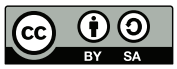
La plataforma de desarrollo GLib/GTK

Una guía de introducción

`Versión 0.9 alpha`

Sébastien Wilmetn | Gerson Benavides

0.1. Licencia



0.2. Prefacio

Este texto es una guía de inicio para dar comienzo con la plataforma de desarrollo GLib/GTK, haciendo uso del lenguaje C. El libro nace en 2016 como un proyecto personal de Sébastien Wilmetn, el cual desarrolla hasta el año 2019. El proyecto es luego retomado por Gerson Benavides a partir del año 2021.

En ocasiones se asumirá que el lector usa un sistema similar a Unix, no obstante, la mayor parte de lo escrito en este libro es aplicable a otros sistemas operativos. Tenga en cuenta que este libro está lejos de estar terminado, en este momento esta leyendo la versión 0.9 alpha. Si tiene algún comentario, no dude en ponerse en contacto escribiendo a gersonbdev@gmail.com. El código fuente y los diferentes formatos del libro se pueden encontrar en los siguientes enlaces:

- Español
  - Web: `https://gersonbdev.github.io/glib-gtk-libro/`
  - PDF: `https://raw.githubusercontent.com/gersonbdev/glib-gtk-libro/master/glib-gtk-libro.pdf`
  - Repositorio: `https://github.com/gersonbdev/glib-gtk-libro`
- Inglés
  - Web: `https://people.gnome.org/~swilmet/glib-gtk-book/`
  - PDF: `https://people.gnome.org/~swilmet/glib-gtk-dev-platform.pdf`
  - Repositorio: `https://github.com/swilmet/glib-gtk-book`

0.3. Agradecimientos

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

0.4. Novedades

- Se adoptaron algunos lineamientos de la normas APA.
- Mejoras en redacción y traducción.
- Se agrego resaltado de sintaxis.
- Se realizaron algunas tablas para presentar mejor el contenido.
- Se cambio Jhbuild por Build-Stream.
- Se agrego una imagen de la estructura de GLib (propuesta por el usuario DarkTrick).
- Se hablo un poco de Rust como alternativa a C.
- Varias actualizaciones en versiones de software.

0.5. Contribución

Si desea contribuir con el desarrollo de este libro puede apoyar con:

- **Soporte en redacción**: Escribiendo o revisando el texto, para hacerlo participe en el repositorio del libro (`https://github.com/gersonbdev/glib-gtk-libro`) dando su opinión o desarrollando las tareas propuestas (revise el archivo TODO).
- **Soporte financiero**: El libro se publica como un documento *libre* y es gratuito. Pero no se materializa en un espacio vacío, se necesita tiempo para escribir. Al donar, demuestra su aprecio por este trabajo y ayuda a su desarrollo futuro.

  Puede encontrar un botón de donación en:

  `https://gersonbdev.github.io/about/`

  ¡Gracias!

Índice general

Capítulo 1

Introducción

1.1. ¿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.

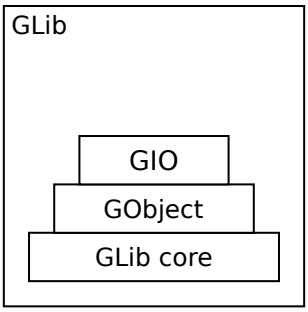


Figura 1.1: Estructura de la biblioteca GLib.

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", 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[1], fue escrita principalmente por Peter Mattis en 1996 para GIMP (Programa de manipulación de imágenes GNU), pero se ha convertido rápidamente en una biblioteca de uso general. Una vez el proyecto se movió fuera del árbol de fuentes de GIMP para distinguir entre la versión original y una nueva versión que agregó características orientadas a objetos se agrego al nombre un "+" denominándose la biblioteca como GTK+ (actualmente este nombre esta en desuso y se conoce simplemente como GTK). 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

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

de contenido asignado dinámicamente[2]. 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.

En el momento de escribir este artículo, hay nuevas versiones estables de GLib y GTK cada seis meses, alrededor de marzo y septiembre. Un número de versión tiene la forma "`major.minor.micro`", donde "`minor`" designa ciclos estables si es par y ciclos de desarrollo (versiones inestables) si es impar. Por ejemplo, las versiones 4.4.* son estables, pero las versiones 4.5.* son inestables. Una nueva versión "`micro`" estable (por ejemplo, 4.4.0 → 4.4.1) no agrega nuevas funciones, solo actualizaciones de traducción, corrección de errores y mejoras de rendimiento. Para una biblioteca, un nuevo número de versión "`major`" generalmente significa que ha habido una ruptura de la API, pero afortunadamente las versiones principales anteriores se pueden instalar en paralelo con la nueva versión. Durante un ciclo de desarrollo (por ejemplo, 4.5), no hay garantías de estabilidad en la 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.

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`

1.2. 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 este último ahora es desarrollado principalmente por desarrolladores de GNOME.

"GNOME" es en realidad un acrónimo: GNU Network Object Model Environment[3]. 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

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

[3]Como con GTK, el nombre completo de GNOME rara vez se usa y no refleja la realidad actual.

antiguas de GNOME incluían una arquitectura de incrustación de objetos llamada Bonobo, y GNOME 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. Estos programas ocultan la línea de comandos 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.

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 41 con el centro de control en la versión 40. En el momento de escribir este libro, las últimas versiones estables son: GLib 2.70, GTK 4.4 y GNOME 41, todas lanzadas en el segundo semestre de 2021.

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

1.3. 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 conoce el lenguaje C. El libro de referencia es *The C Programming Language*, de Brian Kernighan y Dennis Ritchie [1].
- 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 [2].
- 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 [4].
- 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 [5].
- 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 [6].

1.4. ¿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 gran variedad de lenguajes, de manera que puedan ser usadas todas las bibliotecas basadas en GObject por estos. Los enlaces oficiales de GNOME están disponibles para los siguientes lenguajes [4]:

Lenguaje	**v3**	**v4**	**Enlace**
C++	☑	☑	`https://www.gtkmm.org/`
JavaScript	☑	☑	`https://gjs.guide/`
Python	☑	☑	`https://pygobject.readthedocs.io/`
Rust	☑	☑	`https://gtk-rs.org/`
Vala	☑	☑	`https://valadoc.org/`

Una buena alternativa a C es Vala, el cual es un lenguaje de programación que integra las peculiaridades de GObject directamente en su sintaxis similar a C#. De manera que todo el

[4] Aunque existen enlaces a más lenguajes, los expresados en la tabla son los más activos es sus repositorios y con una mayor comunidad.

código hecho en Vala es traducido a código en C, el cual hace uso de GObject directamente, esto puede resultar útil si desea código cercano a C pero haciendo uso de un lenguaje más moderno[5].

Otra alternativa importante a considerar si desea trabajar con GLib/GTK es el lenguaje Rust, un lenguaje moderno que genera programas eficientes en memoria, gracias a un verificador de prestamos que validan las referencias, por lo cual no necesita de un recolector de basura como otros lenguajes de programación para asegurar la seguridad de la memoria.

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?*.

Por ejemplo, para escribir un demonio en un sistema tipo Unix, C es el lenguaje *predeterminado*. Pero es menos obvio que lenguaje usar para una aplicación. Para responder a la pregunta, veamos primero cómo estructurar el código base de una aplicación.

1.4.1. 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 complejas": sus algoritmos y estructuras de 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).

[5]Tenga en cuenta que el lenguaje Vala podría considerarse un lenguaje de nicho, teniendo una comunidad pequeña si es comparado con lenguajes más populares.

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.

1.4.2. Otros aspectos a tener en cuenta

Si tiene dudas sobre el lenguaje 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[6]. 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

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

parte GUI 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.

Elegir un lenguaje de nivel superior tiene mucho sentido.

1.5. 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. 19), luego presenta la Programación Orientada a Objetos en C (parte II p. 49) seguida de un capítulo de Lecturas Adicionales (p. 88)

1.6. 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;
- Las Autotools (Autoconf, Automake y Libtool);
- Las páginas del manual de: El kernel de Linux y glibc [7].

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 BuildStream: `https://wiki.gnome.org/Projects/BuildStream`

Para leer la documentación de la API de GLib y GTK, Devhelp es una aplicación útil, si ha instalado el paquete -dev o -doc. Por otro lado para el editor de texto o IDE, hay muchas opciones (y con ellas una fuente de muchos trolls), como por ejemplo: Vim/Neovim, Emacs, gedit, Anjuta, MonoDevelop/Xamarin Studio, Geany, entre muchos otras. Un prometedor IDE especializado para GNOME es Builder, el cual de manera nativa posee integración con GLib/GTK y con su documentación, además de ofrecer integración con Git.

Para crear una *interfaz gráfica de usuario* o GUI con GTK, puede escribir directamente el código para hacerla 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, la cual esta disponible en

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

línea en:

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

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 1.6 en la p. 17). 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 0.1 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 2.70 en el momento de escribir este artículo)

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)`.

```
#include <glib.h>

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

Listado 2.1: Macros C familiares

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);
```

Listado 2.2: Macros para almacenar enteros en punteros

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 ("Estamos almacenando %d\n", GPOINTER_TO_INT (my_pointer));
```

Pero tenga 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.

```
#include <glib.h>

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

Listado 2.3: Comprobaciones de condiciones previas

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`. 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 mensaje de error que les indicará que los argumentos `NULL` no están permitidos.

```
#include <glib.h>

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

Listado 2.4: Aserciones

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 aserción que siempre falla. Las aserciones llaman a `abort()` para salir del programa y (si su entorno lo admite) descargan un archivo central con fines de depuración.

Las aserciones 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 aserción al final asegura que GLib construyó un objeto correctamente, dados valores adecuados.

`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.

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.

```
#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);
```

Listado 2.5: Asignación de memoria GLib

Es importante hacer coincidir `g_malloc()` con `g_free()`, `malloc()` simple con `free()` y (si estás usando C ++) `new` con `delete`. De lo contrario, pueden suceder comportamientos no esperados, ya que estos asignadores pueden usar diferentes grupos de memoria (y `new`/`delete` llama a constructores y destructores).

[1] Antes del estándar ANSI/ISO C, el tipo de puntero genérico `void *` no existía y `malloc()` devolvía un valor `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 *`.

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!)

```
#include <glib.h>

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

Listado 2.6: Macros de asignación

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.

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

Listado 2.7: Envoltorio de portabilidad

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.

```
#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);
```

Listado 2.8: Asignar cadenas

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 del búfer para arrancar:

```
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);
```

Listado 2.9: Modificaciones de cadenas in situ

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.

```c
#include <glib.h>

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

Listado 2.10: Conversiones de cadenas

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).

```c
#include <glib.h>

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

Listado 2.11: Concatenar cadenas

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.

```c
#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);
```

Listado 2.12: Manipulación de vectores de cadena terminados en `NULL`

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

2.2.1. Listas

GLib proporciona listas genéricas con enlaces simples y dobles, `GSList` y `GList`, respectivamente. Estos se implementan como listas de `gpointer`; puede usarlos para contener enteros con las macros `GINT_TO_POINTER` y `GPOINTER_TO_INT`. `GSList` y `GList` tienen casi las mismas API, excepto que hay una función `g_list_previous()` y no `g_slist_previous()`. Esta sección discutirá `GSList` pero todo también se aplica a la lista doblemente enlazada.

```
typedef struct _GSList GSList;

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

Listado 2.13: Celda `GSList`

Una celda `GSList` es una estructura autoexplicativa que se muestra en el Listado 2.13. Los campos de estructura son públicos, por lo que puede usarlos directamente para acceder a los datos o para recorrer la lista.

En la implementación de GLib, la lista vacía es simplemente un puntero `NULL`. Siempre es seguro pasar `NULL` a las funciones de lista, ya que es una lista válida de longitud 0. El código para crear una lista y agregar un elemento podría verse así:

```
GSList *list = NULL;
gchar *element = g_strdup ("Una cadena de caracteres");
list = g_slist_append (list, element);
```

Las listas GLib tienen una influencia Lisp notable; la lista vacía es un valor especial “nil” por esa razón. `g_slist_prepend()` funciona de forma muy similar a `cons` – es una operación de tiempo constante ($O(1)$) que agrega una nueva celda al principio de la lista.

