# Programación en el entorno GNOME


**Rodrigo Moya**

rodrigo@gnome-db.org

**Álvaro del Castillo**

acs@barrapunto.com

**Roberto Majadas**

phoenix@nova.es

**Gaspar Oriol**

gaspar.oriol@hispalinux.es

**Gregorio Robles**

grex@scouts-es.org

**Yolanda Moreno**

ttacunymoreno@yahoo.es

**Ayose Cazorla**

ayose.cazorla@hispalinux.es

**Germán Poo-Caamaño**

gpoo@ubiobio.cl

**Alejandro Sánchez Acosta**

**raciel@es.gnu.org**

**Roberto Pérez Cubero**

**hylian@jazzfree.com**

**Carlos Garnacho**

**garnacho@tuxerver.net**

**Juanan Pereira**

**chessy@diariolinux.com**

**Antonio Santiago**

**xxx**

**Alejandro Valdés**

**avaldes@utalca.cl**

**Claudio Saavedra V.**

**csaavedra@alumnos.utalca.cl**

**Programación en el entorno GNOME**

por Rodrigo Moya, Álvaro del Castillo, Roberto Majadas, Gaspar Oriol, Gregorio Robles, Yolanda Moreno, Ayose Cazorla, Germán Poo-Caamaño, Alejandro Sánchez Acosta, Roberto Pérez Cubero, Carlos Garnacho, Juanan Pereira, Antonio Santiago, Alejandro Valdés, y Claudio Saavedra V.

Copyright © 2002 The GNOME Foundation

Se otorga permiso para copiar, distribuir y/o modificar este documento bajo los términos de la Licencia de Documentación Libre GNU, versión 1.1 o cualquier versión posterior publicada por la Free Software Foundation. No hay Secciones Invariantes ni Textos de Portada o Contraportada. Puedes consultar una copia de la licencia en http://www.gnu.org/copyleft/fdl.html.

# Tabla de contenidos

**1. Introducción**........1
El proyecto GNOME........1
Historia de GNOME.........1
Organización de GNOME: La Fundación GNOME.........2
El consejo directivo.........3
El consejo consultor.........3
Herramientas y servicios de comunicación en el desarrollo.........3
Sistema de control de versiones.........3
`Bugzilla`.........5
Listas de correo.........5
Internet Relay Chat (IRC).........6
GNOME Hispano.........6
Historia de GNOME Hispano.........6
Servicios y recursos.........6
Obtención de una cuenta de CVS.........6
GUNIH.........7
Otros proyectos.........8
**2. Preparación del entorno**.........9
Introducción.........9
Estructura de un proyecto GNU.........9
Archivos de información de un proyecto.........10
Archivos de programas y configuración.........11
Las herramientas a través de un ejemplo.........12
aclocal.........12
autoheader.........13
autoconf.........14
automake.........15
El programa.........16
Finalizando la configuración.........16
**3. Internacionalización**.........19
Internacionalización de aplicaciones.........19
intltool.........19
**4. GLib**.........23
Tipos de datos de GLib.........23
Mensajes de salida.........25
Mensajes de salida.........25
Funciones de depuración.........26
Funciones de registro.........29
Trabajar con cadenas.........31
Manipular el contenido de una cadena.........31
GString : la otra manera de ver una cadena.........32
Jugando con el tiempo.........38
Funciones de manejo de fechas y horas.........38
Midiendo intervalos de tiempo con GTimer.........39
Miscelánea de funciones.........42
Números aleatorios.........42
Funciones de información sobre entorno.........44
Bucles de ejecución.........45
Alarmas.........46
Tiempos de inactividad.........46
Tratamiento de ficheros y canales de entrada/salida.........47
Obtención de un GIOChannel.........48
Generalidades en el trabajo con GIOChannels.........48
Operaciones básicas.........49
Integración de canales al bucle de eventos.........51
Configuración avanzada de canales.........52

Manejo de memoria dinámica. ....54
Reserva de memoria. ....54
Liberación de memoria. ....54
Realojamiento de memoria. ....55
Estructuras de datos: listas enlazadas, pilas y colas. ....55
Listas enlazadas. ....55
Listas doblemente enlazadas. ....59
GQueue: pilas y colas. ....62
Estructuras de datos avanzadas. ....67
Tablas de dispersión. ....67
Arboles binarios balanceados ....70
GNode : Arboles de orden n. ....72
Caches. ....75
GLib avanzado. ....78
Hilos en Glib. ....78
UTF-8: las letras del mundo. ....80
Como hacer *plugins*. ....82
**5. Sistema de objetos de GLib. ....89**
Gestión dinámica de tipos. ....89
Tipos basados en clases (objetos). ....89
Tipos no instanciables (fundamentales). ....91
Implementación de nuevos tipos. ....92
Interfaces ....92
Herencia ....94
Señales. ....95
GValue ....95
GObject, la clase base. ....96
Parámetros y valores. ....96
**6. Pango ....97**
Caminando hacia Pango ....97
Pango, a modo de introducción ....97
Arquitectura ....97
El API de Pango e Implementación ....97
Pango en GTK+2.0 ....97
El futuro de Pango ....97
Referencias ....97
**7. GTK+ ....99**
Qué es un widget ....99
Widgets básicos ....99
Bucle de ejecución y eventos ....100
Señales ....101
Ejemplo básico ....102
El ejemplo paso a paso ....103
Cómo compilar el ejemplo ....106
Contenedores ....106
Cajas. ....107
Cajas de botones ....114
Tablas ....117
GtkNotebook ....121
GtkAlignment ....126
GtkHPaned/GtkVPaned ....126
GtkLayout ....127
Colocación por coordenadas ....128
Marcos ....131
GtkAspectFrame ....131
GtkViewport ....133
Ventanas. ....133
Ventanas (GtkWindow). ....134

Diálogos (GtkDialog) ....................................................................................134
Ventanas de mensaje (GtkMessageDialog) .....................................................137
Ventanas invisibles (GtkInvisible)..................................................................137
Ventanas embebidas (GtkPlug).......................................................................138
Grupos de ventanas (GtkWindowGroup)......................................................139
Visualización y Entrada de información.............................................................139
Etiquetas...............................................................................................................139
Entrada de datos .................................................................................................142
Ajustes ..................................................................................................................144
Widgets de selección de rango.........................................................................147
Imágenes (GtkImage)..........................................................................................163
Barras de progreso (GtkProgressBar) ..............................................................165
Barras de estado (GtkStatusBar).......................................................................170
Botones........................................................................................................................172
GtkButton.............................................................................................................172
GtkToggleButton..................................................................................................173
GtkCheckButton ..................................................................................................174
GtkRadioButton...................................................................................................174
Menús y barras de herramientas.........................................................................175
Menús ...................................................................................................................175
GtkToolbar ...........................................................................................................182
GtkCombo.............................................................................................................187
Ventanas de selección .............................................................................................190
GtkFileSelection ..................................................................................................190
GtkFileChooser ....................................................................................................193
GtkColorSelection................................................................................................197
GtkColorSelectionDialog...................................................................................197
GtkFontSelection..................................................................................................200
GtkFontSelectionDialog......................................................................................201
Widgets de desplazamiento....................................................................................203
GtkHScrollbar/GtkVSrollbar.............................................................................203
GtkScrolledWindow ...........................................................................................203
**8. GtkTreeView: Árboles y listas en GTK+.................................................................205**
Ejemplos básicos.......................................................................................................205
Ejemplo de lista de datos...................................................................................205
Ejemplo de árbol de datos .................................................................................209
Modelos de datos estándares.................................................................................214
Modelos de datos GtkListStore.........................................................................214
Modelos de datos GtkTreeStore........................................................................214
Generalidades .....................................................................................................214
Refiriendose a las filas........................................................................................216
Modelos de datos ordenados ............................................................................217
Modelos de datos a medida....................................................................................218
La interfaz `MgListModel` ....................................................................................219
MgGroupModel: El modelo de datos completo..............................................222
Implementación completa de modelo de datos ...........................................225
GtkTreeView: Visualización....................................................................................228
Listas.....................................................................................................................228
Árboles .................................................................................................................228
Celdas ...................................................................................................................229
Cortar y pegar en GtkTreeView.............................................................................229
**9. Editor de texto multilínea..........................................................................................231**
Manipulación de texto.............................................................................................231
Mini editor de texto..................................................................................................231

**10. GTK+ avanzado** ....237
El portapapeles GTK....237
Drag and Drop....237
Introducción ....237
Definiendo el widget destino....237
Definiendo el widget fuente....238
Imágenes, botones, menús de stock ....240
Ficheros de recursos....240
Introducción ....241
Estructura de un fichero rc ....241
Como usar los ficheros de recursos....242
Selecciones....243
Resumen....243
Obteniendo la selección ....243
Suministrando la selección....244
Otros widgets....245
**11. Accesibilidad en GNOME** ....247
**12. Interfaces de usuario con Glade y libglade** ....249
Introducción....249
Diseño de interfaces de usuario con Glade ....251
Ventana principal....251
Paleta de herramientas....251
Ventana de propiedades ....253
Ventana de trabajo ....254
La biblioteca libglade....255
Ejemplo básico de uso de libglade ....255
Compilación ....256
Un ejemplo más elaborado....256
Internacionalización de las interfaces....261
Ejemplos más avanzados ....261
Integración de GtkTreeView ....262
Integración de Bonobo ....262
Trabajo sólo con widgets específicos....262
**13. El canvas de GNOME** ....263
Bases....263
Modos del `GnomeCanvas`....263
Grupos....264
Crear un canvas ....264
Zoom....265
Región del canvas ....266
Objetos por defecto ....266
Elipses y cuadrados....267
Polígonos....268
Texto....269
Líneas....270
Imágenes ....271
Widgets....271
Ejemplo sobre los objetos ....272
Trabajar con los objetos....275
Manejando objetos....275
Recibiendo eventos....276
Crear objetos propios....280
Métodos....280
Dibujar en el canvas ....282
El objeto genérico GnomeCanvasShape....284
Un objeto propio....287

**14. CORBA** ........ **291**
Introducción a CORBA ........ 291
La Historia de CORBA en el proyecto GNOME ........ 291
El lenguaje IDL ........ 292
ORBit, la implementación de CORBA de GNOME ........ 294
Cabos y esqueletos ........ 294
Accediendo a los objetos ........ 295
Implementación de nuestro objeto ........ 296
Ejecutando la aplicación ........ 299
**15. Activación de componentes** ........ **301**
libgnorba ........ 301
Instalación de componentes en el sistema ........ 301
Activación de Componentes ........ 303
El lenguaje de consulta de OAF ........ 303
**16. Bonobo** ........ **305**
Â¿Qué es Bonobo? ........ 305
CORBA y Bonobo ........ 305
Implementación de interfaces CORBA con Bonobo ........ 307
Implementación del componente con BonoboObject ........ 307
Factorías de componentes ........ 310
Controles Bonobo ........ 310
Uso de Controles - clientes ........ 313
Creación de nuevos controles ........ 315
Menus y barras de tareas ........ 318
Documentos compuestos ........ 318
Introducción a los Documentos Compuestos ........ 319
Interfaces de comunicación entre contenedor y empotrable ........ 320
Documentación y ejemplos de documentos compuestos ........ 321
El primer ejemplo de un contenedor ........ 322
Añadir empotrables al contenedor ........ 326
Vistas de un Empotrable ........ 333
Integración de menús y barras de herramientas ........ 334
Activación de la vista de un Empotrable ........ 336
Impresión del contenedor ........ 339
Persistencia del contenedor ........ 340
Creación de un empotrable ........ 340
Persistencia de un Empotrable ........ 344
Impresión de un Empotrable ........ 344
Contenedores (BonoboWindow) ........ 345
Monikers ........ 346
Representación de los monikers ........ 346
bonobo-conf ........ 347
Implementación de monikers ........ 348
Otros "monikers" ........ 351
**17. XML en GNOME** ........ **353**
libxml (o GNOME-XML) ........ 353
Características ........ 353
Carga de documentos ........ 354
Creación de ficheros XML ........ 358
Salvando documentos XML ........ 360
**18. GConf, el sistema de configuración** ........ **363**
Almacenes de datos ........ 364
Clientes GConf ........ 364
Notificaciones ........ 367
Gestión de errores ........ 368

**19. gnome-vfs** ..... **369**
URIs (Uniform Resource Identifier) ..... 369
Operaciones básicas ..... 370
E/S Asíncrona ..... 372
Módulos para GNOME-VFS ..... 372
**20. Acceso a BBDD** ..... **375**
libgda ..... 375
Fuentes de datos ..... 375
Clientes GDA ..... 375
Gestión de errores ..... 377
Ejecución de comandos ..... 378
Modelos de datos ..... 379
Metadatos ..... 379
libgnomedb ..... 379
**21. Documentación con DocBook** ..... **381**
Por qué usar SGML? ..... 381
SGML ..... 381
Archivos XML ..... 382
Qué es un DTD? ..... 382
El DTD DocBook ..... 382
Instalación de programas ..... 382
Donde Escribir el documento ..... 383
Paso a otros formatos ..... 383
Tipos de Documentos ..... 383
Elementos ..... 385
Lista con items ..... 385
Listas ordenadas ..... 385
Tablas ..... 386
Imgenes ..... 386
Referencias dentro de un documento ..... 387
Enlaces a internet ..... 387
Notas al pie de página ..... 387
Marcas ..... 388
**A. Cómo migrar aplicaciones a la plataforma Gnome 2.0** ..... **389**
Introducción ..... 389
Â¿A quién está dirigido esto? ..... 389
Además de esta guía ... ..... 389
Â¿Qué significa *migración*? ..... 389
Consideraciones generales ..... 391
Las bibliotecas de la plataforma ..... 391
Preparación del ambiente ..... 392
Requerimientos para compilar los módulos ..... 392
Descargando los paquetes ..... 393
Otros paquetes necesarios para ejecutar GNOME 2 ..... 394
Miscelánea de consejos de configuración ..... 395
Requerimientos de espacio en disco ..... 395
Cambios al entorno de compilación ..... 396
Cambios a `autogen.sh` ..... 396
`Pkg-config` ..... 396
Cambios al `configure.in` ..... 397
Cambios en `Makefile.am` ..... 399

# Capítulo 1. Introducción

En esta introducción se presentará el proyecto GNOME, su historia y su organización. Pretende ser un capítulo no técnico para que el lector vaya familiarizándose con el proyecto en sí, de manera que comprenda su evolución pasada y pueda hacerse una idea de su funcionamiento actual.

## El proyecto GNOME

El proyecto GNOME tiene como principal objetivo crear un sistema de escritorio para el usuario final que sea completo, libre y fácil de usar. Asimismo, se pretende que GNOME sea una plataforma muy potente de cara al desarrollador.

GNOME es el acrónimo en inglés de "GNU Network Object Model Environment". Se han propuesto desde los inicios de GNOME varias formas de traducirlo al español, pero no se ha encontrado ninguna que haya satisfecho a todos. Sin embargo, de su nombre podemos ver que GNOME es parte del proyecto GNU y, por tanto, software libre (algunas veces conocido como Open Source). En la actualidad, todo el código contenido en GNOME debe de estar bajo licencia GNU GPL o GNU LGPL. También vemos que las redes y el modelado orientado a objetos tienen capital importancia. A lo largo de este libro, el lector irá comprobando cada uno de estos atributos. Pero, para empezar, veamos un poco la historia de GNOME.

## Historia de GNOME.

Aunque con probabilidad no fue la primera solución en cuanto a entornos de escritorios "amigables" para el usuario, la difusión a mediados de 1995 del sistema operativo Windows95™ supuso un cambio radical en la interacción de los usuarios de a pie con los ordenadores. De los sistemas unidimensionales de línea de instrucciones (los terminales), se pasó a la metáfora de entorno del escritorio bidimensional, donde el ratón ganó terreno al teclado. Windows95™, más que una innovación tecnológica, debe ser acreditado como el sistema consiguió adentrarse en todos los entornos personales y de oficina, marcando las pautas a seguir (normas que, a principios del siglo XXI, todavía seguimos padeciendo).

Los seguidores del software libre, rápidamente se hicieron eco de este notable éxito y, a la vista de que los entornos UNIX carecían de sistemas tan intuitivos a la vez que libres, decidieron ponerse manos a la obra. Fruto de esta preocupación nació en 1996 el proyecto KDE de las manos de Matthias Ettrich (creador de LyX) y otros hackers. El gran problema fue que los chicos de KDE decidieron utilizar una biblioteca de nombre Qt, propiedad de la firma noruega TrollTech™, que no estaba amparada bajo una licencia de software libre. Se daba, por tanto, la circunstancia de que, a pesar de que las aplicaciones de KDE estaban licenciadas bajo la GPL u otras licencias libres, enlazaban con esta biblioteca de manera que se hacía imposible su redistribución. Consecuentemente, se estaba violando una de las cuatro libertades del software libre enunciadas por Richard Stallman en su Manifiesto del Software Libre.

Mientras se seguía discutiendo acerca de la libertad de KDE, la historia quiso que en el verano de 1997, Miguel de Icaza y Nat Friedman coincidieran en Redmond en unas jornadas organizadas por Microsoft™. Es probable que este encuentro propiciara en ambos un giro radical que supuso tanto la creación de GNOME por parte de Miguel de Icaza a su vuelta a México (junto con Federico Mena Quintero), como su admiración por las tecnologías de objetos distribuidos. De Icaza y Mena decidieron crear un entorno alternativo a KDE, ya que consideraron que una reimplementación de una biblioteca propietaria habría sido una tarea destinada a fracasar. GNOME había nacido.

Desde aquellos tiempos lejanos de 1997 hasta la actualidad, GNOME ha ido creciendo paulatinamente con sus reiteradas publicaciones. En noviembre de 1998 ya se lanzó la versión 0.99, pero la primera realmente popular distribuida prácticamente

por cualquier distribución de GNU/Linux sería GNOME 1.0, en marzo de 1999. Cabe destacar que la experiencia de esta primera versión estable de GNOME no fue muy satisfactoria, ya que muchos la consideraron como llena de erratas críticas. Por eso, GNOME October (GNOME 1.0.55) es tratada como la primera versión del entorno de escritorio GNOME realmente estable. Como se puede observar, con GNOME October se intentó evitar versiones de publicación numeradas para no entrar en una "carrera" de versiones con KDE.

La realización de la primera GUADEC, la conferencia de desarrolladores y usuarios europeos, celebrada en París en el año 2000, no coincidió en el tiempo por poco con la publicación de una nueva publicación de GNOME, llamada GNOME April. Fue la última que llevó un mes como nombre de publicación, ya que se mostró que ese sistema causaba más confusión que otra cosa (por ejemplo, GNOME April es posterior a GNOME October, aunque el sentido común nos haría parecer lo contrario). En octubre de ese año, tras ser debatida durante meses en diferentes listas de correo, se fundó la Fundación GNOME.

GNOME 1.2 fue un paso adelante en cuanto a la arquitectura utilizada por GNOME, que se siguió usando en GNOME 1.4. Esta época estuvo caracterizada por la segunda edición de la GUADEC, esta vez en Copenhaguen. Lo que empezó siendo una reunión minoritaria de algunos hackers, se convirtió en un evento mayoritario que atrajo miradas de toda la industria del software.

Mientras tanto, el litigio sobre la libertad de KDE se resolvió con el cambio de postura de TrollTech™, que terminó licenciando `Qt` bajo una licencia dual, que era de software libre para las aplicaciones que son software libre. Hoy en día no cabe ninguna duda de que tanto GNOME como KDE son entornos de escritorio libres, por lo que podemos considerar que el desarrollo de GNOME ha propiciado el hecho de no tener un sólo entorno de escritorio libre, sino dos.

A la hora de escribir este libro, nos encontramos en una parte importante de la historia: la creación de GNOME2, la segunda versión de la plataforma GNOME. Esta nueva versión se ha venido gestando a lo largo del último año y ha tenido su momento cumbre en la celebración de la tercera edición de la GUADEC en Sevilla, España. GNOME2 abre un mundo nuevo de posibilidades al desarrollador que podréis ir descubriendo poco a poco a lo largo y ancho de este libro. Desde aquí, esperamos que la historia de GNOME de ahora en adelante no se pueda escribir sin nuestros lectores.

## Organización de GNOME: La Fundación GNOME.

El problema más difícil de abordar cuando se oye hablar de GNOME por primera vez, es la organización de los más de 800 contribuyentes al proyecto. Resulta paradójico que un proyecto cuya estructura es más bien anárquica llegue a fructificar y saque adelante unos objetivos complejos al alcance de pocas multinacionales del sector de la informática.

Aunque GNOME nació con una clara intención de realizar un entorno amigable y potente al que se iban añadiendo nuevos programas, pronto se vio la necesidad de crear un órgano que tuviera ciertas competencias que permitieran potenciar el uso, desarrollo y difusión de GNOME: de esta forma, en octubre de 2000, se dio paso a la creación de la Fundación GNOME cuya sede se encuentra en Boston, EE.UU..

La Fundación GNOME es una organización sin fines de lucro, no un consorcio industrial, que tiene las siguientes funciones:

- Coordina las publicaciones.
- Decide qué proyectos son parte de GNOME.
- Es la voz oficial (para la prensa y para organizaciones tanto comerciales como no comerciales) del proyecto GNOME.

- Patrocina conferencias relacionadas con GNOME (como la GUADEC).
- Representa a GNOME en otras conferencias.
- Crea estándares técnicos.
- Promueve el uso y el desarrollo de GNOME.

Además, la Fundación GNOME permite la recepción de fondos económicos con los cuales patrocinar e impulsar las funciones antes mencionadas, hecho que antes de su creación era imposible realizar de manera transparente.

En la actualidad, la Fundación GNOME cuenta con un empleado a tiempo completo que se encarga de solventar todos los trabajos burocráticos y organizativos que se dan en una organización sin fines de lucro que realiza reuniones y conferencias de manera periódica.

En términos generales, la Fundación GNOME se estructura en dos grandes consejos: El consejo directivo y el consejo consultor. A continuación se describirán tanto sus funciones como su composición.

### El consejo directivo.

El consejo directivo (Board of Directors) está integrado a lo sumo por catorce miembros elegidos democráticamente por los miembros de la Fundación GNOME. La membresía sigue un modelo meritocrático, lo que viene a decir que para ser miembro de la Fundación GNOME se debe de haber colaborado de alguna u otra manera con el proyecto GNOME. La aportación no tiene por qué ser código, también existen tareas de traducción, organización, difusión, etc. por las que uno puede pedir ser miembro de la Fundación GNOME y tener derecho a voto. Por tanto, son los miembros de la Fundación los que se pueden presentar al consejo directivo y los que, democráticamente, eligen a sus representantes en el mismo de entre los que se hayan presentado. En la actualidad, la votación se lleva a cabo por correo electrónico. La duración del cargo como consejero director es de un año, periodo tras el que se vuelven a convocar elecciones.

Existen una normas básicas para garantizar la transparencia del consejo directivo. La más llamativa es la limitación de miembros afiliados a una misma empresa, la cual no puede exceder de cuatro empleados. Es importante hacer hincapié en que los miembros del consejo directivo lo hacen siempre a nivel personal y nunca en representación de una compañía. Aún así, y después de una larga discusión, se aceptó incluir esta cláusula para evitar suspicacias.

### El consejo consultor.

El consejo consultor es un órgano sin capacidad de decisión que sirve como vehículo de comunicación con el consejo directivo. Está compuesto por compañías comerciales de la industria del software como Red Hat™, Ximian™, HP™, etc. así como por organizaciones no comerciales como la Fundación del Software Libre o el proyecto Debian. Para formar parte del consejo de consultores se exige una cuota a todas las empresas con más de 10 empleados.

## Herramientas y servicios de comunicación en el desarrollo.

## Sistema de control de versiones.

En los siguientes párrafos, se introducirá brevemente el uso del CVS. Para una explicación más completa y detallada, se recomienda la visita de LuCAS[1], donde existe un interesante manual[2] de introducción al CVS.

El uso de CVS es muy sencillo a la vez que potente. Para empezar, no hay más que definir la variable *CVSROOT* con el valor del repositorio que se va a usar. El repositorio CVS está situado en el servidor y es como el almacén donde se guardan los ficheros y sus sucesivos cambios. Este repositorio tiene dentro del servidor CVS una localización física, que se traduce en un directorio. En el caso del repositorio CVS de GNOME Hispano, habría que introducir la siguiente instrucción en el terminal:

```
export
   CVSROOT=:pserver:usuario@cvs.es.gnome.org:/home/cvs/gnome
```

Donde *usuario* es el nombre de usuario que asignado. En el CVS de GNOME Hispano, existe un usuario llamado *anoncvs* que puede usar todo el mundo con permisos de lectura para tener acceso a todos los fuentes de nuestros proyectos.

Se puede observar que *CVSROOT* necesita saber el tipo de conexión (*psever* viene de *persistent server*), el nombre de usuario, la localización de la máquina donde está el servidor CVS (en el caso de GNOME Hispano se encuentra en cvs.es.gnome.org) y finalmente la localización dentro de esa máquina del repositorio CVS.

La opción *pserver* implica que sólo hará falta autentificarse una sola vez y que se guardarán los parámetros de conexión de manera indefinida en el ordenador local. En caso contrario, cada vez que se interactue con el servidor CVS, será preciso introducir la contraseña.

Como en este caso se ha elegido la opción de una conexión persistente, habrá que autentificarse con la siguiente instrucción:

```
cvs -z3 login
```

A continuación, se pedirá la clave de acceso que, en el caso del usuario *anoncvs* es, efectivamente, *anoncvs*.

Seguidamente, se obtendría una copia de los módulos en los que se esté interesado. Normalmente cada proyecto tiene su propio módulo, que es implementado en el CVS como un subdirectorio. Para ello, se usa la instrucción **checkout**:

```
cvs -z3 checkout módulo
```

Ésto crea el subdirectorio módulo en el directorio actual con la última versión de los ficheros que existían en ese módulo. Una vez se tiene una copia local, no hará falta hacer más veces *checkout* (que podría traducirse como "extracción"). Para actualizar una copia ya existente, se usará el comando **update**

```
cvs -z3 update
```

en el directorio que se quiere actualizar.

Y finalmente, para ver las diferencias que existen entre la copia local y el repositorio CVS (muy útil si se quiere seguir el desarrollo de un proyecto paso a paso o enviar parches a los desarrolladores), se usará el comando **diff**. La salida por pantalla de este comando suele ser bastante extensa, así que es buena idea redirigirla a un fichero o mostrarla paginada mediante las instrucciones **more** y **less**.

Si se tiene permiso de escritura en el repositorio, para lo cual habrá que tener una cuenta con acceso de escritura (situación que no se da con anoncvs), se podrán no sólo bajarse ficheros, sino también añadir ficheros y directorios mediante **add**:

```
cvs -z3 add [fichero | escritorio]
```

Cuando se añade un directorio, éste se actualiza directamente en el repositorio, pero cuando es un fichero, es necesario un segundo paso: el comando **commit**, para que el fichero sea creado en el repositorio.

```
cvs -z3 commit
```

Análogamente, para borrar ficheros y directorios, se usa el comando **remove**. Pero, antes de hacer esto, es necesario eliminar el fichero o el directorio de la copia local, pues si no, CVS no llevará a cabo el borrado del archivo.

Y, como ya se ha indicado, el comando **commit** de CVS es el que se usa para introducir los cambios que se hayan hecho en la copia local del repositorio.

En resumen, se puede ver que el uso común de CVS se suele limitar a los siguientes pasos: exportar la variable de shell *CVSROOT* y autentificase en el sistema; a continuación, se extrae una copia local con la última versión de los ficheros mediante la instrucción **checkout**. Esta secuencia, como se ha comentado ya, es única y se hace la primera vez.

Si se edita un fichero que ya existe en el repositorio, a continuación lo único que habrá que hacer es ejecutar la instrucción **commit** para que el servidor sincronice su versión con los cambios realizados. Si este fichero que se ha editado es nuevo y, por tanto, no está en el repositorio del servidor, habrá que añadirlo mediante la instrucción **add** y posteriormente sincronizar el repositorio con la instrucción **commit**. Este último proceso es el que se repite una y otra vez al desarrollar, así que esperemos que haya quedado claro y el lector haya perdido el miedo a usar el sistema de control de versiones CVS, porque es realmente simple y potente.

Para concluir, sólo hay que comentar que el sistema CVS ofrece muchísimas más posibilidades de las que han sido mostradas aquí. Estas instrucciones son las básicas y con las que cualquiera puede a ponerse a trabajar inmediatamente. La profundización en otras instrucciones, menos frecuentemente usadas, del CVS corre a cargo del lector.

### Bugzilla

`Bugzilla` no es más que una base de datos que sirve para que el usuario reporte todos los bugs o errores que encuentreen su GNOME. Reportar errores es una tarea más valiosa de lo que parece porque, gracias a estos informes de errores, cada vez se puede disfrutar de un GNOME mejor. Además, es una tarea muy sencilla.

Para reportar errores lo primero que hay que hacer es crear una cuenta en el `Bugzilla` de GNOME[3] con una dirección de correo válida. Hace poco se creó un estupendísimo asistente[4] muy detallado para hacer más fácil el envío de errores. Basta con seleccionar la aplicación en la que encontremos el error y seguir las instrucciones. Como ya comentamos anteriormente, es una tarea muy sencilla y de inmenso valor que todo usuario puede hacer. Â¡Hagamos entre todos un GNOME libre de fallos!

## Listas de correo.

Las listas de correo forman una parte muy importante dentro de GNOME. Estas listas sirven para discutir sobre las distintas partes de GNOME, problemas a la hora de

programar o simplemente anunciar la nueva versión del programa que se esté desarrollando. Hay listas para todo y para todos.Aquí están todas las listas de correo disponibles[5].

### Internet Relay Chat (IRC).

Tanto si somos desarrolladores como usuarios de GNOME, el IRC nos puede ser de gran ayuda. GNOME dispone de una red propia de IRC (irc.gnome.org) y de canales para charlar.

Para acceder al IRC, el usuario necesitará un cliente. Recomendamos al lector que se descargue el X-Chat[6].

## GNOME Hispano.

Como se ha visto ya en la parte dedicada a la historia de GNOME, desde sus principios, este proyecto se ha venido caracterizando por tener una amplia presencia hispana. GNOME Hispano pretende unir a todos los desarrolladores, traductores y usuarios de habla hispana para acercar este entorno de escritorio a las sociedades hispanohablantes a lo largo y ancho del planeta. Para ello ofrece una serie de servicios, documentación y estructuras organizativas que serán presentadas a continuación.

### Historia de GNOME Hispano.

GNOME Hispano fue creado por Rodrigo Moya y Álvaro del Castillo como un micro grupo dentro de Barrapunto[7], el portal de noticias sobre software libre más conocido en el ámbito hispano. Su fundación se debió a satisfacer necesidades que el grupo de traducción GNOME-es no cubría: se pretendía crear una comunidad hispana de desarrollo, de información y de expansión de GNOME. Aún hoy, la sección GNOME de Barrapunto es una de las más activas donde se puede seguir la actualidad de GNOME.

### Servicios y recursos.

GNOME Hispano tiene una serie de servicios y recursos que hacen que los proyectos que nos proponemos puedan llevarse a cabo. Estos van desde lo más básico, como son el web y las listas de correo, hasta otros más "avanzados" como el servidor CVS y el `Bugzilla` (que usamos gracias al buen trabajo de la gente de HispaLinux[8]).

El servidor web incluye no sólo la página principal del proyecto GNOME Hispano (con sus respectivas secciones de actualidad, eventos, documentación y traducción, etc.), sino que existe la posibilidad de que proyectos de GNOME Hispano sean alojados con sus propias URLs de manera independiente. En la actualidad, ese es el caso de los proyectos *David* y *gcafe*.

En cuanto a las listas de correo, sucede algo parecido. Existen una serie de listas genéricas del grupo, como son *gnome-desarrollo* y *gnome-traductores*, pero también hay multitud de listas para los diferentes proyectos de GNOME Hispano. Incluso existe una lista de correo gnome-libros[9] que ha sido creada para facilitar la intercomunicación a la hora de crear este libro. Cualquier duda, sugerencia, incluso agradecimientos del lector son bienvenidos en esa lista.

Tenemos un servidor de control de versiones CVS para albergar nuestros proyectos. El contenido del repositorio CVS es accesible desde un navegador en la siguiente dirección[10]. En un anexo del libro se explica el uso del CVS.

## Obtención de una cuenta de CVS.

En cuanto alguien aporte algo a alguno de los proyectos, ya sea con ideas, código, documentación o traducciones, lo más razonable es obtener una cuenta de usuario de CVS con permisos de escritura en el repositorio. De esta manera, se podrá colaborar directamente sin tener que estar enviando las contribuciones a alguien.

Pero, como el mundo no es perfecto, existen algunas restricciones en el uso del CVS, que se enumeran a continuación:

- No está permitido, bajo ningún concepto, la creación de nuevos módulos sin el permiso pertinente del cvsmaster[11], por lo que, si se desea hacerlo, lo mejor es ponerse en contacto con él.
- Cada vez que se de dé alta un nuevo módulo, se asignará a una persona como responsable del mismo. Esta persona se encargará de todo lo relacionado con dicho módulo, como la 'recomendación' de nuevos colaboradores, y el mantenimiento del proyecto en sí. Si dicha persona no pudiera continuar con el mantenimiento del proyecto, deberá recomendar a alguien para que le tome el relevo.
- Cada persona es responsable de un módulo, o de parte de un módulo, por lo que tampoco están permitidas las modificaciones en otro módulo bajo la responsabilidad de otras personas, sin el correspondiente permiso del responsable de dicho módulo.
- Para que un módulo de un proyecto de desarrollo de software sea aceptado en el CVS, éste debe hacer obligatoriamente uso de las herramientas autoconf/automake. Esto no es un capricho de GNOME Hispano, sino que obligamos a que se usen estas herramientas para facilitar la compilación de los proyectos en distintas plataformas, aparte de que no nos gustaría tener las listas de correo[12] saturadas con mensajes de gente con problemas de compilación.

Después de estar un tiempo en el IRC Hispano, hemos decididos pasarnos al servidor oficial de GNOME(irc.gnome.org). Allí nos podrás encontrar en el canal `#gnome-hispano`. Con este cambio, pretendemos acercarnos a los demás proyectos que existen en GNOME, de forma que se cree una interesante sinergia. Además, utilizamos el canal de IRC para llevar a cabo un ciclo de charlas semanales, de manera que seguimos un dinámica de formación continua. Las bitácoras de las charlas antiguas se pueden encontrar en la web de GNOME Hispano en la sección de eventos[13].

A medida que se ha ido uniendo más y más gente a GNOME Hispano, es importante el contar con una organización que nos permita ir asignando tareas de una manera eficiente. Para hacer más fácil la gestión de estas tareas, hemos empezado a hacer uso del sistema de control de errores Bugzilla de Hispalinux[14]. El uso y funcionamiento del Bugzilla será tratado en uno de los anexos a este libro.

## GUNIH

GNOME en las universidades hispanas (GUNIH) es un proyecto del grupo GNOME Hispano que trata de entrar en contacto con la comunidad universitaria de todo país hispano, tanto a los alumnos como los profesores.

Esta iniciativa persigue una serie de objetivos:

- Introducción de GNOME en la comunidad universitaria.
- Desarrollo de software libre y gratuito, en colaboración con la comunidad universitaria, que permita su disponibilidad a cualquier componente de dicha comunidad, ya sean estos profesores o alumnos.
- Creación de un grupo de traductores y artistas que colaboren con el proyecto GNOMEÂ  y con el software desarrollado bajo el proyecto GUNIH.

Para la consecución de estos objetivos, el proyecto GUNIH pretende buscar la colaboración de profesores, alumnos y organismos oficiales de carácter universitario. Y proporcionarles todos los recursos que estén a nuestro alcance (infraestructuras, información, apoyo...).

Para ello, el proyecto GUNIH está creando una red interuniversitaria de carácter hispano por todo el mundo con el objetivo de englobar en un mismo proyecto colaboradores de muchas y muy variadas disciplinas universitarias: Desde ingenieros informáticos y matemáticos, hasta artistas de bellas artes, pasando Â por filólogos ingleses, físicos,ingenieros...

Con esta red podremos desarrollar software libre y gratuito y recursos informáticos asesorados por los mejores profesores y alumnos de la comunidad universitaria hispana.

### Otros proyectos.

## Notas

1. http://lucas.hispalinux.es
2. http://www.olea.org/como-empezar-cvs/
3. http://bugzilla.gnome.org/createaccount.cgi
4. http://bugzilla.gnome.org/simple-bug-guide.cgi
5. http://mail.gnome.org/mailman/listinfo
6. http://xchat.org
7. http://www.barrapunto.org
8. http://www.hispalinux.es
9. mailto:gnome-libros@es.gnome.org
10. http://www.es.gnome.org/cgi-bin/cvsweb?cvsroot=GNOME
11. mailto:cvsmaster@es.gnome.org
12. http:www.es.gnome.org/mailman/listinfo
13. http://www.es.gnome.org/eventos/index.php
14. http://bugzilla.hispalinux.es

# Capítulo 2. Preparación del entorno

## Introducción

Las herramientas como automake y autoconf se encuentran disponibles en la mayoría de los proyectos open sources de hoy en día. La mayor ventaja en el uso de estas herramientas se debe a que ayudan a la portabilidad de las aplicaciones a nivel de código fuente, abstrayéndose en la medida de lo posible, de las versiones de las herramientas tradicionales disponibles en cada sistema operativo tipo Unix.

Cada vez que un usuario descarga el código fuente de una aplicación que se encuentra empaquetada, se encuentra comunmente con un script llamado configure, el cual al ejecutarse realiza todas las verificaciones y definiciones necesarias para que la aplicación se pueda compilar, y posteriormente instalar, con éxito. Por lo tanto, desde el punto de vista del usuario, el script configure es el inicio del proceso que dejará la aplicación funcional en su sistema.

Desde el punto de vista del desarrollador, el script configure constituye el resultado final de un proceso, que facilita la distribucion de su aplicación para dejarla disponible a la comunidad.

En este trabajo, se explicarán las herramientas que permiten al desarrollador facilitar su trabajo para, finalmente, dejar disponible un script de autoconfiguracion de su aplicación. Además, se explicará la estructura de directorios en un proyecto open source, la mejor manera de dividirlo y los lineamientos para enfrentar un proyecto que pueda ser mantenible a través del tiempo con distintos desarrolladores.

Este tema es relevante debido a que existen falencias en los desarrolladores a nivel nacional en el uso de este tipo de herramientas, a pesar que es un tema de interés y que han manifestado en más de una ocasión su intención de aprender. Por otra parte, está la competencia entre aprender a crear un proyecto y comenzar a programar inmediatamente, muchas veces acompañado por las barreras de entrada que impone la mantención de archivos Makefile y posteriormente el uso de macros m4.

En los proyectos en funcionamiento, la cantidad de archivos y directorios, de alguna forma u otra, contribuyen al distanciamiento de los nuevos contribuyentes al aprendizaje del uso de estas herramientas.

- *Las herramientas básicas: aclocal, autoheader, autoconf, automake, libtool.*
- *El primer ejemplo, es una aplicación mínima y el segundo es analizar un proyecto GNOME, con todas las macros añ¡¤©das en GNOME que facilitan la gestión de proyectos de naturaleza abierta.*
- *Definiciones de las herramientas*
- *Inicios*
- *Quien las mantiene*
- *Qué proyecto las usa*
- *Por qué se usan*
- *Cuáles son sus facilidades*
- *Por qué se utilizan macros M4*
- *Trucos básicos en el uso de macros*
- *Estructura de un archivo configure.in*
- *Estructura de un archivo Makefile.am*
- *Directivas y variables predefinidas y su relación (configure.in <-> Makefile.am)*
- *Explicación de cada uno de los archivos del proyecto*
- *Macros de internacionalización*
- *Macros para búsqueda de dependencias*
- *Macros para generación de avisos*
- *Macros para evitar el uso de funciones obsoletas*
- *Archivos de un proyecto: ChangeLog, HACKERS, README, TODO*
- *Ideas:*
- *The programa make lee un conjunto de reglas de un archivo Makefile y las emplea para construir un programa.*

## Estructura de un proyecto GNU

El proyecto GNU ha establecido estándares para el ordenamiento de los proyectos que han sido seguido por muchos otros proyectos, no necesariamente bajo el alero de la Free Software Foundation. Entre ellos se encuentra la disposición básica de archivos. De manera simplificada se muestra un proyecto "hola-mundo" para ejemplificar el contenido básico de gran parte de los proyectos open source.

### Archivos de información de un proyecto

En la figura se muestra el árbol con el contenido de un proyecto. El directorio raíz del proyecto contiene una serie de archivos de textos que permiten mantener un control del proyecto y sirve como primer canal de comunicación con el usuario.

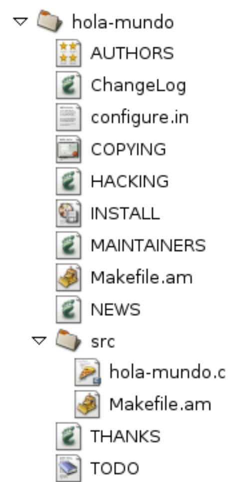



**Figura 2-1. Estructura básica de un proyecto**

#### Archivos de información obligatorios

NEWS,

es un registro de los cambios visibles al usuario donde los cambios más recientes se deben colocar al inicio.

README,

contiene una descripción general del paquete. También es posible indicar instrucciones especiales de instalación o recomendaciones para leer el archivo INSTALL.

AUTHORS,

contiene la lista de nombres de quienes han trabajado en la aplicación.

ChangeLog,

es un registro de todos los cambios que se han efectuado en la aplicación.

COPYING,

especifica los permisos de copia y distribución de la aplicación. Es una buena idea permitir que automake cree este archivo, siempre que se desee licenciar bajo GPL.

INSTALL,

instrucciones de instalación de la aplicación. automake provee un archivo genérico, en donde siempre es aconsejable personalizarlo de acuerdo a los detalles de cada aplicación.

### Archivos de información opcionales

MAINTAINERS,

contiene la lista de nombres de los responsables del proyecto para quien desee ponerse en contacto con ellos.

HACKING,

contiene instrucciones para otros desarrolladores que quieran contribuir a la aplicación. Aquí se incluyen normas de sana convivencia como es el estilo de programación, tipos de autorización para efectuar cambios, etc.

VERSION,

indica la versión del programa.

THANKS,

contiene los créditos a las personas que han contribuido al proyecto pero que no son considerados autores.

TODO,

listado de características que se necesitan llevar a cabo. Permite llevar un orden de prioridades entre los autores y facilitar que potenciales contribuyentes sepan dónde y cómo pueden contribuir. Además, para los usuarios que han solicitado alguna característica es una retroalimentación que indica que sus peticiones están siendo consideradas.

## Archivos de programas y configuración

La estructura que se muestra, corresoponde a la visión del desarrollador. Cuando un programa se distribuye, se entregan en muchos casos archivos generados a partir de ciertos procesos. El caso mas característicos es el script configure, que es un script creado en forma automática.

configure.in,

contiene las reglas de verificación y construcción del proyecto. Es la base para crear el script configure. Las reglas se escriben en macros m4.

Makefile.am,

es el archivo que sirve como entrada al programa automake, quien se encargará de generar en forma automática el archivo Makefile, necesario para construir la aplicación. Debe existir un archivo Makefile.am por cada directorio del proyecto.

src/,

directorio donde se almacena el código fuente de la aplicación. Eventualmente pueden existir más directorios o con otro nombre, pero lo estándar para proyectos que no son bibliotecas o que no son múltiples aplicaciones es denomilarlo src.

autogen.sh,

script que permite automatizar la llamada a cada uno de los programas descritos en la sección. Este script no se muestra en la figura dado que no es obligatorio.

## Las herramientas a través de un ejemplo

A continuación se explica el funcionamiento de las herramientas de autoconfiguración de proyectos de desarrollo a través del proyecto "hola-mundo", que consistirá en armar el proyecto para una aplicación muy sencilla que sirva para mostrar la integración de las herramientas y finalizar con el paquete a distribuir a la comunidad.

Las herramientas funcionan en torno a los archivos configure.in y Makefile.am, y es su definición la que guiará el funcionamiento de cada uno de los programas de automatización. Lo importante, en un principio, es conocer como construir el archivo configure.in y cómo actúan los programas en todo a este archivo.

La secuencia de construcción consiste de los siguientes comandos: aclocal, autoheader, autoconf y automake, lo que finalmente termina generando el script configure y varios plantillas con extensión *.in* (Makefile.in entre los más importantes).

### aclocal

automake, que es la última utilidad en ser invocada, provee macros para autoconf, programa que necesita conocer de su existencia. Para ello, se define como interfaz el archivo aclocal.m4, que será empleado por autoconf para buscar macros de automake.

aclocal es una utilidad que permite generar automáticamente el archivo aclocal.m4. aclocal busca macros en todos los archivos .m4 del directorio en donde se ejecuta y finalmente busca en el archivo configure.in. Todas esas macros se incluirán en el archivo aclocal.m4, así como todas las macros de las cuales dependen. De esta forma, se puede emplear el mismo archivo configure.in para el uso de autoconf y automake simultáneamente.

Para facilitar las personalización en proyectos que requieran macros propias, se ha definido el archivo acinclude.m4. Las macros allí definidas, pueden ser empleadas también en el archivo configure.in.

El archivo aclocal.m4 solo será generado si se encuentran macros de automake, es decir, aquellas que comiencen con el prefijo AM_.

Por ejemplo, un archivo configure.in básico (sin funcionalidad):

**Ejemplo 2-1. configure.in básico (sin funcionalidad)**

```
AC_PREREQ(2.52)
AC_INIT(hola-mundo,0.1)

AM_AUTOMAKE_VERSION(1.7.2)

AC_OUTPUT(Makefile)
```

Después de ejecutar aclocal, se creará el archivo aclocal.m4 bastante pequeño cuya macro, AM_AUTOMAKE_VERSION, sirve para indicar que se requiere la versión 1.7 de la API de automake a utilizar. En este ejemplo, se ha empleado para ilustrar el archivo aclocal.m4 generado.

Sin embargo, con en el ejemplo anterior no es posible utilizar automake, puesto que no se encuentran disponibles todas las macros que automake requiere. Para ello es necesaria la macro AM_INIT_AUTOMAKE[1], que debe ser la primera macro de automake en ser invocada.

Una disgresión entre AC_INIT y AM_INIT_AUTOMAKE. Ambas macros son capaces de inicializar el nombre del paquete y la versión (que más adelante se empleará).

La forma antigua de hacerlo es:

**Ejemplo 2-2. Método antiguo**

```
AC_INIT
AM_INIT_AUTOMAKE(programa,version)
```

La forma nueva de hacerlo es:

**Ejemplo 2-3. Método nuevo**

```
AC_INIT(programa,version)
AM_INIT_AUTOMAKE
```

Actualmente AM_INIT_AUTOMAKE acepta el nombre del paquete y la versión, pero no hay garantía que en el futuro siga funcionando de esa forma. Por lo tanto, es conveniente usar el nuevo formato.

## autoheader

autoheader crea una plantilla con un conjunto de directivas #define que pueden emplearse desde los programas en C. Para indicar el nombre del archivo se debe emplear la macro AC_CONFIG_HEADERS. En versiones antiguas[2], se empleaba AM_CONFIG_HEADER, la cual está obsoleta.

