menos importante, pero útil; la verbosidad de C en combinación con las convenciones GLib / GTK tiene una ventaja: el código se puede buscar fácilmente con un comando como `grep`. Por ejemplo, la función `gtk_widget_show()` contiene el espacio de nombres (`gtk`), la clase (`widget`) y el método (`show`). Con un lenguaje orientado a objetos, la sintaxis es generalmente `object.show()`. Si se busca "show" en el código, probablemente habrá más falsos positivos, por lo que se necesita una herramienta más inteligente. Otra ventaja es que conocer el espacio de nombres y la clase de un método puede ser útil al leer el código, es otra forma de auto-documentación.

Más importante aún, la documentación de la API GLib / GTK está escrita principalmente para el lenguaje C. No es conveniente leer la documentación de C mientras se programa en otro idioma. Algunas herramientas están actualmente en desarrollo para generar la documentación de la API para otros lenguajes de destino, por lo que es de esperar que en el futuro ya no sea un problema.

GLib / GTK están escritos en C. Entonces, cuando se programa en C, no hay capa adicional. Una capa adicional es potencialmente una fuente de errores adicionales y cargas de mantenimiento. Además, usar el lenguaje C probablemente sea mejor para propósitos pedagógicos. Un lenguaje de nivel superior puede ocultar algunos detalles sobre GLib / GTK. Por lo tanto, el código es más corto, pero cuando tiene un problema, debe comprender no solo cómo funciona la función de la biblioteca, sino también cómo funciona el enlace del idioma.

Dicho esto, si (1) no se siente cómodo en C, (2) ya domina un lenguaje de nivel superior con compatibilidad con GObject Introspection, (3) planea escribir solo una pequeña aplicación o complemento, luego elige el lenguaje de nivel superior

tiene mucho sentido.

1.9. Ruta de aprendizaje

Normalmente, esta sección debería llamarse " Estructura del libro ", pero como puede ver, el libro está lejos de estar terminado, por lo que la sección se llama " Ruta de aprendizaje ".

El camino de aprendizaje lógico es:

1. Los fundamentos del núcleo GLib;
2. Programación orientada a objetos en C y los conceptos básicos de GObject;
3. GTK y GIO en paralelo.

Dado que GTK se basa en GLib y GObject, es mejor comprender primero los conceptos básicos de esas dos bibliotecas. Algunos tutoriales se sumergen directamente en GTK, por lo que después de un corto período de tiempo puede mostrar una ventana con texto y tres botones; es divertido, pero conocer GLib y GObject no es un lujo si quiere comprender lo que está haciendo, y una aplicación GTK realista utiliza ampliamente las bibliotecas GLib. GTK y GIO se pueden aprender en paralelo — una vez que comience a usar GTK, verá que algunas partes que no son GUI están basadas en GIO.

Así que este libro comienza con la biblioteca principal GLib (parte I p. 11), luego presenta la Programación Orientada a Objetos en C (parte **??** p. **??**) seguida de un capítulo de Lecturas Adicionales (p. **??**)

1.10. El entorno de desarrollo

Esta sección describe el entorno de desarrollo que se usa normalmente al programar con GLib y GTK en un sistema Unix.

En una distribución GNU / Linux, a menudo se puede instalar un solo paquete o grupo para obtener un entorno de desarrollo C completo, que incluye, entre otros:

- un compilador compatible con C89, GCC por ejemplo;
- el depurador GNU GDB;
- GNU Make;
- los Autotools (Autoconf, Automake y Libtool);
- las páginas de manual de: el kernel de Linux y glibc [9].

Para utilizar GLib y GTK como desarrollador, existen varias soluciones:

- Los encabezados y la documentación se pueden instalar con el administrador de paquetes. El nombre de los paquetes suele terminar con uno de los siguientes sufijos: `-devel`, `-dev` o `-doc`. Por ejemplo `glib2-devel` y `glib2-doc` en Fedora.
- Las últimas versiones de GLib y GTK se pueden instalar con Jhbuild: `https://wiki.gnome.org/Projects/Jhbuild`

[9] No confunda la biblioteca GNU C (glibc) con GLib. El primero es de nivel inferior.