```
#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);
```

Listado 2.14: Cambiar el contenido de la lista vinculada

El listado 2.14 muestra las funciones básicas para cambiar el contenido de `GSList`. Para todos estos, debe asignar el valor de retorno a su puntero de lista en caso de que cambie el encabezado de la lista. Tenga en cuenta que GLib *no* almacena un puntero al final de la lista, por lo que las funciones de agregar, insertar y eliminar se ejecutan en el tiempo $O(n)$, con n la longitud de la lista.

GLib se encargará de los problemas de memoria, desasignando y asignando celdas de lista según sea necesario. Por ejemplo, el siguiente código eliminaría el elemento agregado anteriormente y vaciaría la lista:

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

`list` ahora es `NULL`. Aún tienes que liberar `element` tú mismo, por supuesto.

Para acceder a un elemento de lista, consulte la estructura `GSList` directamente:

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

Para iterar sobre la lista, puede escribir un código como este:

```
GSList *l;

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

```
#include <glib.h>

typedef void (* GDestroyNotify) (gpointer data);

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

Listado 2.15: Liberar listas enteras vinculadas

El Listado 2.15 muestra funciones para borrar una lista completa. `g_slist_free()` elimina todos los enlaces de una sola vez. `g_slist_free()` no tiene valor de retorno porque siempre sería `NULL`, y simplemente puede asignar ese valor a su lista si lo desea. Obviamente, `g_slist_free()` libera solo las celdas de la lista; no tiene forma de saber qué hacer con el contenido de la lista. La función más inteligente `g_slist_free_full()` toma un segundo argumento con un puntero de función de destrucción que se llama en los datos de cada elemento. Para liberar la lista que contiene cadenas asignadas dinámicamente, puede escribir:

```
g_slist_free_full (list, g_free);

/* Si la lista se puede usar mas tarde: */
list = NULL;
```

Esto es equivalente a escribir:

```
GSList *l;

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

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

Construir una lista usando `g_slist_append()` es una *terrible* idea; use `g_slist_prepend()` y luego llame a `g_slist_reverse()` si necesita elementos en un orden en particular. Si prevé agregar con frecuencia a una lista, también puede mantener un puntero al último elemento. El siguiente código se puede usar para realizar agregados eficientes [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);
```

[2]Una forma más conveniente es usar el tipo de datos `GQueue`: una cola de dos extremos que mantiene un puntero a la cabeza, un puntero a la cola y la longitud de la lista doblemente enlazada.

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

Para usar esta función, debe almacenar la lista y su final en algún lugar, y pasar su dirección a `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"));
```

Por supuesto, debe tener cuidado de no utilizar ninguna función de lista que pueda cambiar el final de la lista sin actualizar `list_end`.

```
#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);
```

Listado 2.16: Acceder a datos en una lista vinculada

Para acceder a los elementos de la lista, se proporcionan las funciones del Listado 2.16. Ninguno de estos cambia la estructura de la lista. `g_slist_foreach()` aplica un `GFunc` a cada elemento de la lista.

Usado en `g_slist_foreach()`, su `GFunc` se llamará en cada `list->data` en `list`, pasando el `user_data` que proporcionó a `g_slist_foreach()`. `g_slist_foreach()` es comparable a la función "map" de Scheme.

Por ejemplo, es posible que tenga una lista de cadenas y que desee poder crear una lista paralela con alguna transformación aplicada a las cadenas. Aquí hay algo de código, usando

la función `efficient_append()` de un ejemplo anterior:

```
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 y GTK usan mucho el lenguaje de "puntero de función y datos de usuario". Si tiene experiencia en programación funcional, esto es muy parecido a usar expresiones lambda para crear un *cierre*. (Un cierre combina una función con un *environment* – un conjunto de enlaces nombre-valor. En este caso, el "environment" son los datos de usuario que pasa a `append_foreach()`, y el "cierre" es la combinación del puntero de función y los datos del usuario).

```c
#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);
```

Listado 2.17: Manipular una lista vinculada

Hay algunas prácticas rutinas de manipulación de listas, listadas en Listado 2.17. Con la excepción de `g_slist_copy()`, todos estos afectan las listas en el lugar. Lo que significa que debe asignar el valor de retorno y olvidarse del puntero pasado, tal como lo hace al agregar o eliminar elementos de la lista. `g_slist_copy()` devuelve una lista recién asignada, por lo que puede continuar usando ambas listas y debe liberar ambas listas eventualmente.

```c
#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);
```

Listado 2.18: Listas ordenadas

Finalmente, hay algunas disposiciones para listas ordenadas, que se muestran en Listado 2.18. Para usarlos, debe escribir un `GCompareFunc`, que es como la función de comparación en el estándar C `qsort()`.

Si `a < b`, `GCompareFunc` debería devolver un valor negativo; si `a > b` un valor positivo; si `a == b` debería devolver 0.

Una vez que tenga una función de comparación, puede insertar un elemento en una lista ya ordenada u ordenar una lista completa. Las listas se ordenan en orden ascendente. Incluso puedes reciclar tu `GCompareFunc` para encontrar elementos de la lista, usando `g_slist_find_custom()`.

Tenga cuidado con las listas ordenadas; su mal uso puede volverse muy ineficaz rápidamente. Por ejemplo, `g_slist_insert_sorted()` es una operación $O(n)$, pero si la usa en un bucle para insertar varios elementos, el bucle se ejecuta en tiempo cuadrático ($O(n^2)$). Es mejor

simplemente anteponer todos sus elementos y luego llamar a `g_slist_sort()`. `g_slist_sort()` se ejecuta en $O(n \log n)$.

También puede usar la estructura de datos `GSequence` para datos ordenados. `GSequence` tiene una API de lista, pero se implementa internamente con un árbol binario equilibrado.

2.2.2. Árboles

Hay dos tipos diferentes de árboles en GLib; `GTree` es su árbol binario balanceado básico, útil para almacenar pares clave-valor ordenados por clave; `GNode` almacena datos arbitrarios estructurados en árbol, como un árbol de análisis o taxonomía.

GTree

```
#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);
```

Listado 2.19: Creando y destruyendo árboles binarios balanceados

Para crear y destruir un `GTree`, use un constructor y un destructor que se muestran en el Listado 2.19. `GCompareFunc` es la misma `qsort()`-función de comparación de estilo descrita para `GSList`; en este caso, se utiliza para comparar claves en el árbol. `g_tree_new_full()` es útil para facilitar la gestión de la memoria para claves y valores asignados dinámicamente.

La estructura `GTree` es un tipo de datos opaco. Se accede y modifica a su contenido únicamente con funciones públicas.

```
#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);
```

Listado 2.20: Manipular el contenido de `GTree`

Las funciones para manipular el contenido del árbol se muestran en Listado 2.20. Todo muy sencillo; `g_tree_insert()` sobrescribe cualquier valor existente, así que si no usa `g_tree_new_full()`, tenga cuidado si el valor existente es su único puntero a una porción de memoria asignada. Si `g_tree_lookup()` no encuentra la clave, devuelve `NULL`; de lo contrario, devuelve el valor asociado. Tanto las claves como los valores tienen el tipo `gpointer` o `gconstpointer`, pero las macros `GPOINTER_TO_INT()` y `GPOINTER_TO_UINT()` le permiten usar enteros en su lugar.

```
#include <glib.h>

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

Listado 2.21: Determinar el tamaño de un `GTree`

Hay dos funciones que le dan una idea del tamaño del árbol, que se muestran en el Listado 2.21.

```
#include <glib.h>

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

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

Listado 2.22: Atravesando un `GTree`

Usando `g_tree_foreach()` (Listado 2.22) puedes recorrer todo el árbol. Para usarlo, proporcione un `GTraverseFunc`, al que se le pasa cada par clave-valor y un argumento `data` que le da a `g_tree_foreach()`. El recorrido continúa mientras `GTraverseFunc` devuelva `FALSE`; si alguna vez devuelve `TRUE`, el recorrido se detiene. Puede usar esto para buscar en el árbol por *valor*.

GNode

```
typedef struct _GNode GNode;

struct _GNode
{
```

```
  gpointer data;
  GNode *next;
  GNode *prev;
  GNode *parent;
  GNode *children;
};
```

Listado 2.23: Celda `GNode`

A `GNode` es un árbol N-ario, implementado como una lista doblemente enlazada con listas padre e hijo. Por lo tanto, la mayoría de las operaciones de lista tienen análogos en la API `GNode`. Puedes caminar por el árbol de varias maneras. El listado 2.23 muestra la declaración de un nodo.

```
#include <glib.h>

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

Listado 2.24: Accediendo a `GNode`

Hay macros para acceder a los miembros de `GNode`, que se muestran en el Listado 2.24. Al igual que con `GList`, el miembro `data` está diseñado para usarse directamente. Estas macros devuelven los miembros `next`, `prev` y `children` respectivamente; también comprueban si su argumento es `NULL` antes de eliminar la referencia, y devuelven `NULL` si lo es.

```
#include <glib.h>

GNode * g_node_new (gpointer data);
```

Listado 2.25: Creando un `GNode`

Para crear un nodo, se proporciona la función `_new()` habitual (Listado 2.25). `g_node_new()` crea un nodo sin hijos y sin padres que contiene `datos`. Normalmente, `g_node_new()` se usa solo para crear el nodo raíz; Se proporcionan macros de conveniencia que crean automáticamente nuevos nodos según sea necesario.

```
#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);
```

Listado 2.26: Construyendo un árbol `GNode`

Para construir un árbol se utilizan las operaciones fundamentales que se muestran en el Listado 2.26. Cada operación devuelve el nodo recién agregado, para mayor comodidad al escribir bucles o recuperar el árbol. A diferencia de `GList`, es seguro ignorar el valor de retorno.

```
#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);
```

Listado 2.27: Construyendo un `GNode`

Las macros de conveniencia que se muestran en el Listado 2.27 se implementan en términos de las operaciones fundamentales. `g_node_append()` es análogo a `g_node_prepend()`; el resto toma un argumento `data`, automáticamente le asigna un nodo y llama a la operación básica correspondiente.

```
#include <glib.h>

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

Listado 2.28: Destruyendo un `GNode`

Para eliminar un nodo del árbol, hay dos funciones que se muestran en el Listado 2.28. `g_node_destroy()` elimina el nodo de un árbol, destruyéndolo a él ya todos sus hijos. `g_node_unlink()` elimina un nodo y lo convierte en un nodo raíz; es decir, convierte un subárbol en un árbol independiente.

```
#include <glib.h>

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

Listado 2.29: Predicados para `GNode`

Hay dos macros para detectar la parte superior e inferior de un árbol `GNode`, que se muestran en el Listado 2.29. Un nodo raíz se define como un nodo sin padres ni hermanos. Un nodo hoja no tiene hijos.

```
#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);
```

Listado 2.30: Propiedades de `GNode`

Puede pedirle a GLib que proporcione información útil sobre un `GNode`, incluido el número de nodos que contiene, su nodo raíz, su profundidad y el nodo que contiene un puntero de datos en particular. Estas funciones se muestran en el Listado 2.30.

`GTraverseType` es una enumeración; hay cuatro valores posibles. Estos son sus significados:

- `G_PRE_ORDER` visita el nodo actual, luego recurre a cada hijo por turno.
- `G_POST_ORDER` recurre a cada hijo en orden, luego visita el nodo actual.
- `G_IN_ORDER` primero recurre al hijo más a la izquierda del nodo, luego visita el nodo mismo y luego recurre al resto de los hijos del nodo. Esto no es muy útil; en su mayoría, está diseñado para su uso con un árbol binario.
- `G_LEVEL_ORDER` primero visita el nodo en sí; luego cada uno de los hijos del nodo; luego los hijos de los hijos; luego los hijos de los hijos de los hijos; y así. Es decir, visita cada nodo de profundidad 0, luego cada nodo de profundidad 1, luego cada nodo de profundidad 2, etc.

Las funciones de recorrido de árbol de `GNode` tienen un argumento `GTraverseFlags`. Este es un campo de bits que se utiliza para cambiar la naturaleza del recorrido. Actualmente solo hay tres banderas: puede visitar solo los nodos de hoja, solo los nodos que no son de hoja o todos los nodos:

- `G_TRAVERSE_LEAVES` significa atravesar solo los nodos hoja.
- `G_TRAVERSE_NON_LEAVES` significa atravesar solo nodos que no son hojas.
- `G_TRAVERSE_ALL` es simplemente un atajo para `(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);
```

Listado 2.31: Accediendo a un `GNode`

El listado 2.31 muestra algunas de las funciones restantes de `GNode`. Son sencillos; la mayoría de ellos son simplemente operaciones en la lista de hijos del nodo. Hay dos definiciones de tipos de función exclusivas de `GNode`: `GNodeTraverseFunc` y `GNodeForeachFunc`. Estos se llaman con un puntero al nodo que se está visitando y los datos de usuario que proporciona. Un `GNodeTraverseFunc` puede devolver `TRUE` para detener cualquier recorrido que esté en progreso; por lo tanto, puede usar `g_node_traverse()` para buscar el árbol por valor.

2.2.3. Tablas hash

`GHashTable` es una implementación de tabla hash simple, que proporciona una matriz asociativa con búsquedas en tiempo constante. Para crear y destruir una `GHashTable`, use un constructor y un destructor listados en Listado 2.32. Debe proporcionar una `GHashFunc`, que debería devolver un entero positivo cuando se le pase una clave hash. Cada `guint` devuelto (módulo del tamaño de la tabla) corresponde a un "slot" o "bucket" en el hash; `GHashTable` maneja las colisiones almacenando una lista vinculada de pares clave-valor en cada espacio. Por lo tanto, los valores de `guint` devueltos por su `GHashFunc` deben distribuirse de manera bastante uniforme sobre el conjunto de posibles valores de `guint`, o la tabla hash degenerará en una lista enlazada. Su `GHashFunc` también debe ser rápido, ya que se usa para cada búsqueda.

Además de `GHashFunc`, se requiere una `GEqualFunc` para probar la igualdad de las claves. Se utiliza para encontrar el par clave-valor correcto cuando las colisiones hash dan como resultado más de un par en la misma ranura hash.

Si usa el constructor básico `g_hash_table_new()`, recuerde que GLib no tiene forma de saber cómo destruir los datos contenidos en su tabla hash; solo destruye la tabla misma. Si es necesario liberar las claves y los valores, use `g_hash_table_new_full()`, las funciones de destrucción se llamarán en cada clave y valor antes de destruir la tabla hash.