El archivo configure.in quedará entonces:

**Ejemplo 2-4. configure.in básico**

```
AC_PREREQ(2.52)
AC_INIT(hola-mundo,0.1)

AC_CONFIG_HEADERS(config.h)

AM_AUTOMAKE_VERSION(1.7)
AM_INIT_AUTOMAKE

AC_PROG_CC

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

autoheader creará el archivo config.h.in, una plantilla que en un proceso posterior pasará a ser config.h El nombre del archivo es arbitrario, pero config.h ya es estándar en las aplicaciones. Dicho archivo puede emplearse en los programas a través de la directiva \lstinline[language=C,frame={tb}]{#include <config.h>}.

Por lo tanto, la secuencia de ejecución será:

**Ejemplo 2-5.**

```
$ aclocal
$ autoheader
$ autoconf
```

En este instante se han creado los archivos aclocal.m4, config.h.in y autom4te.cache. Los dos últimos, generados por autoheader. El resultado en config.h.in es el siguiente:

**Ejemplo 2-6.**

```
/* config.h.in.  Generated from configure.in by autoheader.  */

/* Name of package */
#undef PACKAGE

/* Define to the address where bug reports for this package should be sent. */
#undef PACKAGE_BUGREPORT

/* Define to the full name of this package. */
#undef PACKAGE_NAME

/* Define to the full name and version of this package. */
#undef PACKAGE_STRING

/* Define to the one symbol short name of this package. */
#undef PACKAGE_TARNAME

/* Define to the version of this package. */
#undef PACKAGE_VERSION

/* Version number of package */
#undef VERSION
```

Como se puede apreciar, es una plantilla, donde cada directiva \lstinline[language=C]{#undef} será reemplazada por los valores reales y cuyos valores serán obtenidos durante la ejecución del script configure, más adelante. De esta manera, se define en un solo lugar el nombre de la aplicación y su versión, facilitando la mantención y la liberación de nuevas versiones.

## autoconf

autoconf es una herramientas que permite construir paquetes de programas de una manera más portable. Provee una seria de pruebas que se realizan en el sistema donde se instalará para determinar sus características antes de ser compilado, de tal forma que el código fuente de un programa puede adaptarse en mejor manera a los diferentes sistemas.

autoconf permite generar un script, llamado configure, que es el encargado de realizar las comprobaciones para determinar la disponibilidad y ubicación de todos los requisitos para la construcción exitosa de un programa. Cuando se encuentra bien construidos, permite instalar y desinstalar muy fácilmente las aplicaciones.

autoconf genera un script a partir de de las definiciones en configure.in, y sus macros se reconocen por el uso del prefijo AC_.

autoconf primero busca en las macros que tiene en forma predefinida y luego en el archivo aclocal.m4, si es que existe[3].

En estos momentos se pueden detallar las macros no explicadas del programa \ref{configure.in-basico}:

- AC_PREREQ, indica que la versión de autoconf requerida debe ser igual o mayor a 2.52:
- AC_INIT, es la macro encargada de inicializar las funcionalidades de autoconf. Recibe como parámetros el nombre de la aplicación y su respectiva versión. Cada que se libere una nueva versión, se debe incrementar el segundo parámetro, para indicar la nueva versión en desarrollo.
- AC_PROG_CC indica que se trata de programas en C y que se requiere de compilador y las bibliotecas básicas de desarrollo.
- AC_OUTPUT indica los archivos que finalmente deben ser generados por el script configure.

## automake

automake es una herramienta que permite generar en forma automática archivos Makefile, de acuerdo a un conjunto de estándares, simplificando el proceso de organización de las distinas reglas así como proveyendo funciones adicionales que permiten un mejor control sobre los programas.

* *libtool*

* *Libtool es una interfaz entre el compilador y el enlazador para facilitar la generación de bibliotecas estáticas y dinámicas de una forma más portable, independiente de la plataforma en la cual se encuentran ejecutándose.*

La utilidad make es ampliamente usada en el desarrollo de aplicaciones, ya que a partir de un archivo, llamado Makefile, que contiene reglas de dependencia es capaz de determinar las acciones a seguir, comúnmente determinar que programas deben compilarse para obtener la aplicación.

Sin embargo, es tedioso crear las reglas para que construya a través de varios subdirectorios, con seguimiento automático de dependencias. Si se trabaja en varios proyectos, prácticamente significa reinventar la rueda en cada uno de ellos, sin mecionar los problemas de portabilidad.

automake permite automatizar esta labor, para ello emplea un archivo Makefile.am como fuente, en donde se indica lo esencial a construir, y será necesario por cada uno de los directorios en que se necesite realizar alguna tarea. A partir de ello, generará una plantilla llamada Makefile.in, la que finalmente se traducirá en Makefile cuando se ejecute el script configure.

Los archivos Makefile.in que se generarán dependerán exclusivamente de lo que se indique en la macro AC_OUTPUT.

automake puede proveer los archivos que son estándares (ver la sección ) en cada aplicación, y salvo que se desee realizar algún cambio, es posible trabajar transparentemente con lo que sugiera, para ello basta usar la opción '--add-mising' de automake. Esto creará enlaces simbólicos de los archivos, para realizar una copia es necesario añadir la opción '--copy'. Es importante revisar el archivo COPYING, puesto que contiene la licecia de uso de la aplicación. Por omisión, será GPL.

El archivo Makefile.am del directorio raíz contendrá:

**Ejemplo 2-7. Makefile.am del proyecto 'hola mundo'**

En donde se indica, que debe procesar el subdirectorio src. SUBDIRS es una variable genérica, en donde se especifican todos los subdirectorios que se deben procesar. De cierta forma, se indican los directorios en que hay dependencias que se deben satisfacer. Cuando se ejecute la utilidad make, ingresará primero a cada uno de los subdirectorios y así sucesivamente.

Luego, se indican todos los archivos que son parte extra de la aplicación y que se desean distribuir, además de la aplicación ejecutable.

Dentro del directorio src, también se encuentra un archivo Makefile. En este directorio se encuentra el programa propiamente tal, por lo tanto las reglas son distintas.

**Ejemplo 2-8. src/Makefile.am del proyecto 'hola mundo'**

La variable bin_PROGRAMS indica como se llamará la aplicación final, el archivo binario (ejecutable). Si se desea generar una biblioteca y no un programa ejecutable, se debe emplear libexec_PROGRAMS.

Cabe notar que ha sido llamado 'hola-mundo'. Ese mismo texto se empleará como prefijo para otras reglas, tales como: hola_mundo_SOURCES[4], hola_mundo_LDADD, por nombrar las más comunes.

## El programa

Como se trata de una aplicación de ejemplo, el programa "hola-mundo", imprimirá el nombre de la aplicación, su versión y el clásico "hola mundo". El nombre de la aplicación y versión se obtienen de las definiciones de la macro AC_INIT en el archivo configure.in y utilizados a través del archivo config.h.

**Ejemplo 2-9. Programa hola-mundo.c**

## Finalizando la configuración

En este punto, ya se encuentran todos los archivos preparados para ejecutar el script configure. Hay que notar que, al momento de distribuir la aplicación, no es necesario que la contraparte disponga de aclocal, autoheader, autoconf y automake; el script contiene toda la información necesaria para realizar las verificaciones en forma autónoma.

Ya se encuentra todo preparado y resumiendo, la ejecución completa debiera ser:

**Ejemplo 2-10.**

Ya se encuentra nuestro programa compilado y funcionando. Si se desea distribuirlo, basta ejecutar:

**Ejemplo 2-11.**

De esta forma, se ha obtenido la aplicación empaquetada en un archivo listo para ser liberado como la versión 0.1 de hola-mundo.

Entre las reglas mas comunes se encuentran: make install, make uninstall, make clean, make distclean.

## Notas

1. La excepción la constituye AM_AUTOMAKE_VERSION.
2. Es conveniente verificar la versión de automake que se emplea para desarrollar. La versión 1.4 requiere de AM_CONFIG_HEADER. La versión en uso actual (1.7.6) acepta ambas, aunque recomienda AC_CONFIG_HEADERS.
3. Recordar que aclocal.m4 se genera previamente con aclocal.
4. Notar que se reemplazó el símbolo ’-’ por ’_’ en el nombre de la regla.

# Capítulo 3. Internacionalización

## Internacionalización de aplicaciones

Si se desea que la aplicación esté disponible en diferentes idiomas, es necesario internacionalizarla, de tal forma de separar todas las cadenas de textos que aparecen en la aplicación y que puedan ser manipuladas por el equipo de traducción respectivo. El proceso de internacionalizar una aplicación también se conoce como i18n. A su vez, el proceso de traducir mensajes de un programa se conoce como l10n.

La idea básica de i18n es marcar aquellas cadenas de texto para que puedan ser traducidas. Para ello existen funciones especiales, donde la más conocida es gettext.

### intltool

La utilidad gettext es la que permite extraer cadenas desde programas, sin embargo,intltool es una herramienta que extiende su funcionalidad, permitiendo la traducción de archivos desktop (empleados en el menú), glade, gconf, xml, entre otros. Es posible escribir programas traducibles sin emplear intltool, pero es mucho más cómodo utilizar la infraestructura existente en GNOME, aunque se utilice para proyectos no GNOME.

Intltool permite:

1. Detectar las herramientas necesarias para la configuración y construcción de la aplicación.
2. Extraer las cadenas a traducir
3. Mezclar las traducciones en la aplicación final

#### Cambios en la estructura de directorios

Se debe añadir el directorio "po", donde se ubicarán el archivo con todas las cadenas traducibles (hola-mundo.pot) y los archivos en cada idioma (es.po, fr.po, etc.). Los archivos esenciales dentro de este directorio son:

POTFILES.in,

: contiene una lista de todos los archivos en el proyecto que contienen cadenas a traducir.

POTFILES.skip,

: contiene una lista de los archivos que no deben ser considerados como traducibles. Es útil cuando hay archivos de programas que están siendo usados, pero que aún permanecen en el proyecto y así evitar esfuerzo innecesario por parte del equipo de traductores.

ChangeLog,

: un registro de todos los cambios efectuados en el manejo de i18n y l10n.

A través de intltool-update se puede obtener los nombres de los archivos que tienen cadenas a traducir, lo cual se detalla en la sección.

#### Cambios en configure.in

Para añadir soporte de i18n con intltool se requiere añadir algunas macros al archivo configure.in[1], las cuales se muestran en el listado \ref{configure.in-i18n}. Lo primero

es añadir la macro AC_PROG_INTLTOOL, que indica que se hara uso de esta herramienta y, en forma particular, se requiere al menos de la versión 0.23.

**Ejemplo 3-1. configure.in con soporte i18n**

Entre la línea 11 y 13 se muestra la forma para definir símbolos al preprocesador de C. La línea 11 asigna un valor de texto a la variable GETTEXT_PACKAGE, la macro AC_DEFINE_UNQUOTED se encarga de dejar disponible el valor para GETTEXT_PACKAGE para los programas, en términos de programación, será equivalente a:

\lstinputlisting{define.c}

el cual se registrará en el archivo config.h. La macro AC_SUBST permite que se pueda emplear @GETTEXT_PACKAGE@ dentro de los archivos Makefile.am y puedan acceder a su valor. En este caso, será útil para el Makefile que se generará en el directorio "po"

Posteriormente, en la línea 15, se ha definido una cadena vacía para ALL_LINGUAS, debido a que aún no hay idiomas disponibles. Normalmente contendrá los códigos de los idiomas separados por espacios, por ejemplo:

\lstinputlisting{configure-i18n.in}

En la línea 16, la macro AM_GLIB_GNU_GETTEXT se encarga de realizar las verificaciones necesarias para el uso de gettext. Además se encarga de definir los símbolos HAVE_GETTEXT y ENABLE_NLS en el archivo config.h. La macro AM_GLIB_GNU_GETTEXT es propia del entorno de desarrollo GTK+, sin embargo es bastante. Si se quiere evitar su uso, puede ser reemplazada por AM_GNU_GETTEXT, con prestaciones menores.

Finalmente, línea 22, se ha añ¡¤©do un nuevo archivo de salida: po/Makefile.in.

## Cambios en Makefile.am

Se añ el directorio "po" en la lista de subdirectorios en los cuales make debe ingresar. Además, se han añadido las utilidades de intltool como archivos extras.

**Ejemplo 3-2. Makefile.am con soporte i18n**

También es necesario modificar el archivo Makefile.am en el directorio src. Allí se añadir la variable INCLUDES, en donde se define el lugar donde residirán las cadenas traducidas. El símbolo allí indicado será utilizado dentro del programa.

Además, se añade un nuevo archivo dentro de los programas fuentes: hola-mundo-i18n.h, descrito más adelante.

**Ejemplo 3-3. Makefile.am con soporte i18n**

## Cambios en el código fuente

Como se indicó en la sección, se ha añadido un archivo de cabecera, cuya línea esencial es la 9, donde se asocia _() como un sinónimo de la función gettext() y de esta

manera simplificar la lectura de los programas respecto a las cadenas que se han marcado como traducibles.

**Ejemplo 3-4. hola-mundo-i18n.h**

En caso que no este disponible el conjunto de herramientas de internacionalización, se ha definido de tal forma que no provoque problemas en compilar.

El programa hola-mundo.c también sufre modificaciones:

**Ejemplo 3-5. hola-mundo-i18n.c con soporte i18n**

Como se aprecia, se usa la ENABLE_NLS para comprobar si se está compilando con soporte i18n, en cuyo caso define el dominio o ámbito de las traducciones, normalmente reducido al dominio del programa y que se encuentra definido en GETTEXT_PACKAGE, el cual fue declarado previamente en el archivo configure.in (ver listado \ref{configure.in-i18n} en la sección. También se especifica el directorio donde se encuentran las traducciones, la cual fue definida en el archivo src/Makefile.am. Además, se especifica el conjunto de caracteres a emplear, en este caso UTF-8.

Finalmente, en la línea 18, se puede apreciar que la cadena ha sido marcada para ser traducida al colocarla dentro de la función _(). Desde el punto de vista de programación, no recarga con nombres de función largos ni distrae la lectura para lo que es una tarea rutinaria en proyectos grandes.

El texto de la función printf ahora aparece en inglés, lo cual es lo recomendable, pueso que es mucho más sencillo encontrar traductores de inglés a otro idioma que de español a otro idioma.

## Preparación del ambiente

Dentro de la sencuencia de comandos, es necesario añadir glib-gettextize[2], que crea el archivo po/Makefile.in.in, e intltoolize, para disponer de las herramientas para manejar las cadenas a traducir.

**Ejemplo 3-6.**

Con esto se dispone de la infraestructura básica para tener una aplicación disponible en distintos idiomas. Sin embargo, aún falta un trabajo que realizar en el directorio "po", y que consiste en determinar el contenido del archivo POTFILES.in. Para ello, dentro de dicho directorio basta invocar el comando:

**Ejemplo 3-7.**

el cual mostrará los archivos que tienen cadenas a traducir y que no se encuentran en el archivo POTFILES.in. Para facilitar el trabajo, también se genera un archivo llamado "missing"[3] que puede ser añ¡¤©do al final de POTFILES.in.

En estos momentos ya es posible ejecutar el script configure y compilar la aplicación.

### Últimos pasos: traducir las cadenas

A estas alturas se dispone de toda la infraestructura para poder comenzar a traducir la aplicación. Por lo tanto, es necesario generar el archivo maestro que contiene todas las cadenas a traducir y que puede ser empleado para comenzar el soporte para un nuevo idioma. Dentro del directorio "po" basta ejecutar:

**Ejemplo 3-8.**

De esta forma se generará el archivo hola-mundo.pot[4] que es una plantilla, el cual empleamos para comenzar la traducción al español (es.po). Para que las cadenas se actualicen de acuerdo a como cambia la aplicación y que además se distribuya, es necesario agregar "es" en la variable ALL_LINGUAS del archivo configure.in.

Finalmente, tras ejecuta make distcheck, se tendrá la versión 0.2 del programa hola-mundo.

* *Un proyecto GTK+/GNOME. pkg-config*

## Notas

1. Se ha cambiado la versión a 0.2 para esta versión "internacional".
2. En caso que no se disponga, se puede emplear gettexttize que es más limitado.
3. Este archivo puede ser borrado, porque solo es para informar en un instante dado.
4. El nombre hola-mundo se obtiene de la definición de GETTEXT_PACKAGE en el archivo configure.in.

# Capítulo 4. GLib

La librería GLib es una de las más importantes que existen en GNOME. Esta librería es, junto a la librería GTK+, el pilar sobre el que se sustentan todas las aplicaciones.

## Tipos de datos de GLib

Dentro de GLib está implementada una serie de tipos de datos que nos hace más fácil, si cabe, el tratamiento de los datos y además tiene la propiedad de mejorar la portabilidad de nuestros programas. Los tipos que se usan en GLib no son muy diferentes de los que usamos en el estándar de C. Así que realmente no nos resultará nada complicado acostumbrarnos a usarlos.

De momento, vamos a comenzar aprendiendo los tipos de GLib que corresponden con los del estándar de C.

**Tabla 4-1. Tipos de GLib que se corresponden con los del estándar C.**

| Tipos GLib. | Tipos estándar C. |
|---|---|
| gchar | char |
| gint | int |
| gshort | short |
| glong | long |
| gfloat | float |
| gdouble | double |

Como podemos observar en la tabla, estos tipos no son muy diferentes de los que usamos cuando programamos con los tipos del estándar de C, aunque es altamente recomendable poner un poco de esfuerzo para acostumbrarse a estos tipos si se va a trabajar con la plataforma GNOME.

GLib también posee una serie de tipos que nos hacen más sencillo el tratamiento de datos. Algunos tienen su correspondencia en C, pero nos hacen más sencillo su uso; otros nos sirven para mantener el mismo tamaño de datos de una plataforma a otra y, finalmente, hay otros que no tienen correspondencia con el estándar de C, pero que nos resultarán bastante útiles.

**Tabla 4-2. Tipos de GLib que facilitan el uso de tipos del estándar C.**

| Tipos de GLib | Tipos del C estándar. | Definición |
|---|---|---|
| gpointer | void * | gpointer es un puntero sin tipo, que luce mejor que escribir void *. |

| **Tipos de GLib** | **Tipos del C estándar.** | **Definición** |
| --- | --- | --- |
| gconstpointer | | gconstpointer es un puntero a una constante. Los datos a los que apuntan no deberían ser cambiados. Este tipo suele usarse en los prototipos de funciones para indicar que el valor al que apunta la función no debería ser cambiado en el interior de la función. |
| guchar | unsigned char | Como se puede observar, estos tipos son idénticos a los tipos de C que carecen de signo, pero con la ventaja de que son más visuales y fáciles de usar. |
| guint | unsigned int | |
| gushort | unsigned short | |
| gulong | unsigned long | |

**Tabla 4-3. Tipos de GLib que aseguran el tamaño del dato entre plataformas.**

| **Tipo** | **Rango de tamaño del dato.** |
| --- | --- |
| gint8 | -128 a 127 |
| guint8 | 0 a 255 |
| gint16 | -32.768 a 32.767 |
| guint16 | 0 a 65535 |
| gint32 | -2.147.483.648 a 2.147.483.647 |
| guint32 | 0 a 4.294.967.295 |
| gint64 | -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807 |
| guint64 | 0 a 18.446.744.073.709.551.615 |

**Tabla 4-4. Tipos de GLib nuevos que no están en el estándar de C.**

| **Tipos de GLib** | **Definición** |
| --- | --- |
| gboolean | Este tipo es booleano y sólo contendrá los valores `TRUE` o `FALSE`. |
| gsize | Es un entero de treinta y dos bits sin signo que sirve para representar tamaños de estructuras de datos. |
| gssize | Es un entero de 32 bits con signo, que sirve para representar tamaños de estructuras de datos. |

Quizás, nuestro lector más novato haya saltado de su asiento al ver algunos tipos como gint8, gint16, gsize, etc. Para aliviarle esta posible preocupación, es interesante denotar que cuando se programa con GLib, los tipos más usados son los char, int, double, etc., aunque ahora usará los mismos, pero con una ge delante del tipo.

# Mensajes de salida.

## Mensajes de salida.

GLib implementa una serie de funciones para el envío de mensajes al exterior del programa. Al igual que en el ANSI C disponemos de la mítica función `printf ()`, nosotros disponemos en GLib de una función que se asemeja en todo menos en el nombre; esta función es `g_print ()` y su prototipo de función sería:

```
void g_print (const gchar *format, ...);
```

Como se puede ver, el uso de `g_print ()` es idéntico al de `printf ()`. Es una función que recibe como parámetros un formato y las variables que se usan dentro de las características del formato, igual que `printf ()`.

**Ejemplo 4-1. Mensajes de salida.**

```
/* ejemplo del uso de g_print */

#include <glib.h>

int
main (int argc, char *argv[])
{
        gint numero = 1;
        gchar *palabra = "hola";

        g_print ("glib te saluda : %s\nglib te dice un número : %d\n", palabra, numero)

        return 0;
}
```

Además, GLib provee de un método para dar formato todos los mensajes de salida que se emitan con la función `g_print ()`, por lo que se podría decir que, cada vez que mandamos un mensaje con formato a `g_print ()`, éste debe salir por pantalla con algún tipo de mensaje adicional. Esto es debido a la función `g_print_set_handler ()`, que tiene como parámetros un puntero a función. Esta función, que es representada por el puntero a función, será definida por el programador y, con ella, se podrá decir qué mensaje adicional queremos que se vea. Para entenderlo mejor, se mostrará a continuación la sinopsis de esta función, así como un ejemplo.

```
GPrintFunc  g_print_set_handler (GPrintFunc funcion);
```

Esta sería la sinopsis de la función `g_pint_set_handlet ()`, que recibiría como parámetro un puntero a función. Esta función tendría este aspecto

```
void  (* GPrintFunc) (const gchar *  cadena );
```

Éste sería el aspecto que ha de tener la función que se le pasará como parámetro a `g_print_set_handler`. El siguiente ejemplo ha sido concebido con la intención de facilitar su comprensión.

**Ejemplo 4-2. Personalización de mensajes de salida.**

```
/* ejemplo de cómo añadir información a los mensajes de g_print */

#include <glib.h>

void funcion_personal (const gchar *cadena);

/* funcion_personal es la función que le pasaremos a
   g_print_set_handler como parámetro */

void
funcion_personal (const gchar *cadena){

        printf ("** Soy tu programa y te digo ** ");
        printf (cadena);

}

main (){

        /* g_print normal */

        g_print ("Soy un mensaje normal de salida\n");

        /* g_print personalizado */

        g_set_print_handler (funcion_personal);
        g_print ("mensaje de salida personalizado\n");

}
```

GLib también nos provee de una función para comunicar mensajes de salida por la salida de errores estándar (*stderr*). Para el envío de mensajes tenemos la función `g_printerr` que funciona exactamente igual que `g_print`. Existe también una función como `g_set_print_handler`, llamada `g_set_printerr_handler` que funciona igual que la anterior.

Se puede ver, en conclusión, que el formateo personalizado de los mensajes de salida en GLib es bastante simple.

## Funciones de depuración.

La depuración de un programa siempre es un trasiego por el cual todo programador, ya sea principiante o avanzado, ha de pasar. Y para hacer la vida más fácil al programador de GNOME, GLib nos ofrece una serie de funciones que harán más sencilla la depuración del código.

Estas funciones son útiles para comprobar que se cumple una serie de condiciones lógicas delimitadas por el programador. En caso de no ser cumplida, GLib emitirá un mensaje por consola, explicando que la condición que el programador ha puesto no se ha cumplido e indicando el archivo, la línea y la función en las que se ha roto la condición.

```
#define g_assert ( expresion );
```

La primera función de este tipo que vamos a ver es `g_assert`. Esta función recibe como parámetro un enunciado lógico y comprueba su validez. Por poner un ejemplo de su uso, podríamos exponer el caso en el que a una función se le pase un puntero. Y nosotros podríamos poner **`g_assert (ptr != NULL)`** para certificar que ese caso no se dé nunca. De hecho, si se diese ese caso, la función `g_assert` comprobaría que esa expresión no es válida y mandaría interrumpir la ejecución del programa. Para tener más claro este concepto, obsérvese la ejecución del siguiente ejemplo:

**Ejemplo 4-3. Depuración de programas con `g_assert`.**

```
/* ejemplo del uso de g_assert */

#include <glib.h>

void
escribe_linea (gchar *cadena)
{

        /* si a la función se le pasa un puntero a NULL
           el programa cesara en la ejecución */

        g_assert (cadena != NULL);
        g_print ("%s\n", cadena);

}

main ()
{
        gchar *cadena="esto es una cadena";
        gchar *no_es_cadena = NULL;

        /* como no se pasa un NULL sino una cadena, se escribirá
           la cadena "esto es una cadena" por pantalla */

        escribe_linea (cadena);

        /* como se le pasa un NULL el assert de la función
           escribe_linea surtirá efecto */

        escribe_linea (no_es_cadena);
}
```

Si se compila este programa, en la salida del mismo, se podrá observar que cuando entra por segunda vez en la función `escribe_linea`, cesa la ejecución del programa. Además, por la consola saldrá este mensaje o uno parecido.

```
(process:457): ** ERROR **: file
  ejemplo.c: line 9 (escribe_linea):
assertion failed: (cadena !=
  NULL)
aborting...
```

El mensaje da información del proceso donde se rompió la condición y muestra por pantalla información del archivo, de la línea y de la función en las que se viola la condición por la cual `g_assert` ha parado la ejecución. También indica qué expresión se ha incumplido, porque se puede dar el caso de tener varios `g_assert` en una misma función.

Quizá un método tan drástico como parar la ejecución del programa no sea lo mejor. Es probable que sólo se busque algo que termine la ejecución de la función y avise

que una condición no ha sido cumplida. Para estos casos existen varias macros que facilitarán mucho la vida de los programadores. Estas macros son:

```
#define g_return_if_fail ()( expresion );
```

```
#define g_return_val_if_fail ()( expresion , val );
```

Las dos macros son similares en su funcionamiento, ya que comprueban si la expresión lógica que se les pasa como primer parámetro es verdadera o falsa. En el caso de ser falsa, aparecerá por consola un mensaje explicando que esa condición no ha sido cumplida. Esto es muy parecido a lo que hacía `g_assert ()`; la diferencia estriba en que estas dos funciones no finalizan la ejecución del programa, sino que simplemente hacen un retorno dentro de la propia función y, en consecuencia, el resto del programa seguirá ejecutándose.

Como se dijo antes, estas macros son similares. La diferencia es que la segunda, `g_return_val_if_fail`, hace que la función devuelva el valor que se le pasa como segundo parámetro.

Obviamente, `g_return_if_fail` se usará en funciones que no devuelvan ningún valor y `g_return_val_if_fail` en funciones que devuelvan algún valor. Para entender mejor estas funciones obsérvese el siguiente ejemplo.

**Ejemplo 4-4. Depuración de programas con `g_return_if_fail` y `g_return_val_if_fail`.**

```
/* ejemplo del uso de g_return_if_fail y f_return_val_if_fail */
#include <glib.h>

void
escribe_linea (gchar *cadena){

        g_return_if_fail (cadena != NULL);
        g_print ("%s\n", cadena);
}


/* escribe línea 2 devolverá
   TRUE si la ha escrito bien
   FALSE si incumple la expresión */

gboolean
escribe_linea2 (gchar *cadena){

        g_return_val_if_fail (cadena != NULL, FALSE);
        g_print ("%s\n", cadena);
        return  TRUE;
}

main (){

        gchar *cadena="esto es una cadena";
        gchar *no_es_cadena = NULL;
        gboolean resultado ;

        escribe_linea (cadena);
        resultado = escribe_linea2 (cadena);

        if (resultado == TRUE)
                g_print ("\n-- escribe_linea2 ha devuelto TRUE --\n");
```

```
        escribe_linea (no_es_cadena);
        resultado = escribe_linea2 (no_es_cadena);

        if (resultado == FALSE)
                g_print ("\n-- escribe_linea2 ha devuelto FALSE --\n");

}
```

Y, por consiguiente, la salida de este programa sería la siguiente:

```
esto es una cadena
esto es una cadena

-- escribe_linea2 ha devuelto TRUE --

(process:578): ** CRITICAL **: file glib-g_return_fail.c: line 6 (escribe_linea):
assertion ‘cadena != NULL’ failed

(process:578): ** CRITICAL **: file glib-g_return_fail.c: line 18 (escribe_linea2):
assertion ‘cadena != NULL’ failed

-- escribe_linea2 ha devuelto FALSE --
```

Como se puede ver, su ejecución también ofrece información sobre el lugar donde no se ha cumplido la expresión como ya lo hacía la función `g_assert`, salvo que no detiene la ejecución del programa.

Y estas son las funciones para depuración que se pueden encontrar dentro de la librería GLib. Aunque no son las únicas, sí son las más comunes.

## Funciones de registro.

GLib tiene cuatro macros básicas para el tratamiento mensajes, por niveles de importancia. Estas macros permiten al programador realizar una registro del recorrido del programa especificando el nivel de importancia de los sucesos acontecidos dentro del mismo.

Las cuatro macros de las que estamos hablando reciben parámetros de la misma forma que lo hacian `g_print` o `printf`. De este modo, no sólo se tendrá una registro por niveles, sino que, además, se podrá especificar un texto preformateado. De este modo, los mensajes pueden representar por pantalla el contenido de variables o cualquier otra información que nos pudiera ser útil. Las cuatro macros de las que estamos hablando son :

```
#define  g_message (...);
```

```
#define  g_warning (...);
```

```
#define  g_critical (...);
```

```
#define  g_error (...);
```

Como se puede observar, las cuatro macros tienen nombres bastante intuitivos. Para empezar, el lector se habrá dado cuenta, por los nombres de las funciones, de que están ordenadas de menor a mayor importancia, siendo `g_message` menos importante que `g_error`. A continuación se explicará lo que hace cada una de ellas.

`g_message` es una macro que emite un mensaje normal con la información que se le ha pasado como parámetro. La información que emite esta macro no tiene por que estar asociada a un error en el programa. Su principal aplicación es el envío de mensajes con la información que se crea conveniente en el proceso de realización del programa.

`g_warning` emite un mensaje de aviso con la información que le ha sido pasada como parámetro. Esta información puede servir para ser avisados de situaciones peligrosas que se podrían dar en el programa.

`g_critical` cumple la misma función que `g_warning`, sólo que éste además tiene la posibilidad de abortar el programa usando la función `g_log_set_always_fatal()`.

`g_error`emite el mensaje que le haya sido pasado como argumento y además aborta irremediablemente la ejecución del programa. Esta macro se usa para obtener información de donde ha sucedido el error irrecuperable con la seguridad de que, a causa de este error, el programa terminara abortando su ejecución de todas maneras.

Para una mejor comprensión se muestra el siguiente ejemplo. En él se han definido cuatro funciones, cada una de las cuales emite un mensaje diferente. En este ejemplo no se hace que `g_critical` tenga capacidad de abortar el programa porque si no, no veríamos como funciona `g_error`.

**Ejemplo 4-5. Uso de mensajes de registro.**

```
/* ejemplo del uso de g_warning, g_critical, ... */
#include <glib.h>

void
funcion_con_mensaje (){

        g_message ("Yo soy solamente un mensaje de aviso\n");
}

void
funcion_con_aviso (){

        g_warning ("Esto es un aviso, algo puede ir mal\n");
}

void
funcion_con_aviso_critico (){

        g_critical ("Esto se pone difícil :( \n");
}

void
funcion_con_un_error (){

        g_error ("3, 2, 1... acabo de morirme. RIP\n");
}

main (){

        funcion_con_mensaje ();
        funcion_con_aviso ();
        funcion_con_aviso_critico ();
        funcion_con_un_error ();
}
```

Aunque este ejemplo no tiene mucha funcionalidad, sirve para ver en acción a las cuatro macros en un mismo programa y da la posibilidad de contemplar en la salida de este programa los mensajes que envía a la consola. Si se ejecuta el ejemplo su salida sería:

```
Message: Yo soy solamente un mensaje de aviso


(process:603): ** WARNING **: Esto es un aviso, algo puede ir mal


(process:603): ** CRITICAL **: Esto se pone difícil :(


(process:603): ** ERROR **: 3, 2, 1... acabo de morirme. RIP

aborting...
'trap' para punto de parada/seguimiento (core dumped)
```

Como se puede ver, todas las funciones y macros que se han mostrado en este apartado nos serán especialmente útiles a la hora de realizar un programa y nos posibilitarán una vía estandarizada y muy flexible de emisión de información al exterior. Facilitando la corrección del código y haciendo más facil la tarea del programador.

## Trabajar con cadenas.

### Manipular el contenido de una cadena.

La manipulación de cadenas es muy común en la vida diaria de un desarrollador. El desarrollador necesita realizar tareas como duplicar cadenas, transformarlas en mayúsculas o verificar si algún caracter corresponde a un tipo en especial. Para este tipo de tareas, esta librería dispone de una serie de funciones cuyo uso no dista mucho de las ya desarrolladas en la Libc.

#### Duplicación de cadenas.

Si lo que se desea es duplicar cadenas, las funciones `g_strdup>` y `g_strndup` resolverán esa necesidad. Las dos funciones realizan el mismo trabajo: duplican la cadena que se le ha pasado como parámetro. La diferencia radica en que la segunda limita el número de caracteres que devuelve a un número que le es indicado por un segundo parámetro de la función.

```
gchar * g_strdup (const gchar * cadena_para_duplicar );
```

```
gchar * g_strndup (const gchar * cadena_para_duplicar , gsize
 longitud_maxima );
```

### Verificación de caracteres.

El título se refiere al conjunto de funciones capaces de responder a preguntas como: Â¿Este carácter es un dígito? o Â¿este carácter es una letra?... Este tipo de funciones es muy usado en el tratamiento de cadenas y su funcionamiento es muy simple. Estas funciones reciben un carácter y devuelven cierto o falso. La siguiente tabla explica cuáles son esas funciones y cuál es la verificación que realizan. Se recomienda la observación del prototipo de la función `g_ascii_isalnum`, usado como ejemplo para una mejor comprensión de la tabla, ya que todas las funciones comparten el mismo prototipo.

```
gboolean  g_ascii_isalnum (gchar  caracter );
```

**Tabla 4-5. Funciones de verificación de caracteres.**

| Nombre de función | Es cierto si... |
|---|---|
| g_ascii_isalnum | Es un carácter alfanumérico. |
| g_ascii_isalpha | Es un carácter del alfabeto. |
| g_ascii_iscntrl | Es un carácter de control. |
| g_ascii_isdigit | Es un dígito. |
| g_ascii_isgraph | Es un carácter imprimible y no es un espacio. |
| g_ascii_islower | Es una letra minúscula. |
| g_ascii_isprint | Es un carácter imprimible. |
| g_ascii_ispunct | Es un carácter de puntuación. |
| g_ascii_isspace | Es un espacio. |
| g_ascii_isupper | Es una letra mayúscula. |

### Copia y concatenación de cadenas.

Otra acción típica para el programador es la copia de una cadena o la concatenación de de múltiples cadenas. Para realizar esta tarea podemos utilizar la función `g_stpcpy`. Esta función desempeña la tarea de copiar una cadena que nosotros le pasemos como parámetro a una cadena destino, también pasada como parámetro. Esta función tiene la particularidad de devolver un puntero a la última posición de la cadena destino, de tal modo que, si se quisiera concatenar cadenas, sólo se tendría que pasar el puntero devuelto por el primer `g_stpcpy` al segundo como si fuera el destino. Así el segundo `g_stpcpy` entendería que ha de copiar la segunda cadena al final de la primera que hemos copiado anteriormente y así sucesivamente. Siguiendo este método, se podría concatenar una cadena tras otra.

```
gchar *  g_stpcpy (gchar *  cadena_destino , const gchar *
 cadena_origen );
```

## GString : la otra manera de ver una cadena.

GString es un TAD (tipo abstracto de datos) que implementa GLib. Este TAD es un buffer de texto que tiene la capacidad de expandirse, realojando memoria automáticamente, sin que el programador tenga que actuar. Esto es realmente útil para el trabajo con cadenas, porque cuando nosotros reservamos memoria para texto y, más adelante, se nos queda pequeña, el programador tiene que tomarse la molestia de reservar más. Por eso GLib implementa GString, que facilita enormemente el manejo de cadenas sin necesidad de estar pendientes de su gestión de memoria.

Para empezar a estudiar este TAD, lo primero que tenemos que conocer es la estructura de datos sobre la que esta diseñada y después estudiaremos las funciones que interactúan con esa estructura.

```
struct GString
{                                                   (1)
    gchar  *str;                                    (2)
    gsize len;
    gsize allocated_len;
};
```

**(1)** En este puntero a una cadena de caracteres es donde se almacenará el contenido de la cadena que nosotros especifiquemos. De este modo, si en algún momento se necesita el contenido de la cadena, sólo es necesario acudir a este campo.

**(2)** Este campo hace referencia a la longitud total de la cadena que está alojada en el campo str de la estructura GString. De este modo, si en str está alojada la cadena "hola", el campo len contendrá el número 4.

Ahora ya sabemos cómo es la estructura GString. El siguiente paso será ver las funciones que trabajan sobre GString con el fin de sacarle el mayor partido a este tipo abstracto de datos.

### Cómo se aloja una cadena con GString.

Alojar una cadena con GString es extremadamente sencillo e intuitivo. Para ello usaremos la función `g_string_new ()`, que tiene la forma:

```
GString *  g_string_new (const gchar *  cadena_inicial );
```

Esta función recibe como parámetro una cadena y devuelve la estructura GString. De este modo, dentro de la estructura GString, tendremos toda la información y, si el lector se ha percatado, en la creación de la cadena, nosotros no hemos tenido que realizar ningún movimiento para alojar memoria.

**Ejemplo 4-6. Crear una cadena con GString.**

```
/* ejemplo del uso de g_string_new */
#include <glib.h>

main (){

        GString *cadena ;

        cadena = g_string_new ("!Qué fácil es hacer una cadena!");
```

```
        g_print ("la cadena que hemos alojado : %s\n", cadena->str);
        g_print ("la longitud de la cadena es : %d\n", cadena->len);
}
```

Como se puede ver en la ejecución de este ejemplo, la creación de una cadena con GString es muy sencilla. Pero, además, GLib nos proporciona otra función para crear cadenas. La sinopsis de esta función es:

```
GString *  g_string_new_len (const gchar *  cadena_inicial , gssize
 longitud_inicial );
```

La ventaja que tiene el uso de `g_string_new_len` es que nos permite especificar una longitud inicial y esto trae consigo que la cadena que introducimos no tiene necesariamente que estar terminada con el caracter *NULL* ("\0") y, además, puede tener varios caracteres *NULL* embebidos dentro de la cadena en cuestión.

### Cómo puedo liberar un GString

Siendo GString una estructura, lo suyo sería liberar uno a uno todos los campos pero, gracias a la implementación que nos ofrece GLib, disponemos de una función que nos ayuda con la liberación de memoria que hemos reservado con la creación de un GString. La sinopsis de esta función sería:

```
gchar *  g_string_free (GString *  cadena , gboolean
 liberar_contenido_cadena );
```

Con la función `g_string_free`, nosotros podremos liberar el GString que le pasemos como primer argumento a la función. Además, esta función también nos da la posibilidad de liberar o no la cadena que estaría dentro de GString. Y esto se conseguiría si como segundo parámetro le pasásemos el booleano *TRUE*. En caso contrario, si pasásemos un booleano *FALSE*, la cadena de caracteres no sería liberada.

### Añadir, insertar y borrar cadenas en un GString.

Con los conocimientos necesarios para crear y liberar este tipo de datos, ha llegado el momento de aprender a trabajar con la funciones que tratan la información que aloja y, para empezar, vamos aprender como se puede añadir texto a una cadena creada como GString. A la hora de añadir texto, nos puede interesar añadir texto al final o al principio de la cadena y podemos hacerlo con GString. Las funciones que pasaremos a ver ahora son exactamente iguales, salvo que contienen las palabras *append* o *prepend*, que indican que la función añade el texto al final o al principio respectivamente.

```
GString *  g_string_append (GString *  cadena , gchar *
 cadena_para_añadir_al_final );
```

```
GString *  g_string_prepend (GString *  cadena , gchar *
 cadena_para_añadir_al_principio );
```

Estas son las primeras funciones para el añadido de texto que vamos a ver. La primera, `g_string_append`, se encargará de añadir el texto pasado como parámetro al final de la cadena contenida dentro del GString y la segunda, `g_string_prepend`, se encargara de añadir el texto al principio.

Con el siguiente ejemplo, el lector entenderá el uso práctico de estas funciones.

**Ejemplo 4-7. Añadir cadena a un GString.**

```
/* ejemplo del uso de g_string_append y g_string_prepend */
#include <glib.h>

main (){

        GString *cadena ;

        cadena = g_string_new ("*");
        cadena = g_string_append (cadena, "| texto añadido al final");
        cadena = g_string_prepend (cadena, "texto añadido al principio |");

        g_print ("\n%s\n", cadena->str);
}
```

Una vez que el programador sabe añadir texto por delante y por detrás de la cadena contenida dentro del GString, el siguiente paso natural sería aprender cómo se pueden introducir datos dentro de las cadenas, es decir, insertar una cadena. Imagíne que tiene un GString con una cadena. Con la función `g_string_insert` se puede insertar, en la posición elegida, la cadena que desee. Como puede observar, en la sinopsis de la función y en el siguiente ejemplo, Lo único que tiene que hacer es indicar la posición en la que quiere insertar la cadena y la cadena que se insertará.

```
GString * g_string_insert (GString * cadena , gssize posicion ,
gchar * cadena_para_insertar );
```

**Ejemplo 4-8. Insertar una cadena a un GString.**

```
/* ejemplo del uso de g_string_insert */
#include <glib.h>

main (){

        GString *cadena ;
        gchar *cadena_para_insertar = "muy muy ";

        cadena = g_string_new ("El día esta soleado");
        cadena = g_string_insert (cadena, 13, cadena_para_insertar);

        g_print ("\n%s\n", cadena->str);

        g_string_free (cadena, TRUE);

}
```

Con las funciones mostradas anteriormente, podemos añadir información a un GString. Si lo que deseamos es borrar, tenemos la función `g_string_erase`. Sólo hay que indicar la posición desde la que quiere empezar a borrar y el número de caracteres que desea borrar desde esa posición.

```
GString *  g_string_erase (GString *  cadena , gssize  posicion ,
GString *  numero_de_caracteres_que_desea_borrar );
```

**Ejemplo 4-9. Borrar una cadena a un GString.**

```
/* ejemplo del uso de g_string_erase */
#include <glib.h>

main (){

        GString *cadena ;

        cadena = g_string_new ("Esto es una cadena --esto sera borrado--");
        cadena = g_string_erase (cadena, 19, 21);

        g_print ("\n%s\n", cadena->str);
        g_string_free (cadena, TRUE);
}
```

## Introducir texto preformateado a un GString.

Esta es una de las aplicaciones más interesantes que podemos dar a un GString. Este tipo de datos nos permite trabajar con texto preformateado, como cuando trabajamos con un `printf`. Esta opción siempre es interesante, pues una vez que tengamos ese texto dentro del GString, podremos aprovecharnos de las múltiples facetas que posee este tipo.

La siguiente función, `g_string_printf`, recordará al lector a la ya conocida `sprintf()` de C estándar; la diferencia está en que el buffer tiene la capacidad de crecer automáticamente. Otra cosa a tener en cuenta en el uso de esta función es que cuando se usa, el contenido anterior existente (si existiese) es destruido.

```
void  g_string_printf (GString *  cadena , const gchar  formato , ...
);
```

**Ejemplo 4-10. Crear texto preformateado con GString.**

```
/* ejemplo del uso de g_string_printf */
#include <glib.h>

main (){

        GString *cadena ;
        gchar *frase = "lo bueno, si breve, dos veces bueno" ;
        gint numero = 7 ;

        cadena = g_string_new ("");
        g_string_printf (cadena, "FRASE TIPICA : %s\nNUMERO DE LA SUERTE : %d",
                         frase, numero);

        g_print ("%s\n",cadena->str);
        g_string_free (cadena, TRUE);

}
```

GLib también dispone de una función similar a la anterior. La única diferencia estriba en que esta función añade el texto preformateado. Es decir, si tuvieramos un texto dentro del GString, el texto formateado se añadiría detrás del texto existente. Esto le puede resultar útil en el sentido de que la anterior función destruía el contenido anterior y puede que eso no le convenga.

```
void  g_string_append_printf (GString *  cadena , const gchar  formato
, ... );
```

**Ejemplo 4-11. Añadir texto preformateado con GString.**

```
/* ejemplo del uso de g_string_append_printf */
#include <glib.h>

main (){

        GString *cadena ;
        gchar *frase = "lo bueno, si breve, dos veces bueno" ;
        gint numero = 7 ;

        cadena = g_string_new ("Línea ya existente\n");
        g_string_append_printf (cadena, "FRASE TIPICA : %s\nNUMERO DE LA SUERTE : %d",
                                frase, numero);
        g_print ("%s\n",cadena->str);
        g_string_free (cadena, TRUE);

}
```

## Otras funciones útiles para el manejo de GString.

En las secciones anteriores se han explicado las funciones más comunes con las que poder manejar una estructura del tipo GString, pero estas funciones no son las únicas que existen para trabajar con este tipo de datos. En la siguiente sección, se explicarán otras que, aunque no son tan fundamentales, pueden resultar útiles.

Para empezar, puede que al programador no le interese insertar o añadir una cadena y sólo quiera trabajar con un carácter. Para ello dispone de estas funciones que, aunque surten el mismo efecto que si usase `g_string_append` y similares, con un sólo carácter, puede que estas funciones le sirvan para dar más legibilidad a su código en un momento dado.

```
GString *  g_string_append_c (GString *  cadena , gchar  caracter );
```

```
GString *  g_string_prepend_c (GString *  cadena , gchar  caracter);
```

```
GString *  g_string_insert_c (GString *  cadena , gssize  posicion ,
gchar *  caracter );
```

**Ejemplo 4-12. Añadir e insertar caracteres a un GString.**

```
/* ejemplo del uso de g_string_{insert|append|prepend}_c */
#include <glib.h>

main (){

        GString *cadena;
        gchar caracter ;

        cadena = g_string_new ("");

        caracter = 'c' ;
        cadena = g_string_append_c (cadena, caracter);
        caracter = 'a' ;
        cadena = g_string_prepend_c (cadena, caracter);
        caracter = 'b' ;
        cadena = g_string_insert_c (cadena, 1, caracter);

        g_print ("la cadena resultante es... %s\n",cadena->str);
        g_string_free (cadena, TRUE);

}
```

Otras acciones que pueden interesarle son la capacidad de truncar un texto por el lugar que usted desee y la capacidad de convertir todo el texto a mayúsculas o minúsculas. Pues para este tipo de necesidades usted dispone de `g_string_truncate`, cuya sintaxis se muestra a continuación.

```
GString * g_string_truncate (GString * cadena , gsize longitud );
```

*longitud* se refiere al número de caracteres desde el principio de la cadena que desea que permanezcan una vez terminada la función.

```
GString * g_string_up (GString * cadena );
```

```
GString * g_string_down (GString * cadena );
```

La primera función, `g_string_up`, es la que convierte a mayúsculas un texto íntegro del GString y la segunda, `g_string_down` la que lo convierte en minúsculas.

## Jugando con el tiempo.

### Funciones de manejo de fechas y horas.

GLib proporciona una API completa para poder programar, utilizando fechas dentro de nuestras aplicaciones de forma sencilla. El objetivo fundamental es dar soporte a las fechas, cuya programación puede ser pesada sin estas funciones, aunque también se proporcionan algunos métodos para relacionar tiempos y fechas. Para ello,

GLib dispone de la estructura GDate, que es la estructura que incorpora toda la información sobre las fechas y, además, una serie de funciones que interactúan con esa estructura.

Antes de empezar, han de aclararse unos conceptos sobre la expresión del tiempo, en cuanto al formato de las fechas se refiere. Hay dos formas de expresar las fechas: día-mes-año y formato Juliano; este último es simplemente el número de días que han transcurrido desde una fecha de referencia (en el caso de GLib, dicha fecha es el 1 de Enero del año 1).

### Creación de una nueva fecha.

El primer paso para poder utilizar una fecha es crearla. Para crear una fecha disponemos de varias funciones, la primera, `g_date_new`, nos permite crear una estructura GDate sin ningún valor. Esta función resultará útil más adelante cuando veamos como podemos ajustar nuestro GDate a una determinada fecha. Las dos siguientes permiten crear una fecha pero, al contrario que la anterior, que sólo crea la estructura, éstas añaden además la fecha: la función `g_date_new_julian`, que nos permite crear una fecha utilizando el formato juliano y, por último, `g_date_new_dmy`, a la que le pasamos un día, mes y año para crear el GDate.

```
GDate * g_date_new ( void );
```

```
GDate * g_date_new_julian (guint32 dia_juliano );
```

```
GDate * g_date_new_dmy (GDateDay dia , GDateMonth mes , GDateYear año
);
```

En esta última función, es conveniente que se examinen los tipos de datos GDateDay,GDateMonth,GDateYear.

El tipo de datos GDateDay es un entero (entre 1 y 31) que representa el día del mes. La constante *G_DATE_BAD_DAY* representa un día inválido del mes.

El tipo GDateMonth es un tipo enumerado, por tanto nos podemos referir a él con un número del 1 al 12, refiriéndonos a los meses de enero a diciembre o también con los elementos *G_DATE_JANUARY*, *G_DATE_FEBRUARY*, etc. La constante *G_DATE_BAD_MONTH* representa un mes inválido.

Y, por último, está el tipo GDateYear. Este tipo es, en realidad, un entero; por tanto, para representar el año, basta con poner el número del año deseado.

## Midiendo intervalos de tiempo con GTimer.

Dentro del mundo de la programación, medir el tiempo entre dos instantes dentro del programa es una actividad bastante común. Por ejemplo, en programas que transfieren un fichero y miden la velocidad de transferencia o en programas de instalación, para mostrar por pantalla el tiempo de instalación usado. La medida del tiempo es tan necesaria como, a veces, dependiente de la plataforma para la que desarrollemos. Por eso, GLib nos provee con la estructura GTimer y sus funciones asociadas, gracias a lo cual, podremos calcular intervalos de tiempo con una resolución del orden de los microsegundos.

### Descripción de las funciones para manejar GTimer.

La estructura GTimer es opaca para el programador. Todo lo necesario para gestionar GTimer se realiza con las funciones que se describen a continuación, lo que quiere decir que no se mostrará en el libro la estructura en detalle como se hizo en otras secciones, ya que ni el mismo programador la ve. Sólo se interactúa con ella a través de funciones, como ya se ha dicho.

Para empezar, lo primero que se ha de hacer, como en la gran mayoría de estructuras de GLib, es inicializar o crearla. Para este fin se dispone de la función `g_timer_new`. Esta función se encarga única y exclusivamente de inicializar la estructura GTimer.

```
GTimer *  g_timer_new ( void );
```

Una vez inicializada la estructura GTimer, lo siguiente es ver qué se puede hacer con ella. Como se dijo anteriormente, esta estructura está dedicada a medir intervalos de tiempo y se proponía el ejemplo de un programa que transfería un fichero y media el tiempo de transferencia. Si se abstrae el problema en sí y se fija la atención solamente en lo que tendría que hacer el programa para medir los tiempos, se llegaría a la conclusión que se necesitará una función que arranque el contador de tiempo para que, cuando empezase la transferencia, se marcase de alguna manera el tiempo en el que se empezó a transferir, y una función que pare el contador cuando el fichero acabe de transferirse. Y, por último, una función con la cual se pueda consultar el tiempo en un momento dado, una vez arrancado el contador. Pues todas esas funciones están implementadas en GTimer>.

Para empezar, existe la función `g_timer_start`, que tiene la función de marcar el tiempo de inicio en el que se ha arrancado el contador. Luego está `g_timer_stop` que marca el momento final de la medida de tiempo. Y por último `g_timer_elapsed`, que tiene un carácter más complejo, pero que resulta sencillo de comprender. Esta función devuelve el número de segundos que hayan transcurrido y, si se desea más precisión el puntero a *`microsegundos`* devolverá el número de microsegundos. Ahora bien, si se ha iniciado el contador pero aún no se ha detenido con `g_timer_stop`, obtiene el tiempo transcurrido desde que se inició el contador. Si se ha llamado a `g_timer_stop`, obtiene el tiempo transcurrido entre el inicio del contador de tiempo y el momento en que se paró.

```
void  g_timer_start ( GTimer *  contador  );
```

```
void  g_timer_stop ( GTimer *  contador  );
```

```
gdouble  g_timer_elapsed ( GTimer *  contador  ,  gulong *
 microsegundos  );
```

Ahora, en un contexto un poco más avanzado, GTimer también dispone de una función que reinicia el contador: `g_timer_reset`. Esta función es equivalente a llamar a `g_timer_start` si el contador ya está activo, es decir, reinicia el contador de tiempo poniendo como instante inicial, el momento de la llamada a esta función. En el caso de estar parado, lo pone a cero.

```
GTimer *  g_timer_reset ( GTimer *  timer  );
```

Y, por ultimo, y dado que la estructura GTimer es una estructura opaca al desarrollador, necesitamos una función que libere todos los recursos de memoria que haya requerido la estructura, es decir, una función que libere de memoria la estructura GTimer. Para ello está `g_timer_destroy`.

```
GTimer *  g_timer_destroy ( GTimer *  timer  );
```

Como unas líneas de código valen más que mil palabras, aquí se propone un ejemplo que aclara dudas sobre el uso de estas funciones si las hubiere.

**Ejemplo 4-13. Ejemplo de uso de GTimer**

```
/* ejemplo del uso de GTimer */
#include  <glib.h>

int main () {

        GTimer *timer1, *timer2;
        gdouble elapsed;
        gulong usec;

        timer1 = g_timer_new (); /* Llama implícitamente a g_timer_start() */
        timer2 = g_timer_new ();

        g_print ("Duermo 5 segundos...");
        sleep (5);
        g_print ("\n\n");

        g_print ("Paro el timer 1...\n");
        g_timer_stop (timer1);

        elapsed = g_timer_elapsed (timer1, <symbol>&amp;</symbol>usec);
        g_print ("Tiempo transcurrido timer 1: %gs %luus\n", elapsed, usec);

        elapsed = g_timer_elapsed (timer2, <symbol>&amp;</symbol>usec);
        g_print ("Tiempo transcurrido timer 2: %gs %luus\n\n", elapsed, usec);

        g_print ("Duermo 1 segundo más...\n");
        sleep (1);

        elapsed = g_timer_elapsed (timer1, <symbol>&amp;</symbol>usec);
        g_print ("Tiempo transcurrido timer 1 (igual que antes):
                      %gs %luus\n", elapsed, usec);
        elapsed = g_timer_elapsed (timer2, <symbol>&amp;</symbol>usec);
        g_print ("Tiempo transcurrido timer 2 (un segundo más):
                      %gs %luus\n\n", elapsed, usec);

        g_print ("Reseteo el timer 1 (que esta parado).\n");
        g_timer_reset (timer1);
        g_print ("Reseteo el timer 2 (que esta activo).\n");
        g_timer_reset (timer2);

        g_print ("Duermo otro segundo más...\n");
        sleep (1);

        elapsed = g_timer_elapsed (timer1, <symbol>&amp;</symbol>usec);
        g_print ("Tiempo transcurrido timer 1
                      (0, no se ha llamado a g_timer_start()):
                      %gs %luus\n", elapsed, usec);

        elapsed = g_timer_elapsed (timer2, <symbol>&amp;</symbol>usec);
        g_print ("Tiempo transcurrido timer 2 (1 segundo):
```

```
                             %gs %luus\n\n", elapsed, usec);

          g_print ("Llamo a g_timer_start () con timer 1.\n");
          g_timer_start (timer1);

          g_print ("Y vuelvo a dormir, 2 segundos...\n");
          sleep (2);

          elapsed = g_timer_elapsed (timer1, <symbol>&amp;</symbol>usec);
          g_print ("Tiempo transcurrido timer 1 (2 segundos):
                             %gs %luus\n", elapsed, usec);
          elapsed = g_timer_elapsed (timer2, <symbol>&amp;</symbol>usec);
          g_print ("Tiempo transcurrido timer 2 (3 segundos):
                             %gs %luus\n\n", elapsed, usec);

          g_timer_destroy (timer2);
          g_timer_destroy (timer1);

          return 0;
}
```

## Miscelánea de funciones.

### Números aleatorios.

En determinadas ocasiones hace falta tener unos ciertos valores aleatorios, por ejemplo a la hora de programar juegos. En los juegos, muchas veces, hace falta elegir al azar qué carta se va a levantar, dónde se va a poner la mina, etc. Para tomar este tipo de decisiones hacen falta unos valores aleatorios, o sea, al azar o que se acerquen mucho a ello. Por esta necesidad es que se ha implementado en GLib la librería `gran.h`, que consiste en una estructura de datos GRand y un conjunto de funciones. Estas funciones sirven para inicializar la estructura GRand, darle valores iniciales y obtener números aleatorios de distintos tipos.

Lo de los tipos es en el sentido más informático de la palabra, porque todos son números aleatorios igualmente, con la diferencia de que en unos se obtiene un guint32, gdouble, gboolean (este último, evidentemente no es un número, es un booleano). Después algunas funciones establecen rangos para los números. Vamos, un número aleatorio entre el X y el Y.

Cuando se habla de números pseudoaleatorios, hay que hablar irremediablemente de dos conceptos: PRNG(generador de números pseudo aleatorios) y "semilla".

Un PRNG no es más que una estructura de datos con unos algoritmos asociados que, a partir de un valor inicial, genera secuencias de números en un orden, aparentemente aleatorio o, lo que es lo mismo, al azar y es aparentemente porque no es real; si se tuviera la tecnología y el tiempo suficiente, se podría encontrar un cierto orden lógico en las secuencias y predecirlas, pero eso es otro tema; a los efectos, serán considerados como realmente aleatorios.

En el caso de GLIB, este PRNG, será GRand, que es una estructura que, aunque oculta a la visión del programador, es usada internamente por las funciones de la librería `grand.h`.

El otro concepto es el de semilla. La semilla, no es más que ese valor inicial del que se hablaba antes. Un valor, a ser posible, bastante aleatorio en sí mismo. Para conseguir esta aleatoriedad, lo que se hace es pasarle este valor a la función correspondiente. En caso de que no se quiera pasar la semilla, el constructor de GRand, `g_rand_new()`, se

encargará de obtener un valor aleatorio de `/dev/urandom`, si existe, o la hora actual si no existiera.

Para usar este generador de números pseudo aleatorios, es necesario incluir la librería `grand.h`.