Para leer la documentación de la API de GLib y GTK, Devhelp es una aplicación útil, si ha instalado el paquete -dev o -doc. Para el editor de texto o IDE, hay muchas opciones (y una fuente de muchos trolls): Vim, Emacs, gedit, Anjuta, MonoDevelop / Xamarin Studio, Geany,... Un prometedor IDE especializado para GNOME es Builder, actualmente en desarrollo. Para crear una GUI con GTK, puede escribir directamente el código para hacerlo o puede usar Glade para diseñar la GUI gráficamente. Finalmente, GTK-Doc se usa para escribir documentación de API y agregar las anotaciones de GObject Introspection.

Cuando utilice GLib o GTK, preste atención a no utilizar API obsoletas para el código recién escrito. Asegúrese de leer la documentación más reciente. También están disponibles en línea en:
`https://developer.gnome.org/`

1.11. Agradecimientos

Gracias a: Christian Stadelmann, Errol van de l'Isle, Andrew Colin Kissa y Link Dupont.

Parte I

GLib, la biblioteca principal

Capítulo 2

GLib, la biblioteca principal

GLib es la biblioteca central de bajo nivel que forma la base para proyectos como GTK y GNOME. Proporciona estructuras de datos, funciones de utilidad, envoltorios de portabilidad y otras funciones esenciales, como un bucle de eventos e hilos. GLib está disponible en la mayoría de los sistemas similares a Unix y Windows.

Este capítulo cubre algunas de las funciones más utilizadas. GLib es simple y los conceptos son familiares; así que nos moveremos rápidamente. Para obtener una cobertura más completa de GLib, consulte la última documentación de la API que viene con la biblioteca (para el entorno de desarrollo, consulte la sección 1.10 en la p. 9). Por cierto: si tiene preguntas muy específicas sobre la implementación, no tema mirar el código fuente. Normalmente, la documentación contiene suficiente información, pero si encuentra un detalle faltante, por favor presente un error (por supuesto, lo mejor sería con un parche proporcionado).

Las diversas instalaciones de GLib están destinadas a tener una interfaz coherente; el estilo de codificación está orientado a semiobjetos, y los identificadores tienen el prefijo " g " para crear una especie de espacio de nombres.

GLib tiene algunos encabezados de nivel superior:

- `glib.h`, el encabezado principal;
- `gmodule.h` para carga dinámica de módulos;
- `glib-unix.h` para API específicas de Unix;
- `glib/gi18n.h` y `glib/gi18n-lib.h` para la internacionalización;
- `glib/gprintf.h` y `glib/gstdio.h` para evitar tirar de todo stdio.

Nota: en lugar de reinventar la rueda, este capítulo se basa en gran medida en el capítulo correspondiente del libro *GTK+/Gnome Application Development* de Havoc Pennington, con licencia de Open Publication License (consulte la sección 1.1 p. 2). GLib tiene una API muy estable. A pesar de que el libro de Havoc Pennington fue escrito en 1999 (para GLib 1.2), solo se requirieron algunas actualizaciones para adaptarse a las últimas versiones de GLib (versión 2.42 en el momento de escribir este artículo)

```
#include <glib.h>

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

Listing 2.1: Familiar C Macros

2.1. Lo esencial

GLib proporciona sustitutos para muchas construcciones de lenguaje C estándar y de uso común. Esta sección describe las definiciones de tipos fundamentales, macros, rutinas de asignación de memoria y funciones de utilidad de cadena de GLib.

2.1.1. Definiciones de tipo

En lugar de utilizar los tipos estándar de C (`int`, `long`, etc.), GLib define los suyos propios. Estos sirven para una variedad de propósitos. Por ejemplo, se garantiza que `gint32` tiene 32 bits de ancho, algo que ningún tipo C89 estándar puede garantizar. `guint` es simplemente más fácil de escribir que `unsigned`. Algunos de los typedefs existen solo por coherencia; por ejemplo, `gchar` siempre es equivalente al `char` estándar.

Los tipos primitivos más importantes definidos por GLib:

- `gint8`, `guint8`, `gint16`, `guint16`, `gint32`, `guint32`, `gint64`, `guint64` — le dará números enteros de un tamaño garantizado. (Si no es obvio, los tipos `guint` son unsigned, los tipos de gint son signed).
- `gboolean` es útil para hacer su código más legible, ya que C89 no tiene un tipo `bool`.
- `gchar`, `gshort`, `glong`, `gint`, `gfloat`, `gdouble` son puramente cosméticos.
- `gpointer` puede ser más conveniente de escribir que `void *`. `gconstpointer` le da `const void *`. (`const gpointer` no hará lo que normalmente quiere; dedique un tiempo a leer un buen libro sobre C si no ve por qué).
- `gsize` es un tipo entero sin signo que puede contener el resultado del operador `sizeof`.

2.1.2. Macros de uso frecuente

GLib define una serie de macros familiares que se utilizan en muchos programas C, que se muestran en el Listado 2.1. Todos estos deben ser autoexplicativos. `MIN()`/`MAX()` devuelven el menor o mayor de sus argumentos. `ABS()` devuelve el valor absoluto de su argumento. `CLAMP(x, low, high)` significa `x`, a menos que `x` esté fuera del rango [`low`, `high`]; si `x` está por debajo del rango, se devuelve `low`; si `x` está por encima del rango, se devuelve `high`. Además de las macros que se muestran en el Listado 2.1, `TRUE`/`FALSE`/`NULL` se definen como los habituales `1`/`0`/`((void *)0)`.

También hay muchas macros exclusivas de GLib, como las conversiones portátiles `gpointer-to-gint` y `gpointer-to-guint` que se muestran en el Listado 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

La mayoría de las estructuras de datos de GLib están diseñadas para almacenar un `gpointer`. Si desea almacenar punteros a objetos asignados dinámicamente, esto es lo correcto. Sin embargo, a veces desea almacenar una lista simple de números enteros sin tener que asignarlos dinámicamente. Aunque el estándar C no lo garantiza estrictamente, es posible almacenar un `gint` o `guint` en una variable `gpointer` en la amplia gama de plataformas a las que GLib ha sido portado; en algunos casos, se requiere un yeso intermedio. Las macros en Listado 2.2 abstraen la presencia del elenco.

He aquí un ejemplo:

```
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));
```

Pero ten cuidado; estas macros le permiten almacenar un entero en un puntero, pero almacenar un puntero en un entero *no* funcionará. Para hacerlo de forma portátil, debe almacenar el puntero en un `long`. (Sin embargo, sin duda es una mala idea hacerlo).

2.1.3. Macros de depuración

GLib tiene un buen conjunto de macros que puede usar para hacer cumplir invariantes y condiciones previas en su código. GTK los usa generosamente, una de las razones por las que es tan estable y fácil de usar. Todos desaparecen cuando define `G_DISABLE_CHECKS` o `G_DISABLE_ASSERT`, por lo que no hay penalización de rendimiento en el código de producción. Usarlos generosamente es una muy, muy buena idea. Encontrará errores mucho más rápido si lo hace. Incluso puede agregar afirmaciones y verificaciones cada vez que encuentre un error para asegurarse de que el error no vuelva a aparecer en versiones futuras; esto complementa un conjunto de regresión. Las comprobaciones son especialmente útiles cuando el código que está escribiendo será utilizado como caja negra por otros programadores; los usuarios sabrán inmediatamente cuándo y cómo han hecho un mal uso de su código.

Por supuesto, debe tener mucho cuidado de asegurarse de que su código no dependa sutilmente de declaraciones de solo depuración para funcionar correctamente. Las declaraciones que desaparecerán en el código de producción *nunca* deberían tener efectos secundarios.

El Listado 2.3 muestra las verificaciones de condiciones previas de GLib. `g_return_if_fail()` imprime una advertencia y regresa inmediatamente de la función actual si `condition` es `FALSE`. `g_return_val_if_fail()` es similar pero le permite devolver algún `return_value`.

```
#include <glib.h>

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

Listing 2.3: Precondition Checks

```
#include <glib.h>

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

Listing 2.4: Assertions

Estos macros son increíblemente útiles, si las usa libremente, especialmente en combinación con la verificación de tipo en tiempo de ejecución de GObject, reducirá a la mitad el tiempo que dedica a buscar punteros incorrectos y errores tipográficos.

Usar estas funciones es simple; aquí hay un ejemplo de la implementación de la tabla hash GLib:

```
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);
    }
}
```

Sin las comprobaciones, pasar `NULL` como parámetro a esta función resultaría en una misteriosa falla de segmentación. La persona que usa la biblioteca tendría que averiguar dónde ocurrió el error con un depurador y tal vez incluso indagar en el código GLib para ver qué estaba mal. Con las comprobaciones, obtendrán un bonito mensaje de error que les indicará que los argumentos `NULL` no están permitidos.

GLib también tiene macros de aserción más tradicionales, que se muestran en el Listado 2.4. `g_assert()` es básicamente idéntico a `assert()`, pero responde a `G_DISABLE_ASSERT` y se comporta consistentemente en todas las plataformas. También se proporciona `g_assert_not_reached()`; esta es una afirmación que siempre falla. Las afirmaciones llaman a `abort()` para salir del programa y (si su entorno lo admite) descargan un archivo central con fines de depuración.

Las afirmaciones fatales deben usarse para verificar la *consistencia interna* de una función o biblioteca, mientras que `g_return_if_fail()` está destinado a garantizar que se pasen valores cuerdos a las interfaces públicas de un módulo de programa. Es decir, si una aserción falla, normalmente busca un error en el módulo que contiene la aserción; Si falla una comprobación de `g_return_if_fail()`, normalmente busca el error en el código que invoca el módulo.

Este código del módulo de cálculos calendáricos de GLib muestra la diferencia:

```
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;
}
```

La verificación de condiciones previas al principio asegura que el usuario pasa en valores razonables para el día, mes y año; la afirmación al final asegura que GLib construyó un objeto sano, dados valores cuerdos.

`g_assert_not_reached()` debe usarse para marcar situaciones " imposibles "; un uso común es detectar declaraciones de cambio que no manejan todos los valores posibles de una enumeración:

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

  case FOO_TWO:
    break;

  default:
    g_assert_not_reached ();
  }
```

Todas las macros de depuración imprimen una advertencia utilizando la función `g_log()` de GLib, lo que significa que la advertencia incluye el nombre de la aplicación o biblioteca de origen y, opcionalmente, puede instalar una rutina de impresión de advertencias de reemplazo. Por ejemplo, puede enviar todas las advertencias a un cuadro de diálogo o archivo de registro en lugar de imprimirlas en la consola.

```
#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

2.1.4. Memoria

GLib envuelve el estándar `malloc()` y `free()` con sus propias variantes `g_`, `g_malloc()` y `g_free()`, que se muestran en el Listado 2.5. Estos son agradables de varias maneras pequeñas:

- `g_malloc()` siempre devuelve un `gpointer`, nunca un `char *`, por lo que no es necesario emitir el valor de retorno [1].
- `g_malloc()` aborta el programa si el `malloc()` subyacente falla, por lo que no tiene que buscar un valor devuelto `NULL`.
- `g_malloc()` maneja con gracia un `size` de `0`, devolviendo `NULL`.
- `g_free()` ignorará cualquier puntero `NULL` que le pase.

Es importante hacer coincidir `g_malloc()` con `g_free()`, plain `malloc()` con `free()` y (si estás usando C ++) `new` con `delete`. De lo contrario, pueden suceder cosas malas, ya que estos asignadores pueden usar diferentes grupos de memoria (y `new`/`delete` llama a constructores y destructores).

Por supuesto, hay un `g_realloc()` equivalente a `realloc()`. También hay un conveniente `g_malloc0()` que llena la memoria asignada con ceros, y `g_memdup()` que devuelve una copia de `n_bytes` bytes comenzando en `mem`. `g_realloc()` y `g_malloc0()` aceptarán ambos un tamaño de 0, por coherencia con `g_malloc()`. Sin embargo, `g_memdup()` no lo hará.

Si no es obvio: `g_malloc0()` llena la memoria sin procesar con bits no configurados, no el valor 0 para cualquier tipo que pretenda poner allí. De vez en cuando, alguien espera obtener una matriz de números de coma flotante inicializados en 0.0; *no* se garantiza que funcione de forma portátil.

Por último, existen macros de asignación con reconocimiento de tipos, que se muestran en el Listado 2.6. El argumento `type` para cada uno de estos es el nombre de un tipo, y el argumento `count` es el número de bloques de tamaño `type` a asignar. Estas macros le ahorran algo de escritura y multiplicación y, por lo tanto, son menos propensas a errores. Se lanzan automáticamente al tipo de puntero de destino, por lo que intentar asignar la memoria asignada al tipo de puntero incorrecto debería activar una advertencia del compilador. (Si tiene las advertencias activadas, ¡como debería hacerlo un programador responsable!)

[1] Antes del estándar ANSI / ISO C, el `void *` el tipo de puntero genérico no existía y `malloc()` devolvió un valor de `char *`. Actualmente, `malloc()` devuelve un tipo `void *` — que es lo mismo que `gpointer`— y `void *` permite conversiones de puntero implícitas en C. Lanzando el valor de retorno de `malloc()` es necesario si: el desarrollador quiere admitir compiladores antiguos; o si el desarrollador piensa que una conversión explícita aclara el código; o si se usa un compilador de C++, porque en C++ se requiere una conversión del tipo `void *`.