```
#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);
```

Listado 2.32: `GHashTable` constructores y destructores

Se proporcionan funciones de comparación y hash listas para usar para claves comunes: enteros, punteros, cadenas y otros tipos de GLib. Los más comunes se enumeran en el Listado 2.33. Las funciones para enteros aceptan un puntero a `gint`, en lugar de a `gint` en sí. Si pasa `NULL` como argumento de la función hash a `g_hash_table_new()`, `g_direct_hash()` se usa por defecto. Si pasa `NULL` como la función de igualdad de claves, entonces se usa una comparación de puntero simple (equivalente a `g_direct_equal()`, pero sin una llamada de función).

```
#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);
```

Listado 2.33: Hashes/comparaciones preescritos

Manipular la tabla hash es simple. Las rutinas se resumen en Listado 2.34. Las inserciones *no* copian la clave o el valor; estos se ingresan en la tabla exactamente como los proporciona, reemplazando cualquier par clave-valor preexistente con la misma clave ("igual" está definido por sus funciones hash e igualdad, recuerde). Si esto es un problema, debe realizar una búsqueda o eliminar antes de insertar. Tenga especial cuidado si asigna claves o valores de forma dinámica. Si ha proporcionado funciones `GDestroyNotify`, éstas se llamarán automáticamente en el antiguo par clave-valor antes de reemplazarlo.

El `g_hash_table_lookup()` simple devuelve el valor que encuentra asociado con `key`, o `NULL` si no hay ningún valor. A veces esto no funciona. Por ejemplo, `NULL` puede ser un valor válido en sí mismo. Si está utilizando cadenas como claves, especialmente cadenas asignadas dinámicamente, saber que una clave está en la tabla puede no ser suficiente; es posible que desee recuperar el `gchar*` exacto que utiliza la tabla hash para representar la clave `"foo"`. Se proporciona una segunda función de búsqueda para casos como estos. `g_hash_table_lookup_extended()` devuelve `TRUE` si la búsqueda se realizó correctamente; si devuelve `TRUE`, coloca la clave y el valor que encontró en las ubicaciones proporcionadas.

```
#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);
```

Listado 2.34: Manipular una `GHashTable`

2.3. El bucle del evento principal

Las aplicaciones actuales a menudo se basan en eventos. Para las aplicaciones GUI, hay muchas fuentes de eventos: una pulsación de tecla, un clic del mouse, un gesto táctil, un mensaje de otra aplicación, un cambio en el sistema de archivos, estar conectado o desconectado de la red, etc. Una aplicación necesita reaccionar a esos eventos. Por ejemplo, cuando

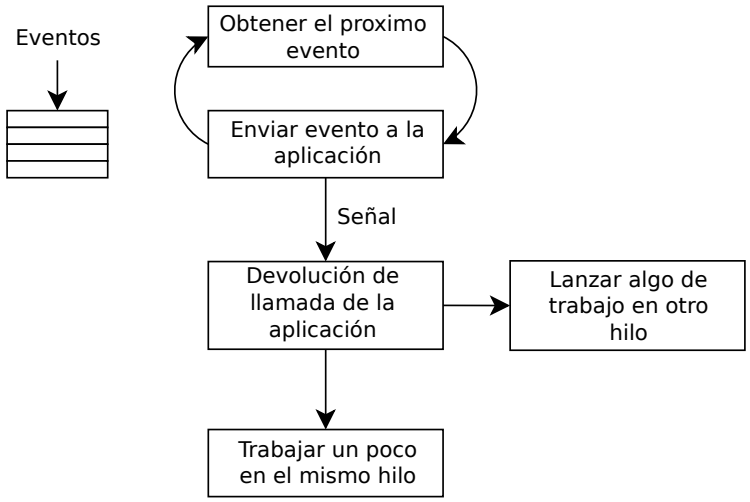


Figura 2.1: Estructura de una aplicación impulsada por eventos, con un bucle de eventos principal

se presiona una tecla cuando una entrada de texto tiene el foco, el carácter debe insertarse y mostrarse en la pantalla.

Pero la programación dirigida por eventos también se aplica a los daemons. La comunicación entre procesos también ocurre entre daemons y aplicaciones. Un daemon podría recibir un evento cuando llega un paquete a una interfaz de red. Un daemon de impresora podría recibir eventos cuando una impresora está conectada, desconectada, tiene poco papel, etc. Un daemon de montaje puede escuchar las memorias USB insertadas. Otro daemon puede escuchar las conexiones de monitores externos para reconfigurar las pantallas, y así sucesivamente.

Un programa impulsado por eventos no está estructurado de la misma manera que un programa por lotes. El trabajo a realizar por un programa por lotes se determina al principio. Luego analiza la entrada, realiza algunos cálculos sobre ella y genera un informe. Por ejemplo, la mayoría de los comandos y scripts de Unix son programas por lotes.

Entonces, ¿cómo estructurar una aplicación que necesita responder a varios eventos que pueden llegar en cualquier momento? Como sugiere el título de esta sección ... ¡con un *bucle de evento principal* por supuesto! Esa es otra parte importante de GLib; Proporciona soporte básico de programación dirigida por eventos, con una abstracción de bucle de eventos principal, una implementación portátil de subprocesos y comunicación asincrónica entre subprocesos. Un bucle de eventos escucha algunas fuentes de eventos. Se asocia una prioridad con cada fuente de eventos. Cuando llega un evento, el bucle de eventos lo envía a la aplicación. El evento se puede tener en cuenta, ya sea en el mismo hilo o en otro hilo. La

Figura 2.1 muestra una vista de alto nivel de lo que es un bucle de eventos principal.

La función `main()` de una aplicación dirigida por eventos se ve así:

```
gint
main (gint   argc,
      gchar *argv[])
{
  /* Cree una ventana principal y adjunte devoluciones de llamada de senal
   . */

  /* Ejecute el bucle de eventos principal. */
  gtk_main ();

  return 0;
}
```

El bucle de eventos GTK tiene un nivel ligeramente más alto que la abstracción del bucle de eventos GLib. `gtk_main()` ejecuta el ciclo de eventos principal hasta que se llama a `gtk_main_quit()`. `gtk_main_quit()` normalmente se llama en la devolución de llamada de la función cuando se hace clic en el botón de cierre o se activa la acción del menú Salir.

Una devolución de llamada es una función que se llama cuando se envía una señal. El sistema de señales está implementado por la biblioteca GObject. Escuchar una señal se logra con la función `g_signal_connect()`:

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

  /* Hacer algo */
}

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

  /* Crear boton */

  /* Adjuntar devolucion de llamada de senal */
  g_signal_connect (button,
```

```
                        "clicked",
                        G_CALLBACK (button_clicked_cb),
                        self);
}
```

Cuando se ejecuta una devolución de llamada, bloquea el bucle principal. Entonces, para no congelar la interfaz de usuario, existen dos soluciones:

1. Las operaciones largas (especialmente las E/S) se pueden iniciar en otro hilo.
2. Las operaciones largas se pueden dividir en fragmentos más pequeños, y cada fragmento se ejecuta en una iteración de bucle principal separada.

Para la segunda solución, GLib proporciona las funciones `g_idle_add()` y `g_timeout_add()` (ver Listado 2.35). Se llamará a una función inactiva cuando el bucle principal esté inactivo, es decir, cuando el bucle principal no tenga nada más que hacer. Se llama a una función de tiempo de espera a intervalos regulares. El valor de retorno booleano de una `GSourceFunc` permite continuar o detener la función. Si continúa, el bucle principal volverá a llamar a la función en el siguiente tiempo de inactividad o tiempo de espera. Puede eliminar manualmente `GSourceFunc` llamando a `g_source_remove()`, que toma como parámetro el ID de origen devuelto por `g_idle_add()` o `g_timeout_add()`. Debe prestar atención para eliminar una `GSourceFunc` cuando se destruye el objeto en el que realiza el cálculo. Por lo tanto, puede almacenar el ID de fuente en un atributo de objeto y llamar a `g_source_remove()` en el destructor si el ID de fuente es diferente de `0`. (Consulte la biblioteca de GObject para crear sus propias clases en C.)

```
#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);
```

Listado 2.35: Inactivos y tiempos de espera

2.4. Otras características

Simplemente no hay espacio para cubrir todas las funciones de GLib en este libro. Vale la pena mirar GLib cada vez que piense: "Realmente *debería* haber una función que ...". Esta

sección enumera otras características que proporciona GLib, pero *no* es exhaustiva.

Parte del soporte de aplicaciones principales que no se ha mencionado aún:

- `GError`: un sistema de notificación de errores, similar a las excepciones en otros lenguajes.
- La función `g_log()` le permite imprimir advertencias, mensajes, etc. con niveles de registro configurables y rutinas de impresión conectables.

Utilidades:

- Un analizador de opciones de línea de comandos.
- Un marco de prueba unitaria.
- Una instalación de temporizador.
- Funciones calendáricas/aritméticas de fechas.
- Manipulación de nombre de archivo, como `g_path_get_basename()` y `g_path_is_absolute()`.
- Un analizador XML simple.
- Expresiones regulares compatibles con Perl.

Una selección de utilidades más pequeñas:

- `G_MAXFLOAT`, etc. equivalentes para muchos tipos numéricos.
- Conversiones por orden de bytes.
- `G_DIR_SEPARATOR` maneja las diferencias de Windows/Unix.
- Rutinas de conveniencia/portabilidad para obtener el directorio de inicio del usuario, obtener el nombre de un directorio `/tmp` y tareas similares.
- `G_VA_COPY` copia una `va_list` de forma portátil.
- Numerosas macros para permitir el uso de extensiones del compilador (especialmente extensiones GCC) de forma portátil.
- Manipulación de campo de bits.
- Portable `g_htonl()` y otras conversiones de host a red.

Y por último, pero no menos importante, otros tipos de datos interesantes:

- Clases mejoradas de cadenas y matrices. Arreglos de punteros y bytes.
- `GQuark` – mapeo bidireccional de cadenas a identificadores enteros.
- `GVariant` – un tipo de datos genérico que almacena un valor junto con información sobre el tipo de ese valor.

Parte II

Programación orientada a objetos en C

Introducción a la Parte II

Ahora que está familiarizado con la biblioteca principal de GLib, ¿cuál es el siguiente paso? Como se explicó en la sección Ruta de aprendizaje (sección 1.5 p. 16), el seguimiento lógico es la programación orientada a objetos (OOP) en C y los conceptos básicos de GObject.

Cada widget GTK es una subclase de la clase base GObject. Entonces, conocer los conceptos básicos de GObject es importante para *usar* un widget GTK u otra utilidad basada en GObject, pero también para *crear* tus propias clases de GObject.

Es importante notar que aunque el lenguaje C no está orientado a objetos, es posible escribir código C "semi-orientado a objetos" fácilmente, sin GObject. Para fines de aprendizaje, con eso comienza esta parte. GObject es entonces más fácil de aprender. Lo que GObject agrega son más características como recuento de referencias, herencia, funciones virtuales, interfaces, señales y más.

Pero, ¿por qué seguir un estilo orientado a objetos en primer lugar? Un código orientado a objetos permite evitar variables globales. Y si ha leído algún tipo de guía de mejores prácticas de programación, sabe que *debería*, si es posible, evitar las variables globales [1]. Porque el uso de datos globales hace que el código sea más difícil de administrar y comprender, especialmente cuando un programa se vuelve más grande. También hace que el código sea más difícil de reutilizar. En cambio, es mejor dividir un programa en partes más pequeñas e independientes, de modo que pueda concentrarse solo en una parte del código a la vez.

Esta parte del libro consta de dos capítulos:

- Capítulo 3, que explica cómo escribir sus propias clases semi-OOP;
- Capítulo 4, que explica los conceptos básicos de GObject.