```
#include <grand.h>
```

Para obtener números aleatorios rápidamente se pueden usar las siguientes funciones:

```
void g_random_set_seed(guint32 semilla);
```

Con esta función se establece la semilla para la generación de aleatorios. Estableciendo esta semilla con un mismo número cada vez que se ejecute nuestro código, se obtendrá el mismo resultado para los números aleatorios que generemos. Si se desea que los aleatorios no coincidan nunca no estableceremos la semilla.

```
gboolean g_random_boolean();
```

Con esta función, se obtiene un booleano aleatorio.

```
guint32 g_random_int(void);
```

Se obtiene un gunt32 distribuido sobre el rango [0..2^32-1].

```
gint32 g_random_int_range(gint32 principio, gint32 end);
```

Se obtiene un gint32 distribuido sobre el rango [principio..fin-1].

```
gdouble g_random_double(void);
```

Se obtiene un gdouble distribuido sobre el rango [0..1).

```
gdouble g_random_double_range(gdouble principio, gdouble fin);
```

Se obtiene un gdouble distribuido sobre el rango[principio..fin).

En todos estos casos, internamente, se ha hecho uso de un tipo de dato: GRand. Si lo que necesitamos es obtener una serie de números aleatorios reproducibles, será una mejor opción trabajar directamente con este dato.

La manera de hacerlo es crear primero un GRand y después trabajar con las funciones del tipo g_rand*.

```
GRand *g_rand_new(void);
```

Crea un nuevo GRand. Los números aleatorios necesitan una semilla para inicializarse, pero en esta función no se le pasa ninguna, así que se tomaría como semilla un valor de `/dev/urandom`, si existe, o la hora actual.

```
GRand *g_rand_new_with_seed(guint32 semilla);
```

En este caso se crea un GRand usando una semilla.

```
void g_rand_set_seed(GRand *rand, guint32 semilla);
```

Con esta función se establece una nueva semilla a un GRand.

```
void g_rand_free(GRand *rand);
```

Libera la memoria usada por un GRand.

```
gboolean g_rand_boolean(GRand *rand);
```

Devuelve el siguiente booleano aleatorio desde rand.

```
guint32 g_rand_int(GRand *rand);
```

Devuelve el siguiente guint32 aleatorio desde rand entre los valores [0..2^32-1].

```
gint32 g_rand_int_range(GRand *rand, gint32 principio, gint32 fin);
```

Devuelve el siguiente gint32 aleatorio desde rand entre los valores [principio..fin-1].

```
gdouble g_rand_double(GRand *rand);
```

Devuelve el siguiente gdouble aleatorio desde rand entre los valores [0..1).

```
gdouble g_rand_double_range(GRand *rand, gint32 principio, gint32
fin);
```

Devuelve el siguiente gdouble aleatorio desde rand entre los valores [principio..fin).

Con estas funciones se pueden obtener números o series de números pseudoaleatorios, de una manera rápida y fácil. Además con la ventaja añadida que ofrece GLIB de la portabilidad.

### Funciones de información sobre entorno.

Muchas veces, en los desarrollos de software, es necesario conocer e interactuar con información del entorno en el cual se esté trabajando. Por ejemplo, en un momento dado, podría ser necesario conocer información del usuario como su nombre, su directorio de trabajo o el directorio donde se pueda almacenar información temporal. Para todas estas acciones e incluso algunas más, se dispone de las siguientes funciones.

```
gchar*  g_get_user_name (void);
```

```
gchar*  g_get_home_dir (void);
```

```
gchar* g_get_tmp_dir(void);
```

```
gchar* g_get_current_dir(void);
```

Después de haber visto la sintaxis de estas funciones no debe ser muy difícil imaginarse para qué sirve cada una, pero por si queda alguna duda, la primera devolverá el nombre del usuario que esté ejecutando el programa en ese momento; la segunda, su directorio de trabajo; la tercera, el directorio temporal habilitado en su sistema para ese fin y la última función nos devolverá el directorio en el que se esté trabajando actualmente.

Estas funciones nos pueden resultar útiles, pero para aquellos que provengan del universo UNIX o GNU/Linux puede que no les sea suficiente. Esto es debido a que, en este tipo de sistemas operativos, el uso de variables de entorno en sus diferentes *shells* es muy extendido y en un momento dado, les interesaría recurrir a ellas. Para ello está la función `g_getenv`. La cual es realmente agradable de usar ya que sólo necesitamos pasarle como parámetro la variable que necesitemos y esta función nos devolverá su valor en una cadena.

```
gchar*  g_getenv (const gchar * variable );
```

## Bucles de ejecución.

Los bucles de ejecución son estructuras que nos permiten realizar programas asíncronos, es decir, no bloqueantes; nos permiten ejecutar un bucle que permanece a la escucha de diversos eventos y que envía "señales" a distintas funciones que se registran como interesadas en determinados eventos.

De esta manera podemos realizar aplicaciones, que por ejemplo, realicen cierta función cada determinado tiempo, esto se logra mediante "alarmas", pero no solo se limita a intervalos de tiempo, tambien hay la posibilidad de ejecutar funciones que respondan a eventos de dispositivos de E/S o intervalos de inactividad de un programa.

La manera de utilizar los bucles de ejcución es:

- Primero creamos la variable GMainloop

```
GMainLoop *bucle;
```

- Luego creamos el bucle

```
bucle = g_main_loop_new(NULL, FALSE);
```

- Finalmente corremos el bucle

```
g_main_loop_run(bucle);
```

- Para detener el bucle se utiliza

```
g_main_loop_quit (bucle);
```

Una vez que utilicemos `g_main_loop_run`, el control de nuestra aplicación pasa a manos de Glib. Para poder utilizar las alarmas tenemos dos opciones, una cuando se cumple cierto intervalos de tiempo, `g_timeout_add`, y la otra cuando nuestra aplicacion no esta haciendo nada, `g_idle_add`.

## Alarmas

Para utilizar un bucle conjuntamente con una alarma se realizan los pasos anteriormente explicados, pero se usa la siguiente función:

```
guint  g_timeout_add (guint intervalGSourceFunc function gpointer
data);
```

Veamos los parámetros de esta función:

- *guint interval*: Es el intervalo de tiempo en que se llama a la función, debe de estar en milisegundos.
- *GSourceFunc function*: Funcion a llamar.
- *gpointer data*: Dato que se pasa a la función como parametro.
- *Retorno:*: El id del evento origen.

## Tiempos de inactividad.

Lo mismo pasa con los tiempo de inactividad. Los tiempo de inactividad, son los tiempos en que nuestra aplicación no hace nada, esta inactiva. Para usarlos se utiliza la siguiente función:

```
guint  g_idle_add (GSourceFunc function gpointer data);
```

Veamos los parámetros de esta función:

- *GSourceFunc function*: Función a llamar.

- *gpointer data*: Dato que se pasa a la función como parámetro.
- *Retorno*: El id del evento origen.

Una cosa que hay que tener en cuenta es que en ambos casos cuando se llama a la función externa, la prioridad de esta es asignada de modo por default, `G_PRIORITY_DEFAULT`. Como ya se habia comentado, la prioridad de la funcion a llamar es asignada automáticamente; asi pues, la función puede ser retrasado por otros procesos de mayor prioridad.

En ambas funciones, la llamada a la función externa tiene que se de cierto tipo, es decir, `GSourceFunc` representa un puntero a una función de la siguiente forma:

```
gboolean Funcion_a_llamar (gpointer data);
```

Asi la función es llamada en varias ocaciones hasta que devuelve FLASE y sale del bucle.

Veamos un ejemplo de la forma de utilizar la función `g_timeout_add` .

```
#include <glib.h>

gboolean func(gpointer data);

int
main (int argc, char *argv[])
{
        GMainLoop *bucle;
        guint b;
        guint i = 5000;

        bucle = g_main_loop_new(NULL, FALSE);
        b = g_timeout_add(i, func, bucle);
        g_main_loop_run(bucle);
        g_print("Ya salimos del bucle.\n");
        return 0;
}

gboolean func(gpointer data)
{
        gint i=0;
        do{
                g_print("Uso de g_timeout_add: %d\n", i);
                i++;
        }while(i<3);
        g_print("Ahora vamos a finalizar el bucle.\n");
        g_main_loop_quit(data);

        return FALSE;
}
```

## Tratamiento de ficheros y canales de entrada/salida.

Para el tratamiento de ficheros y canales de entrada/salida en general, GLib provee un tipo de dato llamado GIOChannel, que permite encapsular ficheros, sockets y pipes. Esta abstracción no sólo posibilita el acceso uniforme a todos estos recursos, sino que también asegura portabilidad e integración al bucle de eventos.

Actualmente sólo hay soporte completo para UNIX, aunque el tratamiento con ficheros puede hacerse independientemente de la plataforma.

## Obtención de un GIOChannel.

Para crear un GIOChannel, primero se debe crear el descriptor de fichero de UNIX por los métodos convencionales (open, socket, etc.) y luego utilizar g_io_channel_unix_new o bien g_io_channel_new_file para acceder a ficheros directamente.

```
GIOChannel *g_io_channel_unix_new(int fd);
```

Esta función devuelve el canal creado a partir del descriptor. Dicho parámetro se puede recuperar después utilizando la función `g_io_channel_unix_get_fd`.

```
GIOChannel *g_io_channel_new_file(const gchar *nombre_fichero, const
gchar *modo, GError **error);
```

Devuelve un nuevo canal para acceder al archivo `nombre_fichero`. El modo tiene la misma forma que el parámetro *modo* de la función de C estándar `fopen`. Esto es:

- *r* para lectura.
- *r+* para lectura y escritura.
- *w* para escritua (Si el fichero no existe, lo crea; en caso contrario lo trunca).
- *w+* para lectura y ecritura (con el fichero hace lo mismo que el parámetro *w*
- *a* para escritura al final del fichero (lo crea si no existe).
- *a+*para lectura y escritura al final del fichero (lo crea si no existe).

El parámetro *error* debe apuntar a una estructura tipo GError previamente creada y es aquí donde se informa de cualquier problema al abrir el fichero. En este último caso la función retorna *NULL*.

## Generalidades en el trabajo con GIOChannels.

Una vez obtenido el GIOChannel, en general, todas las llamadas a funciones para realizar alguna operación sobre el canal (lectura, escritura, etc.) tomarán como primer parámetro al propio canal y como último un puntero, *error*, a una estructura GError, donde se informará de cualquier inconveniente ocurrido al intentar la operación. Además, la mayoría de las funciones retornan un código del tipo GIOStatus que puede tomar los siguientes valores:

- *GIO_STATUS_ERROR*: ocurrió un error (los detalles del mismo se encuentran en el parámetro *error*).
- *G_IO_STATUS_NORMAL*: la operación se realizó con éxito.
- *G_IO_STATUS_EOF*: se alcanzó el fin de fichero.
- *G_IO_STATUS_AGAIN*: el recurso solicitado no está disponible (p.e., se intentó leer datos, pero no estaban en ese momento).

En caso de que el código de retorno sea *G_IO_STATUS_ERROR*, en la estructura GError se almacena otro código, indicando con mayor detalle la naturaleza del problema. Este nuevo código es de tipo GIOChannelError y puede tomar los siguientes valores:

- *G_IO_CHANNEL_ERROR_FBIG*: el fichero es muy grande.
- *G_IO_CHANNEL_ERROR_INVAL*: argumento inválido.
- *G_IO_CHANNEL_ERROR_IO*: error general de entrada/salida.
- *G_IO_CHANNEL_ISDIR*: el fichero es un directorio, pero se intentó acceder a él como unfichero regular.
- *G_IO_CHANNEL_ERROR_NOSPC*: No hay más espacio disponible para escritura.
- *G_IO_CHANNEL_ERROR_NXIO*: No existe tal dispositivo o dirección.
- *G_IO_CHANNEL_ERROR_OVERFLOW*: el valor proporcionado es mayor de lo que permite su tipo.
- *G_IO_CHANNEL_ERROR_PIPE*: algunos de los extremos del pipe lo desconectó.
- *G_IO_CHANNEL_ERROR_FAILED*: otros errores.

Los primeros valores tienen su correspondencia directa con los valores de la variable `errno` estándar de C. El último se reserva para otra clase de errores no identificados.

Los canales tienen un tiempo de vida controlado por conteo de referencias. Inicialmente, los canales se crean con una cuenta de 1 y, cuando la misma cae a 0, se destruyen. Para obtener una referencia se utiliza la función g_io_channel_ref y para liberar una referencia g_io_channel_unref. Cabe aclarar que, aunque el objeto GIOChannel se destruya, no necesariamente se cierra el canal. En concreto, si el objeto fue creado a partir de un descriptor de UNIX, el programador es responsable de cerrarlo (a menos que se utilice la función `g_io_channel_set_close_on_unref`).

Las funciones mencionadas tienen la siguiente forma:

```
void g_io_channel_ref(GIOChannel *canal);
```

```
void g_io_channel_unref(GIOChannel *canal);
```

Para cerrar de un canal se utiliza `g_io_channel_shutdown`:

```
GIOStatus g_io_channel_shutdown(GIOChannel *canal, gboolean descarga,
GError **error);
```

Aquí, el parámetro *descarga* controla si se escriben los datos que pueda haber en el buffer interno del canal antes de cerrarlo (para forzar una descarga en cualquier momento se puede utilizar `g_io_channel_flush`).

## Operaciones básicas.

GLib provee una serie de funciones para acceder a un canal abierto de diversas maneras: por caracteres individuales, por bloques de cantidad de bytes fijos y por líneas.

Si el tipo de canal lo permite, la función g_io_channel_seek_position sirve para establecer la siguiente posición donde se leerá o escribirá. Normalmente sólo los ficheros permiten posicionamiento aleatorio.

```
GIOStatus g_io_channel_seek_position(GIOChannel *canal, glong posicion,
GSeekType tipo_posicion, GError **error);
```

La semántica de posición (que es un número con signo) queda determinada por el *tipo_posicion*, que puede tomar los siguientes valores:

- *G_SEEK_CUR*: la nueva posición es relativa a la actual.
- *G_SEEK_SET*: la nueva posición se toma a partir del principio del fichero.
- *G_SEEK_END*: la nueva posición es absoluta respecto al fin de archivo.

Para efectuar lecturas en un canal se utilizan algunas de las siguientes funciones:

```
GIOStatus g_io_channel_read_chars(GIOChannel *canal, gchar *buffer,
gsize tamaño_buffer, gsize *bytes_leidos, GError **error);
```

Lee una cantidad máxima (*tamaño_buffer*) de bytes del canal como caracteres y los almacena en buffer. Éste debe estar previamente reservado (p.e. con `g_malloc`). *bytes_leidos* contiene la cantidad de bytes efectivamente leídos (puede ser menor que la solicitada).

Es importante notar que si el canal está configurado para trabajar con caracteres Unicode, un carácter puede ocupar más de un byte y, si el buffer no es lo suficientemente grande, puede que no sea posible almacenar ningún carácter. En estas condiciones *bytes_leidos* es cero, pero no hay indicación de error.

```
GIOStatus g_io_channel_read_line(GIOChannel *canal, gchar
**puntero_cadena, gsize *bytes_leidos, gsize *terminador, GError
**error);
```

Lee una línea completa del canal y almacena el puntero a la misma en *puntero_cadena*>. Una vez finalizada la operación con la línea, es responsabilidad del programador liberar la memoria con `g_free`. En *bytes_leidos* se devuelve la cantidad total de bytes leídos (incluyendo caracteres de fin de línea) y en terminador, sólo la cantidad perteneciente a la línea en sí. Tanto *bytes_leidos* como terminador pueden ser nulos si la información no es de interés.

```
GIOStatus g_io_channel_read_line_string(GIOChannel *canal, GString
*cadena, gsize *terminador, GError **error);
```

Similar a g_io_channel_read_line, sólo que, para devolver el resultado, utiliza una estructura GString. *terminador* puede ser nulo.

```
GIOStatus g_io_channel_read_to_end(GIOChannel *canal, gchar
**puntero_cadena, gsize *bytes_leidos, GError **error);
```

Lee todos los bytes disponibles en el canal hasta encontrar el fin de fichero. En *puntero_cadena* se devuelve un puntero al espacio de memoria asignado a tal fin que luego debe ser liberado con `g_free`.

Para la escritura se utiliza la contraparte de g_io_channel_read_chars:

```
GIOStatus g_io_channel_write_chars(GIOChannel *canal, gchar *buffer,
gsize cantidad_bytes, gsize *bytes_escritos, GError **error);
```

Como su propio nombre indica, esta función escribe en el canal `cantidad_bytes` del buffer. Puede que no lleguen a escribirse todos los bytes que se solicitaron, por lo que la función devuelve la cantidad de bytes efectivamente escritos. Si `cantidad_bytes` es -1 se considera al buffer como terminado en nulo.

Se aplican las mismas restricciones en cuanto a codificación del canal que para g_io_channel_read_chars.

## Integración de canales al bucle de eventos.

Uno de los aspectos más interesantes de los GIOChannels es que es posible integrarlos fácilmente al bucle de eventos y así, por ejemplo, hacer que la aplicación reaccione ante la llegada de datos.

Los posibles tipos de eventos que puede generar un canal están dados por el tipo GIOCondition que toma los siguientes valores:

- *G_IO_IN*: hay datos disponibles en el canal.
- *G_IO_OUT*: se pueden escribir datos al canal sin que éste bloquee.
- *G_IO_PRI*: hay datos urgentes para leer.
- *G_IO_ERR*: condición de error.
- *G_IO_HUP*: desconexión; normalmente, se aplica a sockets y pipes. Alguno de los extremos rompió la conexión.
- *G_IO_NVAL*: solicitud no válida (p.e. el descriptor UNIX no está abierto).

La forma básica de integrar un objeto generador de eventos al bucle es obtener una estructura GSource. En el caso de los canales, la función para ello es `g_io_create_watch`:

```
GSource *g_io_create_watch(GIOChannel *canal, GIOCondition condición);
```

Esta función devuelve un GSource que luego se registra en algún bucle de eventos con `g_source_attach`. Sin embargo, por conveniencia, GLib provee un par adicional de funciones para integrar directamente un canal al bucle principal: `g_io_watch` y `f_io_add_watch_full`.

```
guint g_io_add_watch(GIOChannel *canal, GIOCondition condición,
GIOFunc función, gpointer datos_usuario);
```

Esta función, que devuelve el identificador de evento, registra el canal para que sea controlado según las condiciones especificadas. En caso de que se cumpla alguna de las condiciones, se llama a la función con los `datos_usuario` adicionales.

La función especificada en el registro debe tener la siguiente forma:

```
gboolean (* GIOFunc)(GIOChannel *canal, GIOCondition condición,
gpointer datos_usuario);
```

La condición que la función recibe es aquella que causó el evento en primer lugar. `datos_usario` es lo que se especificó en el momento del registro.

La otra función, `g_io_add_watch_full` permite además especificar la prioridad del canal y una función de notificación cuando el canal se desconecta del bucle de eventos.

## Configuración avanzada de canales.

### Codificación

Los GIOChannels son capaces de codificar o decodificar los caracteres que se escriben o leen del mismo. La codificación que se utiliza internamente es UTF-8, por lo que, eventualmente, se puede transformar de UTF-8 a alguna otra codificación para escribir a un fichero, o viceversa.

Si se necesita que un canal no efectúe transformación alguna a los datos (por ejemplo, para tratamiento de datos binarios) se debe optar por la codificación nula (NULL).

Las funciones para manipular la codificación son `g_io_channel_set_encoding` y `g_io_channel_get_encoding`.

```
GIOStatus g_io_channel_set_encoding(GIOChannel *canal, const gchar
*codificación, GError **error);
```

```
G_CONST_RETURN gchar *g_io_channel_get_encoding(GIOChannel *canal);
```

Por defecto, la codificación externa utilizada para ficheros es UTF-8. Para cambiarla se deben tener en cuenta las siguientes condiciones:

- El canal acaba de ser creado.
- El canal es para escritura solamente.
- El canal es sobre un fichero y el puntero de lectura y escritura acaba de ser recolocado con `g_io_channel_seek_position`..
- La codificación actual es UTF-8 o nula.
- Se ha llamado a una función de lectura y ha devuelto *G_IO_STATUS_EOF* o, en el caso de g_io_channel_read_to_end, *G_IO_STATUS_NORMAL*.

### Terminador de línea.

En UNIX, el fin de línea se indica con un carácter de *linefeed* (código ASCII 10), pero esto puede no ser cierto para otras plataformas (p.e. Win32).

Para que las funciones de lectura y escritura por líneas den los resultados esperados para cada plataforma, en caso de ser necesario el acceso a ficheros escritos en otra plataforma, se debe configurar el terminador de línea en el canal. Las funciones para ello son:

```
void g_io_channel_set_line_term(GIOChannel *canal, const gchar
*terminador, gint longitud);
```

```
G_CONST_RETURN gchar *g_io_channel_get_line_term(GIOChannel *canal,
gint *longitud);
```

### Configuración de buffer interno.

La API de GIOChannel nos permite modificar alguno de los parámetros de funcionamiento interno del canal. Es posible modificar el tamaño del buffer interno del mismo para optimizar el rendimiento de lectura y escritura en casos especiales. Para ello se utilizan las funciones `g_io_channel_get_buffer_size` y `g_io_channel_set_buffer_size`.

```
gsize g_io_channel_get_buffer_size(GIOChannel *canal);
```

```
void g_io_channel_set_buffer_size(GIOChannel *canal, gsize tamaño);
```

Si en `g_io_channel_set_buffer_size` se especifica un tamaño 0, GLib elegirá un tamaño adecuado al canal.

Hemos visto antes que, utilizando el bucle de eventos, se puede monitorear un canal y producir llamadas a funciones ante determinados eventos. También es posible verificar manualmente si un canal tiene datos listos para la lectura o si se pueden escribir datos en él. Para ello se utiliza la función`g_io_channel_get_buffer_condition>`.

```
GIOCondition g_io_channel_get_buffer_condition(GIOChannel *canal);
```

Esta función puede devolver solamente *`G_IO_IN`* y *`G_IO_OUT`* y, por supuesto, su significado es exactamente el mismo que se explicó antes.

Por último, con las funciones `g_io_channel_set_flags` y `g_io_channel_get_flags`, es posible saber y, en parte, modificar el comportamiento que tiene un canal.

```
GIOFlags g_io_channel_get_flags(GIOChannel *canal);
```

```
GIOStatus g_io_channel_set_flags(GIOChannel *canal, GIOFlags banderas,
GError **error);
```

Los valores que puede tomar el parámetro banderas son *`G_IO_FLAG_APPEND`* y/o *`G_IO_FLAG_NONBLOCK`* (los demás valores son de sólo lectura). El valor obtenido en `g_io_channel_get_flags` es una composición de:

- *`G_IO_FLAG_APPEND`*: canal en modo de agregado.
- *`G_IO_FLAG_NONBLOCK`*: canal en modo no bloqueante; esto es, si se intenta, p.e., leer y no hay datos disponibles, se devuelve el control inmediatamente con un estado de *`G_IO_STATUS_AGAIN`*.
- *`G_IO_FLAG_IS_READABLE`*: el canal se puede leer.
- *`G_IO_FLAG_IS_WRITEABLE`*: se puede escribir en el canal.
- *`G_IO_FLAG_IS_SEEKABLE`*: se puede mover el puntero cd lectura y/o escritura en el canal con `g_io_channel_seek_position`.

# Manejo de memoria dinámica.

Sobre el tratamiento de memoria, GLib dispone de una serie de instrucciones que sustituyen a las ya conocidas por todos `malloc`, `free`, etc. y, siguiendo con el modo de llamar a las funciones en GLib, las funciones que sustituyen a las ya mencionadas son `g_malloc` y `g_free`.

## Reserva de memoria.

La función `g_malloc` posibilita la reserva de una zona de memoria, con un número de bytes que le pasemos como parámetro. Además, también existe una función similar llamada `g_malloc0` que, no sólo reserva una zona de memoria, sino que, además, llena esa zona de memoria con ceros, lo cual nos puede beneficiar si se necesita un zona de memoria totalmente limpia.

```
gpointer  g_malloc (gulong  numero_de_bytes );
```

```
gpointer  g_malloc0 (gulong  numero_de_bytes );
```

Existe otro conjunto de funciones que nos permiten reservar memoria de una forma parecida a cómo se hace en los lenguajes orientados a objetos. Esto se realiza mediante las siguientes macros definidas en GLib `/gmem.h`:

```
/* Convenience memory allocators
*/
#define g_new(struct_type, n_structs)           \
        ((struct_type *) g_malloc (((gsize) sizeof (struct_type)) * ((gsize) (n_struct
#define g_new0(struct_type, n_structs)          \
                ((struct_type *) g_malloc0 (((gsize) sizeof (struct_type)) * ((gsize) (
#define g_renew(struct_type, mem, n_structs)    \
                ((struct_type *) g_realloc ((mem), ((gsize) sizeof (struct_type)) * ((g
```

Como se puede apreciar, no son más que macros basadas en `g_malloc`, `g_malloc0` y `g_realloc`. La forma de funcionamiento de `g_new` y `g_new0` es mediante el nombre de un tipo de datos y un número de elementos de ese tipo de datos, de forma que se puede hacer:

```
GString *str = g_new (GString,1);
GString *arr_str = g_new (GString, 5);
```

En estas dos líneas de código, se asigna memoria para un elemento de tipo GString, que queda almacenado en la variable `str`, y para un array de cinco elementos de tipo GString, que queda almacenado en la variable `arr_str`).

`g_new0` funciona de la misma forma que `g_new`, con la única diferencia de que inicializa a 0 toda la memoria asignada. En cuanto a `g_renew`, ésta funciona de la misma forma que `g_realloc`, es decir, reasigna la memoria asignada anteriormente.

## Liberación de memoria.

Cuando se hace una reserva de memoria con `g_malloc` y, en un momento dado, el uso de esa memoria no tiene sentido, es el momento de liberar esa memoria. Y el sustituto de `free` es `g_free` que, básicamente, funciona igual que la anteriormente mencionada.

```
void g_free (gpointer memoria_reservada );
```

### Realojamiento de memoria.

En determinadas ocasiones, sobre todo cuando se utilizan estructuras de datos dinámicas, es necesario ajustar el tamaño de una zona de memoria (ya sea para hacerla más grande o más pequeña). Para eso, GLib ofrece la función `g_realloc`, que recibe un puntero a memoria que apunta a una región que es la que será acomodada al nuevo tamaño y devuelve el puntero a la nueva zona de memoria. El anterior puntero es liberado y no se debería utilizar más:

```
gpointer g_realloc (gpointer memoria_reservada , gulong
 numero_de_bytes );
```

## Estructuras de datos: listas enlazadas, pilas y colas.

### Listas enlazadas.

#### Introducción

La lista enlazada es un TDA que nos permite almacenar datos de una forma organizada, al igual que los vectores pero, a diferencia de estos, esta estructura es dinámica, por lo que no tenemos que saber "a priori" los elementos que puede contener.

En una lista enlazada, cada elemento apunta al siguiente excepto el último que no tiene sucesor y el valor del enlace es null. Por ello los elementos son registros que contienen el dato a almacenar y un enlace al siguiente elemento. Los elementos de una lista, suelen recibir también el nombre de nodos de la lista.

```
struct lista {                                          (1)
gint dato;                                              (2)
lista *siguiente;
};
```

**(1)** Representa el dato a almacenar. Puede ser de cualquier tipo; en este ejemplo se trata de una lista de enteros.

**(2)** Es un puntero al siguiente elemento de la lista; con este puntero enlazamos con el sucesor, de forma que podamos construir la lista.

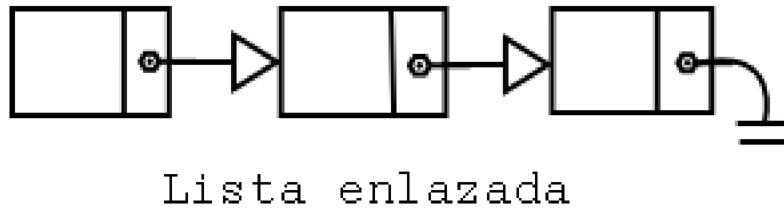



**Figura 4-1. Esquema de un nodo y una lista enlazada.**

Para que esta estructura sea un TDA lista enlazada, debe tener unos operadores asociados que permitan la manipulación de los datos que contiene. Los operadores básicos de una lista enlazada son:

- Insertar: inserta un nodo con dato x en la lista, pudiendo realizarse esta inserción al principio o final de la lista o bien en orden.
- Eliminar: elimina un nodo de la lista, puede ser según la posición o por el dato.
- Buscar: busca un elemento en la lista.
- Localizar: obtiene la posición del nodo en la lista.
- Vaciar: borra todos los elementos de la lista

Después de esta breve introducción, que sólo pretende servir como recordatorio, pasaremos a ver cómo es la estructura GSList que, junto con el conjunto de funciones que la acompañan, forman el TDA lista enlazada en GLib.

### GSList

La definición de la estructura GSList o, lo que es lo mismo, un nodo de la lista, está definido de la siguiente manera:

```
struct GSList {                                         (1)
  gpointer data;                                        (2)
  GSList *next;
};
```

**(1)** Representa el dato a almacenar. Se utiliza un puntero genérico por lo que puede almacenar un puntero a cualquier tipo de dato o bien almacenar un entero utilizando las macros de conversión de tipos.

**(2)** Se trata de un puntero al siguiente elemento de la lista.

Las macros de conversión disponibles son las siguientes:

- `GINT_TO_POINTER ()`
- `GPOINTER_TO_INT ()`
- `GUINT_TO_POINTER ()`
- `GPOINTER_TO_UINT ()`

Más adelante, en esta misma sección, se verán ejemplos del uso de estas macros.

Las funciones que acompañan a la estructura GSList y que implementan los operadores básicos de las listas enlazadas, son las siguientes:

**Tabla 4-6. Operadores de inserción en listas enlazadas.**

| Operador | Funciones asociadas a GSList. |
|---|---|
| Insertar al principio. | GSList* g_slist_prepend (GSList *list, gpointer data) |
| Insertar al final. | GSList* g_slist_append (GSList *list, gpointer data) |
| Insertar en la posición indicada. | GSList* g_slist_insert (GSList *list, gpointer data, gint position) |
| Insertar en orden. | GSList* g_slist_insert_sorted (GSList *list, gpointer data, GCompareFunc func) |

Las funciones de inserción al principio de la lista, `g_slist_prepend`, y al final, `g_slist_append`, son sencillas de usar. Sólo hay que pasarles como parámetros la lista donde queremos añadir el dato así como el dato a insertar y la función devuelve una lista con el nuevo dato insertado.

La función `g_slist_insert` inserta el dato en la posición indicada. Su uso también es sencillo como puede verse en el siguiente ejemplo.

**Ejemplo 4-14. Insertar un nuevo dato en una posición determinada.**

```
/* obtiene el numero de nodos de la lista */
length = g_slist_length (list);

g_print ("\nEscribe el nº de indice donde se insertara el dato (el indice maximo es
scanf ("%d", &index);

/* inserta el valor en la posicion indicada */

if (index < length) {
  list = g_slist_insert (list, GINT_TO_POINTER (value), index);
  print_list (list);
    }
```

En este ejemplo se utiliza la función `g_slist_length` para obtener el número de nodos que contiene la lista. A esta función hay que pasarle como parámetro la lista de la que se desea obtener el número de nodos y devuelve como resultado el número de nodos de ésta.

```
guint * g_slist_length ( GSList * list );
```

La función `g_slist_insert_sorted` inserta los elementos a la lista de forma ordenada. Esta función utiliza el parámetro `GCompareFunc` para insertar el dato en la posición correcta.

`GCompareFunc` es una función que se utiliza para comparar dos valores y saber así cual de ellos hay que insertar primero. En los dos ejemplos que hay a continuación, se puede observar una función de tipo `GCompareFunc` y su uso para insertar datos en una lista en orden creciente.

**Ejemplo 4-15. Parámetro `GCompareFunc` para insertar en orden creciente.**

```
gint compare_value1 (gconstpointer a, gconstpointer b) {
  gint *value1 = (gint *) a;
  gint *value2 = (gint *) b;

  return value1 > value2;
}
```

**Ejemplo 4-16. Insertar elementos en orden creciente.**

```
  gint values[] = {8, 14, 5, 12, 1, 27, 3, 13};
  gint i;
  /* insertando valores en orden creciente */
  for (i = 0; i < 8; i++) {
    list =  g_slist_insert_sorted (list, GINT_TO_POINTER (values[i]),
                                   compare_value1);
  }
```

**Tabla 4-7. Operadores de eliminación en listas enlazadas.**

| Operador | Funciones asociadas a GSList. |
| --- | --- |
| Eliminar un nodo. | GSList* g_slist_remove (GSList *list, gconstpointer data) |
| Eliminar nodos según un patrón. | GSList* g_slist_remove_all (GSList *list, gconstpointer data) |

Las dos funciones expuestas para la eliminación de nodos, si bien tienen una definición prácticamente idéntica, el resultado obtenido es distinto. En el caso de `g_slist_remove`, se eliminará el nodo que contenga el valor *`data`*. Si hay varios nodos con el mismo valor, sólo se eliminará el primero. Si ningún nodo contiene ese valor, no se realiza ningún cambio en el GSList. En el caso de `g_slist_remove_all`, se eliminan todos los nodos de la lista que contengan el valor *`data`* y nos devuelve la nueva lista resultante de la eliminación de los nodos.

**Ejemplo 4-17. Elimina un elemento de la lista.**

```
  if (list2 != NULL) {
    g_print ("\nEl dato %d sera eliminado de la lista.\n", list2->data);

    /* eliminando un elemento de la lista */
    g_slist_remove (list, list2->data);
   }
```

**Tabla 4-8. Operadores de búsqueda en listas enlazadas.**

| Operador | Funciones asociadas a GSList. |
|---|---|
| Buscar un nodo según un valor. | GSList* g_slist_find (GSList *list, gconstpointer data) |
| Buscar un nodo según un criterio. | GSList* g_slist_find_custom (GSList *list, gconstpointer data, GCompareFunc func) |
| Localizar el índice de un nodo. | GSList* g_slist_index (GSList *list, gconstpointer data) |
| Localizar la posición de un nodo. | GSList* g_slist_position (GSList *list, GSList *llink) |
| Obtener el último nodo. | GSList* g_slist_last (GSList *list) |
| Obtener el siguiente nodo. | g_slist_next (slist) |
| Obtener un nodo por su posición. | GSList* g_slist_nth (GSList *list, guint n) |
| Obtener el dato de un nodo según su posición. | gpointer g_slist_nth_data (GSList *list, guint n) |

Todas estas funciones, a excepción de `g_slist_nth_data`, devuelven un nodo de la lista o NULL si el elemento no existe. La función `g_slist_nth_data` devuelve el valor del elemento según la posición que se le pasa como argumento en el parámetro *n* o NULL si la posición que se le pasa está más allá del final de la lista.

La función `g_slist_next`, es una macro que nos devuelve el siguiente nodo. Esta macro la podemos utilizar para recorrer la lista.

**Ejemplo 4-18. Función que imprime una lista.**

```
void print_list (GSList *list) {

  gint i = 0;

  while (list != NULL) {
    g_print ("Node %d content: %d.\n", i, list->data);

    /* apunta al siguiente nodo de la lista */
    list = g_slist_next (list);
    i++;
  }
}
```

**Tabla 4-9. Operador para vaciar la lista.**

| Operador | Funciones asociadas a GSList. |
|---|---|
| Vacía la lista y libera la memoria usada. | void g_slist_free (GSList *list) |

La función `g_slist_free` libera la memoria de la lista que se le pasa como parámetro.

Con estas funciones, quedan definidos los operadores básicos del TDA lista enlazada. GSList trae otras funciones además de los operadores básicos. Para más información sobre estas, está disponible el manual de referencia de GLib[1].

## Listas doblemente enlazadas.

### Introducción.

El TDA lista doblemente enlazada, al igual que la lista enlazada, es un TDA dinámico lineal pero, a diferencia de este, cada nodo de la lista doblemente enlazada contiene dos punteros, de forma que uno apunta al siguiente nodo y el otro al predecesor. Esta característica, permite que se pueda recorrer la lista en ambos sentidos, cosa que no es posible en las listas simples.

La declaración del tipo lista doblemente enlazada de enteros es la siguiente:

```
struct lista_doble {                                        (1)
gint dato;                                                  (2)
lista_doble *siguiente;                                     (3)
lista_doble *anterior;
};
```

**(1)** Representa el dato a almacenar, que puede ser de cualquier tipo. En este ejemplo se trataría de una lista de enteros.

**(2)** Se trata de un puntero al siguiente elemento de la lista. Con este puntero se enlaza con el sucesor, de forma que podamos construir la lista.

**(3)** Es un puntero al elemento anterior de la lista. Este puntero enlaza con el elemento predecesor de la lista y permite recorrerla en sentido inverso.

Sobre este TDA se definen los mismos operadores básicos que en las listas simples.

### GList

La definición de la estructura GList, que es un nodo de la lista doblemente enlazada, está definido de la siguiente manera:

```
struct GList
{                                                           (1)
gpointer data;                                              (2)
GList *next;                                                (3)
GList *prev;
};
```

**(1)** Representa el dato que se va a almacenar. Se utiliza un puntero genérico por lo que puede almacenar un puntero a cualquier tipo de dato o bien almacenar un entero utilizando las macros de conversión de tipos.

**(2)** Se trata de un puntero al siguiente elemento de la lista.

**(3)** Se trata de un puntero al elemento anterior de la lista.

Las funciones que acompañan a la estructura GList, que implementan los operadores básicos de las listas enlazadas, son las siguientes:

**Tabla 4-10. Operadores de inserción en listas doblemente enlazadas.**

| Operador | Funciones asociadas a GList |
|---|---|
| Insertar al principio. | GList* g_list_prepend (GList *list, gpointer data) |
| Insertar al final. | GList* g_list_append (GList *list, gpointer data) |
| Insertar en la posición indicada. | GList* g_list_insert (GList *list, gpointer data, gint position) |
| Insertar en orden. | GList* g_list_insert_sorted (GList *list, gpointer data, GCompareFunc func) |

Como puede observarse en la definición de las funciones, su uso es el mismo que en las listas simples, al igual que las macros de conversión, por lo que todo lo explicado en esa sección es válido en el caso de las listas doblemente enlazadas.

**Ejemplo 4-19. Insertar un nuevo dato en una posición determinada.**

```
    /* obtiene el numero de nodos de la lista */
    length = g_list_length (list);

    g_print ("\nEscribe el numero de indice donde se insertara el dato (el indice maxim
    scanf ("%d", &index);

    /* inserta el valor en la posicion indicada */

    if (index < length) {
      list = g_list_insert (list, GINT_TO_POINTER (value), index);
      print_list (list);
        }
```

**Ejemplo 4-20. Insertar elementos en orden creciente.**

```
  gint values[] = {8, 14, 5, 12, 1, 27, 3, 13};
  gint i;

  for (i = 0; i < 8; i++) {
    list =  g_list_insert_sorted (list, GINT_TO_POINTER (values[i]),
                                  compare_value1);
  }
```

**Tabla 4-11. Operadores de eliminación en listas doblemente enlazadas.**

| Operador | Funciones asociadas a GList |
|---|---|
| Eliminar un nodo. | GList* g_list_remove (GList *list, gconstpointer data) |
| Eliminar nodos según un patrón. | GList* g_list_remove_all (GList *list, gconstpointer data) |

**Ejemplo 4-21. Eliminar un elemento de la lista.**

```
  if (list2 != NULL) {
    g_print ("\nEl dato %d sera eliminado de la lista.\n", list2->data);

    /* eliminando un elemento de la lista */
    g_list_remove (list, list2->data);
    print_list (list);
```

```
}
```

**Tabla 4-12. Operadores de búsqueda en listas doblemente enlazadas.**

| Operador | Funciones asociadas a GList. |
|---|---|
| Buscar un nodo según un valor. | GList* g_list_find (GList *list, gconstpointer data) |
| Buscar un nodo según un criterio. | GList* g_list_find_custom (GList *list, gconstpointer data, GCompareFunc func) |
| Localizar el índice de un nodo. | GList* g_list_index (GList *list, gconstpointer data) |
| Localizar la posición de un nodo. | GList* g_list_position (GList *list, GSList *llink) |
| Obtener el último nodo. | GList* g_list_last (GList *list) |
| Obtener el siguiente nodo. | g_list_next (list) |
| Obtener un nodo por su posición. | GList* g_list_nth (GList *list, guint n) |
| Obtener el dato de un nodo según su posición. | gpointer g_list_nth_data (GList *list, guint n) |

**Ejemplo 4-22. Busca un valor dado en la lista.**

```
g_print ("\nEntra un valor entero a buscar: ");
scanf ("%d", &value);
g_print ("\n");

/* buscando un elemento en la lista */
list2 = g_list_find (list, GINT_TO_POINTER (value));

if (list2 != NULL) {
  index = g_list_index (list, list2->data);
  g_print ("\nEl valor %d esta en el nodo %d.\n", list2->data, index);
```

**Tabla 4-13. Operador para vaciar la lista**

| Operador | Funciones asociadas a GList |
|---|---|
| Vacía la lista y libera la memoria usada. | void g_list_free (GList *list) |

Con estas funciones, quedan definidos los operadores básicos del TDA lista enlazada. Al igual que GSList, GList trae otras funciones además de los operadores básicos. Para más información sobre estas, está disponible el manual de referencia de GLib[2].

## GQueue: pilas y colas.

Otra de las novedades que incorpora esta versión de GLib es la estructura GQueue que, junto con las funciones que la acompañan, nos proporciona la posibilidad de implementar los TDA cola y pila estándar.

GQueue utiliza estructuras GList para almacenar los elementos. La declaración de la estructura de datos es la siguiente.

```
struct GQueue {                                          (1)
   GList  *head;                                         (2)
   GList  *tail;                                         (3)
   guint length;
   };
```

**(1)** Es un puntero al primer elemento de la cola.

**(2)** Es un puntero al último elemento de la cola.

**(3)** Esta variable almacena el número de elementos de la cola.

Aunque para referirnos al TDA GQueue utilizamos el término cola, con esta misma estructura también tenemos la posibilidad de implementar el TDA pila, gracias a las funciones que lo acompañan.

## Colas

Una cola es una estructura de datos donde el primer elemento en entrar es el primero en salir, también denominadas estructuras FIFO (*First In, First Out*).

Esta estructura de datos se puede definir como una lista enlazada con acceso FIFO a la que sólo se tiene acceso al final de la lista para meter elementos y al principio de esta para sacarlos.

Los operadores asociados a este TDA y las funciones que los implementan en GLib son:

**Tabla 4-14. Operadores asociados al TDA Cola.**

| Operador | Funciones asociadas a GQueue. |
|---|---|
| Iniciar cola. | GQueue* g_queue_new (void) |
| Cola vacía. | gboolean g_queue_is_empty (GQueue* queue) |
| Consultar frente cola. | gpointer g_queue_peek_head (GQueue* queue) |
| Consultar final cola. | gpointer g_queue_peek_tail (GQueue* queue) |
| Meter | void g_queue_push_tail (GQueue* queue, gpointer data) |
| Sacar | gpointer g_queue_pop_head (GQueue* queue) |
| Vaciar cola. | void g_queue_free (GQueue* queue) |

### *Iniciar cola.*

El operador "Iniciar cola" es el encargado de crear una nueva cola y ponerla en estado de cola vacía.

**Ejemplo 4-23. Creando una nueva cola.**

```
GQueue* cola;
cola = g_queue_new ();
```

### *Cola vacía.*

Este operador consulta si la cola está vacía. Es necesaria su utilización antes de realizar la operación de "sacar elementos" de la cola.

**Ejemplo 4-24. Función que comprueba si una cola está vacía.**

```
gboolean cola_vacia (GQueue* cola) {
   return g_queue_is_empty (cola);
}
```

### *Consultar el frente.*

Esta operación consulta el contenido del frente de la cola sin sacarlo.

**Ejemplo 4-25. Función que consulta el frente de la cola.**

```
gpointer consultar_frente (GQueue* cola) {
   return g_queue_peek_head (cola);
}
```

### *Consultar el final.*

Esta operación consulta el contenido del final de la cola sin sacarlo.

**Ejemplo 4-26. Función que consulta el final de la cola.**

```
gpointer consultar_final (GQueue* cola) {
   return g_queue_peek_tail (cola);
}
```

### *Meter*

Este operador introduce elementos al final de la cola.

**Ejemplo 4-27. Introducir un nuevo elemento en la cola.**

```
GQueue* meter_cola (GQueue* cola, gpointer dato) {
   g_queue_push_tail (cola, dato);

   return cola;
}
```

### *Sacar*

El operador "sacar" elimina elementos del frente de la cola.

**Ejemplo 4-28. Saca un elemento de la cola.**

```
gpointer sacar_cola (GQueue* cola) {
   gpointer dato;

   dato = g_queue_pop_head (cola);

   return dato;
}
```

*Vaciar cola.*

Elimina el contenido de una cola inicializándola a una cola vacía.

**Ejemplo 4-29. Vacía la cola.**

```
  g_queue_free (cola);
```

## Pilas

Una pila, es una estructura de datos en la que el último elemento en entrar es el primero en salir, opr lo que también se denominan estructuras LIFO (*Last In, First Out*).

En esta estructura sólo se tiene acceso a la cabeza o cima de la pila.

Los operadores asociados a este TDA y las funciones que los implementan en GLib son:

**Tabla 4-15. Operadores asociados al TDA Pila.**

| Operador | Funciones asociadas a GQueue. |
| --- | --- |
| Iniciar pila. | GQueue* g_queue_new (void) |
| Pila vacía. | gboolean g_queue_is_empty (GQueue* queue) |
| Consultar pila. | gpointer g_queue_peek_head (GQueue* queue) |
| Meter. | void g_queue_push_head (GQueue* queue, gpointer data) |
| Sacar. | gpointer g_queue_pop_head (GQueue* queue) |
| Vaciar pila. | void g_queue_free (GQueue* queue) |

*Iniciar pila.*

El operador "iniciar pila" es el encargado de crear una nueva pila y inicializarla al estado de pila vacía.

**Ejemplo 4-30. Creando una nueva pila.**

```
   GQueue* pila;
   pila = g_queue_new ();
```

### *Pila vacía.*

Este operador consulta si la pila está vacía. Es necesaria su utilización antes de realizar la operación de sacar elementos de la pila.

**Ejemplo 4-31. Función que comprueba si una pila está vacía.**

```
gboolean pila_vacia (GQueue* pila) {
   return g_queue_is_empty (pila);
}
```

### *Consultar pila.*

Esta operación, consulta el contenido de la cima de la pila sin sacarlo.

**Ejemplo 4-32. Función que consulta la cima de la pila.**

```
gpointer consultar_pila (GQueue* pila) {
   return g_queue_peek_head (pila);
}
```

### *Meter*

El operador "meter", introduce elementos en la cima de la pila.

**Ejemplo 4-33. Introducir un nuevo elemento en la pila.**

```
GQueue* meter_pila (GQueue* pila, gpointer dato) {
   g_queue_push_head (pila, dato);

   return pila;
}
```

### *Sacar*

El operador sacar, saca elementos de la cima de la pila.

**Ejemplo 4-34. Saca un elemento de la pila.**

```
gpointer sacar_pila (GQueue* pila) {
   gpointer dato;

   dato = g_queue_pop_head (pila);

   return dato;
}
```

### *Vaciar pila.*

Elimina el contenido de una pila inicializándola a una pila vacía.

**Ejemplo 4-35. Vacía la pila**

```
g_queue_free (pila);
```

# Estructuras de datos avanzadas.

## Tablas de dispersión.

### Qué es una tabla de dispersión.

Las tablas de dispersión, más conocidas como tablas *hash*, son unas de las estructuras de datos más frecuentemente usadas. Para tener una idea inicial, las tablas de dispersión posibilitan tener una estructura que relaciona una clave con un valor, como un diccionario. Internamente, las tablas de dispersión son un array. Cada una de las posiciones del array puede contener ninguna, una o varias entradas del diccionario. Normalmente contendrá una como máximo, lo que permite un acceso rápido a los elementos, evitando realizar una búsqueda en la mayoría de los casos. Para saber en qué posición del array se debe buscar o insertar una clave, se utiliza una función de dispersión. Una función de dispersión relaciona a cada clave con un valor entero. Dos claves iguales deben tener el mismo valor de dispersión, también llamado *hash value*, pero dos claves distintas pueden tener el mismo valor de dispersión, lo cual provocaría una colisión.

El valor de dispersión es un entero sin signo entre 0 y el máximo entero de la plataforma, por lo que la tabla de dispersión usa el resto de dividir el valor de dispersión entre el tamaño del array para encontrar la posición. Cuando dos claves tienen que ser almacenadas en la misma posición de la tabla se produce una colisión. Esto puede ser debido a que la función de dispersión no distribuye las claves lo suficiente, o a que hay más claves en la tabla *hash* que el tamaño del array. En el segundo caso, GLib se encargará de redimensionar el array de forma automática.

Para aclarar los conceptos, se examinará el siguiente ejemplo a lo largo de todo el capítulo con el fin de facilitar la comprensión del mismo. Póngase que se desea tener una relación de la información de los activistas de GNOME. Para ello se usará una tabla de dispersión usando como clave el apodo que usa cada desarrollador dentro del proyecto. Y como valor una relación de sus datos personales. Esta relación de datos personales vendría dada por la siguiente estructura.