```
#include <glib.h>

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

Listing 2.6: Allocation macros

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

Listing 2.7: Portability Wrapper

2.1.5. Manejo de string

GLib proporciona una serie de funciones para el manejo de cadenas; algunos son exclusivos de GLib y otros resuelven problemas de portabilidad. Todos interoperan muy bien con las rutinas de asignación de memoria GLib.

Para aquellos interesados en una cadena mejor que `gchar *`, también hay un tipo `GString`. No se trata en este libro; consulte la documentación de la API para obtener más información.

El listado 2.7 muestra un sustituto que GLib proporciona para la función `snprintf()`. `g_snprintf()` envuelve el `snprintf()` nativo en las plataformas que lo tienen y proporciona una implementación en las que no lo tienen.

Preste atención a no usar la función `sprintf()` que causa fallas, crea agujeros de seguridad y generalmente es maligna. Al usar `g_snprintf()` o `g_strdup_printf()` relativamente seguros (ver más abajo), puedes despedirte de `sprintf()` para siempre.

El listado 2.8 muestra la amplia gama de funciones de GLib para asignar cadenas. Como era de esperar, `g_strdup()` y `g_strndup()` producen una copia asignada de `str` o los primeros `n` caracteres de `str`. Para mantener la coherencia con las funciones de asignación de memoria GLib, devuelven `NULL` si se les pasa un puntero `NULL`. Las variantes `printf()` devuelven una cadena formateada. `g_strnfill()` devuelve una cadena de tamaño `length` rellena con `fill_char`.

`g_strdup_printf()` merece una mención especial; es una forma más sencilla de manejar este código común:

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

En su lugar, podría decir esto y evitar tener que averiguar la longitud adecuada

```
#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

```
#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

del búfer para arrancar:

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

Las funciones del Listado 2.9 modifican una cadena en el lugar: `g_strchug ()` y `g_strchomp()` "chug" la cadena (elimina los espacios iniciales), o "chomp" (eliminar los espacios finales). Esas dos funciones devuelven la cadena, además de modificarla en el lugar; en algunos casos, puede ser conveniente utilizar el valor de retorno. Hay una macro, `g_strstrip()`, que combina ambas funciones para eliminar los espacios iniciales y finales.

El listado 2.10 muestra algunas funciones semi-estándar más que envuelve GLib. `g_strtod` es como `strtod()` – convierte la cadena `nptr` en un double – con la excepción de que también intentará convertir el double en la configuración local de `"C"` si no puede convertirlo en la configuración local predeterminada del usuario. `* endptr` se establece en el primer carácter no convertido, es decir, cualquier texto después de la representación numérica. Si la conversión falla, `*endptr` se establece en `nptr`. `endptr` puede ser `NULL`, lo que hace que se ignore.

`g_strerror()` y `g_strsignal()` son como sus equivalentes no `g_`, pero portátiles. (Devuelven una representación de cadena para un `errno` o un número de señal).

GLib proporciona algunas funciones convenientes para concatenar cadenas, que se muestran en el Listado 2.11. `g_strconcat()` devuelve una cadena recién asignada creada concatenando cada una de las cadenas en la lista de argumentos. El último argumento debe ser `NULL`, por lo que `g_strconcat()` sabe cuándo detenerse. `g_strjoin()` es similar, pero `separator` se inserta entre cada cadena. Si `separator` es `NULL`, no se usa ningún separador.

Finalmente, el Listado 2.12 resume algunas rutinas que manipulan matrices de cadenas terminadas en `NULL`. `g_strsplit()` rompe `string` en cada `delimiter`,

```
#include <glib.h>

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

Listing 2.11: Concatenating Strings

```
#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

devolviendo una matriz recién asignada. `g_strjoinv()` concatena cada cadena en la matriz con un `separator` opcional, devolviendo una cadena asignada. `g_strfreev()` libera cada cadena en la matriz y luego la propia matriz.

2.2. Estructuras de datos

GLib implementa muchas estructuras de datos comunes, por lo que no tiene que reinventar la rueda cada vez que desee una lista vinculada. Esta sección cubre la implementación de GLib de listas enlazadas, árboles binarios ordenados, árboles N-arios y tablas hash.