[1] Una variable global en C puede ser una variable `static` declarada en el parte superior de un archivo *.c, al que se puede acceder desde cualquier función en ese archivo *.c. Esto a veces es útil, pero debe evitarse si es posible. Hay otro tipo de variable global en C: una variable `extern` a la que se puede acceder desde cualquier archivo *.c. Este último es mucho peor que el primero.

Capítulo 3

Programación semi-orientada a objetos en C

En el capítulo anterior se explicó que la biblioteca principal de GLib utiliza un estilo de codificación semi-orientado a objetos. Esta sección explica lo que significa y cómo escribir su propio código con este estilo de codificación.

Una de las ideas principales de OOP es *mantener los datos y el comportamiento relacionados en un solo lugar* [1] . En C, los datos se almacenan en una `struct` y el comportamiento se implementa con funciones. Para mantenerlos en un solo lugar, los colocamos en el mismo archivo *.c, con las funciones públicas presentes en el archivo *.h correspondiente (el encabezado).

Otra idea importante de OOP es *ocultar todos los datos dentro de su clase*. En C, significa que la declaración completa de `struct` debe estar presente solo en el archivo *.c, mientras que el encabezado contiene solo un `typedef`. Cómo se almacenan los datos dentro de la clase y qué estructuras de datos se utilizan debe seguir siendo un detalle de implementación. Un usuario de la clase no debe estar al tanto de los detalles de implementación, en su lugar, debe confiar solo en la interfaz externa, es decir, lo que está presente en el encabezado y la documentación pública. De esa manera, la implementación de la clase puede cambiar sin afectar a los usuarios de la clase, siempre que la API no cambie.

[1] Esta es una de las pautas discutidas en *Heurística de diseño orientado a objetos* [2].

3.1. Ejemplo de encabezado

El Listado 3.1 p. 52 muestra un ejemplo de un encabezado que proporciona un simple corrector ortográfico. Este es un código ficticio; si necesita un corrector ortográfico en su aplicación GTK, probablemente usaría hoy en día la biblioteca gspell [2].

```
#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 */
```

Listado 3.1: myapp-spell-checker.h

3.1.1. Espacio de nombres del proyecto

Lo primero a tener en cuenta es el uso del espacio de nombres “ Myapp ”. Cada símbolo del encabezado tiene como prefijo el espacio de nombres del proyecto.

[2] https://gitlab.gnome.org/GNOME/gspell

Es una buena práctica elegir un espacio de nombres para su código, para evitar conflictos de símbolos en el momento del enlace. Es especialmente importante tener un espacio de nombres para una biblioteca, pero también es mejor tener uno para una aplicación. Por supuesto, el espacio de nombres debe ser único para cada base de código; por ejemplo, *no* debe reutilizar los espacios de nombres "G" o "Gtk" para su aplicación o biblioteca.

3.1.2. Espacio de nombres de clase

Además, hay un segundo espacio de nombres con el nombre de la clase, aquí "SpellChecker". Si sigue esa convención de forma coherente, el nombre de un símbolo es más predecible, lo que facilita el trabajo con la API. El nombre de un símbolo siempre será "espacio de nombres del proyecto" + "nombre de la clase" + "nombre del símbolo".

3.1.3. ¿Minúsculas, Mayúsculas o CamelCase?

Dependiendo del tipo de símbolo, su nombre está en mayúsculas, minúsculas o CamelCase. La convención en el mundo GLib es:

- Letras mayúsculas para una constante, ya sea para un valor `#define` o `enum`;
- CamelCase para un tipo `struct` o `enum`;
- Minúsculas para funciones, variables, campos `struct`,...

3.1.4. Incluir guardia

El encabezado contiene el típico protector de inclusión:

```
#ifndef MYAPP_SPELL_CHECKER_H
#define MYAPP_SPELL_CHECKER_H

/* ... */

#endif /* MYAPP_SPELL_CHECKER_H */
```

Evita que el encabezado se incluya varias veces en el mismo archivo *.c.

3.1.5. Soporte de C++

El par `G_BEGIN_DECLS`/`G_END_DECLS` permite que el encabezado se incluya desde el código C ++. Es más importante para una biblioteca, pero también es una buena práctica agregar esas

macros en el código de la aplicación, incluso si la aplicación no usa C ++. De esa manera, una clase de aplicación se podría mover a una biblioteca fácilmente (puede ser difícil notar que faltan las macros `G_BEGIN_DECLS` y `G_END_DECLS`). Y si surge el deseo, la aplicación podría trasladarse a C ++ de forma incremental.

3.1.6. #include

Hay varias formas de organizar los `#include` en una base de código C. La convención en GLib/GTK es que incluir un encabezado en un archivo *.c no debería requerir incluir otro encabezado de antemano [3].

`myapp-spell-checker.h` contiene el siguiente `#include`:

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

Debido a que `glib.h` es necesario para las macros `G_BEGIN_DECLS` y `G_END_DECLS`, para las definiciones de tipos básicos de GLib (`gchar`, `gboolean`, etc.) y `GSList`.

Si se elimina el `#include` en `myapp-spell-checker.h`, cada archivo *.c que incluya `myapp-spell-checker.h` también debería incluir `glib.h` de antemano; de lo contrario, el compilador no podría compilar ese archivo *.c. Pero no queremos agregar tal requisito para los usuarios de la clase.

3.1.7. Definición de tipo

El tipo `MyappSpellChecker` se define de la siguiente manera:

```
typedef struct _MyappSpellChecker MyappSpellChecker;
```

La `struct _MyappSpellChecker` se declarará en el archivo `myapp-spell-checker.c`. Cuando usa `MyappSpellChecker` en otro archivo, no debería necesitar saber qué contiene `struct`, debería usar las funciones públicas de la clase en su lugar. La excepción a la mejor práctica de OOP es cuando llamar a una función sería un problema de rendimiento, por ejemplo, para estructuras de datos de bajo nivel utilizadas para gráficos por computadora (coordenadas, rectángulos,...). Pero recuerde: *la optimización prematura es la raíz de todos los males* (Donald Knuth).

[3] Esa es la regla general, pero existen excepciones: por ejemplo, incluir `glib/gi18n-lib.h` requiere que se defina `GETTEXT_PACKAGE`, este último suele estar presente en el encabezado `config.h`.

3.1.8. Constructor de objetos

`myapp_spell_checker_new()` es el constructor de la clase. Toma un parámetro de código de idioma — por ejemplo `"en_US"`— y devuelve una *instancia* de la clase, también llamada *objeto*. Lo que hace la función es simplemente asignar dinámicamente la `estructura` y devolver el puntero. Ese valor de retorno se usa luego como el primer parámetro de las funciones restantes de la clase. En algunos lenguajes como Python, ese primer parámetro se llama parámetro *self*, ya que hace referencia a "sí mismo", es decir, a su propia clase. Otros lenguajes orientados a objetos como Java y C ++ tienen la palabra clave *this* para acceder a los datos del objeto.

Tenga en cuenta que `myapp_spell_checker_new()` se puede llamar varias veces para crear diferentes objetos. Cada objeto tiene sus propios datos. Esa es la diferencia fundamental con las variables globales: si los datos se almacenan en variables globales, puede haber como máximo una instancia de ellos [4].

3.1.9. Destructor de objetos

`myapp_spell_checker_free()` es el destructor de la clase. Destruye un objeto liberando su memoria y liberando otros recursos asignados.

3.1.10. Otras funciones públicas

Las funciones `myapp_spell_checker_check_word()` y `myapp_spell_checker_get_suggestions()` son las características disponibles de la clase. Comprueba si una palabra está escrita correctamente y obtiene una lista de sugerencias para corregir una palabra mal escrita.

El tipo de parámetro `word_length` es `gssize`, que es un tipo entero GLib que puede contener — por ejemplo— el resultado de `strlen()`, y también puede contener un valor negativo ya que — contrario a `gsize`— `gssize` es un tipo entero *con signo*. El parámetro `word_length` puede ser `-1` si la cadena está terminada en nulo, es decir, si la cadena termina con el carácter especial `'\0'`. El propósito del parámetro `word_length` es poder pasar un puntero a una palabra que pertenece a una cadena más grande, sin la necesidad de llamar, por ejemplo, `g_strndup()`.

[4] A menos que la variable global almacene los "objetos" en una estructura de datos agregados como una matriz, un lista enlazada o una tabla hash, con un identificador como el parámetro "*self*" (por ejemplo, un número entero o una cadena) para acceder a un elemento específico. Pero este es un código realmente desagradable, ¡por favor no lo hagas!

3.2. El archivo *.c correspondiente

Veamos ahora el archivo `myapp-spell-checker.c`:

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

struct _MyappSpellChecker
{
  gchar *language_code;

  /* Ponga aqui otras estructuras de datos utilizadas
   * para implementar la revision ortografica.
   */
};

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

/**
 * myapp_spell_checker_new:
 * @language_code: el lenguaje del codigo a utlizar.
 *
 * Returns: un nuevo objeto #MyappSpellChecker. Libre con
 * 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: un #MyappSpellChecker.
 *
 * Libera @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: un #MyappSpellChecker.
 * @word: la palabra para comprobar.
 * @word_length: la longitud de bytes de @word, o -1 si @word
 * es terminado en nulo.
 *
 * Returns: %TRUE si @word esta escrito correctamente, %FALSE
 * de lo contrario.
 */
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);

  /* ... Compruebe si la palabra esta presente en un diccionario. */

  return TRUE;
}

/**
 * myapp_spell_checker_get_suggestions:
```

```
 * @checker: un #MyappSpellChecker.
 * @word: una palabra mal escrita.
 * @word_length: la longitud de bytes de @word, o -1 si @word
 * es terminado en nulo.
 *
 * Obtiene las sugerencias para @word. Libera el valor de retorno con
 * g_slist_free_full(suggestions, g_free).
 *
 * Returns: (transfer full) (element-type utf8): la lista de sugerencias.
 */
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;
}
```

Listado 3.2: myapp-spell-checker.c

3.2.1. Orden de #include

En la parte superior del archivo, se encuentra la lista habitual de `#include`. Un detalle pequeño pero digno de mención es que el orden de inclusión no se eligió al azar. En cierto archivo *.c, es mejor incluir primero su archivo *.h correspondiente, y luego los otros encabezados [5]. Al incluir la primera `myapp-spell-checker.h`, si falta un `#include` en `myapp-spell-checker.h`, el compilador informará un error. Como se explica en la sec-

[5]Excepto si tiene un archivo `config.h`, en ese caso debería `config.h`, *luego* el *.h correspondiente, y luego los otros encabezados.

ción 3.1.6 p. 54, un encabezado siempre debe tener el mínimo requerido `#includes` para que ese encabezado se incluya a su vez.

Además, dado que `glib.h` ya está incluido en `myapp-spell-checker.h`, no es necesario incluirlo una segunda vez en `myapp-spell-checker.c`.

3.2.2. Comentarios de GTK-Doc

La API pública está documentada con comentarios GTK-Doc. Un comentario de GTK-Doc comienza con `/**`, con el nombre del símbolo a documentar en la siguiente línea. Cuando nos referimos a un símbolo, existe una sintaxis especial a utilizar dependiendo del tipo de símbolo:

- Un parámetro de función tiene el prefijo `@`.
- El *nombre* de una `estructura` o `enum` tiene el prefijo `\#`.
- Una constante — por ejemplo, una `enum` *valor* —tiene el prefijo `\%`.
- Una función tiene el sufijo `()`.

GTK-Doc puede analizar esos comentarios especiales y generar páginas HTML que luego pueden ser navegadas fácilmente por un navegador API como Devhelp. Pero los comentarios especialmente formateados en el código no son lo único que necesita GTK-Doc, también necesita integración con el sistema de compilación de su proyecto (por ejemplo, Autotools), junto con algunos otros archivos para enumerar las diferentes páginas, describe el estructura general con la lista de símbolos y, opcionalmente, proporciona contenido adicional escrito en formato DocBook XML. Estos archivos suelen estar presentes en el directorio `docs/reference/`.

Describir en detalle cómo integrar el soporte GTK-Doc en su código está más allá del alcance de este libro. Para eso, debe consultar el manual de GTK-Doc [9].

Cada biblioteca basada en GLib debe tener una documentación GTK-Doc. Pero también es útil escribir una documentación GTK-Doc para el código interno de una aplicación. Como se explica en la sección 1.4.1 p. 13, es una buena práctica separar el backend de una aplicación de su(s) frontend(s), y escribir el backend como una biblioteca interna o, más tarde, una biblioteca compartida. Como tal, se recomienda documentar la API pública del backend con GTK-Doc, incluso si sigue siendo una biblioteca interna vinculada estáticamente. Porque cuando el código base se vuelve más grande, es de gran ayuda – especialmente para los recién llegados – tener una descripción general de las clases disponibles y saber cómo usar

una clase sin la necesidad de comprender su implementación.

3.2.3. Anotaciones de introspección de GObject

El comentario de GTK-Doc para la función `myapp_spell_checker_get_suggestions()` contiene anotaciones de GObject Introspection para el valor de retorno:

```
/**
 * ...
 * Returns: (transfer full) (element-type utf8): la lista de sugerencias.
 */
```

El tipo de retorno de la función es `GSList *`, que no es suficiente para que los enlaces de idioma sepan qué contiene la lista, y cómo liberarla. `(transferencia completa)` significa que la persona que llama debe liberar completamente el valor de retorno: la lista misma *y* sus elementos. `(tipo de elemento utf8)` significa que la lista contiene cadenas UTF-8.

Para obtener un valor de retorno, si la anotación de transferencia es `(contenedor de transferencia)`, la persona que llama necesita liberar solo la estructura de datos, no sus elementos. Y si la anotación de transferencia es `(transferir ninguna)`, la propiedad del contenido variable no se transfiere y, por lo tanto, la persona que llama no debe liberar el valor de retorno.

Hay muchas otras anotaciones de GObject Introspection (GI), por nombrar solo un par de otras:

- `(nullable)`: el valor puede ser `NULL`;
- `(out)`: un parámetro de función "out", es decir, un parámetro que devuelve un valor.

Como puede ver, las anotaciones GI no solo son útiles para los enlaces de idiomas, sino que también recopilan información útil para el programador de C.

Para obtener una lista completa de anotaciones GI, consulte la wiki de GObject Introspection [10].

3.2.4. Funciones estáticas vs funciones no estáticas

En el código de ejemplo, puede ver que la función `load_dictionary()` se ha marcado como `static`. De hecho, es una buena práctica en C marcar las funciones internas como `static`. Una función `static` solo se puede usar en el mismo archivo *.c. Por otro lado, una función pública debe ser no estática y tener un prototipo en un encabezado.

Existe la opción de advertencia `-Wmissing-prototypes` GCC para garantizar que un fragmento de código siga esta convención [6].

Además, contrariamente a una función pública, una función `static` no requiere que el proyecto y los espacios de nombres de la clase tengan el prefijo (aquí, `myapp_spell_checker`).

3.2.5. Programación defensiva

Cada función pública verifica sus precondiciones con las macros de depuración de `g_return_if_fail()` o `g_return_val_if_fail()`, como se describe en la sección 2.1.3 p. 23. El parámetro *self* también se verifica, para ver si no es `NULL`, excepto por el destructor, `myapp_spell_checker_free()`, para que sea consistente con `free()` y `g_free()` que acepta un parámetro `NULL` por conveniencia.

Tenga en cuenta que las macros `g_return` deben usarse solo para los puntos de entrada de una clase, es decir, en sus funciones públicas. Por lo general, puede asumir que los parámetros pasados a una función `static` son válidos, especialmente el parámetro *self*. Sin embargo, a veces es útil verificar un argumento de una función `static` con `g_assert()`, para hacer que el código sea más robusto y auto-documentado.

3.2.6. Estilo de codificación

Finalmente, vale la pena explicar algunas cosas sobre el estilo de codificación. Se le anima a utilizar desde el principio el mismo estilo de codificación para su proyecto, ya que puede ser algo difícil de cambiar después. Si cada programa tiene diferentes convenciones de código, es una pesadilla para alguien dispuesto a contribuir.

Aquí hay un ejemplo de una definición de función:

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

Vea cómo se alinean los parámetros: hay un parámetro por línea, con el tipo alineado en el paréntesis de apertura y con los nombres alineados en la misma columna. Algunos editores

[6] Cuando se usan las herramientas automáticas, la macro `AX_COMPILER_FLAGS` Autoconf habilita, entre otras cosas, esa bandera GCC.

de texto se pueden configurar para hacerlo automáticamente.

Para una función *call*, si poner todos los parámetros en la misma línea da como resultado una línea demasiado larga, los parámetros también deben estar alineados entre paréntesis, para que el código sea más fácil de leer:

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

A diferencia de otros proyectos como el kernel de Linux, no existe realmente un límite en la longitud de la línea. Un código basado en GObject puede tener líneas bastante largas, incluso si los parámetros de una función están alineados entre paréntesis. Por supuesto, si una línea termina en, digamos, la columna 120, podría significar que hay demasiados niveles de sangría y que debería extraer el código en funciones intermedias.

El tipo de retorno de una función *definition* está en la línea anterior que el nombre de la función, por lo que el nombre de la función está al principio de la línea. Al escribir una función *prototype*, el nombre de la función nunca debe estar al principio de una línea, idealmente debe estar en la misma línea que el tipo de retorno [7]. De esa manera, puede hacer una búsqueda de expresión regular para encontrar la implementación de una función, por ejemplo, con el comando de shell `grep`:

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

O, si el código está dentro de un repositorio de Git, es un poco más conveniente de usar `git grep`:

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

Asimismo, existe una manera fácil de encontrar la declaración de una `struct` pública. La convención es prefijar el nombre del tipo con un guión bajo al declarar la `struct`. Por ejemplo:

```
/* En el encabezado: */
typedef struct _MyappSpellChecker MyappSpellChecker;

/* En el archivo *.c: */
struct _MyappSpellChecker
{
  /* ... */
};
```

[7] En `myapp-spell-checker.h`, los nombres de las funciones están sangrados con dos espacios en lugar de estar en las mismas líneas que los tipos de retorno, porque no hay suficiente espacio horizontal en la página.

Como resultado, para encontrar la declaración completa del tipo `MyappSpellChecker`, puede buscar "`_MyappSpellChecker`":

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

En GLib/GTK, esta convención de prefijo de subrayado se aplica normalmente a cada `struct` que tiene una línea `typedef` separada. La convención no se sigue a fondo cuando se usa una `struct` en un solo archivo *.c. Y la convención generalmente no se sigue para los tipos `enum`. Sin embargo, para su proyecto, nada le impide aplicar consistentemente la convención de prefijo de subrayado a todos los tipos.

Tenga en cuenta que existen herramientas más sofisticadas que `grep` para examinar una base de código C, por ejemplo, Cscope[8].

Para aprender más sobre el estilo de codificación usado en GLib, GTK y GNOME, lea las Pautas de programación de GNOME [11].

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

Capítulo 4

Una suave introducción a GObject

En el capítulo anterior hemos aprendido a escribir código semi-orientado a objetos en C. Así es como se escriben las clases en GLib core. GObject avanza varios pasos en la programación orientada a objetos, con herencia, interfaces, funciones virtuales, etc. GObject también simplifica el paradigma de programación dirigida por eventos, con señales y propiedades.

Se recomienda crear sus propias clases de GObject para escribir una aplicación GLib/GTK. Desafortunadamente, el código es un poco detallado, porque el lenguaje C no está orientado a objetos. El código repetitivo es necesario para algunas funciones, pero no tenga miedo, existen herramientas y scripts para generar el modelo repetitivo.

Sin embargo, este capítulo da un paso atrás del capítulo anterior, es solo una pequeña introducción a GObject; explicará las cosas esenciales para saber cómo *usar* una clase GObject existente (como todos los widgets y clases GTK en GIO). No explicará cómo *crear* sus propias clases de GObject, porque ya está bien cubierto en el manual de referencia de GObject, y el objetivo de este libro no es duplicar todo el contenido de los manuales de referencia, el objetivo es más para que sirva de guía de introducción.

Entonces, para obtener información más detallada sobre GObject y saber cómo crear subclases, la documentación de referencia de GObject contiene capítulos introductorios: "*Conceptos*" y "*Tutorial*", disponibles en:

`https://docs.gtk.org/gobject/`

Para explicar ciertos conceptos, se toman algunos ejemplos de GTK o GIO. Al leer este capítulo, se le anima a abrir Devhelp en paralelo, para mirar la referencia de API y ver por sí mismo cómo se documenta una biblioteca basada en GObject. El objetivo es que seas

autónomo y puedas aprender cualquier clase nueva de GObject, ya sea en GIO, GTK o cualquier otra biblioteca.

4.1. Herencia

Un concepto importante de OOP es la herencia. Una clase puede ser una subclase de una clase principal. La subclase hereda las características de la clase padre, extendiendo o anulando su comportamiento.

La biblioteca GObject proporciona la clase base `GObject`. Cada clase en GIO y GTK hereda, directa o indirectamente, de la clase base `GObject`. Al mirar una clase basada en GObject, la documentación (si está escrita con GTK-Doc) siempre contiene una *Jerarquía de objetos*. Por ejemplo, `GtkApplication` tiene la siguiente jerarquía de objetos:

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

Significa que cuando crea un objeto `GtkApplication`, también tiene acceso a las funciones, señales y propiedades de `GApplication` (implementado en GIO) y `GObject`. Por supuesto, las funciones `g_application_*` toman como primer argumento una variable de tipo "`GApplication *`", no "`GtkApplication *`". Para convertir la variable en el tipo correcto, la forma recomendada es usar la macro `G_APPLICATION()`. Por ejemplo:

```
GtkApplication *app;

g_application_mark_busy (G_APPLICATION (app));
```

4.2. Macros de GObject

Cada clase de GObject proporciona un conjunto de macros estándar. La macro `G_APPLICATION ()` como se demostró en la sección anterior es una de las macros estándar proporcionadas por la clase `GApplication`.

No todas las macros estándar de GObject se explicarán aquí, solo las macros útiles para *usar* un GObject de una manera básica. Las otras macros son más avanzadas y generalmente son útiles solo cuando se subclasifica una clase GObject, cuando se crea una propiedad o una señal, o cuando se reemplaza una función virtual.

Cada clase GObject define una macro de la forma `NAMESPACE_CLASSNAME(object)`, que convierte la variable al tipo "`NamespaceClassname *`" y comprueba en tiempo de ejecución si la variable contiene correctamente un `NamespaceClassname` objeto o una subclase del mismo. Si la variable es `NULL` o contiene un objeto incompatible, la macro imprime un mensaje de advertencia crítico en la consola y devuelve NULL.

Un elenco estándar también funciona, pero la mayoría de las veces no se recomienda porque no hay comprobaciones de tiempo de ejecución:

```
GtkApplication *app;

/* No recomendado */
g_application_mark_busy ((GApplication *) app);
```

Otra macro útil cuando se usa un GObject es `NAMESPACE_IS_CLASSNAME(object)`, que devuelve `TRUE` si la variable es un objeto `NamespaceClassname` o una subclase del mismo.

4.3. Interfaces

Con GObject es posible crear interfaces. Una interfaz es solo una API, no contiene la implementación. Una clase GObject puede implementar una o varias interfaces. Si una clase GObject está documentada con GTK-Doc, la documentación contendrá una sección *Interfaces implementadas*.

Por ejemplo, GTK contiene la interfaz `GtkOrientable` que es implementada por muchos widgets y permite establecer la orientación: horizontal o vertical.

Las dos macros explicadas en la sección anterior también funcionan para interfaces. Un ejemplo con `GtkGrid`:

```
GtkWidget *vgrid;

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

Entonces, cuando busca una determinada característica en la API para una cierta clase de GObject, la característica se puede ubicar en tres lugares diferentes:

- En la propia clase GObject;
- En una de las clases padre en *Jerarquía de objetos*;

- O en una de las *Interfaces implementadas*.

4.4. Recuento de referencias

La gestión de la memoria de las clases de GObject se basa en *recuento de referencias*. Una clase GObject tiene un contador:

- Cuando se crea el objeto, el contador es igual a uno;
- `g_object_ref()` incrementa el contador;
- `g_object_unref()` disminuye el contador;
- Si el contador llega a cero, el objeto se libera.

Permite almacenar el GObject en varios lugares sin necesidad de coordinar cuándo liberar el objeto.

4.4.1. Evitar ciclos de referencia con referencias débiles

Si el objeto A hace referencia al objeto B y el objeto B hace referencia al objeto A, hay un ciclo de referencia y los dos objetos nunca se liberarán. Para evitar ese problema, existe el concepto de referencias "débiles". Al llamar a `g_object_ref()`, es una referencia "sólida". Entonces, en una dirección hay una referencia fuerte, y en la otra dirección debe haber una referencia débil (o ninguna referencia).

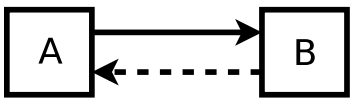


Figura 4.1: Usar una referencia débil para romper el ciclo de referencia entre A y B.

En la Figura 4.1 podemos ver que el objeto A tiene una fuerte referencia al objeto B, y el objeto B tiene una débil referencia al objeto A.

Se puede crear una referencia débil con `g_object_add_weak_pointer()` o `g_object_weak_ref()`. Al igual que con las referencias fuertes, es importante liberar la referencia cuando ya no se necesite, generalmente en el destructor de clases. Una referencia débil debe eliminarse con `g_object_remove_weak_pointer()` o `g_object_weak_unref()`. Entonces, en la Figura 4.1, el destructor de la clase B debe eliminar la referencia débil si aún no lo ha hecho.