```
struct
DatosDesarrollador {
   gchar *apodo;
   gchar *nombre;
   gchar *proyecto;
   gchar *telefono;
};
```

Ahora, si se hace un recuento de datos, se obtiene una curiosa información. Por ejemplo, si tenemos en cuenta que cada apodo tiene como mucho diez caracteres y que el alfabeto tiene veintiseis letras, el resultado es que tendremos 26^10 posibles llaves, lo que supera con creces el número de claves que vamos a utilizar. Con ésto se quiere hacer hincapié en que el uso de esta estructura es útil cuando el número de llaves libres excede con mucho a las llaves que van a ser ocupadas.

### Cómo se crea una tabla de dispersión.

Para crear una tabla de dispersión se tiene que indicar la función de dispersión y la función que se utilizará para comparar dos claves. Estas dos funcione se deben pasar como parámetros a la función `g_hash_table_new`

```
GHashTable*  g_hash_table_new (GHashFunc  funcion_de_dispersion ,
GEqualFunc  funcion_de_comparación );
```

GLib tiene implementadas una serie de funciones de dispersión y comparación. De este modo, el desarrollador no ha de reinventar la rueda. Las funciones de dispersión trabajan siempre de la misma manera. A este tipo de funciones se les pasa un valor y devuelven la clave. En cuanto a las funciones de comparación, se les pasa como parámetros dos valores y devuelve *TRUE* si son los mismos valores y *FALSE* si no son iguales.

Estas funciones son implementadas basándose en el siguiente esquema. Si se diesa el caso de que las existentes en GLib no satisfacen sus necesidades, lo único que tendrá que hacer será implementar un par de funciones que siguieran estos esquemas.

```
guint  (*GHashFunc) (gconstpointer  clave );
```

```
gboolean  (*GEqualFunc) (gconstpointer  clave_A , gconstpointer
 clave_B );
```

Una vez se tiene presente este esquema, se puede comprender mejor el funcionamiento de las funciones que se describirán a continuación. La primera pareja de funciones son `g_str_hash` y `g_str_equal`, que son de dispersión y de comparación respectivamente. Con la función `g_str_hash` podremos convertir una clave en forma de cadena en un valor de dispersión. Lo cual quiere decir que el desarrollador podrá usar, con estas funciones, una clave en forma de cadena para la tabla *hash*.

Otras funciones de carácter similar son `g_int_hash`, `g_int_equal`, `g_direct_hash` o `g_direct_equal`, que posibilitan usar como claves para la tabla *hash* números enteros y punteros genéricos respectivamente.

Ahora, siguiendo con el ejemplo que se comenzó al principio del capitulo, se pasará a ver cómo se creará la tabla de dispersión que contendrá la información de los desarrolladores de GNOME. Recuérdese que se iba a usar, como clave, el apodo del desarrollador. Así que, como función de dispersión y de comparación, se usaran `g_str_hash` y `g_str_equal`, respectivamente. Ya que el apodo es una cadena y estas funciones satisfacen perfectamente el fin que se persigue.

### Cómo manipular un tabla de dispersión.

Ahora se pasará a describir las funciones de manipulación de este TAD. Para añadir y eliminar elementos de una tabla se dispone de estas dos funciones: `g_hash_table_insert` y `g_hash_table_remove`. La primera insertará un valor en la tabla *hash* y le indicará la clave con la que después podrá ser referenciado. En cuanto a la segunda, borrará el valor que corresponda a la clave que se le pase como parámetro a la función.

```
void g_hash_table_insert (GHashTable tabla_hash , gpointer clave ,
 gpointer valor );
```

```
gboolean g_hash_table_remove (GHashTable tabla_hash , gpointer
 clave );
```

En caso que se desee modificar el valor asociado a una clave, hay dos opciones a seguir en forma de funciones: `g_hash_table_insert` (vista anteriormente) o `g_hash_table_replace`. La única diferencia entre ambas es qué pasa cuando la clave ya existe. En el caso de la primera, se conservará la clave antigua mientras que, en el caso de la segunda, se usará la nueva clave. Obsérvese que dos claves se consideran la misma si la función de comparación devuelve *TRUE*, aunque sean diferentes objetos.

```
void g_hash_table_replace (GHashTable tabla_hash , gpointer clave ,
 gpointer valor );
```

### Búsqueda de información dentro de la tabla.

Llegados a este punto, en el que se ha explicado como insertar, borrar y modificar elementos dentro de una tabla de este tipo, parece obvio que el siguiente paso es cómo conseguir encontrar información dentro de una tabla de dispersión. Para ello se dispone de la función `g_hash_table_lookup`. Esta función tiene un funcionamiento muy simple al igual que las funciones anteriormente explicadas. Lo único que necesita es que se le pase como parámetro la tabla de dispersión y la clave que desea buscar en la misma y la función devolverá un puntero al valor asociado a la clave en caso de existir o el valor *NULL* en caso de no existir.

```
gpointer g_hash_table_lookup (GHashTable tabla_hash , gpointer
 clave );
```

También se dispone de otra función para realizar las búsquedas de datos dentro de una tabla de dispersión. Esta función es más compleja en su uso pero provee de un sistema de búsqueda más eficiente que el anterior que ayudará a suplir las necesidades más complejas de un desarrollador.

```
gboolean g_hash_table_lookup_extended (GHashTable tabla_hash ,
 gconstpointer clave_a_buscar , gpointer* clave_original ,
 gpointer* valor );
```

La función anterior devolverá *TRUE* si encuentra el valor buscado, o *FALSE* en caso contrario. En el supuesto de encontrar el valor, el puntero `clave_original` apuntará a la clave con la que fue almacenado en la tabla, y el puntero `valor` apuntará al valor almacenado. Téngase en cuenta que la clave usada para la realizar la búsqueda, *clave_a_buscar*, no tiene por qué ser la misma que la encontrada, *clave_original*, aunque ambas serán iguales según la función de comparación que hayamos indicado al crear la tabla de dispersión.

### Manipulación avanzada de tablas de dispersión.

## Arboles binarios balanceados

Los árboles binarios balanceados son estructuras de datos que almacenan pares clave - dato, de manera ordenada según las claves, y posibilitan el acceso rápido a los datos, dada una clave particular, y recorrer todos los elementos en orden.

Son apropiados para grandes cantidades de información y tienen menor *overhead* que una tabla de dispersión, aunque el tiempo de acceso es de mayor complejidad computacional.

Si bien internamente la forma de almacenamiento tiene estructura de árbol, esto no se exterioriza en la API, haciendo que el manejo de los datos resulte transparente. Se denominan balanceados porque, cada vez que se modifican, las ramas se rebalancean de tal forma que la altura del árbol sea la mínima posible, acortando de esta manera el tiempo promedio de acceso a los datos.

### Creación de un árbol binario.

Para crear un árbol binario balanceado es necesaria una función de comparación que pueda ordenar un conjunto de claves. Para ello se adopta la convención utilizada por `strcmp`. Esto es, dados dos valores, la función devuelve 0 si son iguales, un valor negativo si el primer parámetro es anterior al segundo, y un valor positivo si el primero es posterior al segundo. La utilización de esta función permite flexibilidad en cuanto a claves a utilizar y en como se ordenan. El prototipo es el siguiente:

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

o bien, para el caso en que sea necesario proveer algún dato adicional a la función:

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

Una vez definida dicha función el árbol se crea con alguna de las siguientes formas:

```
GTree *g_tree_new(GCompareFunc key_compare_func);
```

```
GTree *g_tree_new_with_data(GCompareDataFunc key_compare_func,
gpointer key_compare_data);
```

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

donde `key_compare_func` es la función previamente mencionada, `key_compare_data` es un dato arbitrario a enviar a la función de comparación y `key_destroy_func` y `value_destroy_func` funciones de retrollamada cuando alguna clave o dato se eliminan del árbol.

En caso de no utilizar la última función, es tarea del usuario liberar la memoria utilizada por claves y/o datos cuando se destruye el árbol o cuando se elimina algún dato del mismo utilizando `g_tree_remove`.

Para destruir un árbol se utiliza la función `g_tree_destroy`.

```
void g_tree_destroy(GTree *tree);
```

### Agregar y eliminar elementos a un árbol binario.

Para insertar un nuevo elemento en el árbol se utiliza `g_tree_insert` o `g_tree_replace`. La diferencia entre ambas vale sólo en el caso de que en el árbol ya exista una clave igual a la que se intenta agregar y sólo cuando, en la creación del árbol, se especificó una función de destrucción de claves. Para `g_tree_insert` la clave que se pasó como parámetro se destruye, llamando a `key_destroy_func` y la del árbol permanece inalterada. Para el caso de `g_tree_replace`, la clave ya existente se destruye y se reemplaza por la nueva. En ambos casos, de existir el elemento, el dato anterior se destruye llamando a `value_destroy_func`, siempre que se haya especificado en la creación del árbol.

```
void g_tree_insert(GTree *tree, gpointer key, gpointer value);
```

```
void g_tree_replace(GTree *tree, gpointer key, gpointer value);
```

*`key`* es la clave con la cual se recuperará el dato luego y *`value`* el dato en sí. Para eliminar un elemento del árbol se pueden utilizar `g_tree_remove` o `g_tree_steal`. La diferencia entre ambas es que, si se invoca `g_tree_steal` no se destruyen la clave y el valor aunque se hayan especificado funciones para tal fin en la creación del árbol.

```
void g_tree_remove(GTree *tree, gconstpointer key);
```

```
void g_tree_steal(GTree *tree, gconstpointer key);
```

### Búsqueda y recorrida en un árbol binario.

Existen dos funciones para buscar en un árbol binario, similares a las utilizadas en tablas de *hash*: `g_tree_lookup` y `g_tree_lookup_extended`.

```
gpointer g_tree_lookup(GTree *tree, gconstpointer key);
```

En este caso, se busca un elemento del árbol cuya clave sea igual (en los términos de la función especificada al momento de la creación) a *`key`*. Devuelve el dato del elemento encontrado. Si la búsqueda fue infructuosa devuelve `NULL`.

```
gboolean g_tree_lookup_extended(GTree *tree, gconstpointer lookup_key,
gpointer *orig_key, gpointer *value);
```

Esta función no sólo busca el elemento cuya clave coincida con la especificada, sino que además devuelve la clave y dato del elemento encontrado en los parámetros de salida *orig_key* y *value*. A diferencia del caso anterior, la función devuelve un booleano indicando si la búsqueda fue exitosa o no.

La versión extendida de la búsqueda es particularmente útil cuando se quiere eliminar un elemento del árbol sin destruirlo, utilizando la función `g_tree_steal` (p.e. para ser agregado a otro árbol).

Por último para recorrer todos los elementos de un árbol se utiliza `g_tree_foreach`. Los elementos son tomados en orden y pasados como parámetro a la función de iteración especificada.

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

Es importante decir que durante el recorrido, no se puede modificar el árbol (agregar y/o eliminar elementos). Si se quieren eliminar algunos elementos, se deben agregar las claves a una lista (o cualquier estructura auxiliar) y finalizada la iteración proceder a eliminar uno por uno los elementos seleccionados. La función de iteración debe tener la siguiente forma:

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

Si dicha función devuelve `TRUE`, la iteración se interrumpe. *data* es el valor que se especificó como *user_data* en la función `g_tree_foreach`.

## GNode : Arboles de orden n.

Para representar datos de manera jerárquica, GLib ofrece el tipo de dato GNode que permite representar árboles de cualquier orden.

Al igual que con las listas y a diferencia de árboles binarios balanceados o tablas *hash*, los GNode son elementos independientes: sólo hay conexiones entre ellos, pero nada en la estructura dice, por ejemplo, cuál es la raíz del árbol. `NULL` es el árbol vacío. La estructura GNode tiene la siguiente forma:

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

`data` contiene el dato en sí del nodo. `parent` apunta al nodo padre: éste es `NULL` para el nodo raíz. `children` apunta al primer hijo del nodo, accediéndose a los demás hijos mediante los campos `next` y `prev`.

## Agregar nodos a un árbol.

GLib tiene una serie de funciones para agregar nodos a un árbol. Se puede crear el nodo aislado primero e insertarlo en una posición dada, pero también hay macros definidas que crean el nodo y lo insertan directamente. No hay diferencia en el resultado final, simplemente una línea menos de código.

Para crear un nodo se utiliza `g_node_new`:

```
GNode *g_node_new(gpointer data);
```

Una vez obtenido el GNode, se inserta a un árbol especificando el nodo padre (*parent*) ya existente y la posición relativa entre los hijos de dicho padre. Hay cinco funciones diferentes en función de la posición y las condiciones en que se quiera insertar. En todos los casos el valor devuelto es el mismo nodo *node* insertado.

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

*sibling* es un nodo hijo de *parent* que se toma como referencia para insertar el nuevo nodo antes o después de él mismo. En ambos casos, `g_node_insert_after` y `g_node_insert_before`, si *sibling* es `NULL`, el nuevo nodo se inserta al final de la lista de hijos.

Con `g_node_n_children` se obtiene la cantidad de hijos de un nodo y, a partir de este dato, se puede especificar una posición relativa dentro de la lista de hijos *posittion*. Las formas `g_node_append` y `g_node_prepend` agregan el nuevo nodo al final o al principio de la lista de hijos.

Cuatro de estas cinco funciones tienen su equivalente de inserción de nodo directa a partir de el dato. Como se dijo antes, la diferencia es que en estos casos se especifica el dato y la creación del nodo es transparente al usuario aunque, de ser requerido, se retorna en el valor de la función. Estas cuatro nuevas formas son macros que utilizan las funciones anteriores. En todos los casos, *data* es el dato que contendrá el nuevo nodo.

```
GNode *g_node_insert_data(GNode *parent, gint position, gpointer
data);
GNode *g_node_insert_data_before(GNode *parent, GNode *sibling,
gpointer data);
GNode *g_node_append_data(GNode *parent, gpointer data);
GNode *g_node_prepend_data(GNode *parent, gpointer data);
```

## Eliminar Vs. desvincular.

Al igual que con las listas, hay dos formas de quitar un dato de un árbol: `g_tree_unlink` y `g_tree_destroy`.

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

g_node_unlink quita el nodo especificado del árbol, resultando en un nuevo árbol que lo tiene como raíz. g_node_destroy no sólo elimina el nodo del árbol, sino que, además, libera toda la memoria utilizada por el nodo y por sus hijos. GLib, sin embargo, no tiene forma de liberar la memoria de los datos que contenían los nodos: eso es responsabilidad del usuario.

### Información sobre los nodos.

G_NODE_IS_LEAF devuelve TRUE si el nodo es un terminal u hoja del árbol. En otras palabras si el campo children es NULL. Análogamente, G_NODE_IS_ROOT devuelve TRUE si el nodo especificado es la raíz del árbol, o bien, si el campo parent es NULL.

g_node_depth devuelve la profundidad de un nodo (cuantos niveles hay hasta subir a la raíz); g_node_n_children la cantidad de nodos hijos inmediatos y g_node_max_height, la cantidad máxima de niveles inferiores (es decir, iterando hacia los hijos). g_node_depth y g_node_max_height miden alturas. Un árbol vacío tiene altura 0, un único nodo 1 y así sucesivamente.

```
guint g_node_n_nodes(GNode *node, GTraverseFlags flags);
```

g_node_n_nodes recorre el árbol desde el nodo especificado y cuenta los nodos. El parámetro *flags* indica qué nodos debe contar y puede tener como valores:

G_TRAVERSE_LEAFS: Sólo contar los nodos terminales
G_TRAVERSE_NON_LEAFS: Sólo contar los nodos intermedios
G_TRAVERSE_ALL: Contar todos los nodos

```
gboolean g_node_is_ancestor(GNode *node, GNode *descendant);
```

g_node_is_ancestor devolverá TRUE si *node* es un ancestro (padre, padre del padre, etc.) de *descendant*.

g_node_get_root devuelve la raíz del árbol que contiene al nodo especificado.

```
gint g_node_child_index(GNode *node, gpointer data);
gint g_node_child_position(GNode *node, GNode *child);
```

g_node_child_index y g_node_child_position son funciones similares que devuelven la posición de un hijo en la lista de hijos de un nodo (la primera busca el dato en lugar del nodo en sí). En caso de que el nodo no sea hijo del padre especificado, ambas funciones devolverán -1.

Para acceder a los hijos de un nodo dado se pueden utilizar los campos de la estructura o funciones que provee GLib: g_node_first_child, g_node_last_child y g_node_nth_child.

En forma similar, se puede recorrer la lista de hijos usando los campos prev y next, o mediante g_node_first_sibling, g_node_prev_sibling, g_node_next_sibling y g_node_last_sibling.

### Buscar en el árbol y recorrerlo.

Para buscar un nodo determinado, GLib tiene dos funciones:

```
GNode *g_node_find(GNode *node, GTraverseType order, GTraverseFlags
flags, gpointer data);
GNode *g_node_find_child(GNode *node, GTraverseFlags flags, gpointer
data);
```

`g_node_find` y `g_node_find_child` buscan a partir de *node* el nodo del árbol que contenga el *data* especificado. `g_node_find` busca en todo el árbol, mientras que `g_node_find_child` se limita a los hijos inmediatos del nodo.

En ambos casos *flags* especifica que clase de nodos se buscarán, y en `g_node_find`, *order* indica el orden de búsqueda. Este último puede tener uno de estos cuatro valores:

- `G_IN_ORDER`: para cada nodo, visitar el primer hijo, luego el nodo mismo y por último el resto de los hijos.
- `G_PRE_ORDER`: para cada nodo, visitar primero el nodo y luego los hijos.
- `G_POST_ORDER`: para cada nodo, visitar primero los hijos y luego el nodo.
- `G_LEVEL_ORDER`: visitar los nodos por niveles, desde la raíz (es decir la raíz, luego los hijos de la raíz, luego los hijos de los hijos de la raíz, etc.)

Al igual que con otros tipos de dato, es posible recorrer el árbol con dos funciones de GLib, que son análogas a las de búsqueda:

```
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);
```

*max_depth* especifica la profundidad máxima de iteración. Si este valor es -1 se recorre todo el árbol; si es 1 sólo *root*; y así sucesivamente. *order* indica cómo recorrer los nodos y *flags* qué clase de nodos visitar. Notar que al igual que `g_node_find_child`, `g_node_children_foreach` se limita a los hijos directos del nodo.

La forma de las funciones de iteración es la siguiente:

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

Al igual que con las funciones de iteración de los árboles binarios, en caso de necesitar interrumpir la iteración, la función debe devolver `TRUE`.

## Caches

Los cachés son algo que, tarde o temprano, cualquier programador acaba implementando, pues permiten el almacenamiento de datos costosos de conseguir (un fichero a través de la red, un informe generado a partir de datos en una base de datos, etc) para su posterior consulta, de forma que dichos datos sean obtenidos una única vez, y leídos muchas.

Como librería de propósito general que es, GLib incluye, no un sistema de caché superfuncional, si no una infraestructura básica para que se puedan desarrollar cachés

de todo tipo. Para ello, GLib incluye `GCache`, que, básicamente, permite la compartición de estructuras complejas de datos. Esto se usa, principalmente, para el ahorro de recursos en las aplicaciones, de forma que, si se tienen estructuras de datos complejas usadas en distintas partes de la aplicación, se pueda compartir una sola copia de dicha estructura compleja de datos entre todas esas partes de la aplicación.

### Creación del caché.

`GCache` funciona de forma muy parecida a las tablas de claves (`GHashTable`), es decir, mediante el uso de claves para identificar cada objeto, y valores asociados con esas claves.

Pero el primer paso, como siempre, es crear el caché en cuestión. Para ello se usa la función `g_cache_new`, que, a primera vista, parece un tanto compleja:

```
GCache  g_cache_new (GCacheNewFunc value_new_func, GCacheDestroyFunc
value_destroy_func, GCacheDupFunc key_dup_func, GCacheDestroyFunc
key_destroy_func, GHashFunc hash_key_func, GHashFunc
hash_value_func, GEqualFunc key_equal_func);
```

Todos los parámetros que recibe esta función son parámetros a funciones y, para cada uno de ellos, se debe especificar la función que realizará cada una de las tareas, que son:

- Creación de nuevas entradas en el caché (*`value_new_func`*). Esta función será llamada por `g_cache_insert` (ver más adelante) cuando se solicite la creación de un elemento cuya clave no existe en el caché.
- Destrucción de entradas en el caché (*`value_destroy_func`*). En esta función, que es llamada cuando se destruyen entradas en el caché, se debe liberar toda la memoria alojada para la entrada del caché que va a ser destruida. Normalmente, esto significa liberar toda la memoria alojada en `value_new_func`.
- Duplicación de claves. Puesto que las claves son asignadas fuera del interfaz de `GCache` (es decir, por la aplicación que haga uso de `GCache`), es tarea de la aplicación la creación o duplicación de claves. Así, esta función es llamada cada vez que `GCache` necesita duplicar una clave.
- Destrucción de claves. Al igual que en el caso de las entradas, las claves también deben ser destruidas, liberando toda la memoria que estén ocupando.
- Gestión de la tabla de claves incorporada en `GCache`. Al igual que cuando se usan los `GHashTable`, con `GCache` tambien es necesario especificar las funciones de manejo de la tabla de claves asociada, y para ello se usan los tres últimos parámetros de `g_cache_new`.

Como se puede observar, `GCache` está pensado para ser extendido, no para ser usado directamente. Esto es a lo que se hacía referencia en la introducción, de que es una infraestructura para la implementación de cachés. `GCache` implementa la lógica del caché, mientras que el manejo de los datos de ese caché se dejan de la mano de la aplicación, lo cual se obtiene mediante la implementación de todas estas funciones explicadas en este punto.

Es necesario en este punto hacer un pequeño alto y ver con detalle cómo debe ser la implementación de estas funciones.

### Gestión de los datos del caché.

Una vez que el manejo de los datos del caché están claramente definidos en las funciones comentadas, llega el momento de usar realmente el caché. Para ello, se dispone de un amplio conjunto de funciones.

La primera operación que viene a la mente es la inserción y obtención de datos de ese caché. Esto se realiza mediante el uso de una única función: `g_cache_insert`.

```
gpointer  g_cache_insert (GCache *cache, gpointer key);
```

Esta función busca el objeto con la clave *key* especificada. Si lo encuentra ya disponible en el caché, devuelve el valor asociado a esa clave. Si no lo encuentra, lo crea, para lo cual llama a la función especificada en el parámetro *value_new_func* en la llamada a `g_cache_new`. Así mismo, si el objeto no existe, la clave es duplicada, para lo cual, como no podía ser de otra forma, se llama a la función especificada en el parámetro `key_dup_func` de `g_cache_new`.

Podría ser una buena idea usar cadenas como claves para especificar determinados recursos y, mediante esos identificadores, crear la estructura compleja de datos asociada en la función de creación de entradas del caché. De esta forma, por ejemplo, se podría implementar fácilmente un caché de ficheros obtenidos a través de diferentes protocolos: las claves usadas serían los identificadores únicos de cada fichero (URLs), mientras que la entrada en el caché contendría el contenido mismo del fichero.

Pero, puesto que no siempre se accede al contenido del caché conociendo de antemano las claves, GLib incluye funciones que permiten acceder secuencialmente a todos los contenidos del caché. Esto se puede realizar de dos formas, y, por tanto, con dos funciones distintas:

```
void g_cache_key_foreach(GCache *cache, GHFunc func, gpointer
user_data);
void g_cache_value_foreach(GCache *cache, GHFunc func, gpointer
user_data);
```

Ambas funciones operan de la misma forma, que es llamar retroactivamente a la función especificada en el parámetro *func* por cada una de las entradas en el caché. La diferencia entre ellas es que `g_cache_key_foreach` opera sobre las claves y `g_cache_value_foreach` opera sobre los valores.

En cualquiera de los dos casos, cuando se realice una llamada a una de estas funciones, se deberá definir una función con la siguiente forma:

```
void foreach_func (gpointer key, gpointer value, gpointer user_data);
```

En *key* se recibe la clave de cada una de las entradas (una cada vez), mientras que *value* recibe el valor de dicha entrada. Por su parte, *user_data* es el mismo dato que el que se especifica en la llamada a `g_cache_*_foreach`, y que permite pasar datos de contexto a la función de retrollamada.

Otra operación importante que se puede realizar en un caché es la eliminación de entradas. Para ello, se usa la función `g_cache_remove`.

```
void g_cache_remove(GCache *cache, gconstpointer value);
```

Esta función marca la entrada identificada por *value* para ser borrada. No la borra inmediatamente, sino que decrementa un contador interno asociado a cada entrada, que especifica el número de referencias que dicha entrada tiene. Dichas referencias se asignan a 1 cuando se crea la entrada, y se incrementan cada vez que `g_cache_insert` recibe una petición de creación de esa misma entrada. Así, cuando dicho contador llega a 0, significa que no hay ninguna referencia a la entrada del caché, y por tanto, puede ser eliminada. Cuando la entrada es eliminada, se realizan llamadas a las funciones *value_destroy_func* para la destrucción de la entrada y *key_destroy_func* para la destrucción de la clave, tal y como se especificó en la explicación de la función `g_cache_new`.

### Destrucción del caché.

Una vez que no sea necesario por más tiempo el caché, hay que destruirlo, como con cualquier otra estructura de GLib, de forma que se libere toda la memoria ocupada. Esto se realiza con la función `g_cache_destroy`, cuyo único parámetro es el caché que se quiere destruir.

# GLib avanzado.

## Hilos en Glib.

Los hilos permiten que un programa realice varias tareas simultáneamente. Tradicionalmente, los sistemas operativos tipo UNIX/linux han usado los procesos para este propósito. No obstante, éstos son mucho más pesados (requieren más recursos) que los hilos.

Sin embargo, cada sistema operativo ha implementado de una forma diferente el concepto de hilo y, como resultado, las implementaciones no son compatibles, aunque sean muy parecidas. Ésto implica que si, se desea usar hilos en el programa será necesario proveer de una implementación diferente para cada sistemas operativo tipo UNIX.

GLib solventa este problema ocultando las implementaciones especificas de cada plataforma y proveyendo de un API que puede ser accedido de la misma forma desde cualquier sistema operativo. No obstante, internamente se usará las llamadas nativas disponibles en la plataforma donde se encuentre la aplicación. Es decir, GLib abstrae la implementación de los hilos.

Antes de entrar en materia, es necesario hacer una observación: todos los hilos que un programa crea, comparten el mismo espacio de direcciones; esto es, se podrá acceder a las variables globales de nuestro programa (para lectura y escritura) desde cualquier hilo.

### Inicialización y Creación de Hilos.

Antes de poder usar cualquier función de manejo de hilos, hay que preparar el sistema para ejecutar estas operaciones, es decir, es necesario inicializar el sistema de hilos, nada más sencillo.

```
Void g_thread_init (GThreadFunctions *vtable );
```

En la mayoría de las ocasiones bastará con que el parámetro pasado al sistema de hilos sea nulo, de esta forma GLib se encargará de buscar los parámetros más apropiados.

Es importante no llamar a esta función más de una vez pues, si esto sucediese, la ejecución de nuestro programa sería abortada. Para evitar esto la forma más sencilla es hacer:

```
if (!g_thread_supported ())
g_thread_init (NULL);
```

Después, el sistema estará listo para realizar cualquier operación que requiera el uso de hilos. Para empezar, se creará un hilo:

GThread `*hilo`:

```
hilo = g_thread_create (funcion_hilo, NULL, FALSE, NULL);
```

A partir de este punto, el procedimiento *`funcion_hilo`* se ejecutará en un hilo diferente, el programa continuará su ejecución normal y, en paralelo a él, se ejecutará esta función.

### Sincronización entre hilos.

Como ya se ha dicho, las variables globales del programa van a poder ser accedidas desde cualquier hilo. Esto es una gran ventaja, pues compartir información entre todas las hilos será muy sencillo; no obstante, todo tiene su parte negativa: habrá que tener cuidado a la hora de acceder a estos valores, porque puede que un hilo intente modificar alguna variable cuando otro hilo está leyendo esa misma variable, en cuyo caso, el resultado sería impredecible. Es necesario que los hilos se sincronicen para acceder a estas variables.

Existen varios métodos los sincronización disponibles:

- Cerrojos (Mutex)
- Cerrojos estáticos (Static Mutex)
- Variables Condición

Cerrojos: son llaves mutuamente exclusivas, que permiten conocer si se puede o no acceder a una determinada variable. Si una variable está protegida por un cerrojo, se podrá consultar el estado de la llave para conocer si está siendo accedida por otro hilo en ese instante. Si no lo estuviese, será posible obtener la llave, hacer las operaciones precisas sobre esa variable, y después abrir la llave de nuevo para que otro hilo pueda acceder a la variable.

```
g_mutex_new
g_mutex_lock
g_mutex_trylock
g_mutex_unlock
g_mutex_free
```

Los cerrojos estáticos (*Static Mutex*) son iguales que las anteriores, pero no necesitan ser creadas en tiempo de ejecución; son creadas cuando el programa es compilado.

```
g_static_mutex_init
g_static_mutex_lock
g_static_mutex_trylock
g_static_mutex_unlock
g_static_mutex_get_mutex
```

```
g_static_mutex_free
```

Condiciones:

## UTF-8: las letras del mundo.

Antes de explicar el uso de las funciones que proporciona GLib para el uso de cadenas UTF-8, sería conveniente repasar una serie de conceptos que serán útiles para una mejor conmprensión.

### Unicode: Una forma de representación (UCS).

ISO establece un estándar en el que define el conjunto de caracteres universal conocido como UCS. Este conjunto de caracteres asegura la compatibilidad con otros conjuntos de caracteres, de forma que se puedan usar distintas funciones para pasar de un determinado lenguaje a UCS y de este al lenguaje original sin perder contenido.

Gracias al Unicode se pueden representar la enorme mayoría de lenguajes conocidos, tales como latín, griego, cirílico, hebreo y así un largo etcétera. Unicode es un conjunto de caracteres de 16 bits, es decir, posee 65536 posibles combinaciones, representando cada una de estas combinaciones un carácter.

Un valor o carácter UCS o Unicode, indistintamente llamado de una forma u otro, englobará todos estos lenguajes, partiendo en otras codificaciones más pequeñas como BMP (Basic Multilingual Plane) o Plane 0, subconjunto de 16 bits, al igual que otros como UTF-16, de 16 bits, o UTF-8, de 8 bits, ambos utilizados posteriormente por la GLib tanto para comprobaciones de tipos de caracteres como para conversiones entre distintos subconjuntos Unicode, tales como UTF-8 y UTF-16.

### Qué es Unicode.

Al principio, se intentaron dar dos intentos de conseguir un conjunto de caracteres: el proyecto de estandarización ISO 10646 antes comentado y el proyecto Unicode, pero pronto se comprobó que no era bueno tener dos proyectos en marcha y los dos conjuntos de caracteres diferentes se unieron formando una única tabla de caracteres que fue la de Unicode con las extensiones dadas por la ISO 10646 pasando a ser llamado indistintamente Unicode o ISO 10646. Con el paso del tiempo ha habido modificaciones de Unicode adiciones de caracteres y clasificaciones en subconjuntos como pueden ser el ampliamente usado ASCII, UTF-8, UTF-16 y BMP entre otros.

### Qué es UTF-8.

En GLib existe soporte para UTF-8 debido a sus compatibilidades con el resto de códigos, ya que su codificación engloba en sus primeros 7 bits el tan conocido ASCII.

La codificación UTF-8 está definida en el ISO 10646.

- Los primeros 7 bits son utilizados para la codificación de ASCII, para la compatibilidad con éste por su uso extendido. De forma qu,e desde 0000 a 007F (128 caracteres), se usan para la codificación ASCII de 7 bits.
- Los caracteres mayores a 007F son utilizados para el soporte a otros lenguajes.
- Los bytes FE y FF no son utilizados para la codificación UTF-8.

## Conversiones del juego de caracteres.

En determinadas ocasiones, los programas necesitan el uso de más de un juego de caracteres para un correcto tratamiento de los datos que maneja. Para que esto sea posible sin conflictos ni confusiones, GLib proporciona un útil conjunto de funciones para la conversión entre juegos de caracteres.

### *Nombres de archivos y UTF-8.*

Con todo lo explicado anteriormente, es evidente que el juego de caracteres usado para representar nombres de archivos y el juego de caracteres UTF-8 que podamos usar en nuestros programas puede diferir bastante. Para solventar esto existen las funciones `g_filename_to_utf8` y `g_filename_from_utf8`, que realizan la conversión de nombre de archivo a UTF-8 y de UTF-8 a nombre de archivo, respectivamente.

```
gchar *  g_filename_to_utf8 (const gchar * cadena_a_convertir,
gssizelongitud_de_cadena, gsize * bytes_leidos, gsize *
bytes_escritos, GError ** error);
gchar *  g_filename_from_utf8 (const gchar *  cadena_a_convertir
, gssizelongitud_de_cadena, gsize * bytes_leidos, gsize *
bytes_escritos, GError ** error);
```

Como se puede observar, el uso de las dos funciones es idéntica porque devuelven la cadena convertida al nuevo juego de caracteres o `NULL` en caso de que ocurra un error. Los parámetros *`bytes_leidos`*, *`bytes_escritos`* y *`error`* son parámetros de salida, devolviendo respectivamente los bytes convertidos con éxito, los bytes que ocupa la cadena tras la transformación y el posible error que se pueda producir.

### *Configuración local de caracteres y UTF-8.*

De forma parecida al tratamiento de nombres de archivo, las funciones `g_locale_to_utf8` y `g_locale_from_utf8` convierten cadenas en el juego de caracteres usado internamente por el sistema a cadenas en UTF-8 y viceversa.

```
gchar *  g_locale_to_utf8 (const gchar * cadena_a_convertir, gssize
longitud_de_cadena, gsize * bytes_leidos, gsize * bytes_escritos,
GError ** error);
gchar *  g_locale_from_utf8 (const gchar * cadena_a_convertir, gssize
longitud_de_cadena, gsize * bytes_leidos, gsize * bytes_escritos,
GError ** error);
```

El uso de las funciones es idéntico al descrito con `g_filename_to_utf8` y `g_filename_from_utf8`.

## Otras funciones relacionadas con conversión de cadenas.

Además de las funciones de conversión de cadenas, GLib provee además de un conjunto de funciones destinadas a servir de ayuda y apoyo al tratamiento de cadenas en UTF-8.

La función `g_get_charset` es útil para conocer el juego de caracteres que esta usando el sistema sobre el que se ejecuta el programa, pudiendo evitar transformaciones redundantes.

```
gboolean  g_get_charset (G_CONST_RETURN gchar **  juego_de_caracteres
);
```

La función devuelve `TRUE` si el juego de caracteres que usa el sistema es el UTF-8 y `FALSE` en cualquier otro caso. *juego_de_caracteres* es un parámetro de salida que devulve el juego de caracteres usado por el sistema en formato cadena.

La función `g_utf8_validate` sirve determinar si una cadena de caracteres está ya en formato UTF-8 o no.

```
gboolean  g_utf8_validate (gchar *  cadena_a_validar, gssize
 longitud, const gchar **  final);
```

El parámetro *cadena_a_validar* es la cadena que se quiere validar. *longitud* es el número máximo de caracteres que queremos que valide, puede ser `NULL` si queremos validar la cadena entera; por último, en el último parámetro, la función devuelve un puntero al último carácter que ha podido validar. La función devuelve `TRUE` o `FALSE` dependiendo de si la cadena pasada cumple el formato UTF-8 o no.

## Como hacer *plugins*.

Una de las cosas que pueden interesar a la hora de hacer una aplicación es crear un método por el cual se pueda aplicar extensiones. Ésto se conoce por el nombre de *plugin*. Para los desarrolladores habituales, el término "*plugin*" no debería resultar desconocido, ya que existen aplicaciones en GNOME, como por ejemplo `gedit`, `gnumeric` o algunos reproductores multimedia que aprovechan este método para extender la funcionalidad de la aplicación.

Para entender cómo funcionan los *plugins*, hay que tener claro el concepto de "símbolo". En una librería, a cada función o variable se le da un nombre único (símbolo). Los símbolos que interesan ahora, son los que se exportan, que serán los que se utilicen para enlazarla. Por ejemplo, cuando en un código se llama a una función, lo que sehace es especificar el símbolo de la librería y, en consecuencia, a la función que se desea. Ésto se hace automáticamente a la hora de programar. Pero también se puede hacer manualmente. O sea, se puede especificar una librería en concreto (el *plugin*) y llamar a una función en concreto. De este modo se podría crear muchas librerías (*plugins*), que además implementen una función común en todas ellas y se podría cargar manualmente esa función de cualquiera de los *plugins*. Para luego ejecutarla dentro de el programa. Ésto se verá mucho más claro más adelante, con el ejemplo de esta sección.

Para realizar este tipo de acciones se dispone de GModuleÉste es un sistema que permitirá la carga dinámica de símbolos. Además, siguiendo con una de las máximas de GLib, este sistema es portable. Lo que nos permitirá cargar símbolos en sistemas como Linux, Solaris™, Windows™, HP-UX™.

Para la utilización de GModule es necesario tener presente dos cosas. La primera es que, en el momento de la compilación, se debe indicar que se va a usar GModule, y para ello se indicará en el `pkg-config` que incluya la librería gmodule-2.0. Y por supuesto incluir la linea:

```
#include <gmodule.h>
```

La otra tiene que ver con detectar si el sistema operativo en el que se trabaja, soporta este tipo de acciones. Y para ello se usará la función `g_module_supported` que detectará si se puede hacer uso de las siguientes funciones y devolverá `TRUE` si es así.

```
gboolean  g_module_support ( void );
```

## Abrir una librería.

Si el sistema en el que se trabaja soporta el uso de GModule, el siguiente paso será abrir la librería de la cual posteriormente se extraerá el símbolo deseado. Para ello se debe indicar de alguna manera la posición en nuestro sistema de la librería (*plugin*) que se desea abrir. Y con este fin, se usara la función `g_module_build_path`, con la que podremos construir la ruta exacta donde se encuentra la misma. Esta función devolverá una cadena con la ruta anteriormente nombrada.

```
gchar *  g_module_build_path (const gchar *  directorio, const gchar *
 nombre_del_modulo  );
```

En el primer parámetro de esta función se debe especificar el directorio donde se encuentra la librería que se desea abrir. Y en el segundo, el nombre de la misma. Pero hay que tener claro que sólo el nombre, ya que la función se encargará de poner la extensión de la librería según el sistema operativo en el que este corriendo GLib. Por ejemplo, en Linux colocará la extensión ".so" y en Windows™ ".dll".

Una vez que se obtiene la ruta devuelta por `g_module_build_path`, lo siguiente será cargar la librería. Y para ello está la siguiente función.

```
GModule * g_module_open (const gchar *  ruta_de_la_libreria  ,
GModuleFlags  bandera  );
```

El parámetro bandera puede contener dos valores, un cero o la macro `G_MODULE_BIND_LAZY`. El primer valor, el cero, significa que, cuando se ejecuta esta función, se cargan todos los símbolos de la librería. Mientras que con el segundo valor, sólo lo hará cuando sea necesario.

Otra cosa a tener en cuenta es el resultado de esta función. Como se puede observar, devuelve un puntero a una estructura GModule. Esta estructura es opaca a la visión del desarrollador, pero servirá en posteriores funciones como referencia a la librería que hemos abierto. En el caso de que GModule sea igual a `NULL`, ésto significará que la función no lo ha conseguido.

## Obtener un símbolo determinado.

Ahora ha llegado el momento de obtener el símbolo que se desee de la librería. Como se dijo antes, estos símbolos pueden ser funciones, estructuras o variables que hayan sido exportadas en la librería (*plugin*), así que, de momento, se dará por hecho que existe el símbolo al que se va a llamar con la siguiente función.

```
gboolean  g_module_symbol (GModule *  modulo  , const gchar *  símbolo
, gpointer *  puntero_al_simbolo  );
```

Esta función recibirá como parámetros el puntero a la estructura GModule, que hace referencia a la librería que anteriormente se ha abierto. Después se introduce el nombre del símbolo que se desea obtener y, para terminar, la función devolverá en su tercer parámetro un puntero al símbolo que se haya solicitado. En el caso que se haya

solicitado una función, devolverá un puntero a esa función. Y si lo que se ha solicitado es una variable devolverá un puntero a la misma. Para terminar sólo queda decir que el valor boolean que devuelve, verifica si la función se ha cumplido con éxito o no.

### Preparar el *plugin*.

Con las funciones anteriores se ha clarificado el trabajo que se ha de realizar para llamar al símbolo de la librería (*plugin*). Ahora queda tratar la parte contraria, la del *plugin*. Se ha de tener en cuenta que para que la función `g_module_symbol` encuentre el símbolo que se reclame, este ha de estar declarado de alguna manera. Y para este acto se usa la macro `G_NODULE>`G_MODULE_EXPORT. Esta macro, escrita delante de una variable o función, realiza la función que se desea, la de exportar el símbolo. De este modo, ya será posible referenciar el símbolo deseado desde la función `g_module_symbol`.

Dentro de la preparación de *plugin* con GModule existe una manera de inicializar el *plugin*. Para ello se deberá declarar una función llamada `g_module_check_init`. Dentro de esa función, es posible declarar acciones para la inicialización del *plugin*. Pero esta función deberá seguir estrictamente este prototipo.

```
const gchar*  (*GModuleCheckInit)  (GModule *  modulo  );
```

En el momento en el que se abra la librería (*plugin*) con la función `g_module_open`, ésta buscará esta función y la ejecutará. En el caso que esta función no esté declarada, simplemente habrá abierto la librería. En el caso de que exista la función, podrá retornar `NULL` si ha sido exitosa la inicialización o un cadena de caracteres con la información del error contenida en ella.

Y al igual que se puede especificar una manera de inicializar el *plugin*, también se puede especificar la manera en la que finalizará. Para ello se deberá especificar una función llamada `g_module_unload`. Y, al igual que la anterior, deberá seguir el siguiente prototipo.

```
void  (*GModuleUnload)  (GModule *  modulo  );
```

### Ejemplo de uso de GModule.

Llegados a este punto, es hora demostrar prácticamente todo lo anteriormente explicado. Para ello se usará el siguiente ejemplo, que consta de cuatro archivos. Uno de ellos es un `Makefile`, que nos ayudará a compilar dicho ejemplo y hará más familiar el uso de la herramienta **make**.

El siguiente ejemplo explicará cómo funciona un *plugin*. Para ello se cuenta, por un lado, con una aplicación, correspondiente a los ficheros `modulo.c` y `modulo.h`. Esta aplicación llamará al *plugin* generado por el fichero `plugin.c`, que cuando lo compilemos se llamará `libplugin.so`. La aplicación exportará una estructura llamada info_plugin y una función llamada `escribe`. Dentro de la aplicación, se llamará a los símbolos anteriormente citados y se modificará el contenido de la estructura. Una vez ese contenido ha sido modificado, se llamará al puntero a función correspondiente al símbolo de la función `escribe`. Cuando esta función se ejecuta dentro de la aplicación, lo que realmente se esta ejecutando es la función `escribe`, que imprime por pantalla el contenido de la estructura.

**Ejemplo 4-36. Ejemplo de Makefile para usar GModule**

**Ejemplo 4-37. Cabecera del Módulo**

```
#ifndef __MODULO_H__
#define __MODULO_H__

#include <glib.h>
#include <gmodule.h>

typedef struct _datos_plugin datos_plugin;

struct _datos_plugin {
          gchar *nombre;
          gchar *correo;
};


#endif /* __MODULO_H__ */
```

**Ejemplo 4-38. Código del Módulo**

```
#include <stdlib.h>
#include "modulo.h"

/*
*       Se define un tipo, puntero a función nula, para
*       después declarar una variable (escribe), a la que
*       se asignará un símbolo(en este caso una función) del plugin.
*
*/
typedef void (*Func_escritura) (void);

int
main (int argc, char **argv)
{
        gchar *dir;
        GModule *modulo;
        gchar *plugin;
        datos_plugin **info;
        Func_escritura escribe; /*  Variable que apuntará al símbolo */

        /*  Se comprueba que el sistema soporta los plugins */
        if (!g_module_supported ())
                return 0;

        dir = g_get_current_dir ();

        /*  Se obtiene el path del plugin que se usará */
        plugin = g_module_build_path (dir, "libplugin");

        g_free (dir);

        /*  Se carga el plugin  */
        modulo = g_module_open (plugin, G_MODULE_BIND_LAZY);
        if (!modulo)
                g_error ("error: %s", g_module_error ());

        /* Se obtiene el símbolo info_plugin, una estructura de datos */
        if (!g_module_symbol (modulo, "info_plugin", (gpointer *) & info))
                g_error ("error: %s", g_module_error ());
```

```
        (*info)->nombre = "Pepe Lopez";         /*   Aquí se usa el símbolo  */
        (*info)->correo = "pepe@gnome.org";     /*   info_plugin             */

        /* Se obtiene el símbolo plugin_escribe, una función */
        if (!g_module_symbol
            (modulo, "plugin_escribe", (gpointer *) & escribe))
                g_error ("error: %s", g_module_error ());

        escribe ();             /*      Aquí se usa el símbolo plugin_escribe */

        /*      Se descarga el plugin   */
        if (!g_module_close (modulo))
                g_error ("error: %s", g_module_error ());

        return EXIT_SUCCESS;
}
```

**Ejemplo 4-39. Ejemplo de *plugin***

```
#include "modulo.h"

/*
*       Se declara un puntero a una estructura de datos
*       y se exporta como símbolo, para poder ser
*       utilizado por las aplicaciones que carguen
*       este plugin
*
*/
G_MODULE_EXPORT datos_plugin *info_plugin;


/*
*       Esta función inicializa el plugin. Se exporta como simbolo.
*
*       Se ejecuta al ser cargado el plugin -> g_module_open()
*
*/
G_MODULE_EXPORT const gchar *
g_module_check_init (GModule * module)
{
        g_print ("--- Inicializacion del plugin ---\n\n");
        info_plugin = g_malloc (sizeof (datos_plugin));
        return NULL;
}


/*
*       Esta función finaliza el plugin. Se exporta como simbolo.
*
*       Se ejecuta al ser descargado el plugin -> g_module_close()
*
*/
G_MODULE_EXPORT void
g_module_unload (GModule * module)
{
        g_print ("\n--- Finalizacion del plugin ---\n");
        g_free (info_plugin);
}


/*
*       Esta función será un símbolo del plugin
*       que se podrá llamar a voluntad en cualquier
```

```
*       momento, una vez se haya cargado el plugin.
*
*/
G_MODULE_EXPORT void
plugin_escribe (void)
{
        g_return_if_fail (info_plugin != NULL && info_plugin->nombre != NULL &&
                info_plugin->correo != NULL);

        g_print ("Nombre : %s \ncorreo-e : %s\n",
                 info_plugin->nombre, info_plugin->correo);
}
```

Y el resultado de ejecutar el programa obtenido de compilar, tanto el *plugin*, como el modulo con **make**, sería el siguiente:

```
bash$:/var/CVS/librognome/book/code/GModule$

./modulo
--- Inicializacion del plugin ---

    Nombre : Pepe Lopez
    correo-e : pepe@gnome.org

    --- Finalizacion del plugin ---
```

## Notas

1. http://developer.gnome.org/doc/API/2.0/glib/index.html
2. http://developer.gnome.org/doc/API/2.0/glib/index.html

# Capítulo 5. Sistema de objetos de GLib.

Una de las técnicas más modernas y utilizadas en la actualidad en la programación es lo que se denomina “programación orientada a objetos”, o POO, que consiste en enfocar la programación de una forma mucho más cercana a la percepción real de las personas (los programadores), usando el concepto de objetos para encapsular las distintas funcionalidades de las aplicaciones.

La POO permite estructurar los programas de una forma mucho más clara para la percepción humana que la programación “tradicional” (programación estructurada) de forma que, cada funcionalidad específica de la aplicación, está claramente separada y encapsulada (implementación oculta al resto de la aplicación). Esto permite varias cosas:

- Desarrollar cada una de las partes de la aplicación independientemente del resto.

La POO normalmente está asociada a lenguajes de programación que la soportan, tales como Java, C++, SmallTalk, etc. Sin embargo, al ser una técnica realmente potente para el desarrollo de aplicaciones, en el proyecto GNOME, y más concretamente en GTK+, se decidió usar dicha técnica en el lenguaje más usado dentro del proyecto: C. Éste no es un lenguaje preparado para POO, pero con un esfuerzo mínimo, y gracias al trabajo de los desarrolladores de GTK+, se puede usar en este lenguaje. Esto se consigue mediante el sistema de objetos de GLib (`GObject`), que, mediante el uso de las características del lenguaje C, ofrece la posibilidad de usar POO.

Este sistema de objetos de GLib tiene algunas limitaciones propias del uso del lenguaje C (lenguaje no pensado para la POO), pero aun así, su funcionalidad es tal, que es más que probable que jamás se eche ninguna funcionalidad en falta. Dicha funcionalidad incluye:

- herencia, característica imprescindible de cualquier lenguaje orientado a objetos que se precie de serlo; permite la creación de clases que heredan la funcionalidad de otras clases ya existentes. Esto permite crear clases con la funcionalidad básica y, basadas en dicha clase, crear otras que añadan una funcionalidad más específica.
- polimorfismo, que permite tratar a un mismo objeto bajo distintas personalidades.
- interfaces, que permite la definición de interfaces (clases abstractas) y su posterior implementación en clases.

## Gestión dinámica de tipos.

`GLib` incluye un sistema dinámico de tipos, que no es más que una base de datos en la que se van registrando las distintas clases. En esa base de datos, se almacenan todas las propiedades asociadas a cada tipo registrado, información tal como las funciones de inicialización del tipo, el tipo base del que deriva, el nombre del tipo, etc. Todo ello identificado por un identificador único, conocido como GType.

### Tipos basados en clases (objetos).

Para crear instancias de una clase, es necesario que el tipo haya sido registrado anteriormente, de forma que esté presente en la base de datos de tipos de GLib. El registro de tipos se hace mediante una estructura llamada GTypeInfo, que tiene la siguiente forma:

```
struct _GTypeInfo
{
/* interface types, classed types, instantiated types */
guint16                class_size;

GBaseInitFunc          base_init;
```

```
GBaseFinalizeFunc      base_finalize;

/* classed types, instantiated types */
GClassInitFunc         class_init;
GClassFinalizeFunc     class_finalize;
gconstpointer          class_data;

/* instantiated types */
guint16                instance_size;
guint16                n_preallocs;
GInstanceInitFunc      instance_init;

/* value handling */
const GTypeValueTable *value_table;
};
```

La mayor parte de los miembros de esta estructura son punteros a funciones. Por ejemplo, el tipo GClassInitFunc es un tipo definido como puntero a función, que se usa para la función de inicialización de las clases.

Para comprender esta estructura, lo mejor es ver un ejemplo de cómo se usa. Para ello, se usan las siguientes funciones:

```
GType g_type_register_static (GType            parent_type,
                                  const gchar     *type_name,
                                  const GTypeInfo *info,
GTypeFlags       flags);
GType g_type_register_fundamental (GType                       type_id,
                                         const gchar                 *type_name,
                                         const GTypeInfo             *info,
                                         const GTypeFundamentalInfo *finfo,
                                         GTypeFlags                   flags);
```

Con estas dos funciones y la estructura GTypeInfo se realiza el registro de nuevos tipos en GLib y, todo ello normalmente, se realiza en la función `_get_type` de la clase que se esté creando. Esta función es la que se usará posteriormente para referenciar al nuevo tipo, y tiene, normalmente, la siguiente forma:

```
GType
my_object_get_type (void)
{
static GType type = 0;

if (!type) {
static GTypeInfo info = {
sizeof (MyObjectClass),
(GBaseInitFunc) NULL,
(GBaseFinalizeFunc) NULL,
(GClassInitFunc) my_object_class_init,
NULL, NULL,
sizeof (MyObject),
0,
(GInstanceInitFunc) my_object_init
};

type = g_type_register_static (PARENT_TYPE, "MyObject", &info, 0);
}

return type;
}
```

Como se puede apreciar, esta función simplemente tiene una variable (*type*) estática, que se inicializa a 0 y, cuando se le llama, si dicha variable es igual a 0, entonces registra el nuevo tipo (llamando a `g_type_register_static`) y rellena una estructura de tipo GTypeInfo antes, con los datos de la nueva clase, tales como el tamaño de la estructura de la clase (MyObjectClass), la función de inicialización de la clase (`my_object_class_init`), el tamaño de la estructura de las instancias de la clase (MyObject) y la función de inicialización de las instancias que se creen de esta clase (`my_object_init`).

Una vez que se tiene la función que registra la clase, crear instancias de esa clase es tan sencillo como llamar a la función `g_object_new`, que tiene la siguiente forma:

```
gpointer g_object_new (GType        object_type,
                       const gchar *first_property_name,
                       ...);
```

Esta función tiene una lista variable de argumentos, que sirve para especificar una serie de propiedades a la hora de crear el objeto. Pero, de momento, lo único interesante de `g_object_new` es conocer su funcionamiento. Por ejemplo, para crear una instancia de la clase MyObject, una vez que se tiene la función de registro de la clase (`my_object_get_type`), no hay más que llamar a `g_object_new` de la siguiente forma:

```
GObject *obj;

obj = g_object_new (my_object_get_type (), NULL);
```

De esta forma, se le pide a GLib que cree una nueva instancia de la clase identificada por el valor devuelto por la función `my_object_get_type`, que será el valor devuelto por `g_type_register_static`, tal y como se mostraba anteriormente.

## Tipos no instanciables (fundamentales).

Muchos de los tipos que se registran no son directamente instanciables (no se pueden crear nuevas instancias) y no están basados en una clase. Estos tipos se denominan tipos "fundamentales" en la terminología de GLib y son tipos que no están basados en ningún otro tipo, tal y como sí ocurre con los tipos instanciables (u objetos).

Entre estos tipos fundamentales se encuentran algunos viejos conocidos, como por ejemplo gchar y otros tipos básicos, que son automáticamente registrados cada vez que se inicia una aplicación que use GLib.

Como en el caso de los tipos instanciables, para registrar un tipo fundamental es necesaria una estructura de tipo GTypeInfo, con la diferencia que para los tipos fundamentales bastará con rellenar con ceros toda la estructura, excepto el campo *value_table*.