4.4.2. Referencias flotantes

Cuando una clase GObject hereda de `GInitiallyUnowned` (que es el caso de `GtkWidget`), el objeto inicialmente tiene una referencia *floating*. Se debe llamar a `g_object_ref_sink()` para convertir esa referencia flotante en una referencia normal y fuerte.

Cuando un GObject hereda de `GInitiallyUnowned`, significa que GObject debe incluirse en algún tipo de contenedor. El contenedor luego asume la propiedad de la referencia flotante, llamando a `g_object_ref_sink()`. Permite simplificar el código, para eliminar la necesidad de llamar a `g_object_unref()` después de incluir el objeto en el contenedor.

El Listado 4.1 p. 68 muestra cómo se maneja la administración de memoria con un GObject normal. Compare esto con el Listado 4.2, que muestra cómo se maneja la administración de memoria con un GObject derivado de `GInitiallyUnowned`. La diferencia es que `g_object_unref()` no se llama en el último Listado, por lo que acorta el código.

```
/* GObject normal */

a_normal_gobject = normal_gobject_new ();
/* a_normal_gobject tiene ahora un recuento de referencia de 1. */

container_add (container, a_normal_gobject);
/* a_normal_gobject tiene ahora un recuento de referencia de 2. */

/* Ya no necesitamos un a_normal_gobject, por lo que lo anulamos. */
g_object_unref (a_normal_gobject);
/* a_normal_gobject tiene ahora un recuento de referencia de 1. */
```

Listado 4.1: Gestión de la memoria de GObjects normales.

```
/* Objeto GInitiallyUnowned, p. Ej. un GtkWidget */

widget = gtk_entry_new ();
/* widget tiene ahora solo una referencia flotante. */

gtk_container_add (container, widget);
/* El contenedor ha llamado a g_object_ref_sink(), tomando
 * posesion de la referencia flotante. El codigo esta
 * simplificado porque no debemos llamar a g_object_unref().
 */
```

Listado 4.2: Gestión de memoria de GObjects derivados de `GInitiallyUnowned`.

Entonces, es importante saber si un GObject hereda de `GInitiallyUnowned` o no. Para eso, debe mirar la *Object Hierarchy*, por ejemplo, `GtkEntry` tiene la siguiente jerarquía:

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

4.5. Señales y propiedades

Una clase GObject puede emitir señales. Con el bucle de eventos principal GLib (explicado anteriormente en la sección 2.3 p. 43), esta es la base para la programación dirigida por eventos. Un ejemplo de señal es cuando el usuario hace clic en un botón. La aplicación conecta una función de devolución de llamada a la señal para realizar la acción deseada cuando ocurre el evento.

Otro concepto de GObject son las *propiedades*, que está relacionado con las señales. Una propiedad es básicamente una variable de instancia coronada con una señal de `"notify"` que se emite cuando cambia su valor. Un buen ejemplo de una propiedad es el estado de un botón de verificación, es decir, un valor booleano que describe si el botón está actualmente marcado o no. Cuando el estado cambia, se envía la señal `"notify"`.

Para crear sus propias señales o propiedades, se debe crear una subclase GObject. Como se explicó en la introducción de este capítulo, esto está más allá del alcance de este libro, pero debe tener en cuenta que, por supuesto, es posible y recomendable crear sus propias señales o propiedades. De hecho, crear una señal o propiedad de GObject es una buena manera de implementar el patrón de diseño Observer [3]; es decir, uno o varios objetos *observando* cambios de estado de otro objeto, conectando devoluciones de llamada de función. El objeto que *emite* la señal no sabe qué objetos *reciben* la señal. GObject solo realiza un seguimiento de la lista de devoluciones de llamada para llamar. Entonces, agregar una señal permite desacoplar clases.

4.5.1. Conexión de una función de devolución de llamada a una señal

Para hacer las cosas más concretas, si observa la documentación de `GtkButton`, verá que proporciona la señal `"clicked"`. Para realizar la acción deseada cuando se emite la señal, una o más funciones de devolución de llamada deben estar conectadas de antemano.

Para conectar una devolución de llamada a una señal, se puede usar la función `g_signal_connect()`, o una de las otras funciones `g_signal_connect_*()`:

- `g_signal_connect()`
- `g_signal_connect_after()`
- `g_signal_connect_swapped()`
- `g_signal_connect_data()`
- `g_signal_connect_object()`
- Y algunas más avanzadas.

El Listado 4.3 p. 70 muestra el prototipo de la señal `GtkButton::clicked` [1].

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

Listado 4.3: El prototipo de la señal `GtkButton::clicked`.

Cuando se usa `g_signal_connect()`, la función de devolución de llamada debe tener el mismo prototipo que el prototipo de señal. Muchas señales tienen más argumentos y algunas señales devuelven un valor. Si la devolución de llamada tiene un prototipo incompatible, sucederán cosas malas, habrá errores o bloqueos aleatorios.

El Listado 4.4 p. 70 muestra un ejemplo de cómo usar `g_signal_connect()`.

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

  g_message ("Boton pulsado!");
}

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

  /* Crea el boton */
  /* ... */
```

[1] La convención cuando se hace referencia a una señal de GObject es "`ClassName::signal-name`". Así es como se documenta con los comentarios de GTK-Doc.

```
  /* Conectar la funcion de devolucion de llamada */
  g_signal_connect (button,
                    "clicked",
                    G_CALLBACK (button_clicked_cb),
                    my_class);
}
```

Listado 4.4: Cómo conectarse a una señal

La macro `G_CALLBACK()` es necesaria porque `g_signal_connect()` es genérica: se puede usar para conectarse a cualquier señal de cualquier clase de GObject, por lo que el puntero de función debe ser convertido.

Hay dos convenciones principales para nombrar funciones de devolución de llamada:

- Termine el nombre de la función con "`cb`", atajo de "callback". Por ejemplo: `button_clicked_cb()` como en el ejemplo de código anterior.
- Inicie el nombre de la función con "`on`". Por ejemplo: `on_button_clicked()`.

Con una de esas convenciones de nomenclatura — y con el parámetro `gpointer user_data`, que siempre es el último parámetro — es fácil reconocer que una función es una devolución de llamada.

El lenguaje C permite escribir una firma de función de devolución de llamada diferente — pero compatible —, aunque no se considera universalmente como algo bueno:

- Uno o más de los *últimos* argumentos de la función se pueden omitir si no se utilizan. Pero como se explicó anteriormente, el argumento `gpointer user_data` permite reconocer fácilmente que la función es efectivamente una devolución de llamada [2].
- Los tipos de argumentos se pueden modificar a un tipo compatible: p. Ej. otra clase en la jerarquía de herencia, o en el ejemplo anterior, reemplazando "`gpointer`" por "`MyClass *`" (pero hacer eso hace que el código sea un poco menos robusto porque no se llama a la macro `MY_CLASS()`).

4.5.2. Desconexión de controladores de señales

En el Listado 4.4, se llama a `button_clicked_cb()` cada vez que el objeto `button` emite la señal `"clicked"`. Si el objeto `button` todavía está vivo después de que `my_class` se haya liberado, cuando la señal se emita nuevamente habrá un pequeño problema ... Entonces, en

[2] Al igual que con los lenguajes naturales, la redundancia permite comprender mejor y más rápidamente lo que leemos o escuchamos.

el destructor de `MyClass`, el manejador de señales (es decir, la devolución de llamada) debe desconectarse. ¿Como hacer eso?

Las funciones `g_signal_connect*()` en realidad devuelven un ID del manejador de señales, como un entero `gulong` siempre mayor que 0 (para conexiones exitosas). Al almacenar ese ID, es posible desconectar ese manejador de señal específico con la función `g_signal_handler_disconnect()`.

A veces también queremos desconectar el controlador de señales simplemente porque ya no estamos interesados en el evento.

El Listado 4.5 muestra un ejemplo completo de cómo desconectar un manejador de señales cuando se libera su argumento `user_data`. Volvemos a un ejemplo con un corrector ortográfico, porque el ejemplo con el botón GTK no se ajusta bien a la situación [3].

El argumento de devolución de llamada `user_data` es una instancia de `MyTextView`, con `MyTextView` implementado con un estilo semi-OOP. Dado que el objeto del corrector ortográfico puede vivir más tiempo que la instancia de `MyTextView`, la señal debe desconectarse en el destructor `MyTextView`.

```
#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;
```

[3]La mayoría de las veces, un widget GTK no vive más tiempo que el contenedor al que se agrega, y el El objeto que escucha la señal del widget suele ser el propio contenedor. Entonces, si el widget muere al mismo tiempo que el contenedor, no es posible que el widget envíe una señal mientras su contenedor ya ha sido destruido. En ese caso, no tiene sentido desconectar la señal en el destructor del contenedor, ya que en ese punto el widget ya está liberado; y es más difícil para un objeto muerto enviar una señal [4].

[4]Cuando digo que es más difícil, en realidad es imposible, por supuesto.

```
  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().
   */
  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);
}
```

Listado 4.5: Desconectar un manejador de señales cuando se libera su argumento `user_data`.

En realidad, hay otras funciones de `g_signal_handler*()` que permiten desconectar los manejadores de señales:

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

Habría sido posible utilizar una de las funciones anteriores en el Listado 4.5, y habría evitado la necesidad de almacenar `word_added_to_personal_handler_id`. La función básica `g_signal_handler_disconnect()` se ha utilizado con fines de aprendizaje.

Tenga en cuenta también que si `MyTextView` fuera una clase GObject, habría sido posible conectarse a la señal del corrector ortográfico con `g_signal_connect_object()`, y habría

eliminado por completo la necesidad de desconectar manualmente el controlador de señal en el destructor `MyTextView`. Una (pequeña) razón más para aprender a crear subclases de GObject.

4.5.3. Propiedades

Si ha mirado la referencia de la API GTK o GIO, debe haber notado que algunas clases de GObject tienen una o más *propiedades*. Una propiedad es como una variable de instancia, también llamada "attribute", pero es diferente en el contexto de GObject porque tiene propiedades interesantes adicionales [5].

Como se dijo anteriormente al comienzo de la sección 4.5 p. 69, un buen ejemplo de una propiedad es el estado de un botón de verificación, es decir, un valor booleano que describe si el botón está actualmente marcado o no. Si ya conoce un poco de GTK, es posible que haya descubierto que hay un botón de verificación disponible con el widget `GtkCheckButton`. Pero había una pequeña trampa si quería encontrar la propiedad de la que estaba hablando, porque la propiedad está implementada en la clase padre `GtkToggleButton`: la propiedad `GtkToggleButton:active` [6].

La señal `notify`

El atributo principal [7] De una propiedad — además de representar un valor— es que la clase GObject emite la señal `GObject::notify` cuando cambia el valor de una propiedad.

Hay un concepto de señales GObject que aún no se ha explicado y se usa con la señal `GObject::notify`: cuando se emite una señal, se puede proporcionar un *detail*. En el caso de la señal de notificación, el detalle es el nombre de la propiedad cuyo valor ha cambiado. Dado que solo hay una señal para todas las propiedades, gracias al *detail* es posible conectar una devolución de llamada para recibir una notificación solo cuando una determinada propiedad haya cambiado. Si el *detail* no se proporciona al conectar la devolución de llamada, la devolución de llamada se llamará cuando *cualquiera* de las propiedades del objeto cambie, lo que generalmente no es lo que se desea.

Será más claro con un ejemplo. El Listado 4.6 p. 76 muestra cómo conectarse a la señal de notificación. Tenga en cuenta que en lugar de conectarse a la señal detallada `"notify::active "`, en realidad es más conveniente utilizar la señal `GtkToggleButton::toggled`. Hay mejores

[5] Pun intencionado.

[6] De la misma manera como las señales se documentan con GTK-Doc como "`ClassName::signal-name`", las propiedades se documentan como "`ClassName:property-name`".

[7] Pun también intencionado.

casos de uso en el mundo real en los que es necesario conectarse a la señal de notificación, pero al menos el Listado 4.6 es, con suerte, comprensible con solo un conocimiento limitado de GTK (y si mira la documentación en Devhelp en paralelo).

```
/* 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;
}
```

Listado 4.6: Conexión a la señal de notificación para escuchar cambios en la propiedad.

Vinculaciones de propiedad

Otro aspecto útil de las propiedades es que dos propiedades se pueden vincular fácilmente: cuando una propiedad cambia, la otra se actualiza para tener el mismo valor. Lo mismo se puede lograr con la señal `"notify"`, pero existen funciones de nivel superior.

Se pueden vincular dos propiedades de varias formas con una de las funciones `g_object_bind_property *()`. El Listado 4.7 muestra una implementación más simple del Listado 4.6; el código es equivalente, pero usa la función `g_object_bind_property()`.

```
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;
```

```
}
```

Listado 4.7: Vinculando dos propiedades.

Parte III

GTK

Capítulo 5

Ejemplo de una arquitectura de código de aplicación GTK

Este capítulo no está completamente terminado, puede ser difícil de entender sin ningún conocimiento sobre GTK u otro conjunto de herramientas de widgets. Y el capítulo no está bien integrado con los otros capítulos, la introducción y los capítulos de lectura adicional no se han adaptado, por ejemplo. Este capítulo se distribuye con la esperanza de que sea útil, pero SIN NINGUNA GARANTÍA; incluso sin la garantía implícita de COMERCIABILIDAD o APTITUD PARA UN PROPÓSITO EN PARTICULAR.

Para cualquier proyecto de programación, es importante diseñar correctamente la arquitectura de código general. Con Programación Orientada a Objetos, significa definir las clases principales. Este capítulo explica un buen ejemplo de una arquitectura de código para una aplicación GTK, observando una vista simplificada del editor de texto gedit [1].

gedit tiene una interfaz de documento con pestañas: se pueden abrir varios archivos en la misma ventana de gedit, en diferentes pestañas. Como veremos, esto se refleja en la arquitectura del código.

La Figura 5.1 p. 81 muestra el esquema de la clase. Cada clase gedit en el esquema es una subclase de una clase GTK o GtkSourceView. (GtkSourceView [2] es una biblioteca que extiende el widget `GtkTextView`; `GtkTextView` es parte de GTK).

Repasaremos el esquema de la clase, explicando las clases paso a paso, comenzando por la parte superior. Esto permitirá presentar algunas de las clases GTK más importantes, no

[1] `https://wiki.gnome.org/Apps/Gedit`
[2] `https://wiki.gnome.org/Projects/GtkSourceView`

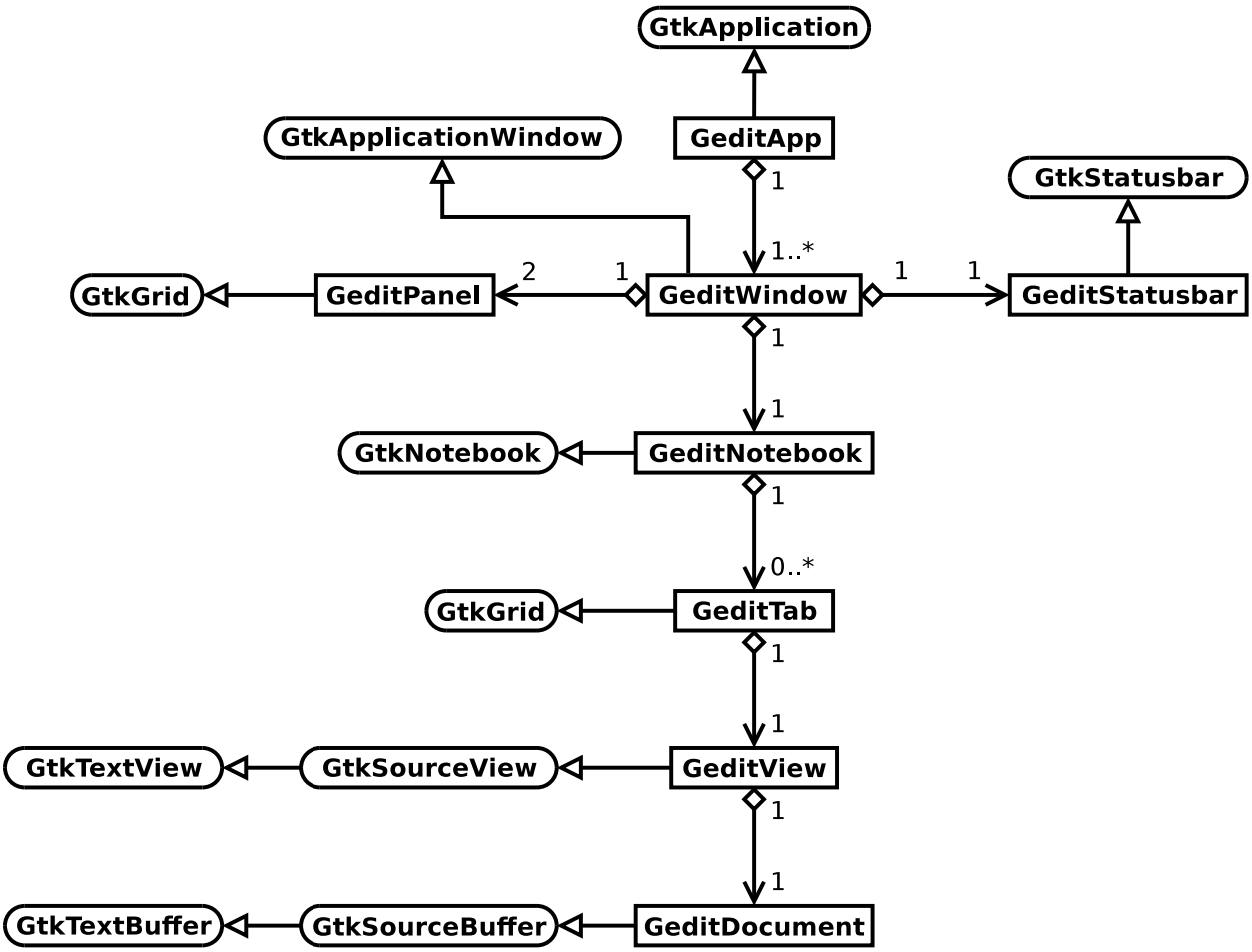


Figura 5.1: Arquitectura de código simplificada del editor de texto gedit

describiéndolas en detalle con muchos ejemplos de código, sino dando una descripción general de alto nivel.

5.1. La función main() y GeditApp

Aunque no está representado en el esquema, el punto de entrada de una aplicación GTK – como para todos los programas en C– es la función `main()`. Para crear una aplicación GTK, lo principal que se debe hacer en `main()` es crear una instancia de `GtkApplication`, o una subclase de la misma. En el esquema vemos que `GeditApp` es una subclase de `GtkApplication`, por lo que la función `main()` de gedit crea un objeto `GeditApp`.

`GtkApplication` es la clase que contiene y representa la aplicación completa. Por lo general, solo hay una instancia de `GtkApplication` por proceso, por lo que puede considerarse una clase singleton. Lo que contiene `GtkApplication` son *windows*, por ejemplo, `GeditWindow` en el caso

de gedit, además de otros tipos de ventanas como ventanas de diálogo.

Ya vimos la jerarquía de clases `GtkApplication` en la sección 4.1 p. 65 al explicar la herencia POO con GObject:

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

`GApplication` es parte de la biblioteca GIO e implementa las funciones que no están relacionadas con la interfaz gráfica de usuario (GUI). Entonces, para un programa que se ejecuta en la terminal, es posible usar solo `GApplication`.

Una característica importante que proporciona `GApplication` es la unicidad del proceso (pero puede desactivarse si no se desea). Lo que hace la unicidad del proceso es tener solo un proceso por aplicación por sesión de usuario. Para que esa característica funcione, se debe proporcionar un ID de aplicación al crear el objeto `GApplication`. Con ese ID, `GApplication` busca si otro proceso ya ejecuta la misma aplicación en la misma sesión de usuario; si es el caso, comunica a la instancia primaria las acciones que deben realizarse (por ejemplo, abrir una nueva ventana, o abrir un nuevo archivo en una ventana existente, etc.). Cuando las acciones se realizan en la instancia principal, el segundo proceso finaliza inmediatamente. En Linux, `GApplication` utiliza el sistema de comunicación entre procesos (IPC) D-Bus para comunicarse entre los dos procesos.

La singularidad del proceso tiene varias ventajas, por dar algunos ejemplos concretos:

- Para una aplicación con una interfaz de documento con pestañas, al hacer clic en un archivo en un administrador de archivos como Nautilus, el archivo se puede abrir en una nueva pestaña en lugar de crear cada vez una nueva ventana. Para que esto funcione, la comunicación entre procesos es necesaria de una forma u otra;

- Una aplicación no necesita sincronizar explícitamente su estado y datos entre diferentes procesos. En aras del argumento, digamos que en gedit el usuario puede crear “ herramientas de compilación ” personalizadas para compilar el archivo o proyecto actual. gedit guarda las herramientas de construcción personalizadas en un archivo XML y se muestran en el menú para ejecutar sus comandos. En Linux, el archivo XML se guarda, por ejemplo, en el directorio `~/.local/share/` del usuario. Sin la unicidad del proceso, si un proceso gedit modifica las herramientas de compilación personalizadas, los otros procesos gedit deben volver a cargar el archivo XML y deben asegurarse de que no haya carreras (dos procesos gedit diferentes no deben modificar el archivo XML al

mismo tiempo) . Con la unicidad del proceso, ese problema no existe, todas las ventanas gedit comparten el mismo estado de la aplicación, y el desarrollador puede asumir que solo un proceso por usuario puede modificar el archivo XML [3] (Por supuesto, el usuario todavía tiene la posibilidad de editar el archivo XML a mano, pero en ese caso la aplicación se puede reiniciar, normalmente se espera que el usuario modifique las herramientas de compilación de la GUI que proporciona gedit).

Otra característica importante de `GApplication` es ejecutar el ciclo de eventos principal. El bucle de eventos principal de GLib se describió en la sección 2.3 p. 43. Con `GApplication`, esto se hace con la función `g_application_run()`. Una versión minimalista de la función `main()` en gedit se vería así:

```
int
main (int    argc,
      char **argv)
{
  GeditApp *app;
  int status;

  /* Init i18n (internationalization) here. */

  app = gedit_app_new ();
  status = g_application_run (G_APPLICATION (app), argc, argv);
  g_object_unref (app);