```
GTypeInfo info = {
0,    /* class_size */
NULL,   /* base_init */
NULL,   /* base_destroy */
NULL,   /* class_init */
NULL,   /* class_destroy */
NULL,   /* class_data */
0,    /* instance_size */
0,    /* n_preallocs */
NULL,   /* instance_init */
NULL,   /* value_table */
};
static const GTypeValueTable value_table = {
    value_init_long0,  /* value_init */
```

```
NULL,   /* value_free */
value_copy_long0,  /* value_copy */
NULL,   /* value_peek_pointer */
"i",   /* collect_format */
value_collect_int, /* collect_value */
"p",   /* lcopy_format */
value_lcopy_char,  /* lcopy_value */
};
info.value_table = &value_table;
type = g_type_register_fundamental (G_TYPE_CHAR, "gchar", &info, &finfo, 0);
```

La mayor parte de los tipos no instanciables estan diseñados para usarse junto con GValue, que permite asociar fácilmente un tipo GType con una posición de memoria. Se usan principalmente como simples contenedores genéricos para tipos sencillos (números, cadenas, estructuras, etc.).

## Implementación de nuevos tipos.

En el apartado anterior se mostraba la forma de registrar una nueva clase en el sistema de objetos de GLib, y se hacía referencia a distintas estructuras y funciones de inicialización. En este apartado, se va a indagar con más detalle en ellos.

Tanto los tipos fundamentales como los no fundamentales quedan definidos por la siguiente información:

- Tamaño de la clase.
- Funciones de inicialización (constructores en C++).
- Funciones de destrucción (destructores en C++), llamadas de finalización en la jerga de GLib.
- Tamaño de las instancias.
- Normas de instanciación de objetos (uso del operador *new* en C++).
- Funciones de copia.

Toda esta información queda almacenada, como se comentaba anteriormente, en una estructura de tipo GTypeInfo. Todas las clases deben implementar, aparte de la función de registro de la clase (`my_object_get_type`), al menos dos funciones que se especificaban en la estructura GTypeInfo a la hora de registrar la clase. Estas funciones son:

- `my_object_class_init`: función de inicialización de la clase, donde se inicializará todo lo relacionado con la clase en sí, tal como las señales que tendrá la clase, los manejadores de los métodos virtuales si los hubiera, etc.
- `my_object_init`: función de inicialización de las instancias de la clase, que será llamada cada vez que se solicite la creación de una nueva instancia de la clase. En esta función, las tareas a desempeñar son todas aquellas relacionadas con la inicialización de una nueva instancia de la clase, tales como los valores iniciales de las variables internas de la instancia.

## Interfaces

Los interfaces GType son muy similares a los interfaces en Java o C#. Es decir, son definiciones de clases abstractas, sin ningún tipo de implementación, que definen una serie de operaciones que debe segir las clases que implementen dichos interfaces deben seguir.

En GLib, para declarar un interfaz, es necesario registrar un tipo de clase no instanciable, que derive de GTypeInterface. Por ejemplo:

```
struct _GTypeInterface
{
GType g_type;         /* iface type */
GType g_instance_type;
};
typedef struct
{
GTypeInterface g_iface;

void (*method_a) (FooInterface *foo);
void (*method_b) (FooInterface *foo);
} FooInterface;

static void foo_interface_method_a (FooInterface *foo)
{
foo->method_a ();
}

static void foo_interface_method_b (FooInterface *foo)
{
foo->method_b ();
}

static void foo_interface_base_init (gpointer g_class)
{
/* Initialize the FooInterface type. */
}

GType foo_interface_get_type (void)
{
GType foo_interface_type = 0;

if (foo_interface_type == 0) {
static const GTypeInfo foo_interface_info =
{
sizeof (FooInterface),      /* class_size */
foo_interface_base_init,    /* base_init */
NULL,                       /* base_finalize */
NULL,
NULL,                       /* class_finalize */
NULL,                       /* class_data */
0,                          /* instance_size */
0,                          /* n_preallocs */
NULL                        /* instance_init */
};

foo_interface_type = g_type_register_static (G_TYPE_INTERFACE, "FooInterface",
&foo_interface_info, 0);
}
return foo_interface_type;
}
```

Un interfaz queda definido por una sola estructura cuyo primer miembro deber ser del tipo GTypeInterface, que es la clase base (ver la la sección de nombre *Herencia*) para los interfaces. Aparte de este primer miembro, la estructura que define el interfaz debe contener punteros a las funciones definidas en el interfaz. Es decir, los métodos del interfaz. Aparte de eso, como muestra el ejemplo anterior, constituye buena práctica incluir funciones que encapsulen el acceso a esos punteros a funciones, tal y como hacen las funciones `foo_interface_method_a` y `foo_interface_method_b` del ejemplo anterior. Aunque esto no es obligatorio, es aconsejable, pues ayuda en la legibilidad del código donde se haga uso del interfaz.

Hasta aquí, la definición del interfaz, que, como se puede apreciar, no contiene ninguna implementación. Simplemente, contiene la definición de los métodos que deben ser implementados para soportar este interfaz. Tras esto, para que el interfaz definido sea útil, es necesario hacer que otras clases definidas implementen dicho interfaz. Para ello, se usa la función `g_type_add_interface_static`, tal y como se muestra en el siguiente fragmento de código, en la función `attach_interface_to_type`:

```
static void
implementation_method_a (FooInterface *iface)
{
/* Implementación de método A */
}

static void
implementation_method_b (FooInterface *iface)
{
/* Implementación de método B */
}


static void
foo_interface_implementation_init (gpointer g_iface,
gpointer iface_data)
{
FooInterface *iface = (FooInterface *) g_iface;
iface->method_a = implementation_method_a;
iface->method_b = implementation_method_b;
}

static void attach_interface_to_type (GType type)
{
static const GInterfaceInfo foo_interface_implementation_info =
{
(GInterfaceInitFunc) foo_interface_implementation_init,
NULL,
NULL
};
g_type_add_interface_static (type,
foo_interface_get_type (),
&foo_interface_implementation_info);
}
```

`g_type_add_interface_static` registra, dentro del sistema de tipos de GLib que un determinado tipo implementa un determinado interfaz. Antes de eso, es necesario rellenar una estructura de tipo GInterfaceInfo, que contiene todos los datos (funciones de inicialización y finalización, así como datos de contexto a los que se podrá acceder desde cualquier punto, a través de la implementación del interfaz) referentes a una implementación concreta del interfaz:

```
struct _GInterfaceInfo
{
GInterfaceInitFunc     interface_init;
GInterfaceFinalizeFunc interface_finalize;
gpointer               interface_data;
};
```

### Herencia

Un sistema orientado a objetos que se precie debe, de alguna forma, ofrecer la posibilidad de crear nuevas clases que contengan la funcionalidad base de otras clases ya existentes. Esto se conoce como herencia en la terminología de la POO y permite la reutilización de funcionalidad ya existente, de forma que, simplemente, haya que añadir la funcionalidad extra requerida.

En GLib, desarrollada en C, esto se consigue mediante el uso de estructuras, como se ha visto anteriormente, y usando, como primer miembro de la estructura que identifica a la nueva clase, un valor del mismo tipo que la clase base. Por ejemplo, si se quisiera crear una clase `ClaseB` que derivara de `ClaseA`, la nueva clase debería definirse de la siguiente manera:

```
struct ClaseA {
...
};
struct ClaseAClass {
...
};
...
struct ClaseB {
struct ClaseA base;
};
struct ClaseBClass {
struct ClaseAClass base_class;
};
```

Éste es el primer paso para usar la herencia. De esta forma, como el primer miembro de la clase derivada es del tipo de la clase base, se puede convertir ("casting" en la terminología del lenguaje C) de un tipo a otro indistintamente. Así, por ejemplo:

```
ClaseA *clasea;
ClaseB *claseb;

claseb = g_object_new (claseb_get_type (), NULL);
clasea = (ClaseA *) claseb;
```

Los demás pasos para conseguir la herencia ya han sido comentados anteriormente y se reducen a especificar la clase base de la nueva clase al registrarla (`g_type_register_static`).

## Señales

Las señales constituyen la forma que tienen las clases en GLib de informar de determinados cambios de estado o de requerir la ejecución de determinadas acciones. Pero antes de entrar en detalle sobre qué son y para qué sirven las señales, es necesario el estudio de una serie de conceptos, que se detallan a continuación.

### GValue

GValue representa una forma única de contener un valor de cualquier tipo. Viendo su definición se comprueba mejor qué significa esto:

```
struct _GValue
{
GType  g_type;

/* public for GTypeValueTable methods */
```

```
union {
gint v_int;
guint v_uint;
glong v_long;
gulong v_ulong;
gint64      v_int64;
guint64     v_uint64;
gfloat v_float;
gdouble v_double;
gpointer v_pointer;
} data[2];
};
GValue* g_value_init (GValue *value,
GType g_type);
void g_value_unset (GValue *value);
```

Como puede observarse, GValue permite albergar valores de distintos tipos, todo ello en una unión de C. Lo interesante es que, a partir de esta simple organización (una estructura que contiene el tipo del valor almacenado y el valor), se pueden pasar valores de distintos tipos de un sitio a otro, usando un método común entre todas las partes involucradas.

Existen multitud de funciones relacionados con GValue, que permiten desde copiarlo (`g_value_copy`) a alterar su contenido fácilmente (`g_value_get/_set`).

## GObject, la clase base.

La base de todo este sistema de objetos es `GObject`, una clase que implementa todo lo básico, y que forma la base de todas las demás clases, tanto en GTK+ como en el resto de librerías de la plataforma GNOME.

`GObject` es una clase abstracta, es decir, una clase de la que no podemos crear instancias directamente, aunque sí contiene alguna implementación, como es toda la gestión del control de vida del objeto, de sus señales y manejadores, etc.

## Parámetros y valores.

# Capítulo 6. Pango

Una de las novedades más importantes de GTK2 es el uso de Pango para representar el texto por pantalla.

Pango es una librería que nos facilita la representación y visualización del texto internacional en la pantalla. Soporta casi todos los lenguajes que pueden existir en el mundo, y puede funcionar a la cabeza de múltiples sistemas de ventanas - incluyendo las tradicionales X fonts, o las fuentes OpenType. Pango se usa en todos los programas que están siendo migrado para la versión 2.0 de GNOME que usan el toolkit de las GTK.

## Caminando hacia Pango

Problemas que habia (muchos lenguajes, unicode como ASCII no es la solución, textos complicado...) Lenguajes dificiles (Texto bidireccional -Hebréo, Árabe- y texto complejo -Thailandés, Coreano-)

## Pango, a modo de introducción

Motivaciones para Pango (No hay soporte en las Xlib, crea una librería potente y versátil, Un solo API) Similares a los de los sistemas propietarios (Uniscribe -Microsoft-, Texto Java 2d, ATSUI -apple-)

## Arquitectura

Arquitectura de Pango, introducción

- Pango Core
- Renderización de librerías
- Módulos (Módulos de lenguaje y módulos de forma)

## El API de Pango e Implementación

Nivel bajo del API (Muchos pasos, más control)

Nivel alto del API (Pocos pasos, menos control) Opera en el objeto PangoLayout Dibujo {pangox_draw_layout()}

## Pango en GTK+2.0

Convirtiendo GTK+ (Muchos widgets se hacen facilísimos, el widget de texto no es conveniente portarlo a Pango, tiene ke ser removido de cualquier forma)

Convirtiendo aplicaciones disponibles

## El futuro de Pango

Escritura vertical, Las mejores fuentes para las X, Tipografía bonita

## Referencias

Referencias de Inet

# Capítulo 7. GTK+

GTK+ es la librería gráfica (toolkit) sobre la que se sustenta todo el interfaz gráfico de GNOME. Es una librería que contiene todo lo necesario para el desarrollo de interfaces gráficas, permitiendo la posibilidad de crear todo tipo de widgets, desde los más básicos, como botones, etiquetas, cajas de texto, hasta cosas más complejas como selectores de ficheros, colores, fuentes, cajas de texto multilínea con soporte para todo tipo de efectos sobre el texto, etc.

## Qué es un widget

En términos de ingeniería del software, un widget es un componente software visible y personalizable. Visible porque está pensado para ser usado en los interfaces gráficos de los programas, y personalizable porque el programador puede cambiar muchas de sus propiedades. De esta forma se logra una gran reutilización del software, un objetivo prioritario en ingeniería del software. Los widgtes se combinan para construir los interfaces gráficos de usuario. El programador los adapta según sus necesidades sin tener que escribir más código que el necesario para definir los nuevos valores de las propiedades de los widgets.

La librería GTK+ sigue el modelo de programación orientado a objetos. La jerarquía de objetos comienza en `GObject` de la librería Glib del que hereda `GtkObject`. Todos los widgets heredan de la clase de objetos `GtkWidget`, que a su vez hereda directamente de `GtkObject`. La clase `GtkWidget` contiene las propiedades comunes a todos los widgets; cada widget particular le añade sus propias propiedades.

Los widgets se definen mediante punteros a una estructura `GtkWidget`. En GTK+, los widgets presentan una relación padre/hijo entre sí. Las aplicaciones suelen tener un widget "ventana" de nivel superior que no tiene padre, pero aparte de él, todos los widgets que se usen en una aplicación deberán tener un widget padre. Al widget padre se le denomina contenedor. El proceso de creacción de un widget consta de dos pasos: el de creación propiamente dicho y el de visualización. La función de creación de un widget tiene un nombre que sigue el esquema `gtk_nombre_new` donde "nombre" debe sustituirse por el nombre del widget que se desea crear. La función `gtk_widget_show` hará visible el widget creado.

La función de creación de un widget `gtk_nombre_new` devuelve un puntero a un objeto de tipo `GtkWidget` y no un puntero a un widget del tipo creado. Por ejemplo, la función `gtk_button_new` devuelve un puntero a un objeto de `GtkWidget` y no una referencia a un botón. Esta referencia puede convertirse a una referencia a un objeto `GtkButton` mediante la macro GTK_BUTTON, si se desea utilizar en lugares donde se requieran objetos botones. Aunque sería posible pasar en esos casos la referencia genérica, el compilador se quejará si se hace así posibilitando un control de tipos de objetos. Todo widget tiene una macro de conversión de una referencia genérica a una referencia al tipo propio. Eso sí, la macro únicamente funcionará correctamente si el widget referenciado fue creado con la función de creación del tipo apropiado, lo que incluye el propio tipo del widget o un descendiente.

El interfaz gráfico de una aplicación se construye combinando diferentes widgets (ventanas, cuadros combinados, cuadros de texto, botones, ...) y se establecen diversas retrollamadas (callbacks) para estos widgets, de esta forma se obtiene el procesamiento requerido por el programa a medida que se producen ciertas señales que a su vez provocan las retrollamadas. Las señales se producen por diversos sucesos como oprimir el boton de un ratón que se encuentra sobre un widget botón, pasar el cursor por encima de un widget u oprimir una tecla.

GTK+ utiliza GDK para visualizar los widgets. GDK es una interfaz de programación (API) de aplicaciones que se situa por encima de la API gráfica nativa como Xlib o Win32. De esta forma portando GDK pueden utilizarse las aplicaciones construidas con GTK+ en otras plataformas.

## Widgets básicos

El conjunto de "widgets" básicos incluye todos los que nos podríamos esperar de una librería como GTK, es decir, botones, etiquetas, marcos, barras de tareas y de menús, ventanas, cajas de diálogo, etc. Todos y cada uno de ellos, implementado como una clase (es decir, un objeto derivado de GObject, aunque indirectamente, pues todos los "widgets" están basados en la clase GtkWidget), permiten una personalización bastante amplia, lo cual nos permite desarrollar nuestros interfaces de usuario al más mínimo detalle.

Cada clase en GTK ofrece una función _new que nos permite crear una instancia de esa clase. Así, por ejemplo, tenemos las siguientes funciones:

```
GtkWidget *gtk_list_new (void);
GtkWidget *gtk_toolbar_new (GtkOrientation orientation, GtkToolbarStyle style);
GtkWidget *gtk_button_new (void);
GtkWidget *gtk_button_new_with_label (const gchar *label);
...
```

Una vez que hemos creado una instancia de una clase GTK, podemos alterar su aspecto/comportamiento mediante las funciones disponibles para esa clase, así como las funciones disponibles en las clases de las que, directa o indirectamente, hereda la primera clase.

Esto último, tratar a una clase como otra, se consigue mediante macros de conversión. Así, por ejemplo, la clase GtkCheckButton, que deriva de GtkToggleButton puede ser tratada de la siguiente forma:

```
GtkWidget *check_button;
check_button = gtk_check_button_new_with_label("un check button");
...
if (gtk_toggle_button_get_active(GTK_TOGGLE_BUTTON(check_button))) {
/* el "check button" está activado */
...
}
```

Como se ve en este ejemplo, primero creamos una instancia de la clase GtkCheckButton mediante la función gtk_check_button_new_with_label, para luego comprobar si está activo mediante la función gtk_toggle_button_get_active. Esto se debe a que, como coméntabamos anteriormente, la clase GtkCheckButton hereda de la clase GtkToggleButton y, por tanto, podemos usar todas las funciones de esta última clase con cualquier objeto de la clase GtkCheckButton.

Este mecanismo de herencia es el equivalente a la herencia en los lenguajes orientados a objetos, en los que no es necesaria la conversión de los objetos, sino que es posible acceder a los métodos/propiedades del objeto directamente. Así, para entenderlo mejor, veamos el equivalente, en C++, del ejemplo anterior.

```
Gtk::CheckButton check_button = new Gtk::CheckButton("un check button");
...
if (check_button->get_active()) {
  ...
}
```

## Bucle de ejecución y eventos

Como es habitual en cualquier librería de desarrollo de interfaces gráficos, en GTK+ se usa el modelo de programación por eventos. Este modelo consiste en la asociación de determinadas acciones ("eventos") a determinadas operaciones ("funciones"). Dichas acciones, o eventos, marcan un cambio de estado de determinado objeto, de forma que la aplicación pueda ser informada de ello.

Los eventos en GTK se refieren, en terminología `GObject`, a señales especificadas en la definición de las clases de los distintos widgets. Así, por ejemplo, el widget `GtkButton` tiene definida una señal llamada "clicked" que es emitida (emitir una señal es la terminología usada en GTK) cuando el botón en cuestión es pulsado por el usuario, de forma que la aplicación no tiene más que "conectarse" a dicha señal para ser informada por GTK+ cada vez que dicho botón es pulsado.

Para que éste sistema de señales funcione correctamente, se usa lo que se denomina el bucle de eventos, que no es más que un bucle interno de GTK+, en el que se van, una y otra vez, comprobando los estados de cada uno de los elementos de la aplicación, e informando de dichos cambios a los elementos que se hayan registrado para ser informados. Este concepto es exactamente idéntico al del bucle de ejecución contenido en la librería GLib, y, de hecho, no es más que una ampliación de dicho concepto para hacerlo funcionar en aplicaciones con interfaz gráfica. Este bucle de eventos GTK+ se traduce básicamente en dos funciones, que son:

```
void       gtk_main                 (void);
void       gtk_main_quit            (void);
```

Estas dos funciones, análogas a `g_main_run` y `g_main_quit` representan el interfaz que tienen las aplicaciones para usar el bucle de eventos. Como en el caso de su equivalente en GLib, `gtk_main` ejecuta el bucle de eventos. Esto significa que, una vez que se haya realizado la llamada a `gtk_main`, se cede todo el control de la aplicación a GTK. Es decir, `gtk_main` no retorna hasta que, dentro de algun manejador (instalado *ANTES* de llamar a `gtk_main`) se haga una llamada a `gtk_main_quit`, que, como su nombre indica, termina el bucle de eventos de GTK+.

Un ejemplo de una típica aplicación GTK+ sería:

```
int main (int argc, char *argv[])
{
    gtk_init (&argc, &argv);
    /* creación del interfaz principal */
    /* conexión a las distintas señales */

    gtk_main ();

    return 0;
}
```

Como puede comprobarse, el programa inicializa GTK+, crea el interfaz básico, conecta funciones a las distintas señales en las que esté interesado (llamadas a `g_signal_connect`), para seguidamente llamar a `gtk_main` para que se ejecute la aplicación. Cuando en algun manejador de señal realicemos una llamada a `gtk_main_quit`, `gtk_main` retornará, tras lo cual la aplicación termina.

### Señales

Las señales en GTK+ no son más que una muestra de uso extensivo del sistema de objetos de `GLib`. Las señales son el medio mediante el cual GTK+ informa a las aplicaciones de los eventos producidos en el interfaz gráfico. Estos eventos o señales normalmente estan relacionados con un cambio de estado en determinado widget, como el ejemplo de la clase `GtkButton` que se mencionaba en el apartado anterior,

que tiene definida una señal llamada "clicked" para informar de la pulsación, por parte del usuario, del botón en cuestión.

## Ejemplo básico

Antes de seguir adelante, lo mejor es ver un ejemplo completo de un programa (básico) en GTK, de forma que puedan apreciarse claramente los pasos básicos.

```
#include <gtk/gtk.h>

/* This is a callback function. The data arguments are ignored
 * in this example. More on callbacks below. */
void hello( GtkWidget *widget, gpointer   data )
{
        g_print ("Hello World\n");
}

gint delete_event( GtkWidget *widget,
                   GdkEvent  *event,
                   gpointer   data )
{
        /* If you return FALSE in the "delete_event" signal handler,
         * GTK will emit the "destroy" signal. Returning TRUE means
         * you don't want the window to be destroyed.
         * This is useful for popping up 'are you sure you want to quit?'
         * type dialogs. */

        g_print ("delete event occurred\n");

        /* Change TRUE to FALSE and the main window will be destroyed with
         * a "delete_event". */

        return TRUE;
}

/* Another callback */
void destroy( GtkWidget *widget, gpointer   data )
{
        gtk_main_quit ();
}

int main( int   argc, char *argv[] )
{
        /* GtkWidget is the storage type for widgets */
        GtkWidget *window;
        GtkWidget *button;

        /* This is called in all GTK applications. Arguments are parsed
         * from the command line and are returned to the application. */
        gtk_init (&argc, &argv);

        /* create a new window */
        window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

        /* When the window is given the "delete_event" signal (this is given
         * by the window manager, usually by the "close" option, or on the
         * titlebar), we ask it to call the delete_event () function
         * as defined above. The data passed to the callback
         * function is NULL and is ignored in the callback function. */
        g_signal_connect (G_OBJECT (window), "delete_event",
                          G_CALLBACK (delete_event), NULL);

        /* Here we connect the "destroy" event to a signal handler.
```

```
     * This event occurs when we call gtk_widget_destroy() on the window,
     * or if we return FALSE in the "delete_event" callback. */
    g_signal_connect (G_OBJECT (window), "destroy",
                      G_CALLBACK (destroy), NULL);

    /* Sets the border width of the window. */
    gtk_container_set_border_width (GTK_CONTAINER (window), 10);

    /* Creates a new button with the label "Hello World". */
    button = gtk_button_new_with_label ("Hello World");

    /* When the button receives the "clicked" signal, it will call the
     * function hello() passing it NULL as its argument.  The hello()
     * function is defined above. */
    g_signal_connect (G_OBJECT (button), "clicked",
                      G_CALLBACK (hello), NULL);

    /* This will cause the window to be destroyed by calling
     * gtk_widget_destroy(window) when "clicked".  Again, the destroy
     * signal could come from here, or the window manager. */
    g_signal_connect_swapped (G_OBJECT (button), "clicked",
                              G_CALLBACK (gtk_widget_destroy),
                              G_OBJECT (window));

    /* This packs the button into the window (a gtk container). */
    gtk_container_add (GTK_CONTAINER (window), button);

    /* The final step is to display this newly created widget. */
    gtk_widget_show (button);

    /* and the window */
    gtk_widget_show (window);

    /* All GTK applications must have a gtk_main(). Control ends here
     * and waits for an event to occur (like a key press or
     * mouse event). */
    gtk_main ();

    return 0;
}
```

## El ejemplo paso a paso

Ahora que ya conocemos la teoría básica, explicaremos paso a paso el ejemplo mostrado en el punto anterior.

Aquí tenemos la función que será llamada cuando se haga "click" sobre el botón. En este ejemplo, el cuerpo de la función ignorará tanto el widget como los datos que recoge como parámetros, pero no sería difícil saber como utilizarlos. El siguiente ejemplo usará el argumento data para controlar qué botón se pulsó.

```
static void hello( GtkWidget *widget,
                   gpointer   data )
{
    g_print ("Hello World\n");
}
```

La siguiente función de conexión es un poco especial. El gestor de ventanas emite el evento "delete_event" a la aplicación y en ese momento se llamará a la función que hemos definido con el nombre `"delete_event"` . Ahora podemos reaccionar de varias formas dentro de la función: podemos ignorar el evento, procesarlo o símplemente cerrar la aplicación.

El valor que devuelva esta función de conexión le permite a GTK conocer qué acción debe llevar a cabo. Si devolvemos TRUE, estamos indicando que no queremos que se emita la señal "destroy", lo que permitirá que nuestra aplicación siga ejecutándose. Si devolvemos FALSE, indicaremos que se emita la señal "destroy", evento que será recogido por nuestra función de conexión "destroy".

```
static gboolean delete_event( GtkWidget *widget,
                              GdkEvent  *event,
                              gpointer   data )
{
    g_print ("delete event occurred\n");

    return TRUE;
}
```

Aquí tenemos otra función callback que provoca que el programa termine mediante la llamada a la función `gtk_main_quit()`. Esta función indica a GTK que debe salir de `gtk_main` cuando se le devuelva el control desde la función.

```
static void destroy( GtkWidget *widget,
                     gpointer   data )
{
    gtk_main_quit ();
}
```

Asumimos que el lector conoce la función main()... sí, como sucede con otras aplicaciones, todas las aplicaciones GTK también hacen uso de main.

```
int main( int   argc,
          char *argv[] )
{
```

El siguiente trozo de programa declara punteros a estructuras de tipo `GtkWidget`. Serán usadas más abajo para crear una ventana y un botón

```
    GtkWidget *window;
    GtkWidget *button;
```

Aquí tenemos de nuevo a `gtk_init()`. Como antes, esto inicializa el toolkit, y parsea los argumentos que encuentre en la línea de comandos. Cualquier argumento que reconozca en la línea de comandos será eliminado de la lista y de argc y argv, permitiendo a tu aplicación parsear el resto de argumentos.

```
    gtk_init (&argc, &argv);
```

Crear una nueva ventana es un proceso bastante directo. Se asigna memoria para la estructura `GtkWidget *window` de tal forma que apunte a una estructura válida. Se inicializa una nueva ventana, pero no se muestra hasta que se llame a

```
gtk_widget_show(window)
```

cerca del final del program.

```
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
```

Aquí tenemos dos ejemplos sobre cómo conectar un manejador de señal a un objeto, en este caso, una ventana. Aquí, las señales "delete_event" y "destroy" son capturadas y tratadas. La primera es emitida cuando usamos el gestor de ventanas para cerrar

la ventan, o cuando usamos la llamada gtk_widget_destroy() pasando el widget de la ventana como el objeto a destruir. La segunda es emitida cuando, en el manejador "delete_event", devolvemos el valor FALSE. G_OBJECT y G_CALLBACK son macros que permiten realizar la conversión y comprobación de tipos de forma sencilla, además de facilitar la legibilidad del código.

```
g_signal_connect (G_OBJECT (window), "delete_event",
                  G_CALLBACK (delete_event), NULL);
g_signal_connect (G_OBJECT (window), "destroy",
                  G_CALLBACK (destroy), NULL);
```

La siguiente función se usa para asignar un atributo a un objeto contenedor. Lo que hacemos es ajustar la ventana para que tenga un área blanca alrededor de su perímetro de 10 pixels de ancho, donde no podrá ir ningún otro widget. Existen otras funciones similares que estudiaremos más adelante.

De nuevo, GTK_CONTAINER es una macro que permite la conversión de tipos.

```
gtk_container_set_border_width (GTK_CONTAINER (window), 10);
```

La siguiente llamada crea un nuevo botón. Asigna espacio para una nueva estructura GtkWidget en memoria, la inicializa, y apunta el puntero button hacia él. Tendrá la etiqueta "Hello World" cuando sea mostrado.

```
button = gtk_button_new_with_label ("Hello World");
```

Aquí haremos que el botón tenga alguna utilidad. Le asignaremos un manejador de señal de tal forma que cuando emita el evento "clicked", se llame a nuestra función hello(). El parámetro data se ignora, por lo que pasamos simplemente un NULL en esa posición a la función de conexión hello(). Obviamente, la señal "clicked" se emitirá cuando se haga click sobre el botón con el ratón.

```
g_signal_connect (G_OBJECT (button), "clicked",
                  G_CALLBACK (hello), NULL);
```

También usaremos este botón para salir de nuestro programa. Esto ilustrará lo indicado antes sobre la señal "destroy": puede venir tanto del gestor de ventanas como de nuestro propio programa. Cuando se pulse el botón, al igual que antes, se llama primero a la función hello() y posteriormente a la función gtk_widget_destroy() en el mismo orden en el que se hayan definido las funciones callback. Puedes tener tantas funciones callback como desees, y todas será ejecutadas en el orden en el que fueron conectadas. Dado que la funcion gtk_widget_destroy() acepta únicamente un GtkWidget *widget como argumento, usaremos la función g_signal_connect_swapped() en lugar de g_signal_connect().

```
g_signal_connect_swapped (G_OBJECT (button), "clicked",
                          G_CALLBACK (gtk_widget_destroy),
                          G_OBJECT (window));
```

La siguiente es una llamada a una función de empaquetamiento, que serán explicadas en el siguiente capítulo. Pero es bastante sencillo de entender. Símplemente indica a GTK que el botón debe situarse en la ventana en la que será mostrado. Obsérvese que un contenedor GTK únicamente puede contener un widget. Existen otros widgets, descritos más adelante, diseñados para albergar múltiples widgets en distintas posiciones.

```
gtk_container_add (GTK_CONTAINER (window), button);
```

Ya tenemos todo dispuesto de la forma que queríamos. Con todos los manejadores de señal inicializados y el botón insertado en la ventana correctamente, sólo nos queda indicar a GTK que muestre los widgets en pantalla. El widget de la ventana se muestra el último, de tal forma que toda la ventana con todo su contenido salga al final "a escena", para evitar mostrar primero la ventana y después ver que el botón se coloca encima. Aunque con un ejemplo tan sencillo, nunca se notaría ningún efecto "raro".

```
gtk_widget_show (button);

gtk_widget_show (window);
```

Por supuesto, queda finalmente llamar a `gtk_main()` que se ejecurará en bucle, esperando eventos que vengan del servidor X y llamando a los widgets para que emitan señales cuando lleguen estos eventos.

```
gtk_main ();
```

Para terminar el programa, se debe devolver 0 si todo ha ido correctamente, cuando se devuelva el control tras una llamada `gtk_quit()`.

```
return 0;
```

Ahora, cuando hagamos click con el botón del ratón en un botón GTK, el widget emitirá una señal "clicked". Para poder usar esta información, nuestro programa ha inicializado un manejador de señal para tratar este evento, que ejecutará la función que hayamos especificado. En nuestro ejemplo, cuando el botón creado sea pulsado, se llamará a a la función `hello()` con NULL como argumento; posteriormente, se ejecutará el siguiente manejador para esta señal. Esto llamará a `gtk_widget_destroy()`, pasándole el widget de la ventana como argumento, destruyéndolo. Esto provoca que la ventana emita la señal "destroy", que será interceptada, y llamará a nuestra función callback `destroy()`, que símplemente saldrá de GTK.

Otro posible camino de ejecución podría ser el usar el gestor de ventanas para cerrar la ventana, lo que provocará que se emita la señal "delete_event". Ésta será interceptada por nuestro manejador "delete_event". Si aquí devolvemos TRUE, la ventana seguirá como si nada hubiera ocurrido. Devolver FALSE provocará que GTK emita la señal "destroy" que por supuesto, llamará al callback "destroy", terminando la aplicación GTK.

### Cómo compilar el ejemplo

Para compilar los ejemplos del capítulo sobre GTK 2.0, podemos usar la siguiente orden:

```
gcc `pkg-config --cflags --libs gtk+-2.0` hello-world.c -o hello-world
```

## Contenedores

GTK utiliza los contenedores para colocar los widgets de una forma determinada. Cuando se desarrolla una aplicación, normalmente se necesita colocar más de un widget dentro de una ventana. En el ejemplo anterior de helloworld se usa sólo un

contenedor (gtk_container_add (GTK_CONTAINER (window), button);), para colocar el botón dentro de la ventana donde será mostrado. Pero, Â¿qué pasa si se quiere usar más de un widget dentro de una ventana?, Â¿cómo se puede controlar la posición de los widgets?

En otros sistemas gráficos (como por ejemplo MS Windows), la colocación de los widgets dentro de las ventanas se hace por medio de coordenadas relativas. Esto hace necesario un nivel de detalle a la hora de diseñar las ventanas de las aplicaciones que lo hacen indeseable. En GTK+, al igual que en todos los toolkits gráficos provinientes del mundo UNIX (Motif, QT, GTK, AWT de Java), está basado en el modelo de contenedores, donde no es necesario el uso de coordenadas. Simplemente se crean distintos tipos de contenedores (cada uno de los cuales coloca los widgets dentro de si mismo de una forma determinada) para cada caso concreto, y simplemente se colocan widgets dentro de dichos contenedores. La forma en que se metan los widgets dentro del contenedor define cómo se comportarán dichos widgets cuando la ventana contenedora cambie de tamaño.

## Cajas

Existen varios tipos de contenedores que se irán explicando a lo largo de este capítulo, pero los widgets `GtkHBox` y `GtkVBox` son los más usados. `GtkHBox` y `GtkVBox` son cajas invisibles que sirven para empaquetar los widgets que se vayan a colocar dentro de una ventana u otro contenedor. Cuando se empaquetan widgets en una caja horizontal (`GtkHBox`) se insertan horizontalmente de izquierda a derecha o de derecha a izquierda, dependiendo de la función que se utilice después. En una caja vertical (`GtkVBox`) se insertan de arriba a abajo o vice versa. También se puede usar una combinación de cajas dentro o al lado de otras cajas para crear el efecto deseado.

Para crear una caja horizontal se hace una llamada a la función `gtk_hbox_new`(), y para cajas verticales, `gtk_vbox_new`(). Las funciones `gtk_box_pack_start`() y `gtk_box_pack_end` se usan para colocar los widgets dentro de las cajas creadas. La función `gtk_box_pack_start` coloca los widgets de arriba a abajo en una caja vertical y de izquierda a derecha en una horizontal, mientras que `gtk_box_pack_end`() hace lo contrario, que es colocar los widgets de abajo a arriba en una caja vertical y de derecha a izquierda en una horizontal. Usando estas funciones será posible alinear los widgets a la derecha o izquierda de la caja según se desee.

Usando estas funciones, GTK sabe en qué posición colocar los widgets y así poder cambiar el tamaño de los mismos automáticamente, cuando se cambia el tamaño del contenedor. Además cuentan con varias opciones para poder cambiar el estilo de colocación de los widgets. En la siguiente figura se muestran los 5 estilos diferentes de colocación.

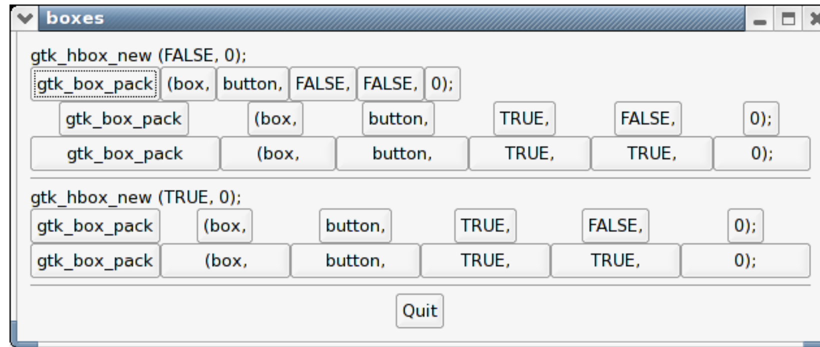



**Figura 7-1. Cinco estilos diferentes de colocación**

Cada línea contiene una caja horizontal (hbox) con varios botones. La llamada a la función gtk_box_pack es para colocar los botones en la caja horizontal.

Las dos funciones para añadir widgets a las cajas tienen la siguiente forma:

```
void  gtk_box_pack_start (GtkBox *box, GtkWidget *child, gboolean
expand, gboolean fill, guint padding);
```

```
void  gtk_box_pack_end (GtkBox *box, GtkWidget *child, gboolean
expand, gboolean fill, guint padding);
```

El primer argumento se refiere a la caja en la que se va a colocar el objeto, el segundo argumento es el objeto. Los objetos en este ejemplo son los botones, así que se colocarán los botones dentro de las cajas, pero puede ser cualquier otro widget.

El argumento *expand* controla que los botones se extiendan hasta rellenar todo el espacio dentro de la caja (TRUE), o que la caja se ajuste al tamaño de los botones (FALSE). Con *expand* con una valor de FALSE se pueden alinear los botones a la izquierda o a la derecha.

El argumento *fill* de las funciones `gtk_box_pack` controla si el espacio extra se coloca en los botones (TRUE), o como espacio extra entre cada botón (FALSE). Esto sólo tiene validez si el argumento *expand* está a TRUE.

El argumento *padding* controla el espacio añadido a cada lado del botón. En la siguiente figura se puede ver mejor el resultado.

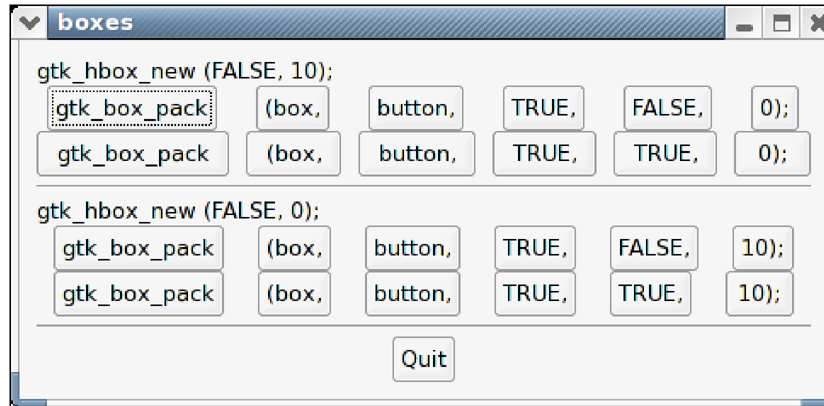



**Figura 7-2. Diferencias entre padding y spacing**

Para crear una caja, las funciones tienen el siguiente formato:

```
GtkWidget *  gtk_hbox_new (gboolean homogenous, gint spacing);
```

```
GtkWidget *  gtk_vbox_new (gboolean homogenous, gint spacing);
```

El argumento `homogeneous` controla si cada botón dentro de la caja tiene el mismo tamaño (la misma anchura en una hbox y la misma altura en una vbox). Si este argumento está a TRUE, las funciones de gtk_box_pack() funcionan como si el argumento `expand` estuviera siempre a TRUE.

El argumento `spacing` controla el espacio añadido entre los botones. La figura anterior muestra el resultado y la diferencia con el argumento padding de las funciones `gtk_box_pack`().

## Ejemplo de uso de cajas

El código que se muestra a continuación es el usado para crear las figuras mostradas anteriormente.

```
#include <stdio.h>
#include <stdlib.h>
#include "gtk/gtk.h"

gint delete_event( GtkWidget *widget,
                   GdkEvent  *event,
                   gpointer   data )
{
        gtk_main_quit ();
        return FALSE;
}

/* Crea una hbox con botones con etiquetas.
 * Se muestran todos los widgets (gtk_widget_show()) a excepción de la caja. */
GtkWidget *make_box( gboolean homogeneous,
                     gint     spacing,
                     gboolean expand,
                     gboolean fill,
                     guint    padding )
```

```
{
        GtkWidget *box;
        GtkWidget *button;
        char padstr[80];

        /* Crea una hbox con los parámetros establecidos
         * para homogeneous y spacing */
        box = gtk_hbox_new (homogeneous, spacing);

        /* Crea una serie de botones con etiqueta */
        button = gtk_button_new_with_label ("gtk_box_pack");
        gtk_box_pack_start (GTK_BOX (box), button, expand, fill, padding);
        gtk_widget_show (button);

        button = gtk_button_new_with_label ("(box,");
        gtk_box_pack_start (GTK_BOX (box), button, expand, fill, padding);
        gtk_widget_show (button);

        button = gtk_button_new_with_label ("button,");
        gtk_box_pack_start (GTK_BOX (box), button, expand, fill, padding);
        gtk_widget_show (button);

        /* Crea un botón con etiqueta dependiendo del valor del parámetro
         * expand. */
        if (expand == TRUE)
                button = gtk_button_new_with_label ("TRUE,");
        else
                button = gtk_button_new_with_label ("FALSE,");

        gtk_box_pack_start (GTK_BOX (box), button, expand, fill, padding);
        gtk_widget_show (button);

        /* Lo mismo con el botón de "expand"
         * pero esta vez de la forma abreviada. */
        button = gtk_button_new_with_label (fill ? "TRUE," : "FALSE,");
        gtk_box_pack_start (GTK_BOX (box), button, expand, fill, padding);
        gtk_widget_show (button);

        sprintf (padstr, "%d);", padding);

        button = gtk_button_new_with_label (padstr);
        gtk_box_pack_start (GTK_BOX (box), button, expand, fill, padding);
        gtk_widget_show (button);

        return box;
}

int main( int   argc,
          char *argv[])
{
        GtkWidget *window;
        GtkWidget *button;
        GtkWidget *box1;
        GtkWidget *box2;
        GtkWidget *separator;
        GtkWidget *label;
        GtkWidget *quitbox;
        int which;


        gtk_init (&argc, &argv);

        if (argc != 2) {
                fprintf (stderr, "usage: packbox num, where num is 1, 2, or 3.\n");
                /* Esto finaliza GTK y sale con un estatus de 1. */
                exit (1);
```

```
}

which = atoi (argv[1]);

/* Crea una ventana */
window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

/* Simpre se debe conectar la señal delete_event a la ventana
 * principal, para así poder cerrarla.  */
g_signal_connect (G_OBJECT (window), "delete_event",
                  G_CALLBACK (delete_event), NULL);
gtk_container_set_border_width (GTK_CONTAINER (window), 10);

/* Crea una caja vertical (vbox) para colocar las cajas horizontales dentro.
 * Esto permite colocar las cajas horizontales llenas de botones una
 * encima de la otra dentro de la caja vertical (vbox). */
box1 = gtk_vbox_new (FALSE, 0);

/* Muestra una de las Figuras de arriba. */
switch (which) {
case 1:
        /* crea una etiqueta. */
        label = gtk_label_new ("gtk_hbox_new (FALSE, 0);");

        /* Alínea la etiqueta a la izquierda.  Se hablará más adelante de esta
         * función. */
        gtk_misc_set_alignment (GTK_MISC (label), 0, 0);

        /* Coloca la etiqueta dentro de la caja vertical (vbox box1).  Los
         * widgets que se añaden a una vbox se colocan, por orden, uno
         * encima del otro. */
        gtk_box_pack_start (GTK_BOX (box1), label, FALSE, FALSE, 0);

        /* Muestra la etiqueta */
        gtk_widget_show (label);

        /* Llamada a la función make box, para crear un hbox con
         * una serie de botones - homogeneous = FALSE, spacing = 0,
         * expand = FALSE, fill = FALSE, padding = 0 */
        box2 = make_box (FALSE, 0, FALSE, FALSE, 0);
        gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, FALSE, 0);
        gtk_widget_show (box2);

        /* Llamada a la función make box, para crear otra hbox con
         * una serie de botones - homogeneous = FALSE, spacing = 0,
         * expand = TRUE, fill = FALSE, padding = 0 */
        box2 = make_box (FALSE, 0, TRUE, FALSE, 0);
        gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, FALSE, 0);
        gtk_widget_show (box2);

        /* Los argumentos son: homogeneous, spacing, expand, fill, padding */
        box2 = make_box (FALSE, 0, TRUE, TRUE, 0);
        gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, FALSE, 0);
        gtk_widget_show (box2);

        /* Crea un separador, se verán más adelante,
         * aunque son bastante sencillos. */
        separator = gtk_hseparator_new ();

        /* Coloca el separador en una vbox. Todos estos
         * widgets han sido colocados en una vbox, así que estarán
         * ordenados verticalmente. */
        gtk_box_pack_start (GTK_BOX (box1), separator, FALSE, TRUE, 5);
        gtk_widget_show (separator);

        /* Crea otra etiqueta y la muestra. */
```

```
            label = gtk_label_new ("gtk_hbox_new (TRUE, 0);");
            gtk_misc_set_alignment (GTK_MISC (label), 0, 0);
            gtk_box_pack_start (GTK_BOX (box1), label, FALSE, FALSE, 0);
            gtk_widget_show (label);

            /* Los argumentos son: homogeneous, spacing, expand, fill, padding */
            box2 = make_box (TRUE, 0, TRUE, FALSE, 0);
            gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, FALSE, 0);
            gtk_widget_show (box2);

            /* Los argumentos son: homogeneous, spacing, expand, fill, padding */
            box2 = make_box (TRUE, 0, TRUE, TRUE, 0);
            gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, FALSE, 0);
            gtk_widget_show (box2);

            /* Otro separador. */
            separator = gtk_hseparator_new ();
            /* Los tres últimos argumentos de gtk_box_pack_start son:
             * expand, fill, padding. */
            gtk_box_pack_start (GTK_BOX (box1), separator, FALSE, TRUE, 5);
            gtk_widget_show (separator);

            break;

        case 2:

            /* Crea una etiqueta, box1 es una
             * vbox que ya ha sido creada    */
            label = gtk_label_new ("gtk_hbox_new (FALSE, 10);");
            gtk_misc_set_alignment (GTK_MISC (label), 0, 0);
            gtk_box_pack_start (GTK_BOX (box1), label, FALSE, FALSE, 0);
            gtk_widget_show (label);

            /* Los argumentos son: homogeneous, spacing, expand, fill, padding */
            box2 = make_box (FALSE, 10, TRUE, FALSE, 0);
            gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, FALSE, 0);
            gtk_widget_show (box2);

            /* Los argumentos son: homogeneous, spacing, expand, fill, padding */
            box2 = make_box (FALSE, 10, TRUE, TRUE, 0);
            gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, FALSE, 0);
            gtk_widget_show (box2);

            separator = gtk_hseparator_new ();
            /* Los tres últimos argumentos de gtk_box_pack_start son:
             * expand, fill, padding. */
            gtk_box_pack_start (GTK_BOX (box1), separator, FALSE, TRUE, 5);
            gtk_widget_show (separator);

            label = gtk_label_new ("gtk_hbox_new (FALSE, 0);");
            gtk_misc_set_alignment (GTK_MISC (label), 0, 0);
            gtk_box_pack_start (GTK_BOX (box1), label, FALSE, FALSE, 0);
            gtk_widget_show (label);

            /* Los argumentos son: homogeneous, spacing, expand, fill, padding */
            box2 = make_box (FALSE, 0, TRUE, FALSE, 10);
            gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, FALSE, 0);
            gtk_widget_show (box2);

            /* Los argumentos son: homogeneous, spacing, expand, fill, padding */
            box2 = make_box (FALSE, 0, TRUE, TRUE, 10);
            gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, FALSE, 0);
            gtk_widget_show (box2);

            separator = gtk_hseparator_new ();
            /* Los tres últimos argumentos de gtk_box_pack_start son: expand, fill,
```

```
        gtk_box_pack_start (GTK_BOX (box1), separator, FALSE, TRUE, 5);
        gtk_widget_show (separator);
        break;

case 3:

        /* Esto demuestra la capacidad de gtk_box_pack_end() para
         * alinear a la derecha los widgets. Primero, se crea una caja. */
        box2 = make_box (FALSE, 0, FALSE, FALSE, 0);

        /* Crea una etiqueta colocada al final. */
        label = gtk_label_new ("end");
        /* Coloca la etiqueta usando gtk_box_pack_end(), así que se situa a la
         * derecha de la hbox creada con la llamada a make_box(). */
        gtk_box_pack_end (GTK_BOX (box2), label, FALSE, FALSE, 0);
        /* Show the label. */
        gtk_widget_show (label);

        /* Coloca box2 dentro de box1 (vbox) */
        gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, FALSE, 0);
        gtk_widget_show (box2);

        /* Crea un separador. */
        separator = gtk_hseparator_new ();
        /* Establece las medidas del separador, una anchura de 400 pixels por 5
         * de altura. La hbox que se creó anteriormente también tendrá 400 pixe
         * anchura,y la etiqueta "end" se separa de las otras etiquetas en la
         * hbox. */
        gtk_widget_set_size_request (separator, 400, 5);
        /* coloca el separador en la vbox (box1) */
        gtk_box_pack_start (GTK_BOX (box1), separator, FALSE, TRUE, 5);
        gtk_widget_show (separator);
}

/* Crea otra hbox.. (se pueden crear tantas cajas como se necesiten) */
quitbox = gtk_hbox_new (FALSE, 0);

/* Crea el botón para salir del programa. */
button = gtk_button_new_with_label ("Quit");

/* Establece la señal para terminar el programa cundo se pulsa el botón ("quit"
g_signal_connect_swapped (G_OBJECT (button), "clicked",
                          G_CALLBACK (gtk_main_quit),
                          G_OBJECT (window));
/* Coloca el botón en la hbox (quitbox).
 * Los 3 últimos argumentos de gtk_box_pack_start son:
 * expand, fill, padding. */
gtk_box_pack_start (GTK_BOX (quitbox), button, TRUE, FALSE, 0);
/* Coloca la hbox (quitbox) en la vbox (box1) */
gtk_box_pack_start (GTK_BOX (box1), quitbox, FALSE, FALSE, 0);

/* Coloca la vbox (box1), la cual contiene ahora todos los widgets, en la
 * ventana principal. */
gtk_container_add (GTK_CONTAINER (window), box1);

/* Se muestra todo lo restante */
gtk_widget_show (button);
gtk_widget_show (quitbox);

gtk_widget_show (box1);
/* Se muestra la ventana en último lugar, así todo se muestra a la vez. */
gtk_widget_show (window);

/* La función principal. */
gtk_main ();
```

```
        /* El control vuelve aquí cuando se llama a gtk_main_quit(), pero no cuando
         * se usa exit(). */

        return 0;
}
```

## Cajas de botones

Uno de los usos más comunes que se le dan a las cajas citadas en el apartado anterior es para agrupar botones. Al ser esta una tarea muy común, GTK incluye las cajas de botones (Gtk?ButtonBox).

Las cajas de botones son una utilidad que permite crear grupos de botones de una forma rápida y sencilla. Estas cajas de botones pueden ser horizontales o verticales.

Las siguientes funciones son para crear una caja de botones horizontal y vertical.