  return status;
}
```

Lo que hace `GeditApp` es básicamente lo que debería hacerse en `main()` si no hubiera una subclase `GtkApplication`. Esto incluye:

- Configurando correctamente el objeto `GtkApplication`, por ejemplo, dando el ID de la aplicación;

[3] Tenga en cuenta que esto no sería cierto si fuera posible abrir *varias* sesiones gráficas para el mismo usuario, en la misma máquina (con soporte para múltiples puestos) o al menos compartir el almacenamiento de respaldo para el directorio de inicio (por ejemplo, con montajes NFS). Pero GNOME y la mayoría de las aplicaciones no admiten esto, un usuario puede abrir como máximo una sesión gráfica a la vez para el mismo directorio de inicio. Para los inicios de sesión en la misma máquina física, esto se aplica mediante GDM (el administrador de pantalla GNOME y la pantalla de inicio de sesión) y D-Bus. En el caso de montajes NFS, esto no se aplica, pero si el mismo usuario abre varias sesiones gráficas en diferentes equipos, es posible que algunos programas no funcionen correctamente. Entonces, aunque la unicidad del proceso de `GApplication` está documentada como por *sesión de usuario*, en la práctica podemos decir que es simplemente por *usuario*.

- Conectando devoluciones de llamada a algunas señales [4], por ejemplo para crear una `GeditWindow` cuando sea necesario;
- Implementando `GAction` en toda la aplicación. `GAction` es una parte de clase de GIO que representa una acción que el usuario puede desencadenar. Una acción en toda la aplicación es, por ejemplo, salir de la aplicación o abrir el cuadro de diálogo de preferencias (porque las preferencias se aplican a toda la aplicación).

Cuando comienzas a escribir una nueva aplicación GTK, no ves directamente la necesidad de una subclase `GtkApplication`, ya que el código en `main()`, más las devoluciones de llamada, aún son pequeñas. Pero cuando se agregan más y más características, es una buena idea en algún momento mover el código a una subclase `GtkApplication`. O para crear una subclase directamente. Una subclase es especialmente útil cuando surge la necesidad de almacenar datos adicionales.

5.2. GeditWindow

`GeditWindow` es una subclase de `GtkApplicationWindow`. Y no lo vemos en el esquema, pero `GtkApplicationWindow` es una subclase de `GtkWindow`, que es un widget de nivel superior. Un widget de nivel superior no puede incluirse en otro widget. Una `GtkApplicationWindow` está contenida en una `GtkApplication`, pero `GtkApplication` no es una subclase de `GtkWidget`.

En el esquema, la notación "`1`" y "`1..*` " significa que un objeto `GeditApp` *contiene* uno o varios `GeditWindow` objetos, y que una `GeditWindow` está contenida exactamente en una `GeditApp` (una `GeditWindow` no puede estar contenida en varios objetos `GeditApp`, de todos modos solo hay una instancia de `GeditApp` por proceso).

`GeditWindow` es responsable de crear la interfaz de usuario principal, creando otros widgets y ensamblándolos en un contenedor `GtkGrid`, por ejemplo. Otra cosa que hace `GeditWindow` es implementar las `GActions` que tienen un efecto solo en la ventana actual, por ejemplo, una acción para cerrar la ventana o guardar el documento actual. Al implementar una `GAction`, `GeditWindow` puede, por supuesto, delegar la mayor parte de su trabajo a otras clases contenidas en `GeditWindow`.

En la parte superior de la ventana principal de una aplicación, generalmente hay una `GtkHeaderBar`, que muestra el título de la ventana, algunos botones y un menú de "hambur-

[4]Pero tenga en cuenta que en una subclase de GObject, en lugar de conectar devoluciones de llamada a señales de una clase principal con p. Ej. `g_signal_connect()`, es mejor anular las funciones virtuales en su lugar.

guesa". Alternativamente, una aplicación puede tener una barra de menú y una barra de herramientas tradicionales.

Además de la barra de encabezado, `GeditWindow` crea un widget `GeditStatusbar` y lo agrega al final de la ventana. También crea dos `GeditPanels`, uno en el lado izquierdo de la ventana y el otro en la parte inferior, encima de la `GeditStatusbar`. Cada panel puede contener varios elementos. Por ejemplo, el panel lateral contiene un explorador de archivos integrado y el panel inferior puede contener una terminal, entre otras cosas [5].

`GeditWindow` también crea un `GeditNotebook`, la parte principal de la ventana.

5.3. GeditNotebook y lo que contiene

`GeditNotebook` es una subclase de `GtkNotebook`, que es el widget que muestra pestañas y también contiene su contenido.

En el esquema de la clase, podemos ver que el contenido de una pestaña es un widget `GeditTab`, una subclase de `GtkGrid`. El elemento principal dentro de una `GeditTab` es `GeditView`. Más precisamente — se omitió en el esquema por concisión — el `GeditView` está contenido en una `GtkScrolledWindow` que está contenido en el `GeditTab`. Pero `GeditTab` puede contener otros widgets, por ejemplo, barras de información en la parte superior del documento.

`GeditView` es una subclase de `GtkSourceView`, que en sí misma es una subclase de `GtkTextView` . `GtkTextView` — que es parte de GTK — es la base para un editor de texto multilínea. La biblioteca GtkSourceView agrega características útiles para el código fuente, como el resaltado de sintaxis. `GtkTextView` sigue un patrón Modelo-Vista-Controlador. `GtkTextBuffer` es el modelo, es decir, contiene los datos.

5.4. ¿Por qué y cuándo crear subclases de widgets GTK?

Si buscamos, por ejemplo, en `GeditTab`, contiene — por composición — a `GeditView`. `GeditView` es una subclase de `GtkSourceView`. En su lugar, `GeditTab` podría usar por composición directamente un objeto `GtkSourceView` y mover el código de `GeditView` a `GeditTab`. Pero, por lo general, sucede o debería ocurrir exactamente lo contrario.

[5] El código gedit actual en realidad ya no contiene una clase `GeditPanel`, pero fue el caso en una versión anterior. Se agregó `GeditPanel` al diagrama para mostrar una posible implementación de paneles en una aplicación. Si su aplicación contiene solo un elemento en un panel, no es necesario tener una clase `Panel`, puede agregar directamente el elemento a la ventana.

Cuando la base de código de una aplicación GTK aún es pequeña, por ejemplo, si comienza a escribir un equivalente de `GeditTab`, puede crear un objeto `GtkSourceView` directamente en `GeditTab` y almacenar la `GtkSourceView` objeto en una variable de instancia. Luego, al implementar nuevas características, agrega nuevas funciones que usan casi exclusivamente la variable de instancia `GtkSourceView`. Incluso puede tener funciones `static` que toman directamente un argumento `GtkSourceView` en lugar del parámetro `GeditTab` *self*. También puede almacenar datos adicionales útiles solo para las funciones relacionadas con `GtkSourceView`. Si la clase `GeditTab` aún es pequeña (por ejemplo, 500 líneas de código) y no contiene muchas variables de instancia, no hay problema. Por otro lado, si la clase `GeditTab` se vuelve más grande (por ejemplo, más de 2000 líneas de código), entonces probablemente sea una señal de que la clase debería delegar parte de su trabajo a una nueva clase; en nuestro caso, `GeditView`. Tenga en cuenta que 2000 líneas de código para una clase pueden estar bien, no hay un límite claro sobre cuándo se debe dividir una clase. Pero si la clase `GeditView` resultante contendría al menos varios cientos de código no estándar, probablemente sea una buena idea hacer la refactorización.

De lo que se trata OOP es de empaquetar datos y comportamiento juntos, y delegar parte del trabajo a otras clases. La herencia de clases tiene sentido cuando queremos agregar más comportamiento a una clase existente, con posibles datos adicionales relacionados con el comportamiento agregado. `GeditView` es una subclase de `GtkSourceView` porque `GeditView` *es a* `GtkSourceView`; es decir, `GeditView` opera con los mismos datos base que `GtkSourceView`. Además, permite a `GeditTab` delegar parte de su trabajo, con el objetivo de tener clases más pequeñas y manejables. Más pequeño de dos formas: menos código y menos variables de instancia.

Por lo tanto, durante la vida útil de una aplicación GTK, el programador a menudo necesita refactorizar el código, crear nuevas clases y delegar más trabajo. Lo contrario puede suceder cuando el código de la aplicación se mueve a la biblioteca subyacente; por ejemplo, si todas las características de `GeditView` se agregan a la clase `GtkSourceView`; en ese caso, la subclase `GeditView` ya no tiene sentido.

5.5. Widgets compuestos

Los widgets compuestos son contenedores que ya contienen una colección útil de widgets secundarios en un paquete agradable. Implementar un widget compuesto es fácil [6], solo necesita:

[6] Una vez que sepa cómo subclasificar una clase GObject.

1. Subclase un contenedor como `GtkGrid` o `GtkBin` o `GtkWindow`;

2. En el constructor de la clase, cree los widgets secundarios y agréguelos al contenedor.

En el esquema de la clase gedit, los widgets compuestos son las subclases de `GtkGrid` (`GeditPanel` y `GeditTab`) y `GeditWindow`.

`GeditWindow` es una subclase indirecta de `GtkBin`, por lo que puede contener como máximo un widget hijo. Es por eso que `GeditWindow` usa un `GtkGrid` como su widget hijo, de modo que `GtkGrid` puede contener a su vez todos los elementos de la ventana.

Por defecto, una `GeditTab` tiene solo un widget secundario, la `GtkScrolledWindow` que contiene la `GeditView`. Pero `GeditTab` tiene una función para agregar un `GtkInfoBar` en la parte superior, mostrando por ejemplo un mensaje de error.

Entonces, mientras `GtkGrid` es un contenedor de uso general que inicialmente no contiene ningún widget secundario, un widget compuesto es un contenedor especializado que ya contiene widgets secundarios específicos. Escribir widgets compuestos es una forma conveniente de codificar aplicaciones.

Parte IV

Lectura adicional

Capítulo 6

Lecturas adicionales

En este punto, debe conocer los conceptos básicos de GLib core y GObject. No necesitas saber *todo* sobre GLib core y GObject para continuar, pero tener al menos un conocimiento básico te permitirá aprender más fácilmente GTK y GIO, o cualquier otra biblioteca basada en GObject.

6.1. GTK y GIO

GTK y GIO se pueden aprender en paralelo.

Debería poder usar cualquier clase de GObject en GIO, solo lea la descripción de la clase y hojee la lista de funciones para tener una descripción general de las características que proporciona una clase. Entre otras cosas interesantes, GIO incluye:

- `GFile` para manejar archivos y directorios.
- `GSettings` para almacenar la configuración de la aplicación.
- `GDBus`: una API de alto nivel para el sistema de comunicación entre procesos D-Bus.
- `GSubprocess` para iniciar procesos secundarios y comunicarse con ellos de forma asincrónica.
- `GCancellable`, `GAsyncResult` y `GTask` para usar o implementar tareas asincrónicas y cancelables.
- Muchas otras funciones, como flujos de E/S, soporte de red o soporte de aplicaciones.

Para crear aplicaciones gráficas con GTK, no se preocupe, la documentación de referencia

tiene una guía de introducción, disponible con Devhelp o en línea en:
`https://developer.gnome.org/gtk3/stable/`

Después de leer la guía de introducción, lea toda la referencia de la API para familiarizarse con los widgets, contenedores y clases base disponibles. Algunos widgets tienen una API bastante grande, por lo que también están disponibles algunos tutoriales externos, por ejemplo, para `GtkTextView` y `GtkTreeView`. Consulte la página de documentación en:
`http://www.gtk.org`

También hay una serie de pequeños tutoriales sobre varios temas GLib/GTK:
`https://wiki.gnome.org/HowDoI`

6.2. Escribir sus propias clases de GObject

El capítulo 4 explica cómo *usar* una clase GObject existente, que es muy útil para aprender GTK, pero no explica cómo *crear* tus propias clases GObject. Escribir sus propias clases de GObject permite contar con referencias, puede crear sus propias propiedades y señales, puede implementar interfaces, anular funciones virtuales (si la función virtual no está asociada a una señal), etc.

Como se explicó al principio del capítulo 4, si desea obtener información más detallada sobre GObject y saber cómo crear subclases, la documentación de referencia de GObject contiene capítulos introductorios: "*Concepts*" y "*Tutorial*", disponibles como de costumbre en Devhelp o en línea en:
`https://developer.gnome.org/gobject/stable/`

6.3. Sistema de compilación

Un Makefile básico generalmente no es suficiente si desea instalar su aplicación en diferentes sistemas. Por tanto, se necesita una solución más sofisticada. Para un programa basado en GLib/GTK, existen dos alternativas principales: Autotools y Meson.

GNOME y GTK históricamente usan Autotools, pero a partir de 2017 algunos módulos (incluido GTK) están migrando a Meson. Para un nuevo proyecto, se puede recomendar Meson.

6.3.1. Las Autotools

Las Autotools comprenden tres componentes principales: Autoconf, Automake y Libtool. Está basado en scripts de shell, macros m4 y `make`.

Las macros están disponibles para varios propósitos (la documentación del usuario, estadísticas de cobertura de código para pruebas unitarias, etc.). El libro más reciente sobre el tema es *Autotools*, de John Calcote [7].

Pero Autotools tiene la reputación de ser difícil de aprender.

6.3.2. Meson

Meson es un sistema de construcción bastante nuevo, es más fácil de aprender que Autotools y también resulta en construcciones más rápidas. Algunos módulos de GNOME ya usan Meson. Consulte el sitio web para obtener más información:
`http://mesonbuild.com/`

6.4. Mejores prácticas de programación

Se recomienda seguir las Pautas de programación de GNOME [11].

La siguiente lista no tiene nada que ver con el desarrollo de GLib/GTK, pero es útil para cualquier proyecto de programación. Después de tener algo de práctica, es interesante aprender más sobre las *mejores* prácticas de programación. Escribir código de buena calidad es importante para prevenir errores y para mantener una pieza de software a largo plazo.

- *El* libro sobre las mejores prácticas de programación es *Code Complete*, de Steve McConnell [8]. Muy recomendable [1].
- Para obtener pautas sobre POO específicamente, consulte *Heurística de diseño orientado a objetos*, de Arthur Riel [2].
- Una excelente fuente de información es la web de Martin Fowler: refactorización, metodología ágil, diseño de código, ...
  `http://martinfowler.com/`

[1] Aunque el editor de *Código completo* es Microsoft Press, el libro no está relacionado con Microsoft o Windows. El autor a veces explica cosas relacionadas con el código abierto, UNIX y Linux, pero uno puede lamentar la ausencia total de la mención "software libre/free" y todos los beneficios de la libertad, en particular para este tipo de libros: poder aprender leyendo el código de otros. Pero si está aquí, es de esperar que ya sepa todo esto.

Más relacionados con GNOME, los artículos de Havoc Pennington tienen buenos consejos que vale la pena leer, incluidos "*Trabajando en software libre*", "*Interfaz de usuario de software libre*" y "*Mantenimiento de software libre : Adición de funciones*":
`http://ometer.com/writing.html`

Bibliografía

[1] Brian Kernighan y Dennis Ritchie (1988).
The C Programming Language (2da ed.). Prentice Hall.

[2] Arthur Riel (1996).
Object-Oriented Design Heuristics (1ra ed.). Addison-Wesley.

[3] Gamma E., Helm R., Johnson R. y Vlissides J. (1994).
Design Patterns: Elements of Reusable Object-Oriented Software (1ra ed.). Addison-Wesley Professional.

[4] Steven Skiena (2008).
The Algorithm Design Manual (2da ed.). Springer.

[5] Paul Abrahams (1995).
UNIX for the Impatient (2da ed.). Addison-Wesley.

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

[7] John Calcote (2010).
Autotools: A Practitioner's Guide to GNU Autoconf, Automake, and Libtool (1ra ed.). No Starch Press.

[8] Steve McConnell (2004).
Code Complete: A practical handbook of software construction (2da ed.). Microsoft Press.

[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/