```
GtkWidget *gtk_hbutton_box_new( void );

GtkWidget *gtk_vbutton_box_new( void );
```

Para añadir los botones a la caja de botones, se usa la siguiente función:

```
gtk_container_add (GTK_CONTAINER (button_box), child_widget);
```

El siguiente ejemplo muestra las diferentes layout settings de las cajas de botones.

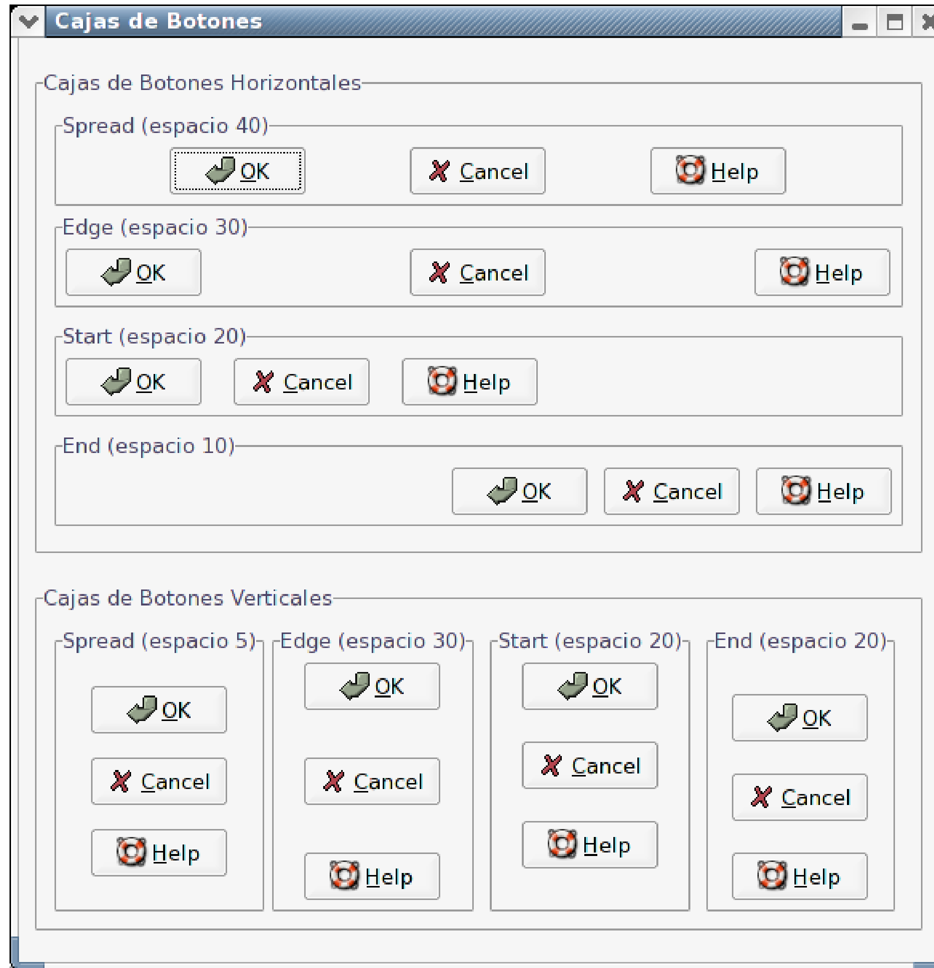



**Figura 7-3. Diferentes layout settings de las cajas de botones**

```
#include <gtk/gtk.h>

/* Crea una Caja de Botones con los parámetros específicos */
GtkWidget *create_bbox( gint  horizontal,
                        char *title,
                        gint  espacio,
                        gint  child_w,
                        gint  child_h,
                        gint  layout )
{
  GtkWidget *frame;
  GtkWidget *bbox;
  GtkWidget *button;

  frame = gtk_frame_new (title);

  if (horizontal)
    bbox = gtk_hbutton_box_new ();
  else
    bbox = gtk_vbutton_box_new ();

  gtk_container_set_border_width (GTK_CONTAINER (bbox), 5);
  gtk_container_add (GTK_CONTAINER (frame), bbox);
```

```
  /* Establece la apariencia de la Caja de Botones */
  gtk_button_box_set_layout (GTK_BUTTON_BOX (bbox), layout);
  gtk_box_set_spacing (GTK_BOX (bbox), espacio);
  /*gtk_button_box_set_child_size (GTK_BUTTON_BOX (bbox), child_w, child_h);*/

  button = gtk_button_new_from_stock (GTK_STOCK_OK);
  gtk_container_add (GTK_CONTAINER (bbox), button);

  button = gtk_button_new_from_stock (GTK_STOCK_CANCEL);
  gtk_container_add (GTK_CONTAINER (bbox), button);

  button = gtk_button_new_from_stock (GTK_STOCK_HELP);
  gtk_container_add (GTK_CONTAINER (bbox), button);

  return frame;
}

int main( int   argc,
          char *argv[] )
{
  static GtkWidget* window = NULL;
  GtkWidget *main_vbox;
  GtkWidget *vbox;
  GtkWidget *hbox;
  GtkWidget *frame_horz;
  GtkWidget *frame_vert;

  /* Inicializa GTK */
  gtk_init (&argc, &argv);

  window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  gtk_window_set_title (GTK_WINDOW (window), "Cajas de Botones");

  g_signal_connect (G_OBJECT (window), "destroy",
                    G_CALLBACK (gtk_main_quit),
                    NULL);

  gtk_container_set_border_width (GTK_CONTAINER (window), 10);

  main_vbox = gtk_vbox_new (FALSE, 0);
  gtk_container_add (GTK_CONTAINER (window), main_vbox);

  frame_horz = gtk_frame_new ("Cajas de Botones Horizontales");
  gtk_box_pack_start (GTK_BOX (main_vbox), frame_horz, TRUE, TRUE, 10);

  vbox = gtk_vbox_new (FALSE, 0);
  gtk_container_set_border_width (GTK_CONTAINER (vbox), 10);
  gtk_container_add (GTK_CONTAINER (frame_horz), vbox);

  gtk_box_pack_start (GTK_BOX (vbox),
         create_bbox (TRUE, "Spread (espacio 40)", 40, 85, 20, GTK_BUTTONBOX_SPREAD),
                    TRUE, TRUE, 0);

  gtk_box_pack_start (GTK_BOX (vbox),
         create_bbox (TRUE, "Edge (espacio 30)", 30, 85, 20, GTK_BUTTONBOX_EDGE),
                    TRUE, TRUE, 5);

  gtk_box_pack_start (GTK_BOX (vbox),
         create_bbox (TRUE, "Start (espacio 20)", 20, 85, 20, GTK_BUTTONBOX_START),
                    TRUE, TRUE, 5);

  gtk_box_pack_start (GTK_BOX (vbox),
         create_bbox (TRUE, "End (espacio 10)", 10, 85, 20, GTK_BUTTONBOX_END),
                    TRUE, TRUE, 5);

  frame_vert = gtk_frame_new ("Cajas de Botones Verticales");
```

```
  gtk_box_pack_start (GTK_BOX (main_vbox), frame_vert, TRUE, TRUE, 10);

  hbox = gtk_hbox_new (FALSE, 0);
  gtk_container_set_border_width (GTK_CONTAINER (hbox), 10);
  gtk_container_add (GTK_CONTAINER (frame_vert), hbox);

  gtk_box_pack_start (GTK_BOX (hbox),
           create_bbox (FALSE, "Spread (espacio 5)", 5, 85, 20, GTK_BUTTONBOX_SPREAD),
                      TRUE, TRUE, 0);

  gtk_box_pack_start (GTK_BOX (hbox),
           create_bbox (FALSE, "Edge (espacio 30)", 30, 85, 20, GTK_BUTTONBOX_EDGE),
                      TRUE, TRUE, 5);

  gtk_box_pack_start (GTK_BOX (hbox),
           create_bbox (FALSE, "Start (espacio 20)", 20, 85, 20, GTK_BUTTONBOX_START),
                      TRUE, TRUE, 5);

  gtk_box_pack_start (GTK_BOX (hbox),
           create_bbox (FALSE, "End (espacio 20)", 20, 85, 20, GTK_BUTTONBOX_END),
                      TRUE, TRUE, 5);

  gtk_widget_show_all (window);


  gtk_main ();

  return 0;
}
```

## Tablas

Otro tipo de contenedor son las tablas, muy útiles en algunas ocasiones.

Usando tablas, se crea una rejilla en la que se pueden colocar widgets. Los widgets pueden ocupar los espacios que se especifiquen (1 o más celdas).

La función para crear tablas es `gtk_table_new`, y tiene la siguiente forma:

```
GtkWidget *  gtk_table_new (guint rows, guint columns, gboolean
homogeneous);
```

El primer argumento es el número de filas de la tabla, el segundo, obviamente, el número de columnas. El argumento *homogeneous* especifica las medidas de las cajas de la tabla. Si su valor es TRUE, las cajas de la tabla se ajustan al tamaño del widget más largo que esté en la tabla. Por contra, si vale FALSE, las cajas de la tabla se ajustan al tamaño del widget más alto de la fila y el más ancho de la columna.

Las filas y columnas empiezan de 0 a n, siendo “n” el número especificado en la llamada a `gtk_table_new`(). Si se especifica por ejemplo, 2 filas (rows = 2) y 2 columnas (columns = 2), la estructura quedaría como se ve en la siguiente imagen:

```
 0          1          2
0+----------+----------+
 |          |          |
1+----------+----------+
 |          |          |
2+----------+----------+
```

Hay que tener en cuenta que el sistema de coordenadas comienza en la esquina superior izquierda (0,0).

Para colocar widgets dentro de la tabla, se usa la función `gtk_table_attach`, que tiene la siguiente forma:

```
void  gtk_table_attach (GtkTable * table, GtkWidget * child,
guint left_attach, guint right_attach, guint top_attach, guint
bottom_attach, GtkAttachOptions xoptions, GtkAttachOptions yoptions,
guint xpadding, guint ypadding);
```

El primer argumento es la tabla que se ha creado y el segundo el widget que se va a colocar en la tabla. Los argumentos *left_attach* y *right_attach* especifican dónde colocar el widget, y cuántas celdas usar. Si, por ejemplo, se quiere posicionar un botón en la parte inferior derecha de la tabla anterior de 2x2 y rellenar únicamente esa entrada, los valores de los argumentos serían los siguientes: left_attach = 1, right_attach = 2, top_attach = 1, bottom_attach = 2.

Ahora bien, si se quiere posicionar un widget que ocupe la fila superior entera de la tabla de 2x2, los valores de los argumentos serían los siguientes: left_attach = 0, right_attach = 2, top_attach = 0, bottom_attach = 1.

Los argumentos *xoptions* y *yoptions* especifican las opciones de colocación en forma de máscara de bits, donde los valores existentes para el tipo GtkAttachOptions pueden ser agrupados mediante el operador binario OR (|). Estas opciones son:

- GTK_FILL: Si la caja de la tabla es más larga que el widget, y se ha especificado GTK_FILL, el widget se extenderá hasta ocupar todo el sitio que ocupa la caja.
- GTK_SHRINK: Cuando se reduce el tamaño de la tabla, los widgets, normalmente, no cambian su tamaño junto con el de la tabla, de forma que las partes inferior y derecha de dichos widgets desaparezca a medida que se reduce el tamaño de la tabla. Usando GTK_SHRINK los widgets reducirán su tamaño junto con el de la tabla.
- GTK_EXPAND: La tabla se extiende hasta ocupar todo el espacio de la ventana.

El argumento *padding* funciona de la misma forma que con cajas, es decir, especifica el espacio, en pixels, alrededor del widget.

Al tener tantas opciones la función `gtk_table_attach`, GTK ofrece una función extra que permite añadir widgets a la tabla usando los valores por defecto de colocación. Dicha función es `gtk_table_attach_defaults`, cuya sintaxis es la siguiente:

```
void  gtk_table_attach_defaults (GtkTable * table, GtkWidget * child,
guint left_attach, guint right_attach, guint top_attach, guint
bottom_attach);
```

Como puede apreciarse, los parámetros para especificar las opciones de colocación y de espaciado alrededor del widget han sido omitidos en esta función con respecto a `gtk_table_attach`. Eso se debe a que `gtk_table_attach_defaults` toma unos valores por defecto para esos parámetros, que son *GTK_FILL | GTK_EXPAND* en el caso de *xoptions* y *yoptions*, y 0 en el caso de *xpadding* e *ypadding*. El resto de argumentos son idénticos a los comentados para la función `gtk_table_attach`.

Las funciones `gtk_table_set_row_spacing()` y `gtk_table_set_col_spacing()`, añaden espacio entre las filas y las columnas.

```
void gtk_table_set_row_spacing(GtkTable * table, guint row, guint
spacing);
```

```
void gtk_table_set_col_spacing(GtkTable * table, guint column, guint
spacing);
```

Hay que tener en cuenta que el espacio se sitúa a la derecha de la columna, y en las filas, se sitúa debajo de la fila.

También es posible establecer un espacio para las filas y columnas a la vez, con las siguientes funciones:

```
void gtk_table_set_row_spacings(GtkTable * table, guint spacing);
```

```
void gtk_table_set_col_spacings(GtkTable * table, guint spacing);
```

Con estas funciones, no se deja espacio en la última fila y la última columna.

## Ejemplo de uso de tablas

A continuación se muestra el código para crear una ventana con tres botones en una tabla de 2x2. Los primeros dos botones se colocan en la fila superior de la tabla. El tercero, el botón "quit", se coloca en la fila inferior, ocupando las dos columnas.

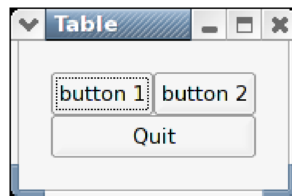



**Figura 7-4. Tabla con 3 botones**

```
#include <gtk/gtk.h>

/* Los datos pasados a esta función se imprimen en la salida estándar (stdout) */
void callback( GtkWidget *widget,
               gpointer   data )
{
    g_print ("Hello again - %s was pressed\n", (char *) data);
}

/* Función para terminar el programa */
gint delete_event( GtkWidget *widget,
                   GdkEvent  *event,
                   gpointer   data )
{
    gtk_main_quit ();
    return FALSE;
}

int main( int   argc,
          char *argv[] )
```

```
{
    GtkWidget *window;
    GtkWidget *button;
    GtkWidget *table;

    gtk_init (&argc, &argv);

    /* Crea una ventana */
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

    /* Establece el título de la ventana */
    gtk_window_set_title (GTK_WINDOW (window), "Table");

    /* Establece un manejador para "delete_event" que inmediatamente
     * cerrará GTK. */
    g_signal_connect (G_OBJECT (window), "delete_event",
                      G_CALLBACK (delete_event), NULL);

    /* Establece el tamaño del borde de la ventana. */
    gtk_container_set_border_width (GTK_CONTAINER (window), 20);

    /* Crea una tabla de 2x2 */
    table = gtk_table_new (2, 2, TRUE);

    /* Coloca la tabla en la ventana principal */
    gtk_container_add (GTK_CONTAINER (window), table);

    /* Crea el primer botón */
    button = gtk_button_new_with_label ("button 1");

    /* cuando se pulsa el botón, se llama a la función que se ha creado anteriormente
     * "callback", con un puntero a "button 1" como argumento */
    g_signal_connect (G_OBJECT (button), "clicked",
                      G_CALLBACK (callback), (gpointer) "button 1");


    /* Coloca el primer botón (button 1) en el cuadrante superior izquierdo de la tabla
    gtk_table_attach_defaults (GTK_TABLE (table), button, 0, 1, 0, 1);

    gtk_widget_show (button);

    /* Crea el segundo botón */

    button = gtk_button_new_with_label ("button 2");

    /* cuando se pulsa el botón, se llama a la función que se ha creado anteriormente
     * "callback", con un puntero a "button 2" como argumento */
    g_signal_connect (G_OBJECT (button), "clicked",
                      G_CALLBACK (callback), (gpointer) "button 2");
    /* Coloca el segundo botón (button 2) en el cuadrante superior derecho de la tabla.
    gtk_table_attach_defaults (GTK_TABLE (table), button, 1, 2, 0, 1);

    gtk_widget_show (button);

    /* Crea el botón de "Quit" */
    button = gtk_button_new_with_label ("Quit");

    /* Cuando se pulsa el botón, se hace una llamada a función "delete_event" y el
     * programa termina. */
    g_signal_connect (G_OBJECT (button), "clicked",
                      G_CALLBACK (delete_event), NULL);

    /* Coloca el botón "quit" ocupando los dos cuadrantes inferiores de la tabla.*/
    gtk_table_attach_defaults (GTK_TABLE (table), button, 0, 2, 1, 2);

    gtk_widget_show (button);
```

```
    gtk_widget_show (table);
    gtk_widget_show (window);

    gtk_main ();

    return 0;
}
```

## GtkNotebook

El widget `GtkNotebook` es una colección de "páginas" que se solapan entre ellas. Cada página, sólo visible una cada vez, contiene una información (widgets) determinada. Este widget ha sido cada vez más común en la programación de interfaces gráficos, ya que es una buena forma de mostrar bloques de similar información uno a uno.

La función que se necesita para la creación de un widget notebook es la siguiente:

```
GtkWidget * gtk_notebook_new(void);
```

GTK también tiene varias funciones para establecer el formato del widget notebook. La primera de estas funciones sirve para posicionar los indicadores de página o pestañas, los cuales pueden ser colocados arriba, abajo, a derecha o izquierda.

```
void gtk_notebook_set_tab_pos(GtkNotebook * notebook, GtkPositionType
pos);
```

Los diferentes tipos de la variable de tipo GtkPositionType son los siguientes:

- GTK_POS_LEFT (posición a la izquierda)
- GTK_POS_RIGHT (posición a la derecha)
- GTK_POS_TOP (posición arriba)
- GTK_POS_BOTTOM (posición abajo)
- GTK_POS_TOP es la opción por defecto.

El siguiente paso sería añadir páginas al notebook. Existes tres formas de añadir páginas. Las dos primeras que se muestran a continuación, y son muy similares.

```
void gtk_notebook_append_page(GtkNotebook * notebook, GtkWidget *
child, GtkWidget * tab_label);
```

```
void gtk_notebook_prepend_page(GtkNotebook * notebook, GtkWidget *
child, GtkWidget * tab_label);
```

La primera función (`gtk_notebook_append_page`), añade las páginas al final del notebook y la segunda (`gtk_botebook_prepend_page`), las añade al principio. El argumento *`child`* es el widget que será colocado dentro de la página del notebook, y *`tab_label`* es la etiqueta de la página que será añadida. El widget *`child`* deberá ser creado por separado y normalmente suele ser un contenedor que contiene otros widgets, aunque, por supuesto, puede ser cualquier widget (cualquier `GtkWidget`).

La última función para añadir páginas al notebook contiene todas las propiedades de las dos anteriores, pero además permite especificar en qué posición se quiere colocar

la pagína dentro del notebook. Es decir, permite insertar una página antes de otra ya existente en el notebook.

```
void gtk_notebook_insert_page(GtkNotebook * notebook, GtkWidget *
child, GtkWidget * tab_label, gint position);
```

El parámetro extra (`position`) se usa para especificar la posición en la que se desea insertar la nueva página. Hay que tener en cuenta que 0 representa la primera página.

Para borrar una página del notebook, se utiliza la siguiente función:

```
void gtk_notebook_remove_page(GtkNotebook * notebook, gint page_num);
```

Esta función borra del notebook, la página que se haya especificado en el parámetro `page_num`.

Para saber en que página se encuentra el notebook se utiliza la siguiente función:

```
gint gtk_notebook_get_current_page(GtkNotebook * notebook);
```

Esta función devuelve la posición de la página actual dentro del notebook, siendo 0 la primera página.

Las suguientes funciones sirven para mover las hojas del notebook hacia adelante o hacia atrás. Si el notebook está en la última página y se llama a la función `gtk_notebook_next_page` éste volverá a la primera página. Igualmente si se llama a la función `gtk_notebook_prev_page` y notebook está en la primera página, éste volverá a la última página.

```
void gtk_notebook_next_page(GtkNoteBook * notebook);
```

```
void gtk_notebook_prev_page(GtkNoteBook * notebook);
```

La siguiente función sirve para activar una página determinada del notebook. Si por ejemplo se necesita activar la página 5 del notebook, se usaría esta función. Si no se usa esta función, el notebook se sitúa siempre en la primera página al ser creada ésta y las siguientes que se vayan añadiendo.

```
void gtk_notebook_set_current_page(GtkNotebook * notebook, gint
page_num);
```

Las siguientes funciones sirven para mostrar u ocultar los indicadores o pestañas y los bordes del notebook respectivamente.

```
void gtk_notebook_set_show_tabs(GtkNotebook * notebook, gboolean
show_tabs);
```

```
void gtk_notebook_set_show_border(GtkNotebook * notebook, gboolean
show_border);
```

Ambas funciones reciben un valor de tipo boolean que especifica si se debe (TRUE) o no (FALSE) mostrar los indicadores o los bordes.

La siguiente función es muy útil cuando se tiene un gran número de páginas y las pestañas no caben en la página. La función crea dos botones con flechas, para así permitir al usuario el ir recorriendo las múltiples pestañas.

```
void gtk_notebook_set_scrollable(GtkNotebook * notebook, gboolean
scrollable);
```

El siguiente ejemplo se ha sacado del fichero `testgtk.c` que viene junto con la distribución de GTK. Este pequeño programa, crea una ventana con un notebook que contiene seis botones. El notebook contiene 11 páginas, que son añadidas de tres maneras diferentes (append, insert y prepend). Los botones permiten cambiar la posición de la pestaña, borrar o añadir los bordes y pestañas del notebook y cambiar a la anterior o siguiente página.

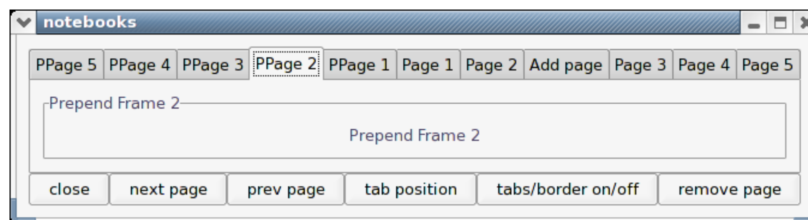



**Figura 7-5. GtkNotebook en acción**

```
#include <stdio.h>
#include <gtk/gtk.h>

/* Esta función va cambiando la posición de las pestañas
This function rotates the position of the tabs */
void rotate_book( GtkButton   *button,
                  GtkNotebook *notebook )
{
    gtk_notebook_set_tab_pos (notebook, (notebook->tab_pos + 1) % 4);
}

/* Añade o borra las pestañas y los bordes del notebook */
void tabsborder_book( GtkButton   *button,
                      GtkNotebook *notebook )
{
    gint tval = FALSE;
    gint bval = FALSE;
    if (notebook->show_tabs == 0)
            tval = TRUE;
    if (notebook->show_border == 0)
            bval = TRUE;

    gtk_notebook_set_show_tabs (notebook, tval);
    gtk_notebook_set_show_border (notebook, bval);
}

/* Borra una página del notebook */
void remove_book( GtkButton   *button,
                  GtkNotebook *notebook )
{
    gint page;

    page = gtk_notebook_get_current_page (notebook);
    gtk_notebook_remove_page (notebook, page);
```

```
    /* Con la siguiente función se vuelve a dibujar el notebook
     * para que no se vea la página que ha sido borrada. */
    gtk_widget_queue_draw (GTK_WIDGET (notebook));
}

gint delete( GtkWidget *widget,
             GtkWidget *event,
             gpointer   data )
{
    gtk_main_quit ();
    return FALSE;
}

int main( int argc,
          char *argv[] )
{
    GtkWidget *window;
    GtkWidget *button;
    GtkWidget *table;
    GtkWidget *notebook;
    GtkWidget *frame;
    GtkWidget *label;
    GtkWidget *checkbutton;
    int i;
    char bufferf[32];
    char bufferl[32];

    gtk_init (&argc, &argv);

    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

    g_signal_connect (G_OBJECT (window), "delete_event",
                      G_CALLBACK (delete), NULL);

    gtk_container_set_border_width (GTK_CONTAINER (window), 10);

    table = gtk_table_new (3, 6, FALSE);
    gtk_container_add (GTK_CONTAINER (window), table);

    /* Crea un notebook, y especifica la posición de las pestañas */
    notebook = gtk_notebook_new ();
    gtk_notebook_set_tab_pos (GTK_NOTEBOOK (notebook), GTK_POS_TOP);
    gtk_table_attach_defaults (GTK_TABLE (table), notebook, 0, 6, 0, 1);
    gtk_widget_show (notebook);

    /* El siguiente bucle, añade 5 páginas al notebook (append)*/
    for (i = 0; i < 5; i++) {
        sprintf(bufferf, "Append Frame %d", i + 1);
        sprintf(bufferl, "Page %d", i + 1);

        frame = gtk_frame_new (bufferf);
        gtk_container_set_border_width (GTK_CONTAINER (frame), 10);
        gtk_widget_set_size_request (frame, 100, 75);
        gtk_widget_show (frame);

        label = gtk_label_new (bufferf);
        gtk_container_add (GTK_CONTAINER (frame), label);
        gtk_widget_show (label);

        label = gtk_label_new (bufferl);
        gtk_notebook_append_page (GTK_NOTEBOOK (notebook), frame, label);
    }

    /* Se añade un página para Now let's add a page to a specific spot */
    checkbutton = gtk_check_button_new_with_label ("Check me please!");
    gtk_widget_set_size_request (checkbutton, 100, 75);
```

```
gtk_widget_show (checkbutton);

label = gtk_label_new ("Add page");
gtk_notebook_insert_page (GTK_NOTEBOOK (notebook), checkbutton, label, 2);

/* El siguiente bucle, añade 5 páginas al notebook (prepend)*/
for (i = 0; i < 5; i++) {
    sprintf (bufferf, "Prepend Frame %d", i + 1);
    sprintf (bufferl, "PPage %d", i + 1);

    frame = gtk_frame_new (bufferf);
    gtk_container_set_border_width (GTK_CONTAINER (frame), 10);
    gtk_widget_set_size_request (frame, 100, 75);
    gtk_widget_show (frame);

    label = gtk_label_new (bufferf);
    gtk_container_add (GTK_CONTAINER (frame), label);
    gtk_widget_show (label);

    label = gtk_label_new (bufferl);
    gtk_notebook_prepend_page (GTK_NOTEBOOK (notebook), frame, label);
}

/* Con esta función, se mostrará una página específica "page 4" */
gtk_notebook_set_current_page (GTK_NOTEBOOK (notebook), 3);

/* Crea los botones */
button = gtk_button_new_with_label ("close");
g_signal_connect_swapped (G_OBJECT (button), "clicked",
                          G_CALLBACK (delete), NULL);
gtk_table_attach_defaults (GTK_TABLE (table), button, 0, 1, 1, 2);
gtk_widget_show (button);

button = gtk_button_new_with_label ("next page");
g_signal_connect_swapped (G_OBJECT (button), "clicked",
                          G_CALLBACK (gtk_notebook_next_page),
                          G_OBJECT (notebook));
gtk_table_attach_defaults (GTK_TABLE (table), button, 1, 2, 1, 2);
gtk_widget_show (button);

button = gtk_button_new_with_label ("prev page");
g_signal_connect_swapped (G_OBJECT (button), "clicked",
                          G_CALLBACK (gtk_notebook_prev_page),
                          G_OBJECT (notebook));
gtk_table_attach_defaults (GTK_TABLE (table), button, 2, 3, 1, 2);
gtk_widget_show (button);

button = gtk_button_new_with_label ("tab position");
g_signal_connect (G_OBJECT (button), "clicked",
                  G_CALLBACK (rotate_book),
                  (gpointer) notebook);
gtk_table_attach_defaults (GTK_TABLE (table), button, 3, 4, 1, 2);
gtk_widget_show (button);

button = gtk_button_new_with_label ("tabs/border on/off");
g_signal_connect (G_OBJECT (button), "clicked",
                  G_CALLBACK (tabsborder_book),
                  (gpointer) notebook);
gtk_table_attach_defaults (GTK_TABLE (table), button, 4, 5, 1, 2);
gtk_widget_show (button);

button = gtk_button_new_with_label ("remove page");
g_signal_connect (G_OBJECT (button), "clicked",
                  G_CALLBACK (remove_book),
                  (gpointer) notebook);
gtk_table_attach_defaults (GTK_TABLE (table), button, 5, 6, 1, 2);
```

```
    gtk_widget_show (button);

    gtk_widget_show (table);
    gtk_widget_show (window);

    gtk_main ();

    return 0;
}
```

## GtkAlignment

El widget `GtkAlignment` permite colocar widgets en una posición y tamaño relativos al tamaño de sí mismo. Esto puede ser muy útil, por ejemplo, a la hora de centrar un widget en una ventana.

Sólo hay dos funciones asociadas a este widget, que son:

```
GtkWidget * gtk_alignment_new(gfloat xalign, gfloat yalign, gfloat
xscale, gfloat yscale);
```

```
void gtk_alignment_set(GtkAlignment * alignment, gfloat xalign, gfloat
yalign, gfloat xscale, gfloat yscale);
```

La primera función crea un nuevo widget `GtkAlignment` con los parámetros especificados. La segunda función permite cambiar los parámetros de un widget `GtkAlignment` ya existente.

Los cuatro parámetros de estas funciones son de tipo gfloat con un valor que puede ir de 0.0 a 1.0. Los argumentos *`xalign`* e *`yalign`* determinan la posición del widget dentro del `GtkAlignment`. Los argumentos *`xscale`* e *`yscale`* determinan la cantidad de espacio asignado al widget.

También se puede añadir un widget al `GtkAlignment` usando la siguiente función (perteneciente a la clase `GtkContainer`, de la que deriva `GtkAlignment`):

```
gtk_container_add (GTK_CONTAINER (alignment), child_widget);
```

Se puede ver un ejemplo del uso del widget GtkAlignment en el programa de ejemplo del uso del widget de barra de progreso (GtkProgressBar).

## GtkHPaned/GtkVPaned

Los widgets "paned" se utilizan cuando se quiere dividir un área en dos partes, con el tamaño de las dos partes controlado por el usuario. Un separador, con un manejador que el usuario puede arrastrar y cambiar el ratio, separa las dos partes. La división puede ser horizontal (GtkHPaned) o vertical (GtkVPaned).

Para crear una ventana "paned", horizontal o vertical, se necesita una de las siguientes funciones:

```
GtkWidget *gtk_hpaned_new (void);

GtkWidget *gtk_vpaned_new (void);
```

Después de crear una ventana "paned", se necesita añadir widgets a las dos mitades. Para ello GTK cuenta con las siguientes funciones:

```
void gtk_paned_add1 (GtkPaned *paned, GtkWidget *child);

void gtk_paned_add2 (GtkPaned *paned, GtkWidget *child);
```

La función `gtk_paned_add1` añade el widget hijo a la izquierda o arriba de una de las partes de la ventana "paned", mientras que la función `gtk_paned_add2` añade el widget hijo a la derecha o abajo de una de las partes de la ventana "paned".

El siguiente ejemplo crea parte de un interfaz de usuario de un programa de correo imaginario. La ventana se divide en dos partes verticales. La parte de arriba muestra una lista de los mensajes recibidos y la parte de abajo el texto de los mensajes. Hay que tener dos cosas en cuenta: no se puede añadir texto a un widget de texto hasta no indicarlo explícitamente a GTK ("realize"). Ésto puede hacerse llamando a la `gtk_widget_realize()`, pero como demostración de una técnica alternativa, conectaremos un manejador a la señal "realize" para añadir texto al widget. Además, necesitaremos añadir la opción GTK_SHRINK a algunos elementos de la tabla que contiene la ventana de texto y sus barras de scroll, de tal forma que cuando la parte inferior se haga pequeña, se encoja también la porción adecuada en lugar de salir por debajo de la ventana.

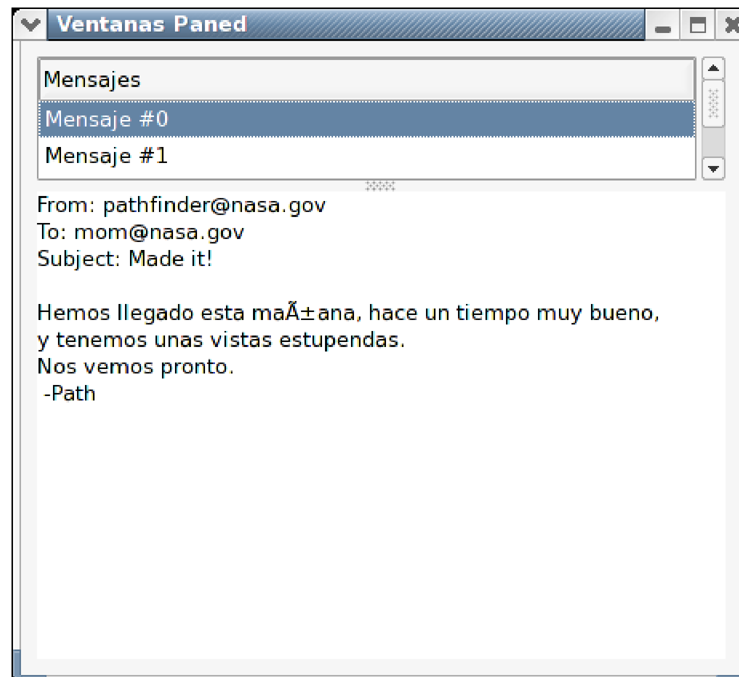



**Figura 7-6. Ejemplo de Paned Window**

## GtkLayout

El contenedor `GtkLayout` es similar al contenedor `GtkFixed`, salvo que el primero implementa un área de scroll muy amplia (hasta 2^32). El sistema X window tiene una limitación ya que las ventanas sólo pueden ser de 32767 pixels tanto de ancho como de largo. El contenedor `GtkLayout` salva esta limitación pudiendo tener un amplio área de scroll, incluso cuando se tienen muchos widgets hijos dentro de este área.

Para crear un contenedor Layout se utiliza la siguiente función:

```
GtkWidget *gtk_layout_new( GtkAdjustment *hadjustment,
                           GtkAdjustment *vadjustment );
```

Como puede verse, se puede, al crear un `GtkLayout`, especificar los `GtkAdjustment`, tanto horizontal como vertical, que serán usados para controlar el desplazamiento del contenido del contenedor. Dichos parámetros son opcionales, por lo que en la mayor parte de los casos se usará esta función especificando NULL para ambos.

Las siguientes funciones sirven para añadir y mover widgets en el contenedor `GtkLayout`:

```
void gtk_layout_put( GtkLayout *layout,
                     GtkWidget *widget,
                            gint       x,
                            gint       y );

void gtk_layout_move( GtkLayout *layout,
                             GtkWidget *widget,
                             gint       x,
                             gint       y );
```

Para establecer el tamaño de un `GtkLayout` se usa la siguiente función:

```
void gtk_layout_set_size( GtkLayout *layout,
                                 guint      width,
                                 guint      height );
```

Y para terminar con el `GtkLayout`, las cuatro funciones siguientes se usan para manipular el alineamiento horizontal y vertical de los widgets.

```
GtkAdjustment* gtk_layout_get_hadjustment( GtkLayout *layout );

GtkAdjustment* gtk_layout_get_vadjustment( GtkLayout *layout );

void gtk_layout_set_hadjustment( GtkLayout     *layout,
                                 GtkAdjustment *adjustment );

void gtk_layout_set_vadjustment( GtkLayout     *layout,
                                 GtkAdjustment *adjustment);
```

## Colocación por coordenadas

Una de las cosas que hacen especial a GTK+ (al igual que a Java Swing y otros toolkits), es que la colocación de los widgets en las ventanas se hace por medio de contenedores, como se comenta en esta sección, que controlan la forma en que son colocados los distintos widgets controlados por el contenedor.

Sin embargo, hay ocasiones en las que puede ser necesario el usar una colocación de los distintos widgets en coordenadas exactas. Para dicho menester, GTK incluye el widget `GtkFixed`, que permite colocar los widgets en una posición fija dentro de la ventana, relativa a la esquina superior izquierda de la misma. La posición de los widgets se puede cambiar dinámicamente.

Las funciones asociadas a este widget son las siguientes:

```
GtkWidget* gtk_fixed_new( void );

void gtk_fixed_put( GtkFixed  *fixed,
                    GtkWidget *widget,
                    gint       x,
                    gint       y );

void gtk_fixed_move( GtkFixed  *fixed,
                            GtkWidget *widget,
                            gint       x,
                            gint       y );
```

La primera función, `gtk_fixed_new` permite crear un nuevo widget `GtkFixed`

La función `gtk_fixed_put` coloca el widget en el contenedor `GtkFixed` en la posición especificada por los parámetros *`x`* e *`y`*.

`gtk_fixed_move` permite mover el widget especificado en el parámetro *`widget`* a una nueva posición.

```
void gtk_fixed_set_has_window( GtkFixed  *fixed,
                               gboolean   has_window );

gboolean gtk_fixed_get_has_window( GtkFixed *fixed );
```

En las últimas versiones de GTK, los widgets `GtkFixed` no tenían su propia ventana X, pero ahora la función `gtk_fixed_set_has_window` permite crearlos con su propia ventana. No obstante, esta función tendrá que ser llamada antes de que se visualice el widget.

El siguiente ejemplo muestra cómo usar el contenedor `GtkFixed`.

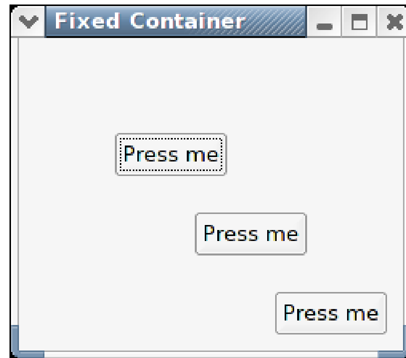



**Figura 7-7. Ejemplo de GtkFixed**

```
#include <gtk/gtk.h>

/* Para agilizar el trabajo, se usarán algunas variables globales para
 * guardar la posición del widget dentro del contenedor fixed */
gint x = 50;
gint y = 50;

/* Esta función mueve el botón a una nueva posición
 * del contenedor Fixed. */
void move_button( GtkWidget *widget,
                  GtkWidget *fixed )
{
  x = (x + 30) % 300;
  y = (y + 50) % 300;
  gtk_fixed_move (GTK_FIXED (fixed), widget, x, y);
}

int main( int   argc,
          char *argv[] )
{
  /* GtkWidget es el tipo utilizado para widgets */
  GtkWidget *window;
  GtkWidget *fixed;
  GtkWidget *button;
  gint i;

  /* Inicializa GTK */
  gtk_init (&argc, &argv);

  /* Crea una ventana */
  window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  gtk_window_set_title (GTK_WINDOW (window), "Fixed Container");

  /* Conecta el evento "destroy" a un manejador de señales */
  g_signal_connect (G_OBJECT (window), "destroy",
                    G_CALLBACK (gtk_main_quit), NULL);

  /* Establece la anchura del borde de la ventana. */
  gtk_container_set_border_width (GTK_CONTAINER (window), 10);

  /* Crea un Contenedor Fixed */
  fixed = gtk_fixed_new ();
  gtk_container_add (GTK_CONTAINER (window), fixed);
  gtk_widget_show (fixed);

  for (i = 1 ; i <= 3 ; i++) {
    /* Crea un botón con la etiqueta "Press me" */
```

```
    button = gtk_button_new_with_label ("Press me");

    /* Cuando el botón recive la señal "clicked", se llama a la función
     * move_button() pasando el Contenedor Fixed como argumento */
    g_signal_connect (G_OBJECT (button), "clicked",
                      G_CALLBACK (move_button), (gpointer) fixed);

    /* Coloca el botón dentro de la ventana de los contenedores fixed. */
    gtk_fixed_put (GTK_FIXED (fixed), button, i*50, i*50);

    /* El último paso es mostrar el nuevo widget que se ha creado. */
    gtk_widget_show (button);
  }

  /* Muestra la ventana */
  gtk_widget_show (window);

  /* Entra en el bucle de eventos */
  gtk_main ();

  return 0;
}
```

## Marcos

### GtkAspectFrame

`GtkAspectFrame` es igual que el widget `GtkFrame`, salvo que aquél además especifica el ratio entre la anchura y la altura del widget hijo, para así poder tener un valor determinado y añadir espacio extra si fuera necesario. Esto es muy útil cuando se quiere visualizar una imagen y ampliarla o reducirla segun cambie el tamaño de la ventana contenedora. El tamaño de la imagen debe cambiar cuando el usuario maximice o minimice la ventana, pero el ratio siempre deberá coincidir con la imagen original.

La siguiente función crea un `GtkAspectFrame` nuevo.

```
GtkWidget * gtk_aspect_frame_new(const gchar * label, gfloat xalign,
gfloat yalign, gfloat ratio, gboolean obey_child);
```

Los argumentos *xalign* e *yalign* establecen la posición al igual que se comentó con el widget definido en la sección de nombre *GtkAlignment*. Si el valor del argumento *obey_child* es TRUE, el ratio del widget hijo coincidirá con el ratio del tamaño necesario. En caso contrario, el valor del ratio vendrá dado por el valor del argumento del mismo nombre.

La siguiente función cambia las opciones de un `GtkAspectFrame`

```
void gtk_aspect_frame_set(GtkAspectFrame * aspect_frame, gfloat
xalign, gfloat yalign, gfloat ratio, gboolean obey_child);
```

El siguiente programa usa un `GtkAspectFrame` para presentar un área de dibujo cuyo ratio será siempre 2:1, independientemente que el usuario cambie el tamaño de la ventana principal.

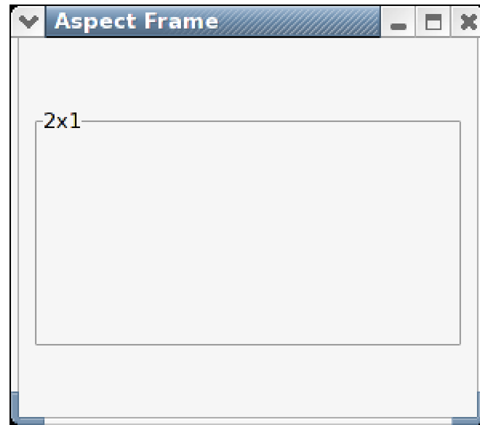



**Figura 7-8. GtkAspectFrame: al redimensionar la ventana, el ratio se mantiene**

```
#include <gtk/gtk.h>

int main( int argc,
          char *argv[] )
{
    GtkWidget *window;
    GtkWidget *aspect_frame;
    GtkWidget *drawing_area;
    gtk_init (&argc, &argv);

    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title (GTK_WINDOW (window), "Aspect Frame");
    g_signal_connect (G_OBJECT (window), "destroy",
                      G_CALLBACK (gtk_main_quit), NULL);
    gtk_container_set_border_width (GTK_CONTAINER (window), 10);

    /* Crea un widget aspect_frame y lo añade a la ventana principal */

    aspect_frame = gtk_aspect_frame_new ("2x1", /* etiqueta */
                                         0.5, /* centra x */
                                         0.5, /* centra y */
                                         2, /* el ratio: xsize/ysize = 2 */
                                         FALSE /* este argumento se ignora */);

    gtk_container_add (GTK_CONTAINER (window), aspect_frame);
    gtk_widget_show (aspect_frame);

    /* Se añade un widget child al widget aspect frame */

    drawing_area = gtk_drawing_area_new ();

    /* el tamaño de la ventana se establece a 200x200, aunque el widget AspectFrame
     * establece el tamaño a 200x100 ya que el ratio es de 2x1 */
    gtk_widget_set_size_request (drawing_area, 200, 200);
    gtk_container_add (GTK_CONTAINER (aspect_frame), drawing_area);
    gtk_widget_show (drawing_area);

    gtk_widget_show (window);
    gtk_main ();
    return 0;
}
```

### GtkViewport

El widget `GtkViewport` se suele usar muy poco directamente. Normalmente se usan las ventanas con scroll, las cuales usan el widget `Viewport`.

`GtkViewport` permite la colocación de un widget mayor dentro de él, de tal forma que es posible visualizar todo el widget por partes. Se usa la alineación para definir el área que se va a ver.

La siguiente función crea un widget Viewport:

```
GtkWidget *gtk_viewport_new( GtkAdjustment *hadjustment,
                             GtkAdjustment *vadjustment );
```

Como se puede ver, se puede especificar la alineación horizontal o vertical que va a usar el widget. Si el valor de los argumentos es NULL, creará la alineación internamente.

Con las siguientes funciones se puede obtener y cambiar la alineación de un widget que ya ha sido creado anteriormente.

```
GtkAdjustment *gtk_viewport_get_hadjustment (GtkViewport *viewport );

GtkAdjustment *gtk_viewport_get_vadjustment (GtkViewport *viewport );

void gtk_viewport_set_hadjustment( GtkViewport   *viewport,
                                   GtkAdjustment *adjustment );

void gtk_viewport_set_vadjustment( GtkViewport   *viewport,
                                   GtkAdjustment *adjustment );
```

Por último, se usará la siguiente función para cambiar la apariencia del widget viewport.

```
void gtk_viewport_set_shadow_type( GtkViewport   *viewport,
                                   GtkShadowType  type );
```

Los posibles valores para el parámetro *`type`* son:

- GTK_SHADOW_NONE
- GTK_SHADOW_IN
- GTK_SHADOW_OUT
- GTK_SHADOW_ETCHED_IN
- GTK_SHADOW_ETCHED_OUT

## Ventanas

Las ventanas son la parte básica de todo interfaz, pues es el widget sobre el que se muestra el interfaz de la aplicación al usuario.

Las ventanas, todas ellas, derivan de la clase `GtkWindow`, que a su vez deriva de `GtkContainer`. Por tanto, se puede decir que las ventanas son un caso de contenedores especial, pues son contenedores que no están contenidos en ningún otro contenedor, como si que ocurre con el resto de contenedores comentados en el apartado anterior.

### Ventanas (GtkWindow)

GtkWindow es el objeto ventana de nivel más alto, puede contener a otros widgets.

`gtk_window_new()` crear un nuevo objeto `GtkWindow`, que normalmente será una ventana del nivel más alto (el tipo de la ventana debería de ser GTK_WINDOW_TOPLEVEL). Si estás implementando algo como un menu pop-up desde cero (una mala idea, sería mejor que usar GtkMenu), podrías usar el parámetro GTK_WINDOW_POPUP. GTK_WINDOW_POPUP no está diseñado para cuadros de diálogo, aunque en otros toolkits llaman "popups" a los cuadros de diálogo. En `GTK+`, GTK_WINDOW_POPUP significa un menú pop-up o una ventanita tooltip. En `X11`, las ventanas popup no se controlan por el gestor de ventanas.

Si lo que quieres símplemente es quitar la decoración de una ventana (ventanas sin bordes), usa la función `gtk_window_set_decorated()`, no uses GTK_WINDOW_POPUP.

### Diálogos (GtkDialog)

Los diálogos son ventanas temporales, que se usan para obtener o mostrar determinada información del/al usuario. Habitualmente son ventanas que estan activas durante un espacio de tiempo limitado, el que tarda el usuario en actuar ante lo que se le indica en el cuadro de diálogo. Ejemplos típicos de diálogos son una ventana para confirmar una acción con el usuario, para mostrar un mensaje de error, etc.

#### Creación de diálogos

En GTK+, la creación de diálogos se hace con `GtkDialog`, que contiene dos funciones para la creación de los mismos:

```
GtkWidget* gtk_dialog_new (void);
GtkWidget* gtk_dialog_new_with_buttons (const gchar *title,
                                         GtkWindow *parent,
                                         GtkDialogFlags flags,
                                       const gchar *first_button_text,
                                         ...);
```

La primera función crea un diálogo completamente vacío, sin botones, y sin ningun contenido. La segunda, nos permita especificar distintas propiedades del diálogo en el momento de su creación.

Con `gtk_dialog_new`, la creación del diálogo consta de dos pasos. Uno es la creación del diálogo vacío, lo cual se realiza con una llamada a `gtk_dialog_new`, mientras que el otro consiste en añadir los botones al mismo. Un ejemplo típico de esto sería:

```
GtkWidget *dialog;

dialog = gtk_dialog_new ();
gtk_dialog_add_button (GTK_DIALOG (dialog), "Cerrar", GTK_RESPONSE_CLOSE);
```

Esta nueva función, `gtk_dialog_add_button`, añade un nuevo botón al diálogo, en su parte inferior derecha. A cada botón es necesario asociarle tanto una cadena de texto ("Cerrar" en el ejemplo anterior), que será el texto que se muestre dentro del botón, y un código de respuesta, que permite identificar el botón pulsado, como se verá más adelante. Estos códigos de respuesta son denominados "responses" (respuestas), y se incluyen unos cuantos por defecto, tales como GTK_RESPONSE_CLOSE, aunque, por supuesto, se pueden definir nuevos códigos.

Como se comentaba anteriormente, `gtk_dialog_new_with_button` permite hacer todo lo anteriormente expuesto en un solo paso. Es decir, crear el diálogo con botones directamente, sin necesidad del paso adicional. Aparte de especificar la lista de botones a crear en el diálogo, esta función permite especificar el título de la ventana y la ventana padre del diálogo, así como distintas opciones de creación del diálogo. Así, se podría crear el mismo diálogo explicado anteriormente con el siguiente código:

```
GtkWidget *dialog;
GtkWindow *parent_window;

dialog = gtk_dialog_new_with_buttons ("Título del diálogo",
                                      parent_window,
                                      0,
                                      "Cerrar", GTK_RESPONSE_CLOSE,
                                      NULL);
```

`gtk_dialog_new_with_buttons` usa una lista variable de argumentos. Los primeros son fijos, mientras que los últimos especifican la lista de botones a crear, todos ellos identificados con dos parámetros, como en el caso anterior: texto a mostrar en el botón y código de respuesta. El último parámetro debe ser siempre NULL, que marca el fin de la lista de botones.

Se podría usar esta misma función para crear varios botones de la siguiente forma:

```
GtkWidget *dialog;
GtkWindow *parent_window;

dialog = gtk_dialog_new_with_buttons ("Título del diálogo",
                                      parent_window,
                                      0,
                                      "Aceptar", GTK_RESPONSE_OK,
                                      "Cancelar", GTK_RESPONSE_CANCEL,
                                      NULL);
```

Esto crearía un diálogo con dos botones. Por supuesto, se puede extender este ejemplo y añadir cuantos botones sean necesarios, siempre y cuando cada uno de ellos esté correctamente identificado (texto y código de respuesta asociados) y que el último parámetro sea NULL.

Una vez está creado el diálogo, es momento de añadirle contenido. Para ello, se usa un widget interno de la `GtkDialog`, que no es más que un `GtkBox` en el que se pueden añadir los distintos widgets que forman el diálogo.

`GTK+` trata las cajas de diálogo como una ventana dividida verticalmente en dos partes. La sección superior es una GtkVBox, y es donde se pueden colocar widgets como `GtkLabel` o `GtkEntry`. El área inferior es conocida como action_area. Se usa normalmente para empaquetar botones como "cancelar", "ok" o "aplicar" en el diálogo. Las dos áreas están separadas por un widget `GtkHSeparator`.

Se puede acceder a las partes superior e inferior de un nuevo 'dialogo' mediante GTK_DIALOG(dialog)->vbox y GTK_DIALOG(dialog)->action_area, como se puede ver en el ejemplo siguiente. También se pueden crear diálogos modales (es

decir, un diálogo que bloquea el resto de la aplicación hasta que el usuario responda a lo que se le pide en el diálogo) mediante una llamada a `gtk_window_set_modal()` en el propio diálogo. Se debe usar la macro `GTK_WINDOW()` para convertir el tipo del widget devuelto por `gtk_dialog_new()` a una `GtkWindow`. Si se usa `gtk_dialog_new_with_buttons()` también se puede pasar el flag GTK_DIALOG_MODAL como argumento para construir un diálogo modal.

```
#include "gtk/gtk.h"

gint delete_event( GtkWidget *widget,
                   GdkEvent  *event,
                   gpointer   data )
{
        gtk_main_quit ();
        return FALSE;
}


/* Función para abrir una caja de diálogo que muestre un mensaje dado como parámetro. *

void quick_message (gchar *message, GtkWidget *parent) {

   GtkWidget *dialog, *label;

   /* Create the widgets */

   dialog = gtk_dialog_new_with_buttons ("Título del cuadro de diálogo",
                                         GTK_WINDOW (parent),
                                         GTK_DIALOG_DESTROY_WITH_PARENT,
                                         GTK_STOCK_OK,
                                         GTK_RESPONSE_NONE,
                                         NULL);
   label = gtk_label_new (message);

   /* Ensure that the dialog box is destroyed when the user responds. */

   g_signal_connect_swapped (GTK_OBJECT (dialog),
                             "response",
                             G_CALLBACK (gtk_main_quit),
                             GTK_OBJECT (dialog));

   /* Add the label, and show everything we've added to the dialog. */

   gtk_container_add (GTK_CONTAINER (GTK_DIALOG(dialog)->vbox),
                      label);
   gtk_widget_show_all (dialog);
}



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

        gtk_init (&argc, &argv);

        /* Crea una ventana */
        window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

        /* Siempre se debe conectar la señal delete_event a la ventana
         * principal, para así poder cerrarla.  */
        g_signal_connect (G_OBJECT (window), "delete_event",
                          G_CALLBACK (delete_event), NULL);
        gtk_container_set_border_width (GTK_CONTAINER (window), 10);
```

```
        quick_message("Mensaje en una botella.", window);

        /* La función principal. */
        gtk_main ();

        /* El control vuelve aquí cuando se llama a gtk_main_quit(), pero no cuando
         * se usa exit(). */

        return 0;
}
```

## Ventanas de mensaje (GtkMessageDialog)

GtkMessageDialog muestra un diálogo con una imagen que representa el tipo de mensaje (Error, Pregunta, etc.) además de algún texto aclaratorio. Es símplemente un widget para facilitar el trabajo; se puede construir la ventana equivalente a la ofrecida por GtkMessageDialog sin mucho esfuerzo, pero GtkMesageDialog nos ahorrará teclear código en balde. La forma más sencilla de crear un diálogo modal consiste en usar `gtk_dialog_run()`, aunque es posible hacerlo también pasando el parámetro GTK_DIALOG_MODAL, `gtk_dialog_run()` construirá automáticamente el diálogo modal y esperará a que el usuario responda a lo indicado. `gtk_dialog_run()` devolverá el control cuando el usuario pulse cualquiera de los botones del cuadro de diálogo.

Ejemplo de un cuadro de diálogo modal.

```
dialog = gtk_message_dialog_new (main_application_window,
                                  GTK_DIALOG_DESTROY_WITH_PARENT,
                                  GTK_MESSAGE_ERROR,
                                  GTK_BUTTONS_CLOSE,
                                  "Error loading file 's': s",
                                  filename, g_strerror (errno));
gtk_dialog_run (GTK_DIALOG (dialog));
gtk_widget_destroy (dialog);
```

También puedes construir un cuadro de diálogo no-modal `GtkMessageDialog` como sigue:

```
dialog = gtk_message_dialog_new (main_application_window,
                                  GTK_DIALOG_DESTROY_WITH_PARENT,
                                  GTK_MESSAGE_ERROR,
                                  GTK_BUTTONS_CLOSE,
                                  "Error loading file 's': s",
                                  filename, g_strerror (errno));

/* Destruir el diálogo cuando el usuario pulse un botón */
g_signal_connect_swapped (GTK_OBJECT (dialog), "response",
                          G_CALLBACK (gtk_widget_destroy),
                          GTK_OBJECT (dialog));
```

## Ventanas invisibles (GtkInvisible)

`GtkInvisible` es un widget usado internamente por `GTK+` que no será mostrado en pantalla. Probablemente, no sea útil para los desarrolladores de aplicaciones. Es usado por `GTK+` para la correcta gestión del puntero del ratón y gestión de selecciones en el código interno de arrastrar-y-soltar ("drag-and-drop").

La gestión del puntero del ratón se usa en operaciones que necesitan de un completo control sobre los eventos originados por el ratón, incluso si el puntero del ratón deja la aplicación (para pasar a otra). Por ejemplo en `GTK+` se usa para el control de Drag and Drop, para arrastrar el manejador en los widgets `GtkHPaned` y `GtkVPaned` y para cambiar el tamaño de las columnas en los widgets `GtkCList`.

Ofrece los siguientes métodos:

```
GtkWidget*  gtk_invisible_new            (void);
GtkWidget*  gtk_invisible_new_for_screen (GdkScreen *screen);
void        gtk_invisible_set_screen     (GtkInvisible *invisible,
                                          GdkScreen *screen);
GdkScreen*  gtk_invisible_get_screen     (GtkInvisible *invisible);
```

## Ventanas embebidas (GtkPlug)

Junto a `GtkSocket`, `GtkPlug` ofrece la posibilidad de incrustar widgets de un proceso en otro proceso diferente de una forma transparente al usuario. Un proceso creará un widget `GtkSocket` y pasará el ID de la ventana de ese widget a otro proceso, que posteriormente creará un `GtkPlug` tomando como parámetro el ID recogido. Cualquier widget contenido en el `GtkPlug` aparecerá entonces en la ventana de la primera aplicación.

GtkPlug ofrece los siguientes métodos:

```
void        gtk_plug_construct             (GtkPlug *plug,
                                            GdkNativeWindow socket_id);
void        gtk_plug_construct_for_display (GtkPlug *plug,
                                            GdkDisplay *display,
                                            GdkNativeWindow socket_id);
GtkWidget*  gtk_plug_new                   (GdkNativeWindow socket_id);
GtkWidget*  gtk_plug_new_for_display       (GdkDisplay *display,
                                            GdkNativeWindow socket_id);
GdkNativeWindow gtk_plug_get_id            (GtkPlug *plug);
```

El ID de ventana del socket se obtiene usando `gtk_socket_get_id()`. Antes de usar esta función, el socket debe haber sido puesto a la vista "realize", y por tanto, debe haber sido añadido a su padre.

Example 1. Cómo obtener el ID de ventana de un socket.

```
GtkWidget *socket = gtk_socket_new ();
gtk_widget_show (socket);
gtk_container_add (GTK_CONTAINER (parent), socket);

/* The following call is only necessary if one of
 * the ancestors of the socket is not yet visible.
 */
gtk_widget_realize (socket);
g_print ("The ID of the sockets window is %x\n",
         gtk_socket_get_id (socket));
```

Obsérvese que si se pasa el ID de ventana del socket a otro proceso que se conecte (plug) a dicho socket, debemos asegurarnos de que el widget socket no sea destruído hasta que se cree la conexión. La violación de esta regla puede causar efectos impredecibles, siendo el más probable que la ventana plug aparezca como una ventana separada.

Se puede comprobar que la conexión ha sido creada examinando el campo plug_window de la estructura `GtkSocket`. Si este campo es distinto a NULL, la conexión con el socket se habrá creado satisfactoriamente.

Cuando `GTK+` reciba la notificación de que la ventana incrustada ha sido destruída, destruirá el socket automáticamente. Por lo tanto, el programador debe estar preparado para tratar la destrucción del socket en cualquier momento dentro del bucle main principal.

La comunicación entre GtkSocket y GtkPlug sigue las normas impuestas por el protocolo XEmbed. Este protocolo ha sido así mismo implementado en otros toolkits de desarrollo como Qt, permitiendo el mismo nivel de integración al incrustar un widget Qt en GTK o viceversa.

Un socket también puede ser usado para incrustar cualquier ventana pre-existente de primer nivel usando la función `gtk_socket_steal()`, aunque el grado de integración cuando se realiza esta operación no es tan bueno como cuando se usa GtkPlug en conjunto con GtkSocket.

### Grupos de ventanas (GtkWindowGroup)

Un GtkWindowGroup se define para limitar el efecto del seguimiento del puntero del ratón entre ventanas. GtkWindowGroup deriva directamente de GObject y ofrece los siguientes métodos:

```
GtkWindowGroup* gtk_window_group_new        (void);
```

Crea un nuevo objeto `GtkWindowGroup`. Los seguimientos añadidos con `gtk_grab_add()` sólo afectan a las ventanas dentro del mismo `GtkWindowGroup`.

```
void        gtk_window_group_add_window     (GtkWindowGroup *window_group,
                                            GtkWindow *window);
void        gtk_window_group_remove_window  (GtkWindowGroup *window_group,
                                            GtkWindow *window);
```

## Visualización y Entrada de información

### Etiquetas

Las etiquetas son widgets que nos permiten mostrar información (texto) en cualquier parte de nuestro interfaz.

### GtkLabel

El tipo más básico de etiqueta que podemos crear es GtkLabel, que permite crear porciones de texto (incluido multilínea) en nuestro interfaz.

En GTK+, para crearlas, se usa una de las siguientes funciones:

```
GtkWidget*            gtk_label_new               (const char    *str);
GtkWidget*            gtk_label_new_with_mnemonic (const char    *str);
```

Ambas funciones reciben una cadena como parámetro, pero en cada una su significado es distinto. En `gtk_label_new`, la cadena de texto representa el texto que queremos que sea mostrado en la etiqueta, mientras que en `gtk_label_new_with_mnemonic` dicha cadena representa ese mismo texto a representar en pantalla, pero con una diferencia, que es que esa cadena puede contener el carácter '_' delante de algunas letras, lo que causa que dichas letras sean mostradas con el carácter de subrayado. Esto, más adelante veremos para qué puede servir.

### GtkAccelLabel

Los widgets `GtkAccelLabel` se usan principalmente en la construcción de menús. Muestran una etiqueta (como un widget `GtkLabel` normal) más una combinación de teclas a su derecha. Por ejemplo, en el siguiente trozo de código:

```
GtkAccelLabel* = gtk_accel_label_new("entrada");
gtk_accel_label_set_accel_widget(label, entry);
```

creamos un widget de tipo `GtkAccelLabel` que contendrá la etiqueta "entrada". En la segunda línea de código, indicamos que la etiqueta que acabamo de crear debe vigilar y mostrar constantemente el atajo de teclado que tenga definido el widget entry. Así, si en el código asignas el atajo ctrl+P como la señal de activación de entry, `GTK+` mostrará esa combinación de teclas en la etiqueta del widget `GtkAccelLabel`, al lado del texto "entrada". El `GtkAccelLabel` vigilará los cambios que se realicen en tiempo de ejecución en la combinación de teclas. Si cambiáramos el atajo de teclado por programación, AccelLabel se enteraría y mostraría el nuevo atajo al lado del texto "entrada". Para ver las diferencias existentes entre GtkAccelLabel y GtkLabel es interesante comparar el trozo de código anterior con el siguiente:

```
GtkLabel* label = gtk_label_new_with_mnemonic("_something");
gtk_label_set_mnemonic_widget(label, entry);
```

Este trozo de código crea una etiqueta con la letra s de "something" subrayada, lo que indica que GTK+ creará un atajo de teclado automáticamente con la combinación de teclas alt+s para el widget entry. Es decir, cuando pulsemos alt+s, se activará el widget entry. * FIXME * Debe haber una referencia que indique: para conocer un caso práctico del uso de gtk-accel-label, obsérvese el ejemplo gtk-accel-label-example * FIXME * A continuación, veremos un ejemplo de código que muestra el uso de gtk-accel-label y la creación manual de menús.

```
#include <gtk/gtk.h>

gint Delete (GtkWindow *widget, gpointer data) {
        gtk_main_quit();
        return FALSE;
}


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

        GtkWidget *window;      /* Top-level window */
```

```
GtkWidget *menubar;     /* To hold our sample "File" menu */
GtkWidget *menuitem;    /* Used for the "File","New","Open" MenuItems */
GtkWidget *submenu;     /* The actual "menu" holding "New", "Open" */

GtkAccelGroup *accel;   /* Our accel group */
GtkWidget *accel_label; /* Our GtkAccelLabel */

gtk_init(&argc, &argv);

/* Create top-level window */
window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
gtk_window_set_position(GTK_WINDOW (window), GTK_WIN_POS_CENTER_ALWAYS);
g_signal_connect(G_OBJECT (window), "delete_event",
         G_CALLBACK (Delete), NULL);

/* Create our accel group, attach it to our top-level window */
accel = gtk_accel_group_new();
gtk_window_add_accel_group(GTK_WINDOW (window), accel);
/* The accels are added down below... */

/* create the menu bar as the only widget in our "window" */
menubar = gtk_menu_bar_new();
gtk_container_add(GTK_CONTAINER(window), menubar);
gtk_widget_show(menubar);

/* Create the "File" GtkMenuItem */
menuitem = gtk_menu_item_new_with_label("File");
gtk_menu_bar_append(GTK_MENU_BAR(menubar), menuitem);
gtk_widget_show(menuitem);

/* Create the actual drop-down menu (called "submenu") */
submenu = gtk_menu_new();
/* set it to be the submenu of "File" */
gtk_menu_item_set_submenu(GTK_MENU_ITEM(menuitem), submenu);

/* Create "New" entry in our submenu: */
menuitem = gtk_menu_item_new_with_label("New");
gtk_menu_append(GTK_MENU(submenu), menuitem);

/* Finally, add an accelerator for "New" of CTRL-N */
gtk_widget_add_accelerator(menuitem,
"activate", accel, ’N’, GDK_CONTROL_MASK, GTK_ACCEL_VISIBLE);

/* Wrap up the "New" GtkMenuItem by showing it: */
gtk_widget_show(menuitem);


/* Create a second GtkMenuItem ("Open") */
/* Now with GtkAccelLabel... */
menuitem = gtk_menu_item_new();

accel_label = gtk_accel_label_new("Open");

/* We attach it to ’menuitem’, which is the GtkMenuItem */

gtk_accel_label_set_accel_widget(GTK_ACCEL_LABEL(accel_label), menuitem);
gtk_container_add(GTK_CONTAINER(menuitem), accel_label);
gtk_widget_show(accel_label);

/* All done--add this menuitem to the submenu and show it... */
gtk_menu_append(GTK_MENU(submenu), menuitem);
gtk_widget_show(menuitem);

/* Now, add an accelerator to the Open menuitem */
gtk_widget_add_accelerator(menuitem,
"activate", accel, ’P’, GDK_CONTROL_MASK, GTK_ACCEL_VISIBLE);
```

```
        gtk_widget_show(window);

        gtk_main();

        return 0;

}
```

## Entrada de datos

GtkEntry - a single line text entry field. GtkHScale - a horizontal slider widget for selecting a value from a range. GtkVScale - a vertical slider widget for selecting a value from a range. GtkSpinButton - retrieve an integer or floating-point number from the user. GtkEditable - Interface for text-editing widgets.

### GtkEntry

El widget Entry permite mostrar e introducir texto en una línea de un cuadro de diálogo. El texto se puede poner con llamadas a funciones que permiten reemplazar, preañadir o añadir el texto al contenido actual del widget Entry. Hay dos funciones para crear widget Entry:

```
GtkWidget *gtk_entry_new( void );

GtkWidget *gtk_entry_new_with_max_length( guint16 max );
```

La primera sirve para crear un nuevo widget Entry, mientras que la segunda crea el widget y además establece un límite en la longitud del texto que irá en el mismo. hay varias funciones que sirven para alterar el que texto que se está mostrando en el widget Entry.

```
void gtk_entry_set_text( GtkEntry    *entry,
                         const gchar *text );

void gtk_entry_append_text( GtkEntry    *entry,
                            const gchar *text );

void gtk_entry_prepend_text( GtkEntry    *entry,
                             const gchar *text );
```

La función `gtk_entry_set_text` cambia el contenido actual del widget Entry. Las funciones `gtk_entry_append_text` y `gtk_entry_prepend_text` permiten añadir texto por delante o por detrás. Las función siguientes permiten decir donde poner el punto de inserción.

```
void gtk_entry_set_position( GtkEntry *entry,
                             gint      position );
```

Se pueden obtener los contenidos del widget llamando a la función que se describe a continuación. Obtener los contenidos del widget puede ser útil en las funciones de llamada descritas más adelante.

```
gchar *gtk_entry_get_text( GtkEntry *entry );
```

Si quiere impedir que alguien cambie el contenido del widget escribiendo en él, utilice la función

```
void gtk_entry_set_editable( GtkEntry *entry,
                             gboolean  editable );
```

Esta función permite camiar el estado de edición de un widget Entry, siendo el argumento editable TRUE o FALSE. Si estamos utilizando el widget Entry en un sitio donde no queremos que el texto que se introduce sea visible, como por ejemplo cuando estamos introduciendo una clave, podemos utilizar la función siguiente, que también admite como argumento una bandera booleana.

```
void gtk_entry_set_visibility( GtkEntry *entry,
                               gboolean  visible );
```

Se puede seleccionar una región del texto utilizando la siguiente función. Esta función se puede utilizar después de poner algún texto por defecto en el widget, haciéndole fácil al usuario eliminar este texto.

```
void gtk_entry_select_region( GtkEntry *entry,
                              gint      start,
                              gint      end );
```

Si queremos saber el momento en el que el usuario ha introducido el texto, podemos conectar con las señales `activate` o `changed`. `activate` se activa cuando el usuario aprieta la tecla enter en el widget. `changed` se activa cuando cambia algo del texto, p.e. cuando se introduce o se elimina algún carácter.

Ejemplo de código de utilización del widget `gtk-entry`:

```
#include <gtk/gtk.h>

void enter_callback(GtkWidget *widget, GtkWidget *entry)
{
  gchar *entry_text;
  entry_text = gtk_entry_get_text(GTK_ENTRY(entry));
  printf("Entry contents: %s\n", entry_text);
}

void entry_toggle_editable (GtkWidget *checkbutton,
                                   GtkWidget *entry)
{
  gtk_entry_set_editable(GTK_ENTRY(entry),
                         GTK_TOGGLE_BUTTON(checkbutton)->active);
}

void entry_toggle_visibility (GtkWidget *checkbutton,
                                   GtkWidget *entry)
{
  gtk_entry_set_visibility(GTK_ENTRY(entry),
                         GTK_TOGGLE_BUTTON(checkbutton)->active);
}

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

    GtkWidget *window;
    GtkWidget *vbox, *hbox;
    GtkWidget *entry;
    GtkWidget *button;
    GtkWidget *check;

    gtk_init (&argc, &argv);

    /* crear una nueva ventana */
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
```

```
    gtk_widget_set_usize( GTK_WIDGET (window), 200, 100);
    gtk_window_set_title(GTK_WINDOW (window), "GTK Entry");
    gtk_signal_connect(GTK_OBJECT (window), "delete_event",
                       (GtkSignalFunc) gtk_exit, NULL);

    vbox = gtk_vbox_new (FALSE, 0);
    gtk_container_add (GTK_CONTAINER (window), vbox);
    gtk_widget_show (vbox);

    entry = gtk_entry_new_with_max_length (50);
    gtk_signal_connect(GTK_OBJECT(entry), "activate",
                       GTK_SIGNAL_FUNC(enter_callback),
                       entry);
    gtk_entry_set_text (GTK_ENTRY (entry), "hello");
    gtk_entry_append_text (GTK_ENTRY (entry), " world");
    gtk_entry_select_region (GTK_ENTRY (entry),
                             0, GTK_ENTRY(entry)->text_length);
    gtk_box_pack_start (GTK_BOX (vbox), entry, TRUE, TRUE, 0);
    gtk_widget_show (entry);

    hbox = gtk_hbox_new (FALSE, 0);
    gtk_container_add (GTK_CONTAINER (vbox), hbox);
    gtk_widget_show (hbox);

    check = gtk_check_button_new_with_label("Editable");
    gtk_box_pack_start (GTK_BOX (hbox), check, TRUE, TRUE, 0);
    gtk_signal_connect (GTK_OBJECT(check), "toggled",
                        GTK_SIGNAL_FUNC(entry_toggle_editable), entry);
    gtk_toggle_button_set_state(GTK_TOGGLE_BUTTON(check), TRUE);
    gtk_widget_show (check);

    check = gtk_check_button_new_with_label("Visible");
    gtk_box_pack_start (GTK_BOX (hbox), check, TRUE, TRUE, 0);
    gtk_signal_connect (GTK_OBJECT(check), "toggled",
                        GTK_SIGNAL_FUNC(entry_toggle_visibility), entry);
    gtk_toggle_button_set_state(GTK_TOGGLE_BUTTON(check), TRUE);
    gtk_widget_show (check);

    button = gtk_button_new_with_label ("Close");
    gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                               GTK_SIGNAL_FUNC(gtk_exit),
                               GTK_OBJECT (window));
    gtk_box_pack_start (GTK_BOX (vbox), button, TRUE, TRUE, 0);
    GTK_WIDGET_SET_FLAGS (button, GTK_CAN_DEFAULT);
    gtk_widget_grab_default (button);
    gtk_widget_show (button);

    gtk_widget_show(window);

    gtk_main();
    return(0);
}
```

## Ajustes

Existen diferentes widgets en GTK+ que pueden ser ajustados visualmente por el usuario mediante el ratón o el teclado. Un ejemplo son los widgets de selección descritos en la sección la sección de nombre *Funciones y señales*. También hay otros widgets que pueden ser ajustados parcialmente, por ejemplo el widget de texto o el viewport.

Como es lógico el programa tiene que poder reaccionar a los cambios que el usuario realiza. Mediante señales especiales se podría conseguir saber qué y cuánto ha sido ajustado, pero en el caso de que se quiera conectar la señal con diferentes widgets, de manera que todos cambien al mismo tiempo, el proceso puede ser un poco tedioso.

El ejemplo más obvio es conectar una barra de desplazamiento a una región con texto. Si cada widget posee su propia forma de establecer u obtener sus valores de ajuste el programador puede que tenga que escribir sus propios controladores de señales para traducir el resultado de la señal producida por un widget como el argumento de una función usada para determinar valores en otro widget.

Para resolver este problema GTK+ usa objetos del tipo GtkAdjustment. Con ellos se consigue almacenar y traspasar información de una forma abstracta y flexible. El uso más obvio es el de almacenes de parámetros para widgets de escala (barras deslizantes y escalas). Como los GtkAdjustment derivan de GtkObject poseen cualidades intrínsecas que les permiten ser algo más que simples estructuras de datos. Lo más importante es que pueden emitir señales que a su vez pueden ser usadas tanto para reaccionar frente al cambio de datos introducidos por el usuario como para transferir los nuevos valores de forma transparente entre widgets ajustables.

### Creando un ajuste

Los ajustes se pueden crear usando:

```
GtkObject *gtk_adjustment_new( gfloat value,
                               gfloat lower,
                               gfloat upper,
                               gfloat step_increment,
                               gfloat page_increment,
                               gfloat page_size );
```

El argumento value es el valor inicial que le queremos dar al ajuste. Normalmente se corresponde con las posiciones situadas más arriba y a la izquierda de un widget ajustable. El argumento lower especifica los valores más pequeños que el ajuste puede contener. A su vez con step_increment se especifica el valor más pequeño en el que se puede variar la magnitud en cuestión (valor de paso asociado), mientras que page_increment es el mayor. Con page_size se determina el valor visible de un widget.

### Forma sencilla de usar los ajustes

Los widgets ajustables se pueden dividir en dos categorias diferentes, aquellos que necesitan saber las unidades de la cantidad almacenada y los que no. Este último grupo incluye los widgets de tamaño (barras deslizantes, escalas, barras de estado, o botones giratorios). Normalmente estos widgets son ajustados "directamente" por el usuario. Los argumentos lower y upper serán los limites dentro de los cuales el usuario puede manipular los ajustes. Por defecto sólo se modificará el value (valor) de un ajuste.

El otro grupo incluye los widgets de texto, la lista compuesta o la ventana con barra deslizante. Estos widgets usan valores en pixels para sus ajustes, y normalmente son ajustados "indirectamente" mediante barras deslizantes. Aunque todos los widgets pueden crear sus propios ajustes o usar otros creados por el programador con el

segundo grupo suele ser conveniente dejarles que creen sus propios ajustes. Normalmente no tendrán en cuenta ninguno de los valores de un ajuste proporcionado por el programador, excepto value, pero los resultados son, en general, indefinidos (entiéndase que tendrá que leer el código fuente para saber que pasa con cada widget).

Probablemente ya se habrá dado cuenta de que como los widgets de texto (y todos los widgets del segundo grupo), insisten en establecer todos los valores excepto value, mientras que las barras deslizantes sólo modifican value, si se comparte un objeto de ajuste entre una barra deslizante y un widget de texto al manipular la barra se modificará el widget de texto. Ahora queda completamente demostrada la utilidad de los ajustes. Veamos un ejemplo:

```
  /* creamos un ajuste */
     viewport = gtk_viewport_new (NULL, NULL);
  /* lo usamos con la barra deslizante */
     vscrollbar = gtk_vscrollbar_new (gtk_viewport_get_vadjustment (viewport));
```

## Descripción detallada de los ajustes

Puede que se esté preguntando cómo es posible crear sus propios controladores para responder a las modificaciones producidas por el usuario y cómo obtener el valor del ajuste hecho por este. Para aclarar esto y otras cosas vamos a estudiar la estructura del ajuste

```
struct _GtkAdjustment
{
  GtkData data;

  gfloat lower;
  gfloat upper;
  gfloat value;
  gfloat step_increment;
  gfloat page_increment;
  gfloat page_size;
};
```

Lo primero que hay que aclarar es que no hay ninguna macro o función de acceso que permita obtener el value de un GtkAdjustment, por lo que tendrá que hacerlo usted mismo. Tampoco se preocupe mucho porque la macro GTK_ADJUSTMENT (Object) comprueba los tipos durante el proceso de ejecución (como hacen todas las macros de GTK+ que sirven para comprobar los tipos). Cuando se establece el value de un ajuste normalmente se quiere que cualquier widget se entere del cambio producido. Para ello GTK+ posee una función especial:

```
void gtk_adjustment_set_value( GtkAdjustment *adjustment,
                               gfloat         value );
```

Tal y como se mencionó antes GtkAdjustment es una subclase de GtkObject y por tanto puede emitir señales. Así se consigue que se actualicen los valores de los ajustes cuando se comparten entre varios widgets. Por tanto todos los widgets ajustables deben conectar controladores de señales a sus señales del tipo value_changed. Esta es la definición de la señal como viene en struct _GtkAdjustmentClass

```
  void (* value_changed) (GtkAdjustment *adjustment);
```

Todos los widgets que usan GtkAdjustment deben emitir esta señal cuando cambie el valor de algún ajuste. Esto sucede cuando el usuario cambia algo o el programa modifica los ajustes mediante. Por ejemplo si queremos que rote una figura cuando modificamos un widget de escala habría que usar una respuesta como esta:

```
void cb_rotate_picture (GtkAdjustment *adj, GtkWidget *picture)
{
  set_picture_rotation (picture, adj->value);
...
```

y conectarla con el ajuste del widget de escala mediante:

```
gtk_signal_connect (GTK_OBJECT (adj), "value_changed",
                    GTK_SIGNAL_FUNC (cb_rotate_picture), picture);
```

\x{00BF}Qué pasa cuando un widget reconfigura los valores upper o lower (por ejemplo cuando se añade más texto)? Simplemente que se emite la señal changed, que debe ser parecida a:

```
  void (* changed)       (GtkAdjustment *adjustment);
```

Los widgets de tamaño normalmente conectan un controlador a esta señal, que cambia el aspecto de éste para reflejar el cambio. Por ejemplo el tamaño de la guía en una barra deslizante que se alarga o encoge según la inversa de la diferencia de los valores lower y upper.

Probablemente nunca tenga que conectar un controlador a esta señal a no ser que esté escribiendo un nuevo tipo de widget. Pero si cambia directamente alguno de los valores de GtkAdjustment debe hacer que se emita la siguiente señal para reconfigurar todos aquellos widgets que usen ese ajuste:

```
gtk_signal_emit_by_name (GTK_OBJECT (adjustment), "changed");
```

## Widgets de selección de rango

Este tipo de *widgets* incluye a las barras de desplazamiento (*scrollbar*) y la menos conocida escala (*scale*). Ambos pueden ser usados para muchas cosas, pero como sus funciones y su implementación son muy parecidas los describimos al mismo tiempo. Principalmente se utilizan para permitirle al usuario escoger un valor dentro de un rango ya prefijado.

Todos los *widgets* de selección comparten elementos gráficos, cada uno de los cuales tiene su propia ventana X window y recibe eventos. Todos contienen una guía y un rectángulo para determinar la posición dentro de la guía (en una procesador de textos con entorno gráfico se encuentra situado a la derecha del texto y sirve para situarnos en las diferentes partes del texto). Con el ratón podemos subir o bajar el rectángulo, mientras que si hacemos ‘click’ dentro de la guía, pero no sobre el rectángulo, este se mueve hacia donde hemos hecho el click. Dependiendo del botón pulsado el rectángulo se moverá hasta la posición del click o una cantidad prefijada de ante mano.

Tal y como se mencionó en la sección de nombre *Ajustes* todos los *widgets* usados para seleccionar un rango estan asociados con un objeto de ajuste, a partir del cual calculan la longitud de la barra y su posición. Cuando el usuario manipula la barra de desplazamiento el widget cambiará el valor del ajuste.

## El *widget* barra de desplazamiento

El *widget* barra de desplazamiento solamente debe utilizarse para hacer *scroll* sobre otro *widget*, como una lista, una caja de texto, o un viewport (y en muchos es más fácil utilizar el *widget* GtkScrolledWindow). Para el resto de los casos, debería utilizar los *widgets* de escala, ya que son más sencillos y potentes.

Hay dos tipos separados de barras de desplazamiento, según sea horizontal o vertical. Realmente no hay mucho que añadir. Puede crear estos *widgets* utilizar las funciones siguientes, definidas en `<gtk/gtkhscrollbar.h>` y `<gtk/gtkvscrollbar.h>`:

```
GtkWidget* gtk_hscrollbar_new( GtkAdjustment *adjustment );

GtkWidget* gtk_vscrollbar_new( GtkAdjustment *adjustment );
```

y esto es todo lo que hay, como puede comprobarse leyendo los ficheros de cabecera para estas funciones. El argumento *`adjustment`* puede ser un puntero a un ajuste ya existente, o puede ser NULL, en cuyo caso se creará uno. Es útil especificar NULL si quiere pasar el ajuste recién creado a la función constructora de algún otro *widget* (como por ejemplo el *widget* texto) que se ocupará de configurarlo correctamente por usted.

### *Creación de un widget de escala*

Existen dos tipos de *widgets* de escala: GtkHScale (que es horizontal) y GtkVscale (vertical). Como funcionan de la misma manera los vamos a describir a la vez. Las funciones definidas en `<gtk/gtkvscale.h>` y `<gtk/gtkhscale.h>`, crean *widgets* de escala verticales y horizontales respectivamente.

```
GtkWidget* gtk_vscale_new( GtkAdjustment *adjustment );

GtkWidget* gtk_hscale_new( GtkAdjustment *adjustment );
```

El `ajuste` (adjustment) puede ser tanto un ajuste creado mediante `gtk_adjustment_new()` como `NULL`. En este último caso se crea un GtkAdjustment anónimo con todos sus valores iguales a `0.0`. Si no ha quedado claro el uso de esta función consulte la sección la sección de nombre *Ajustes* para una discusión más detallada.

### *Funciones y señales*

Los *widgets* de escala pueden indicar su valor actual como un número. Su comportamiento por defecto es mostrar este valor, pero se puede modificar usando:

```
void gtk_scale_set_draw_value( GtkScale *scale,
                               gint      draw_value );
```

Los valores posibles de *`draw_value`* son `TRUE` o `FALSE`. Con el primero se muestra el valor y con el segundo no.

El valor mostrado por un *widget* de escala por defecto se redondea a un valor decimal (igual que con *value* en un GtkAdjustment). Se puede cambiar con:

```
void gtk_scale_set_digits( GtkScale *scale,
                            gint     digits );
```

donde *digits* es el número de posiciones decimales que se quiera. En la práctica sólo se mostrarán 13 como máximo.

Por último, el valor se puede dibujar en diferentes posiciones con respecto a la posición del rectangulo que hay dentro de la guía:

```
void gtk_scale_set_value_pos( GtkScale        *scale,
                               GtkPositionType  pos );
```

Si ha leido la sección acerca del *widget* libro de notas entonces ya conoce cuales son los valores posibles de *pos*. Estan definidos en <gtk/gtkscale.h> como enum GtkPositionType y son auto explicatorios. Si se escoge un lateral de la guía, entonces seguirá al rectángulo a lo largo de la guía.

Todas las funcioenes precedentes se encuentran definidas en: <gtk/gtkscale.h>.

## Funciones comunes

La descripción interna de la clase GtkRange es bastante complicada, pero al igual que con el resto de las «clases base» sólo es interesante si se quiere «hackear». Casi todas las señales y funciones sólo son útiles para desarrollar derivados. Para un usuario normal las funciones interesantes son aquellas definidas en: <gtk/gtkrange.h> y funcionan igual en todos los *widgets* de rango.

### *Estableciendo cada cúanto se actualizan*

La política de actualización de un *widget* define en que puntos de la interacción con el usuario debe cambiar el valor *value* en su GtkAdjustment y emitir la señal «value_changed». Las actualizaciones definidas en <gtk/gtkenums.h> como enum GtkUpdateType, son:

- GTK_UPDATE_POLICY_CONTINUOUS - Este es el valor por defecto.La señal «value_changed» se emite continuamente, por ejemplo cuando la barra deslizante se mueve incluso aunque sea un poquito.
- GTK_UPDATE_POLICY_DISCONTINUOUS - La señal Â«value_changedÂ» sólo se emite cuando se ha parado de mover la barra y el usuario ha soltado el botón del ratón.
- GTK_UPDATE_POLICY_DELAYED - La señal sólo se emite cuando el usuario suelta el botón del ratón o si la barra no se mueve durante un periodo largo de tiempo.

Para establecer la política de actualización se usa la conversión definida en la macro

```
void gtk_range_set_update_policy( GtkRange      *range,
                                  GtkUpdateType  policy) ;
```

### Obteniendo y estableciendo Ajustes

Para obtener o establecer el ajuste de un *widget* de rango se usa:

```
GtkAdjustment* gtk_range_get_adjustment( GtkRange *range );

void gtk_range_set_adjustment( GtkRange      *range,
                               GtkAdjustment *adjustment );
```

La función `gtk_range_get_adjustment()` devuelve un puntero al ajuste al que `range` esté conectado.

La función `gtk_range_set_adjustment()` no hace nada si se le pasa como argumento el valor *range* del ajuste que esta siendo usado (aunque se haya modificado algún valor). En el caso de que sea un ajuste nuevo (GtkAdjustment) dejará de usar el antiguo (probablemente lo destruirá) y conectará las señales apropiadas al nuevo. A continuación llamará a la función `gtk_range_adjustment_changed()` que en teoría recalculará el tamaño y/o la posición de la barra, redibujándola en caso de que sea necesario. Tal y como se mencionó en la sección de los ajustes si se quiere reusar el mismo GtkAdjustment cuando se modifican sus valores se debe emitir la señal Â«changedÂ». Por ejemplo:

```
gtk_signal_emit_by_name (GTK_OBJECT (adjustment), "changed");
```

## Enlaces con el teclado y el ratón

Todos los *widgets* de rango reaccionan más o menos de la misma manera a las pulsaciones del ratón. Al pulsar el botón 1 sobre el rectángulo de la barra el *value* del ajuste aumentará o disminuirá según *page_increment*. Con el botón 2 la barra se desplazará al punto en el que el botón fue pulsado. Con cada pulsación de cualquier botón sobre las flechas el valor del ajuste se modifica una cantidad igual a *step_increment*.

Acostumbrarse a que tanto las barras deslizantes como los *widgets* de escala puedan tomar la atención del teclado puede ser un proceso largo. Si que se cree que los usuarios no lo van a entender se puede anular mediante la función GTK_WIDGET_UNSET_FLAGS y con GTK_CAN_FOCUS como argumento:

```
GTK_WIDGET_UNSET_FLAGS (scrollbar, GTK_CAN_FOCUS);
```

Los enlaces entre teclas (que sólo estan activos cuando el *widget* tiene la atención (focus)) se comportan de manera diferente para los *widgets* de rango horizontales que para los verticales. También son diferentes para los *widgets* de escala y para las barras deslizantes. (Simplemente para evitar confusiones entre las teclas de las barras deslizantes horizontales y verticales, ya que ambas actúan sobre la misma área)

### Widgets de rango vertical

Todos los *widgets* de rango pueden ser manipulados con las teclas arriba, abajo, `Re Pág`, `Av Pág`. Las flechas mueven las barras la cantidad fijada

mediante *step_increment*, mientras que `Re Pág` y `Av Pag` lo hacen según *page_increment*.

El usuario también puede mover la barra de un extremo al otro de la guía mediante el teclado. Con el *widget* GtkVScale podemos ir a los extremos utilizando las teclas `Inicio` y `Final` mientras que con el *widget* GtkVScrollbar habrá que utilizar `Control-Re Pág` y `Control-Av Pág`.

### *Widgets de rango horizontal*

Las teclas izquierda y derecha funcionan tal y como espera que funcionen en estos *widgets*: mueven la barra una cantidad dada por *step_increment*. A su vez `Inicio` y `Final` sirven para pasar de un extremo al otro de la guía. Para el *widget* GtkHScale el mover la barra una cantidad dada por *page_increment* se consigue mediante `Control-Izquierda` y `Control-derecha`, mientras que para el *widget* GtkHScrollbar se consigue con `Control-Inicio` y `Control-Final`.

## Ejemplo

Este ejemplo es una versión modificada del test Â«range controlsÂ» que a su vez forma parte de `testgtk.c`. Simplemente dibuja una ventana con tres *widgets* de rango conectados al mismo ajuste, y un conjunto de controles para ajustar algunos de los parámetros ya mencionados. Así se consigue ver como funcionan estos *widgets* al ser manipulados por el usuario.

```
#include <gtk/gtk.h>

GtkWidget *hscale, *vscale;

void cb_pos_menu_select( GtkWidget       *item,
                         GtkPositionType  pos )
{
    /* Establece el valor position en los widgets de escala */
    gtk_scale_set_value_pos (GTK_SCALE (hscale), pos);
    gtk_scale_set_value_pos (GTK_SCALE (vscale), pos);
}

void cb_update_menu_select( GtkWidget     *item,
                            GtkUpdateType  policy )
{
    /* Establece la política de actualización para los widgets
     * de escala */
    gtk_range_set_update_policy (GTK_RANGE (hscale), policy);
    gtk_range_set_update_policy (GTK_RANGE (vscale), policy);
}

void cb_digits_scale( GtkAdjustment *adj )
{
    /* Establece el número de cifras decimales a las que se
     * redondeará adj->value */
    gtk_scale_set_digits (GTK_SCALE (hscale), (gint) adj->value);
    gtk_scale_set_digits (GTK_SCALE (vscale), (gint) adj->value);
}

void cb_page_size( GtkAdjustment *get,
                   GtkAdjustment *set )
{
    /* Establece el tamaño de la página y el incremento del
     * ajuste al valor especificado en la escala "Page Size" */
    set->page_size = get->value;
```

```
    set->page_increment = get->value;
    /* Ahora emite la señal "changed" para reconfigurar todos los
     * widgets que están enlazados a este ajuste */
    gtk_signal_emit_by_name (GTK_OBJECT (set), "changed");
}

void cb_draw_value( GtkToggleButton *boton )
{
    /* Activa o desactiva el valor display en los widgets de escala
     * dependiendo del estado del botón de comprobación */
    gtk_scale_set_draw_value (GTK_SCALE (hscale), boton->active);
    gtk_scale_set_draw_value (GTK_SCALE (vscale), boton->active);
}

/* Funciones varias */

GtkWidget *make_menu_item( gchar         *name,
                           GtkSignalFunc  callback,
                           gpointer       data )
{
    GtkWidget *item;

    item = gtk_menu_item_new_with_label (name);
    gtk_signal_connect (GTK_OBJECT (item), "activate",
                        callback, data);
    gtk_widget_show (item);

    return(item);
}

void scale_set_default_values( GtkScale *scale )
{
    gtk_range_set_update_policy (GTK_RANGE (scale),
                                 GTK_UPDATE_CONTINUOUS);
    gtk_scale_set_digits (scale, 1);
    gtk_scale_set_value_pos (scale, GTK_POS_TOP);
    gtk_scale_set_draw_value (scale, TRUE);
}

/* crea la ventana principal */

void create_range_controls( void )
{
    GtkWidget *ventana;
    GtkWidget *caja1, *caja2, *caja3;
    GtkWidget *boton;
    GtkWidget *scrollbar;
    GtkWidget *separator;
    GtkWidget *opt, *menu, *item;
    GtkWidget *etiqueta;
    GtkWidget *scale;
    GtkObject *adj1, *adj2;

    /* creación estándar de una ventana */
    ventana = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_signal_connect (GTK_OBJECT (ventana), "destroy",
                        GTK_SIGNAL_FUNC(gtk_main_quit),
                        NULL);
    gtk_window_set_title (GTK_WINDOW (ventana), "range controls");

    caja1 = gtk_vbox_new (FALSE, 0);
    gtk_container_add (GTK_CONTAINER (ventana), caja1);
    gtk_widget_show (caja1);

    caja2 = gtk_hbox_new (FALSE, 10);
    gtk_container_border_width (GTK_CONTAINER (caja2), 10);
```

```
gtk_box_pack_start (GTK_BOX (caja1), caja2, TRUE, TRUE, 0);
gtk_widget_show (caja2);

/* value, lower, upper, step_increment, page_increment, page_size */
/* Observe que el valor de page_size solo sirve para los widgets
 * barras de desplazamiento (scrollbar), y que el valor más
 * alto que obtendrá será (upper - page_size). */
adj1 = gtk_adjustment_new (0.0, 0.0, 101.0, 0.1, 1.0, 1.0);

vscale = gtk_vscale_new (GTK_ADJUSTMENT (adj1));
scale_set_default_values (GTK_SCALE (vscale));
gtk_box_pack_start (GTK_BOX (caja2), vscale, TRUE, TRUE, 0);
gtk_widget_show (vscale);

caja3 = gtk_vbox_new (FALSE, 10);
gtk_box_pack_start (GTK_BOX (caja2), caja3, TRUE, TRUE, 0);
gtk_widget_show (caja3);

/* Reutilizamos el mismo ajuste */
hscale = gtk_hscale_new (GTK_ADJUSTMENT (adj1));
gtk_widget_set_usize (GTK_WIDGET (hscale), 200, 30);
scale_set_default_values (GTK_SCALE (hscale));
gtk_box_pack_start (GTK_BOX (caja3), hscale, TRUE, TRUE, 0);
gtk_widget_show (hscale);

/* Reutilizamos de nuevo el mismo ajuste */
scrollbar = gtk_hscrollbar_new (GTK_ADJUSTMENT (adj1));
/* Observe que con esto conseguimos que la escala siempre se
 * actualice de una forma continua cuando se mueva la barra de
 * desplazamiento */
gtk_range_set_update_policy (GTK_RANGE (scrollbar),
                             GTK_UPDATE_CONTINUOUS);
gtk_box_pack_start (GTK_BOX (caja3), scrollbar, TRUE, TRUE, 0);
gtk_widget_show (scrollbar);

caja2 = gtk_hbox_new (FALSE, 10);
gtk_container_border_width (GTK_CONTAINER (caja2), 10);
gtk_box_pack_start (GTK_BOX (caja1), caja2, TRUE, TRUE, 0);
gtk_widget_show (caja2);

/* Un botón para comprobar si el valor se muestra o no*/
boton = gtk_check_button_new_with_label("Display value on scale widgets");
gtk_toggle_button_set_state (GTK_TOGGLE_BUTTON (boton), TRUE);
gtk_signal_connect (GTK_OBJECT (boton), "toggled",
                    GTK_SIGNAL_FUNC(cb_draw_value), NULL);
gtk_box_pack_start (GTK_BOX (caja2), boton, TRUE, TRUE, 0);
gtk_widget_show (boton);

caja2 = gtk_hbox_new (FALSE, 10);
gtk_container_border_width (GTK_CONTAINER (caja2), 10);

/* Una opción en el menú para cambiar la posición del
 * valor */
etiqueta = gtk_label_new ("Scale Value Position:");
gtk_box_pack_start (GTK_BOX (caja2), etiqueta, FALSE, FALSE, 0);
gtk_widget_show (etiqueta);

opt = gtk_option_menu_new();
menu = gtk_menu_new();

item = make_menu_item ("Top",
                       GTK_SIGNAL_FUNC(cb_pos_menu_select),
                       GINT_TO_POINTER (GTK_POS_TOP));
gtk_menu_append (GTK_MENU (menu), item);

item = make_menu_item ("Bottom", GTK_SIGNAL_FUNC (cb_pos_menu_select),
```

```
                            GINT_TO_POINTER (GTK_POS_BOTTOM));
    gtk_menu_append (GTK_MENU (menu), item);

    item = make_menu_item ("Left", GTK_SIGNAL_FUNC (cb_pos_menu_select),
                            GINT_TO_POINTER (GTK_POS_LEFT));
    gtk_menu_append (GTK_MENU (menu), item);

    item = make_menu_item ("Right", GTK_SIGNAL_FUNC (cb_pos_menu_select),
                             GINT_TO_POINTER (GTK_POS_RIGHT));
    gtk_menu_append (GTK_MENU (menu), item);

    gtk_option_menu_set_menu (GTK_OPTION_MENU (opt), menu);
    gtk_box_pack_start (GTK_BOX (caja2), opt, TRUE, TRUE, 0);
    gtk_widget_show (opt);

    gtk_box_pack_start (GTK_BOX (caja1), caja2, TRUE, TRUE, 0);
    gtk_widget_show (caja2);

    caja2 = gtk_hbox_new (FALSE, 10);
    gtk_container_border_width (GTK_CONTAINER (caja2), 10);

    /* Sí, otra opción de menú, esta vez para la política
     * de actualización de los widgets */
    etiqueta = gtk_label_new ("Scale Update Policy:");
    gtk_box_pack_start (GTK_BOX (caja2), etiqueta, FALSE, FALSE, 0);
    gtk_widget_show (etiqueta);

    opt = gtk_option_menu_new();
    menu = gtk_menu_new();

    item = make_menu_item ("Continuous",
                            GTK_SIGNAL_FUNC (cb_update_menu_select),
                            GINT_TO_POINTER (GTK_UPDATE_CONTINUOUS));
    gtk_menu_append (GTK_MENU (menu), item);

    item = make_menu_item ("Discontinuous",
                             GTK_SIGNAL_FUNC (cb_update_menu_select),
                             GINT_TO_POINTER (GTK_UPDATE_DISCONTINUOUS));
    gtk_menu_append (GTK_MENU (menu), item);

    item = make_menu_item ("Delayed",
                            GTK_SIGNAL_FUNC (cb_update_menu_select),
                            GINT_TO_POINTER (GTK_UPDATE_DELAYED));
    gtk_menu_append (GTK_MENU (menu), item);

    gtk_option_menu_set_menu (GTK_OPTION_MENU (opt), menu);
    gtk_box_pack_start (GTK_BOX (caja2), opt, TRUE, TRUE, 0);
    gtk_widget_show (opt);

    gtk_box_pack_start (GTK_BOX (caja1), caja2, TRUE, TRUE, 0);
    gtk_widget_show (caja2);

    caja2 = gtk_hbox_new (FALSE, 10);
    gtk_container_border_width (GTK_CONTAINER (caja2), 10);

    /* Un widget GtkHScale para ajustar el número de dígitos en
     * la escala. */
    etiqueta = gtk_label_new ("Scale Digits:");
    gtk_box_pack_start (GTK_BOX (caja2), etiqueta, FALSE, FALSE, 0);
    gtk_widget_show (etiqueta);

    adj2 = gtk_adjustment_new (1.0, 0.0, 5.0, 1.0, 1.0, 0.0);
    gtk_signal_connect (GTK_OBJECT (adj2), "value_changed",
                        GTK_SIGNAL_FUNC (cb_digits_scale), NULL);
    scale = gtk_hscale_new (GTK_ADJUSTMENT (adj2));
    gtk_scale_set_digits (GTK_SCALE (scale), 0);
```

```
    gtk_box_pack_start (GTK_BOX (caja2), scale, TRUE, TRUE, 0);
    gtk_widget_show (scale);

    gtk_box_pack_start (GTK_BOX (caja1), caja2, TRUE, TRUE, 0);
    gtk_widget_show (caja2);

    caja2 = gtk_hbox_new (FALSE, 10);
    gtk_container_border_width (GTK_CONTAINER (caja2), 10);

    /* Y un último widget GtkHScale para ajustar el tamaño de la
     * página de la barra de desplazamiento. */
    etiqueta = gtk_label_new ("Scrollbar Page Size:");
    gtk_box_pack_start (GTK_BOX (caja2), etiqueta, FALSE, FALSE, 0);
    gtk_widget_show (etiqueta);

    adj2 = gtk_adjustment_new (1.0, 1.0, 101.0, 1.0, 1.0, 0.0);
    gtk_signal_connect (GTK_OBJECT (adj2), "value_changed",
                        GTK_SIGNAL_FUNC (cb_page_size), adj1);
    scale = gtk_hscale_new (GTK_ADJUSTMENT (adj2));
    gtk_scale_set_digits (GTK_SCALE (scale), 0);
    gtk_box_pack_start (GTK_BOX (caja2), scale, TRUE, TRUE, 0);
    gtk_widget_show (scale);

    gtk_box_pack_start (GTK_BOX (caja1), caja2, TRUE, TRUE, 0);
    gtk_widget_show (caja2);

    separator = gtk_hseparator_new ();
    gtk_box_pack_start (GTK_BOX (caja1), separator, FALSE, TRUE, 0);
    gtk_widget_show (separator);

    caja2 = gtk_vbox_new (FALSE, 10);
    gtk_container_border_width (GTK_CONTAINER (caja2), 10);
    gtk_box_pack_start (GTK_BOX (caja1), caja2, FALSE, TRUE, 0);
    gtk_widget_show (caja2);

    boton = gtk_button_new_with_label ("Quit");
    gtk_signal_connect_object (GTK_OBJECT (boton), "clicked",
                               GTK_SIGNAL_FUNC(gtk_main_quit),
                               NULL);
    gtk_box_pack_start (GTK_BOX (caja2), boton, TRUE, TRUE, 0);
    GTK_WIDGET_SET_FLAGS (boton, GTK_CAN_DEFAULT);
    gtk_widget_grab_default (boton);
    gtk_widget_show (boton);

    gtk_widget_show (ventana);
}

int main( int   argc,
          char *argv[] )
{
    gtk_init(&argc, &argv);

    create_range_controls();

    gtk_main();

    return(0);
}
```

Observe que el programa no llama a `gtk_signal_connect` para conectar el "delete_event", y que sólo conecta la señal "destroy". Con esto seguimos realizando

la función deseada, ya que un "delete_event" no manejado desemboca en una señal "destroy" para la ventana.

### GtkSpinButton

El widget botón spin se utiliza normalmente para permitir que el usuario elija un valor de un rango de valores. Consiste en una caja para la entrada de texto con una flecha para arriba y otra para abajo justo al lado de la caja. Si utilizamos alguna de las flechas haremos que el valor suba o baje dentro del rango de los valores posibles. También podemos introducir directamente un valor específico (utilizando la caja de texto).

El widget botón spin permite tener valores con un número de cifras decimales (o sin cifras decimales) y la posibilidad de incrementarlo/decrementarlo en pasos configurables. La acción de matener pulsado uno de los botones puede resultar (es opcional) en una aceleración del cambio en el valor, de acuerdo con el tiempo que se mantenga pulsado.

El botón spin utiliza un objeto Ajuste para conservar la información referente al rango de valores que puede tomar el botón spin. Esto hace que el widget botón spin sea muy poderoso.

Recuerde que un widget ajuste puede crearse con la siguiente función, que ilustra la información que se guarda:

```
GtkObject *gtk_adjustment_new( gfloat valor,
                              gfloat inferior,
                              gfloat superior,
                              gfloat paso,
                              gfloat incremento_pagina,
                              gfloat tamano_pagina );
```

Estos atributos de un ajuste se utilizan en un botón spin de la forma siguiente:

- valor: valor inicial del botón spin
- inferior: valor inferior del rango
- superior: valor superior del rango
- paso: valor a incrementar/decrementar cuando pulsemos el botón 1 en una flecha
- incremento_pagina: valor a incrementar/decrementar cuando pulsemos el botón 2 en una flecha
- tamano_pagina: no se utiliza

Además, se puede utilizar el botón 3 para saltar directamente a los valores superior o inferior cuando se pulsa en una de las flechas. Veamos como crear un botón spin:

```
  GtkWidget *gtk_spin_button_new( GtkAdjustment *ajuste,
                                  gfloat         aceleracion,
                                  guint          digitos );
```

El argumento aceleracion toma un valor entre 0.0 y 1.0 e indica la aceleración que tendrá el botón spin. El argumento digitos especifica el número de cifras decimales con que se mostrará el valor.

Se puede reconfigurar un botón spin después de su creación utilizando la función:

```
void gtk_spin_button_configure( GtkSpinButton *boton_spin,
```

```
                              GtkAdjustment *ajuste,
                              gfloat         aceleracion,
                              guint          digitos );
```

El argumento boton_spin especifica el botón spin que va a reconfigurarse. El resto de argumentos son los que acabamos de explicar.

Podemos establecer y obtener el ajuste utilizando las dos funciones siguientes:

```
void gtk_spin_button_set_adjustment( GtkSpinButton  *boton_spin,
                                    GtkAdjustment  *ajuste );


GtkAdjustment *gtk_spin_button_get_adjustment( GtkSpinButton *boton_spin );
```

El número de cifras decimales también puede alterarse utilizando:

```
void gtk_spin_button_set_digits( GtkSpinButton *boton_spin,
                                guint          digitos) ;
```

El valor que un botón spin está mostrando actualmente puede cambiarse utilizando las siguientes funciones:

```
void gtk_spin_button_set_value( GtkSpinButton *boton_spin,
                               gfloat         valor );
```

El valor actual de un botón spin puede obtenerse como un entero o como un flotante con las funciones siguientes:

```
 gfloat gtk_spin_button_get_value_as_float( GtkSpinButton *boton_spin );

 gint gtk_spin_button_get_value_as_int( GtkSpinButton *boton_spin );
```

Si quiere alterar el valor de un spin de forma relativa a su valor actual, puede utilizar la siguiente función:

```
void gtk_spin_button_spin( GtkSpinButton *boton_spin,
                          GtkSpinType    direccion,
                          gfloat         incremento );
```

El parámetro direccion puede tomar uno de los valores siguientes:

- GTK_SPIN_STEP_FORWARD
- GTK_SPIN_STEP_BACKWARD
- GTK_SPIN_PAGE_FORWARD
- GTK_SPIN_PAGE_BACKWARD
- GTK_SPIN_HOME

- GTK_SPIN_END
- GTK_SPIN_USER_DEFINED

Trataré de explicar todas las posibilidades que ofrece esta función. Algunos de los valores que puede utilizar direccion hacen que se utilicen valores que están almacenados en el objeto Ajuste que está asociado con el botón spin.

GTK_SPIN_STEP_FORWARD y GTK_SPIN_STEP_BACKWARD aumentan o disminuyen (respectivamente) el valor del botón spin por la cantidad especificada por incremento, a menos que incremento sea igual a 0, en cuyo caso el valor se aumentará o disminuirá por el valor especificado en paso dentro del Ajuste.

GTK_SPIN_PAGE_FORWARD y GTK_SPIN_PAGE_BACKWARD sencillamente alteran el valor del botón spin por la cantidad incremento.

GTK_SPIN_HOME hace que el botón spin tenga el mismo valor que el valor inferior del rango Ajuste.

GTK_SPIN_END hace que el botón spin tenga el mismo valor que el valor superior del rango Ajuste.

GTK_SPIN_USER_DEFINED cambia el valor del botón spin por la cantidad especificada.

Ahora vamos a dejar de lado las funciones para establecer y obtener el rango de los atributos del botón spin, y vamos a pasar a las funciones que afectan a la apariencia y al comportamiento del widget botón spin en sí mismo.

La primera de estas funciones se utiliza para restringir el contenido de la caja de texto de un botón spin a un valor numérico. Esto evita que un usuario introduzca cualquier valor no númerico.

```
void gtk_spin_button_set_numeric( GtkSpinButton *boton_spin,
                                  gboolean       numerico );
```

Puede indicar si un botón spin pasará del límite superior al inferior utilizando la siguiente función:

```
void gtk_spin_button_set_wrap( GtkSpinButton *boton_spin,
          gboolean       wrap );
```

Puede hacer que un botón spin redondee su valor al paso más cercano, que se indica cuando creamos el Ajuste que se utiliza con el botón spin. Para hacer que redondee tenemos que utilizar la función siguiente:

```
void gtk_spin_button_set_snap_to_ticks( GtkSpinButton  *boton_spin,
     gboolean        redondear );
```

Para política de actualización de un botón spin puede cambiarse con la siguiente función:

```
void gtk_spin_button_set_update_policy( GtkSpinButton  *boton_spin,
        GtkSpinButtonUpdatePolicy politica );
```

Los valores posibles de politica son o GTK_UPDATE_ALWAYS o GTK_UPDATE_IF_VALID.

Estas políticas afectan al comportamiento de un botón spin cuando se lee el texto insertado en la caja de texto y se sincroniza con los valores del Ajuste.

En el caso de GTK_UPDATE_IF_VALID el valor de un botón spin cambiará si el texto introducido es un valor numérico contenido dentro del rango especificado por el Ajuste. En caso contrario el texto introducido se convierte al valor del botón spin.

En caso de utilizar GTK_UPDATE_ALWAYS se ignorarán los errores que puedan ocurrir en la conversión del texto en un valor numérico.

El aspecto de los botones utilizados en un botón spin pueden cambiarse utilizando las siguientes funciones:

```
void gtk_spin_button_set_shadow_type( GtkSpinButton *boton_spin,
          GtkShadowType  tipo_sombra );
```

Como siempre, el tipo_sombra puede ser uno de los siguientes:

- GTK_SHADOW_IN
- GTK_SHADOW_OUT
- GTK_SHADOW_ETCHED_IN
- GTK_SHADOW_ETCHED_OUT

Finalmente, puede pedir de forma explícita que un botón spin se actualice a sí mismo:

```
void gtk_spin_button_update( GtkSpinButton  *boton_spin );
```

Es hora de un nuevo ejemplo.

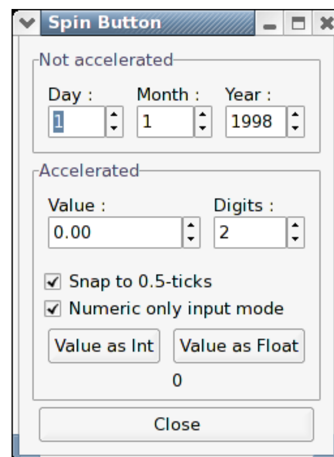



**Figura 7-9. Todos los ajustes de un botón spin al descubierto**

```
#include <stdio.h>
#include <gtk/gtk.h>
```

```
static GtkWidget *spinner1;

static void toggle_snap( GtkWidget     *widget,
                         GtkSpinButton *spin )
{
  gtk_spin_button_set_snap_to_ticks (spin, GTK_TOGGLE_BUTTON (widget)->active);
}

static void toggle_numeric( GtkWidget *widget,
                            GtkSpinButton *spin )
{
  gtk_spin_button_set_numeric (spin, GTK_TOGGLE_BUTTON (widget)->active);
}

static void change_digits( GtkWidget *widget,
                           GtkSpinButton *spin )
{
  gtk_spin_button_set_digits (GTK_SPIN_BUTTON (spinner1),
                              gtk_spin_button_get_value_as_int (spin));
}

static void get_value( GtkWidget *widget,
                       gpointer data )
{
  gchar *buf;
  GtkLabel *label;
  GtkSpinButton *spin;

  spin = GTK_SPIN_BUTTON (spinner1);
  label = GTK_LABEL (g_object_get_data (G_OBJECT (widget), "user_data"));
  if (GPOINTER_TO_INT (data) == 1)
    buf = g_strdup_printf ("%d", gtk_spin_button_get_value_as_int (spin));
  else
    buf = g_strdup_printf ("%0.*f", spin->digits,
                           gtk_spin_button_get_value (spin));
  gtk_label_set_text (label, buf);
  g_free (buf);
}


int main( int   argc,
          char *argv[] )
{
  GtkWidget *window;
  GtkWidget *frame;
  GtkWidget *hbox;
  GtkWidget *main_vbox;
  GtkWidget *vbox;
  GtkWidget *vbox2;
  GtkWidget *spinner2;
  GtkWidget *spinner;
  GtkWidget *button;
  GtkWidget *label;
  GtkWidget *val_label;
  GtkAdjustment *adj;

  /* Initialise GTK */
  gtk_init (&argc, &argv);

  window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

  g_signal_connect (G_OBJECT (window), "destroy",
                    G_CALLBACK (gtk_main_quit),
                    NULL);

  gtk_window_set_title (GTK_WINDOW (window), "Spin Button");
```

```
main_vbox = gtk_vbox_new (FALSE, 5);
gtk_container_set_border_width (GTK_CONTAINER (main_vbox), 10);
gtk_container_add (GTK_CONTAINER (window), main_vbox);

frame = gtk_frame_new ("Not accelerated");
gtk_box_pack_start (GTK_BOX (main_vbox), frame, TRUE, TRUE, 0);

vbox = gtk_vbox_new (FALSE, 0);
gtk_container_set_border_width (GTK_CONTAINER (vbox), 5);
gtk_container_add (GTK_CONTAINER (frame), vbox);

/* Day, month, year spinners */

hbox = gtk_hbox_new (FALSE, 0);
gtk_box_pack_start (GTK_BOX (vbox), hbox, TRUE, TRUE, 5);

vbox2 = gtk_vbox_new (FALSE, 0);
gtk_box_pack_start (GTK_BOX (hbox), vbox2, TRUE, TRUE, 5);

label = gtk_label_new ("Day :");
gtk_misc_set_alignment (GTK_MISC (label), 0, 0.5);
gtk_box_pack_start (GTK_BOX (vbox2), label, FALSE, TRUE, 0);

adj = (GtkAdjustment *) gtk_adjustment_new (1.0, 1.0, 31.0, 1.0,
                                            5.0, 0.0);
spinner = gtk_spin_button_new (adj, 0, 0);
gtk_spin_button_set_wrap (GTK_SPIN_BUTTON (spinner), TRUE);
gtk_box_pack_start (GTK_BOX (vbox2), spinner, FALSE, TRUE, 0);

vbox2 = gtk_vbox_new (FALSE, 0);
gtk_box_pack_start (GTK_BOX (hbox), vbox2, TRUE, TRUE, 5);

label = gtk_label_new ("Month :");
gtk_misc_set_alignment (GTK_MISC (label), 0, 0.5);
gtk_box_pack_start (GTK_BOX (vbox2), label, FALSE, TRUE, 0);

adj = (GtkAdjustment *) gtk_adjustment_new (1.0, 1.0, 12.0, 1.0,
                                            5.0, 0.0);
spinner = gtk_spin_button_new (adj, 0, 0);
gtk_spin_button_set_wrap (GTK_SPIN_BUTTON (spinner), TRUE);
gtk_box_pack_start (GTK_BOX (vbox2), spinner, FALSE, TRUE, 0);

vbox2 = gtk_vbox_new (FALSE, 0);
gtk_box_pack_start (GTK_BOX (hbox), vbox2, TRUE, TRUE, 5);

label = gtk_label_new ("Year :");
gtk_misc_set_alignment (GTK_MISC (label), 0, 0.5);
gtk_box_pack_start (GTK_BOX (vbox2), label, FALSE, TRUE, 0);

adj = (GtkAdjustment *) gtk_adjustment_new (1998.0, 0.0, 2100.0,
                                            1.0, 100.0, 0.0);
spinner = gtk_spin_button_new (adj, 0, 0);
gtk_spin_button_set_wrap (GTK_SPIN_BUTTON (spinner), FALSE);
gtk_widget_set_size_request (spinner, 55, -1);
gtk_box_pack_start (GTK_BOX (vbox2), spinner, FALSE, TRUE, 0);

frame = gtk_frame_new ("Accelerated");
gtk_box_pack_start (GTK_BOX (main_vbox), frame, TRUE, TRUE, 0);

vbox = gtk_vbox_new (FALSE, 0);
gtk_container_set_border_width (GTK_CONTAINER (vbox), 5);
gtk_container_add (GTK_CONTAINER (frame), vbox);

hbox = gtk_hbox_new (FALSE, 0);
gtk_box_pack_start (GTK_BOX (vbox), hbox, FALSE, TRUE, 5);
```

```
vbox2 = gtk_vbox_new (FALSE, 0);
gtk_box_pack_start (GTK_BOX (hbox), vbox2, TRUE, TRUE, 5);

label = gtk_label_new ("Value :");
gtk_misc_set_alignment (GTK_MISC (label), 0, 0.5);
gtk_box_pack_start (GTK_BOX (vbox2), label, FALSE, TRUE, 0);

adj = (GtkAdjustment *) gtk_adjustment_new (0.0, -10000.0, 10000.0,
                                            0.5, 100.0, 0.0);
spinner1 = gtk_spin_button_new (adj, 1.0, 2);
gtk_spin_button_set_wrap (GTK_SPIN_BUTTON (spinner1), TRUE);
gtk_widget_set_size_request (spinner1, 100, -1);
gtk_box_pack_start (GTK_BOX (vbox2), spinner1, FALSE, TRUE, 0);

vbox2 = gtk_vbox_new (FALSE, 0);
gtk_box_pack_start (GTK_BOX (hbox), vbox2, TRUE, TRUE, 5);

label = gtk_label_new ("Digits :");
gtk_misc_set_alignment (GTK_MISC (label), 0, 0.5);
gtk_box_pack_start (GTK_BOX (vbox2), label, FALSE, TRUE, 0);

adj = (GtkAdjustment *) gtk_adjustment_new (2, 1, 5, 1, 1, 0);
spinner2 = gtk_spin_button_new (adj, 0.0, 0);
gtk_spin_button_set_wrap (GTK_SPIN_BUTTON (spinner2), TRUE);
g_signal_connect (G_OBJECT (adj), "value_changed",
                  G_CALLBACK (change_digits),
                  (gpointer) spinner2);
gtk_box_pack_start (GTK_BOX (vbox2), spinner2, FALSE, TRUE, 0);

hbox = gtk_hbox_new (FALSE, 0);
gtk_box_pack_start (GTK_BOX (vbox), hbox, FALSE, TRUE, 5);

button = gtk_check_button_new_with_label ("Snap to 0.5-ticks");
g_signal_connect (G_OBJECT (button), "clicked",
                  G_CALLBACK (toggle_snap),
                  (gpointer) spinner1);
gtk_box_pack_start (GTK_BOX (vbox), button, TRUE, TRUE, 0);
gtk_toggle_button_set_active (GTK_TOGGLE_BUTTON (button), TRUE);

button = gtk_check_button_new_with_label ("Numeric only input mode");
g_signal_connect (G_OBJECT (button), "clicked",
                  G_CALLBACK (toggle_numeric),
                  (gpointer) spinner1);
gtk_box_pack_start (GTK_BOX (vbox), button, TRUE, TRUE, 0);
gtk_toggle_button_set_active (GTK_TOGGLE_BUTTON (button), TRUE);

val_label = gtk_label_new ("");

hbox = gtk_hbox_new (FALSE, 0);
gtk_box_pack_start (GTK_BOX (vbox), hbox, FALSE, TRUE, 5);
button = gtk_button_new_with_label ("Value as Int");
g_object_set_data (G_OBJECT (button), "user_data", val_label);
g_signal_connect (G_OBJECT (button), "clicked",
                  G_CALLBACK (get_value),
                  GINT_TO_POINTER (1));
gtk_box_pack_start (GTK_BOX (hbox), button, TRUE, TRUE, 5);

button = gtk_button_new_with_label ("Value as Float");
g_object_set_data (G_OBJECT (button), "user_data", val_label);
g_signal_connect (G_OBJECT (button), "clicked",
                  G_CALLBACK (get_value),
                  GINT_TO_POINTER (2));
gtk_box_pack_start (GTK_BOX (hbox), button, TRUE, TRUE, 5);

gtk_box_pack_start (GTK_BOX (vbox), val_label, TRUE, TRUE, 0);
```

```
  gtk_label_set_text (GTK_LABEL (val_label), "0");

  hbox = gtk_hbox_new (FALSE, 0);
  gtk_box_pack_start (GTK_BOX (main_vbox), hbox, FALSE, TRUE, 0);

  button = gtk_button_new_with_label ("Close");
  g_signal_connect_swapped (G_OBJECT (button), "clicked",
                            G_CALLBACK (gtk_widget_destroy),
                            G_OBJECT (window));
  gtk_box_pack_start (GTK_BOX (hbox), button, TRUE, TRUE, 5);

  gtk_widget_show_all (window);

  /* Enter the event loop */
  gtk_main ();

  return 0;
}
```

## Imágenes (GtkImage)

GtkImage es un widget que nos permitirá mostrar una imagen por pantalla. Existen varios tipos de objeto que pueden ser mostrados como una imagen; lo más típico es cargar una imagen ya preexistente desde fichero, y mostrarla a continuación. Existe una función muy útil para hacer esto, `gtk_image_new_from_file()`, que podemos usar así:

```
GtkWidget *image;
image = gtk_image_new_from_file ("myfile.png");
```

Internamente, gtk_image_new_from_file está haciendo uso de GdkPixbuf para cargar un buffer de pixels, proveniendo este buffer de imagenes de tipo GIF, PNG, JPEG, BMP, etc.

Si en el ejemplo anterior el fichero no se carga correctamente, la imagen contendrá un icono de "imagen rota" similiar al que muestran muchos navegadores web cuando no encuentran en el servidor la imagen a mostrar. Si deseáramos gestionar nosotros mismos posibles errores al cargar una imagen, por ejemplo, si quisiéramos mostrar un mensaje de error cuando detectaramos algún problema, deberíamos cargar la imagen con `gdk_pixbuf_new_from_file()`, y luego crear el widget GtkImage con `gtk_image_new_from_pixbuf()`.

El fichero de imagen que cargamos podría contener alguna animación y, en ese caso, GtkImage también la mostrará, usando GdkPixbufAnimation en lugar de una imagen estática.

`GtkImage` es una subclase de `GtkMisc`, por lo que es posible alinear la imagen (center, left, right) y añadirle padding, con los métodos de `GtkMisc`.

Veamos un pequeño ejemplo de programa que carga una pequeña imagen de nombre gnome.png y la muestra en una ventana:

En nuestro programa de ejemplo, crearemos un objeto GtkImage usando la función `gtk_image_new_from_file()`. El objeto `GtkImage` cargará nuestra imagen automáticamente, usando internamente GdkPixbuf:

```
#include <gtk/gtk.h>

int
```

```
main (int argc, char **argv)
{
  GtkWidget *window;
  GtkWidget *image;

  gtk_init (&argc, &argv);
  window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  g_signal_connect (G_OBJECT (window), "destroy", gtk_main_quit, NULL);

  image = gtk_image_new_from_file ("gnome.gif");
  gtk_container_add (GTK_CONTAINER (window), image);

  gtk_widget_show_all (window);

  gtk_main ();

  return 0;
}
```

**Figura 7-10. Resultado de ejecutar el programa GtkImage.c**

`GtkImage` es un widget sin ventana asoaciada, (no tiene un GdkWindow propio), por lo que, por defecto, no recibe eventos. Si deseamos programar algo con un GtkImage que reciba eventos, como clics de botón, deberemos poner la imagen dentro de GtkEventBox, y luego conectar la imagen a los eventos de dicha caja. Veamos un ejemplo:

```
static gboolean
button_press_callback (GtkWidget      *event_box,
                       GdkEventButton *event,
                       gpointer        data)
{
  g_print ("Event box clicked at coordinates %f,%f\n",
           event->x, event->y);

  /* Returning TRUE means we handled the event, so the signal
   * emission should be stopped (don't call any further
   * callbacks that may be connected). Return FALSE
   * to continue invoking callbacks.
   */
  return TRUE;
}

static GtkWidget*
create_image (void)
{
  GtkWidget *image;
  GtkWidget *event_box;

  image = gtk_image_new_from_file ("myfile.png");
```

```
    event_box = gtk_event_box_new ();

    gtk_container_add (GTK_CONTAINER (event_box), image);

    g_signal_connect (G_OBJECT (event_box),
                      "button_press_event",
                      G_CALLBACK (button_press_callback),
                      image);

    return image;
  }
```

Al gestionar eventos generados en un GtkEventBox, recuerda que las coordenadas en la imagen podrían ser diferentes a las de la caja, debido al alineamiento y el padding de la imagen dentro de la caja. La forma más simple de arreglar este detalle consiste en ajustar el alineamiento a 0.0 (izquierda/arriba), y establecer el padding a cero. En ese caso, el origen de la imagen será el mismo que el origen de la caja que recibe eventos.

En ocasiones, una aplicación podría querer evitar tener dependencias de ficheros de datos externos, como los ficheros de imágenes que carga dentro de un menú. En este caso, GTK+ ofrece un programa para evitar estas dependencias: gdk-pixbuf-csource. Este programa permite convertir una imagen en una declaración de variable en C, que luego puede ser cargada en un objeto GdkPixbuf, usando la función `gdk_pixbuf_new_from_inline()`.

## Barras de progreso (GtkProgressBar)

Las barras de progreso se usan para mostrar el estado de una operación. Son muy fáciles de usar, como puedes observar con el cÃ³dig que mostramos a continuación. Pero comencemos primero con las llamadas para crear una nueva barra de progreso.

```
GtkWidget *gtk_progress_bar_new( void );
```

Ahora que se ha creado la barra de progreso, podremos usarla con funciones como la siguiente:

```
void gtk_progress_bar_set_fraction ( GtkProgressBar *pbar,
                                     gdouble        fraction );
```

El primer argumento es la barra de progreso con la que deseas operar, y el segundo argumento es la cantidad "completada", es decir, la cantidad de relleno de la barra de progreso, un valor de 0 a 100%. Se le pasa como parámetro a la función, en forma de un número real entre 0 y 1.

GTK v1.2 ha añadido nueva fucionalidad a la barra de progreso que le permite mostrar su valor de diferentes formas, e informar al usuario de su valor actual y su rango.

Una barra de progreso puede ser definida con una orientación determinada, usando la función:

```
void gtk_progress_bar_set_orientation( GtkProgressBar *pbar,
                                       GtkProgressBarOrientation orientation );
```

El argumento orientación puede tomar cualquiera de los siguientes valores, para indicar la dirección en la que dicha barra de progreso se mueve:

- GTK_PROGRESS_LEFT_TO_RIGHT
- GTK_PROGRESS_RIGHT_TO_LEFT
- GTK_PROGRESS_BOTTOM_TO_TOP
- GTK_PROGRESS_TOP_TO_BOTTOM

Además de indicar la cantidad de progreso ocurrido, la barra de progreso puede definirse simplemente para indicar que hay algún tipo de actividad. Esto puede ser útil en situaciones donde el progreso realizado no puede ser medido en un rango de valores. La siguiente función indica que se ha realizado algun tipo de progreso:

```
void gtk_progress_bar_pulse ( GtkProgressBar *progress );
```

Este tipo de barra permite definir el tamañoo del paso que se dibujará en cada pulso:

```
void gtk_progress_bar_set_pulse_step( GtkProgressBar *pbar,
                                      gdouble         fraction );
```

Cuando no se está en modo actividad, la barra de progreso también puede mostrar una cadena de texto configurable, usando la siguiente función:

```
void gtk_progress_bar_set_text( GtkProgressBar *progress,
                                const gchar    *text );
```

Obsérvese que

```
gtk_progress_set_text()
```

no soporta el formateo de texto usando printf() como lo hacía la barra de la versión GTK+ 1.2.

Puedes evitar mostrar esta cadena de texto llamando a la función

```
gtk_progress_bar_set_text()
```

otra vez con NULL como segundo argumento.

Puedes obtener la cadena de texto actualmente en uso por la barra de progreso usando la siguiente función (no liberar la memoria de la cadena devuelta):

```
const gchar *gtk_progress_bar_get_text( GtkProgressBar *pbar );
```

Las barras de progreso se usan normalmente con timeouts u otras funciones para dar la sensación de multitarea (ver sección sobre "Timeouts, I/O y funciones Idle). Todas ellas usarán `gtk_progress_bar_set_fraction()` ó `gtk_progress_bar_pulse` de la misma forma.

En el siguiente ejemplo podemos ver cómo mostrar una barra de progreso, actualizada usando timeouts. Este código muestra cómo resetear la barra de progreso.

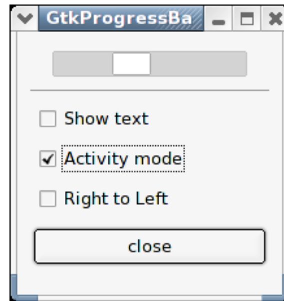



**Figura 7-11. Barra de progreso, actualizada mediante el uso de timeouts.**

```
#include <gtk/gtk.h>

typedef struct _ProgressData {
  GtkWidget *window;
  GtkWidget *pbar;
  int timer;
  gboolean activity_mode;
} ProgressData;

/* Update the value of the progress bar so that we get
 * some movement */
static gboolean progress_timeout( gpointer data )
{
  ProgressData *pdata = (ProgressData *)data;
  gdouble new_val;

  if (pdata->activity_mode)
    gtk_progress_bar_pulse (GTK_PROGRESS_BAR (pdata->pbar));
  else
    {
      /* Calculate the value of the progress bar using the
       * value range set in the adjustment object */

      new_val = gtk_progress_bar_get_fraction (GTK_PROGRESS_BAR (pdata->pbar)) + 0.01;

      if (new_val > 1.0)
        new_val = 0.0;

      /* Set the new value */
      gtk_progress_bar_set_fraction (GTK_PROGRESS_BAR (pdata->pbar), new_val);
    }

  /* As this is a timeout function, return TRUE so that it
   * continues to get called */
  return TRUE;
}

/* Callback that toggles the text display within the progress bar trough */
static void toggle_show_text( GtkWidget    *widget,
                              ProgressData *pdata )
{
  const gchar *text;

  text = gtk_progress_bar_get_text (GTK_PROGRESS_BAR (pdata->pbar));
  if (text && *text)
    gtk_progress_bar_set_text (GTK_PROGRESS_BAR (pdata->pbar), "");
  else
    gtk_progress_bar_set_text (GTK_PROGRESS_BAR (pdata->pbar), "some text");
```

```
}

/* Callback that toggles the activity mode of the progress bar */
static void toggle_activity_mode( GtkWidget    *widget,
                                  ProgressData *pdata )
{
  pdata->activity_mode = !pdata->activity_mode;
  if (pdata->activity_mode)
      gtk_progress_bar_pulse (GTK_PROGRESS_BAR (pdata->pbar));
  else
      gtk_progress_bar_set_fraction (GTK_PROGRESS_BAR (pdata->pbar), 0.0);
}


/* Callback that toggles the orientation of the progress bar */
static void toggle_orientation( GtkWidget    *widget,
                                ProgressData *pdata )
{
  switch (gtk_progress_bar_get_orientation (GTK_PROGRESS_BAR (pdata->pbar))) {
  case GTK_PROGRESS_LEFT_TO_RIGHT:
    gtk_progress_bar_set_orientation (GTK_PROGRESS_BAR (pdata->pbar),
                                      GTK_PROGRESS_RIGHT_TO_LEFT);
    break;
  case GTK_PROGRESS_RIGHT_TO_LEFT:
    gtk_progress_bar_set_orientation (GTK_PROGRESS_BAR (pdata->pbar),
                                      GTK_PROGRESS_LEFT_TO_RIGHT);
    break;
  default:;
    /* do nothing */
  }
}


/* Clean up allocated memory and remove the timer */
static void destroy_progress( GtkWidget    *widget,
                              ProgressData *pdata)
{
    g_source_remove (pdata->timer);
    pdata->timer = 0;
    pdata->window = NULL;
    g_free (pdata);
    gtk_main_quit ();
}

int main( int   argc,
          char *argv[])
{
    ProgressData *pdata;
    GtkWidget *align;
    GtkWidget *separator;
    GtkWidget *table;
    GtkWidget *button;
    GtkWidget *check;
    GtkWidget *vbox;

    gtk_init (&argc, &argv);

    /* Allocate memory for the data that is passed to the callbacks */
    pdata = g_malloc (sizeof (ProgressData));

    pdata->window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_resizable (GTK_WINDOW (pdata->window), TRUE);

    g_signal_connect (G_OBJECT (pdata->window), "destroy",
                      G_CALLBACK (destroy_progress),
                      (gpointer) pdata);
```

```
gtk_window_set_title (GTK_WINDOW (pdata->window), "GtkProgressBar");
gtk_container_set_border_width (GTK_CONTAINER (pdata->window), 0);

vbox = gtk_vbox_new (FALSE, 5);
gtk_container_set_border_width (GTK_CONTAINER (vbox), 10);
gtk_container_add (GTK_CONTAINER (pdata->window), vbox);
gtk_widget_show (vbox);

/* Create a centering alignment object */
align = gtk_alignment_new (0.5, 0.5, 0, 0);
gtk_box_pack_start (GTK_BOX (vbox), align, FALSE, FALSE, 5);
gtk_widget_show (align);

/* Create the GtkProgressBar */
pdata->pbar = gtk_progress_bar_new ();

gtk_container_add (GTK_CONTAINER (align), pdata->pbar);
gtk_widget_show (pdata->pbar);

/* Add a timer callback to update the value of the progress bar */
pdata->timer = g_timeout_add (100, progress_timeout, pdata);

separator = gtk_hseparator_new ();
gtk_box_pack_start (GTK_BOX (vbox), separator, FALSE, FALSE, 0);
gtk_widget_show (separator);

/* rows, columns, homogeneous */
table = gtk_table_new (2, 3, FALSE);
gtk_box_pack_start (GTK_BOX (vbox), table, FALSE, TRUE, 0);
gtk_widget_show (table);

/* Add a check button to select displaying of the trough text */
check = gtk_check_button_new_with_label ("Show text");
gtk_table_attach (GTK_TABLE (table), check, 0, 1, 0, 1,
                  GTK_EXPAND | GTK_FILL, GTK_EXPAND | GTK_FILL,
                  5, 5);
g_signal_connect (G_OBJECT (check), "clicked",
                  G_CALLBACK (toggle_show_text),
                  (gpointer) pdata);
gtk_widget_show (check);

/* Add a check button to toggle activity mode */
check = gtk_check_button_new_with_label ("Activity mode");
gtk_table_attach (GTK_TABLE (table), check, 0, 1, 1, 2,
                  GTK_EXPAND | GTK_FILL, GTK_EXPAND | GTK_FILL,
                  5, 5);
g_signal_connect (G_OBJECT (check), "clicked",
                  G_CALLBACK (toggle_activity_mode),
                  (gpointer) pdata);
gtk_widget_show (check);

/* Add a check button to toggle orientation */
check = gtk_check_button_new_with_label ("Right to Left");
gtk_table_attach (GTK_TABLE (table), check, 0, 1, 2, 3,
                  GTK_EXPAND | GTK_FILL, GTK_EXPAND | GTK_FILL,
                  5, 5);
g_signal_connect (G_OBJECT (check), "clicked",
                  G_CALLBACK (toggle_orientation),
                  (gpointer) pdata);
gtk_widget_show (check);

/* Add a button to exit the program */
button = gtk_button_new_with_label ("close");
g_signal_connect_swapped (G_OBJECT (button), "clicked",
                          G_CALLBACK (gtk_widget_destroy),
                          G_OBJECT (pdata->window));
```

```
    gtk_box_pack_start (GTK_BOX (vbox), button, FALSE, FALSE, 0);

    /* This makes it so the button is the default. */
    GTK_WIDGET_SET_FLAGS (button, GTK_CAN_DEFAULT);

    /* This grabs this button to be the default button. Simply hitting
     * the "Enter" key will cause this button to activate. */
    gtk_widget_grab_default (button);
    gtk_widget_show (button);

    gtk_widget_show (pdata->window);

    gtk_main ();

    return 0;
}
```

## Barras de estado (GtkStatusBar)

Las barras de estado son simples widgets usados para mostrar un mensaje de texto. Funcionan como una pila (estructura software) que guarda los mensajes que metemos en ella, de tal forma que si extraermos el último (pop) la barra de estado mostrará el anterior mensaje de texto de la pila.

Para permitir que diferentes partes de una misma aplicación usen la misma barra de estado para mostrar sus mensajes, el widget barra de estado asigna Identificadores de Contexto, que son usados para identificar a los diferentes "usuarios". El mensaje que se encuentre en lo alto de la pila será el que se muestre en pantalla, sin importar a qué Contexto pertenece. Los mensajes se ordenan como en toda pila, es decir, el último en entrar es el primero en salir, no en función del Identificador de Contexto.

Podemos crear una nueva barra de estado con la función: `GtkWidget *gtk_statusbar_new( void );`

Podemos solicitar un nuevo Identificador de Contexto mediante una llamada a la siguiente función, con una pequeña descripción textual del contexto: `guint gtk_statusbar_get_context_id( GtkStatusbar *statusbar, const gchar *context_description );`

Disponemos de tres funciones distintas para operar con una barra de estado: `guint gtk_statusbar_push( GtkStatusbar *statusbar, guint context_id, const gchar *text );`

`void gtk_statusbar_pop( GtkStatusbar *statusbar) guint context_id );`

`void gtk_statusbar_remove( GtkStatusbar *statusbar, guint context_id, guint message_id );`

La primera, gtk_statusbar_push(), se usa para añadir un nuevo mensaje a la barra de estado. Devuelve un Identificador de Mensaje, que puede ser pasado más adelante como parámetro a la función gtk_statusbar_remove para eliminar el mensaje que se asocie con el Identificador de Contexto y de Mensaje de la pila de la barra de estado.

La función gtk_statusbar_pop() elimina el mensaje más alto en la pila asociado al Identificador de Contexto que le llega como parámetro.

Además de mostrar mensajes, una barra de estado también puede mostrar un control para aumentar o disminuir el tamaño de la barra, al igual que redimensionamos el tamaño de una ventana, estirando con el ratón del borde de la misma.

Las siguientes funciones controlan la visualización de este control: `void gtk_statusbar_set_has_resize_grip( GtkStatusbar *statusbar, gboolean setting );`

```
gboolean gtk_statusbar_get_has_resize_grip( GtkStatusbar *statusbar
);
```

El siguiente ejemplo crea una barra de estado y dos botones, uno para insertar elementos en la barra de estado y el otro para sacar el último insertado.

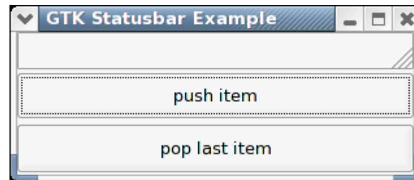



**Figura 7-12. Ejemplo de utilización del widget statusbar**

```
#include <stdlib.h>
#include <gtk/gtk.h>
#include <glib.h>

GtkWidget *status_bar;

static void push_item( GtkWidget *widget,
                       gpointer   data )
{
  static int count = 1;
  gchar *buff;

  buff = g_strdup_printf ("Item %d", count++);
  gtk_statusbar_push (GTK_STATUSBAR (status_bar), GPOINTER_TO_INT (data), buff);
  g_free (buff);
}

static void pop_item( GtkWidget *widget,
                      gpointer   data )
{
  gtk_statusbar_pop (GTK_STATUSBAR (status_bar), GPOINTER_TO_INT (data));
}

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

    GtkWidget *window;
    GtkWidget *vbox;
    GtkWidget *button;

    gint context_id;

    gtk_init (&argc, &argv);

    /* create a new window */
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_widget_set_size_request (GTK_WIDGET (window), 200, 100);
    gtk_window_set_title (GTK_WINDOW (window), "GTK Statusbar Example");
    g_signal_connect (G_OBJECT (window), "delete_event",
                      G_CALLBACK (exit), NULL);

    vbox = gtk_vbox_new (FALSE, 1);
    gtk_container_add (GTK_CONTAINER (window), vbox);
```

```
    gtk_widget_show (vbox);

    status_bar = gtk_statusbar_new ();
    gtk_box_pack_start (GTK_BOX (vbox), status_bar, TRUE, TRUE, 0);
    gtk_widget_show (status_bar);

    context_id = gtk_statusbar_get_context_id(
                          GTK_STATUSBAR (status_bar), "Statusbar example");

    button = gtk_button_new_with_label ("push item");
    g_signal_connect (G_OBJECT (button), "clicked",
                      G_CALLBACK (push_item), GINT_TO_POINTER (context_id));
    gtk_box_pack_start (GTK_BOX (vbox), button, TRUE, TRUE, 2);
    gtk_widget_show (button);

    button = gtk_button_new_with_label ("pop last item");
    g_signal_connect (G_OBJECT (button), "clicked",
                      G_CALLBACK (pop_item), GINT_TO_POINTER (context_id));
    gtk_box_pack_start (GTK_BOX (vbox), button, TRUE, TRUE, 2);
    gtk_widget_show (button);

    /* always display the window as the last step so it all splashes on
     * the screen at once. */
    gtk_widget_show (window);

    gtk_main ();

    return 0;
}
```

## Botones

### GtkButton

Existen varias formas de crear botones. Se puede usar la función `gtk_button_new_with_label()` o `gtk_button_new_with_mnemonic()` para crear un botón con etiqueta, `gtk_button_new_with_stock()` para crear un botón que contenga la imagen y el texto desde un stock item, o `gtk_button_new()` para crear un botón en blanco. Si se desea, se puede añadir una etiqueta o una imagen en este nuevo botón. Para hacerlo, se crea una nueva caja, se empaquetan los objetos en la caja usando `gtk_box_pack_start()`, y se usa `gtk_container_add()` para incluir la caja en el botón.

Aquí se muestra un ejemplo de uso de `gtk_button_new()`, para crear un botón que contenga una imagen y una etiqueta. Se ha separado la parte del código para crear una caja, para poder reusarlo otros programas. Habrá ejemplos más específicos del uso de imágenes, a medida que se avanza en el libro.

La función `xpm_label_box()` puede ser usada para empaquetar imágenes y etiquetas en cualquier widget que pueda ser un contenedor.

El widget `GtkButton`, dispone de las siguiente señales:

- *pressed* - emitido cuando el puntero es pulsado en el widget `GtkButton`.
- *released* - emitido cuando el puntero es levantado del widget `GtkButton`.

- *clicked* - emitido cuando el puntero el pulsado y levantado del widget `GtkButton`.
- *enter* - emitido cuando el puntero entra en el widget `GtkButton`.
- *leave* - emitido cuando el puntero abandona el widget `GtkButton`.

## GtkToggleButton

Este tipo de botones derivan de los normales y son muy similares, con la execepción que siempre estarán en un estado de los dos posibles, alternados mediante un click. Puede estar pulsado, y cuando se vuelva a hacer click de nuevo, se despulsará. Un nuevo click, y volverá a estar pulsado.

El widget `GtkToggleButton` es la base para los widgets `GtkCheckButton` y `GtkRadioButton`, de tal forma, que muchas de las llamadas del `GtkToggleButton` son heredadas por estos.

Para crear un `GtkButtonToggle`, se usan las siguientes funciones:

```
GtkWidget* gtk_toogle_button_new   (void);
GtkWidget* gtk_toogle_button_new_with_label (const gchar *label);
GtkWidget* gtk_toogle_button_new_with_mnemonic (const gchar *label);
```

Como se puede ver, estas funciones son similares a las de los botones normales. La primera crea un botón, mientras que la segunda crea un botón con una etiqueta ya empaquetada. La variante `_mnemonic` analiza la etiqueta, para incluir caracteres mnemotécnicos.

Para saber cual es el estado de un `GtkToggleButton`, incluidos `GtkRadioButton` y `GtkCheckButton`, se usa una de las macros del siguiente ejemplo. Estas comprueban el estado del `GtkToggleButton` accediendo a un campo de la estructura del botón, despues de usar la macro GTK_TOGGLE_BUTTON, para moldear el puntero a widget, a un puntero a `GtkToggleButton`. La señal de interés emitida por el `GtkToggleButton` y sus hijos, es la señal *toggled*. Para comprobar el estado de estos botones, normalmente se utiliza un capturador para la señal *toggled* y se accede a la estructura para determinar su estado. La función de callback, será algpo parecido a:

```
void toggle_button_callback (GtkWidget *widget, gpointer data)
{
 if (gtk_toggle_button_get_activate (GTK_TOGGLE_BUTTON (widget)))
 {
  /* Si el control llega aquí, es que el botón se encuentra pulsado */
 } else {
  /* Si se llega aquí, el botón no está pulsado */
 }
}
```

Para forzar el estado de un `GtkToggleButton`, se usa la siguiente función:

```
void gtk_toggle_button_set_activate (GtkToggleButton *toggle_button,
         gboolean        is_activate);
```

El primer argumento es el botón al que se quiere imponer un estado, y el segundo será TRUE cuando se quiere que el botón esté pulsado, o FALSE cuando no lo esté. Por defecto será no pulsado o FALSE.

Cabe resaltar, que cuando se usa la función `gtk_toogle_button_set_activate ()`, y se cambia el estado actual, hace que el botón emita las señales *clicked* y *toggled*, según corresponda.

```
gboolean gtk_toogle_button_get_activate (GtkToggleButton *toggle_button);
```

Esta función devuelve el estado del `GtkToggleButton`, mediante un valor gboolean TRUE o FALSE.

## GtkCheckButton

El `GtkCheckButton` hereda muchas funciones y propiedades del `GtkToggleButton` anterior, pero tiene diferente apariencia. En vez de ser botones con el texto incluido, son pequeños cuadrados con el texto a su derecha. Son usados normalmente para seleccionar o quitar opciones en las aplicaciones.

Las funciones para crearlos son similares a las usadas para el `GtkButton`

```
GtkWidget* gtk_check_button_new   (void);
GtkWidget* gtk_check_button_new_with_label  (const gchar *label);
GtkWidget* gtk_check_button_new_with_mnemonic (const gchar *label);
```

La función `gtk_check_button_with_label()` crea un botón con la etiqueta a su derecha.

Para comprobar el estado de un `GtkCheckButton` se usan las mismas funciones que para el `GtkToggleButton`.

## GtkRadioButton

`GtkRadioButton` es similar a `GtkCheckButton` salvo porque éstos se encuentran agrupados de tal forma que sólo uno de ellos puede estar seleccionado. Esto sirve para aplicaciones donde se tiene que seleccionar una opción dentro de una lista.

Para crear un `GtkRadioButton` se usa una de las siguientes funciones:

```
GtkWidget* gtk_radio_button_new   (GSList *group );
GtkWidget* gtk_radio_button_new_from_widget (GtkRadioButton *group );
GtkWidget* gtk_radio_button_new_with_label  (GSList *group,
  const gchar  *label );
GtkWidget* gtk_radio_button_new_with_label_from_widget (GtkRadioButton *group,
 const gchar    *label );
GtkWidget* gtk_radio_button_new_with_mnemonic (GSList *group,
 const gchar  *label );
GtkWidget* gtk_radio_button_new_with_mnemonic_from_widget (GtkRadioButton *group,
 const gchar  *label );
```

Hay que resaltar que hay un nuevo argumento en estas funciones. Necesitan una lista para poder funcionar correctamente. La primera vez que se llame a `gtk_radio_button_new ()` o `gtk_radio_button_new_with_label ()`, se le debe pasar NULL como primer argumento. Luego, se creará un grupo usando la función:

```
GSList* gtk_radio_button_get_group( GtkRadioButton *radio_button);
```

Lo más importante que hay recordar aquí es que `gtk_radio_button_get_group` debe ser usada cada vez que se añada un nuevo botón, con el botón nuevo previamente usado como argumento. El resultado, es utilizado para la siguiente llamada a `gtk_radio_button_new ()` o `gtk_radio_button_new_with_label ()`. Esto permite establecer una cadena de botones. El siguiente ejemplo debería de aclarar esto.

Se puede suprimir todo esto usando la siguiente sintaxis, que elimina la necesidad de la variable para mantener la lista de botones.

```
button2 = gtk_radio_button_new_with_label(
   gtk_radio_button_get_group (GTK_RADIO_BUTTON (button1)),
          "button2");
```

La variante `_from_widget` de las funciones de creación permite acortar esto más aún, omitiendo la llamada a `gtk_radio_button_get_group`. Esta forma es usada en el ejemplo para crear el tercer botón.

```
button2 = gtk_radio_button_new_with_label_from_widget(
                  GTK_RADIO_BUTTON (button1),
    "button2");
```

Es también una buena idea establecer el botón que debe de estar por defecto pulsado con:

```
void gtk_toggle_button_set_active( GtkToggleButton *toggle_button,
                                   gboolean        state );
```

Esto se describe en la zona de `GtkToggleButton` y funciona exactamente de la misma forma. Una vez que los `GtkRadioButton` están agrupados, sólo uno del grupo debe de estar seleccionado. Si el usuario selecciona otro `GtkRadioButton`, y luego en otro, el primero emitirá la señal *toggled* (para indicar que ha sido desactivado) y el segundo la señal *toggled* (para indicar que ha sido activado).

El siguiente ejemplo muestra la creación de un grupo de tres `GtkRadioButtons`:

# Menús y barras de herramientas

## Menús

Existen dos formas de crear menús: Una sencilla y otra un poco más complicada, ya que utiliza el método manual. La forma sencilla utiliza la factoría de menús (`GtkItemFactory`), que facilita la creación de menús. El método manual crea todos los menús usando las llamadas directamente. Aunque usando `GtkItemFactory` facilita la creación de menús, no es posible añadir imágenes o el carácter '/'.

### Creación Manual de Menús

Para crear un Menú, se necesitan los siguientes widgets:

- Item del menú: lo que el usuario selecciona, ej. "Guardar".
- Menú: actúa como contenedor de los item del menú, ej. "Archivo".
- Barra de menús: un contenedor para cada menú individual.

Las siguientes funciones se usan para crear menús y barras de menús:

```
GtkWidget *gtk_menu_bar_new (void); // crea una barra de menus.
GtkWidget *gtk_menu_new (void);  // devuelve un puntero al nuevo menú. No es neces
                                 // mostrarlo (con gtk_widget_show()), ya que es s
                                 // un contenedor para los items del menú.
```

Las siguientes funciones crean items de menú que son colocados dentro del menú y de la barra de menús.

```
GtkWidget *gtk_menu_item_new (void); // crea un item de menú sin título
GtkWidget *gtk_menu_item_new_with_label (const char *label); // crea un item de me
GtkWidget *gtk_menu_item_new_with_mnemnonic (const char *label); // crea un item d
```

Una vez se han creado los items de menú, hay que colocarlos dentro del widget menú usando la función `gtk_menu_append`. Además, para saber cuándo el item ha sido seleccionado por el usuario es necesario conectar la señal activada.

En el siguiente ejemplo, se muestra el código necesario para crear un menú Archivo estándar, con las opciones Abrir, Guardar y Salir.

```
file_menu = gtk_menu_new ();    /* Crea un menú */

/* Crea items de menú */
open_item = gtk_menu_item_new_with_label ("Abrir");
save_item = gtk_menu_item_new_with_label ("Guardar");
quit_item = gtk_menu_item_new_with_label ("Salir");

/* Añade los items al menú */
gtk_menu_append (GTK_MENU (file_menu), open_item);
gtk_menu_append (GTK_MENU (file_menu), save_item);
gtk_menu_append (GTK_MENU (file_menu), quit_item);

/* Attach the callback functions to the activate signal */
g_signal_connect_swapped (G_OBJECT (open_item), "activate",
                          G_CALLBACK (menuitem_response),
                          (gpointer) "file.open");
g_signal_connect_swapped (G_OBJECT (save_item), "activate",
                          G_CALLBACK (menuitem_response),
                          (gpointer) "file.save");

/* We can attach the Quit menu item to our exit function */
g_signal_connect_swapped (G_OBJECT (quit_item), "activate",
                          G_CALLBACK (destroy),
                          (gpointer) "file.quit");

/* Es necesario mostrar los items de menú */
gtk_widget_show (open_item);
gtk_widget_show (save_item);
gtk_widget_show (quit_item);
```

El siguiente paso de este ejemplo es la creación de una barra de menús y un item para la opción Archivo, al que se le añadirá el menú (file_menu).

```
/* Crea una barra de menús */
menu_bar = gtk_menu_bar_new ();
gtk_container_add (GTK_CONTAINER (window), menu_bar);
gtk_widget_show (menu_bar);

/* Crea el item "Archivo" */
file_item = gtk_menu_item_new_with_label ("Archivo");
gtk_widget_show (file_item);
```

Lo siguiente es asociar el menú (file_menu) con la opción Archivo (file_item), para ello se necesita la función:

```
void gtk_menu_item_set_submenu (GtkMenuItem *menu_item,
                                GtkWidget   *submenu);
```

En el ejemplo:

```
gtk_menu_item_set_submenu (GTK_MENU_ITEM (file_item), file_menu);
```

Por último, se añade el menú a la barra de menús, usando la función:

```
void gtk_menu_bar_append (GtkMenuBar *menu_bar,
                          GtkWidget  *menu_item);
```

En el ejemplo:

```
gtk_menu_bar_append (GTK_MENU_BAR (menu_bar), file_item);
```

Para alinear el menú a la derecha de la barra de menús, como por ejemplo el menú Ayuda, se usa la función void `gtk_menu_item_right_justify`.

Para que queden claros todos los conceptos, a continuación se resumen los pasos a seguir para la creación de una barra de menús con menús.

1. Crear un menú con la función `gtk_menu_new`
2. Crear los items que van a estar en el menú con la función `gtk_menu_item_new`
3. Colocar los items creados en el menú usando la función `gtk_menu_append`
4. Crear un item de menú con la función `gtk_menu_item_new_with_label`. Este será el menú raíz, es decir, el texto que aparece en la barra de menús.
5. Asociar el menú al menú raíz usando la función `gtk_menu_item_set_submenu`.
6. Crear una barra de menús con `gtk_menu_bar_new` que contendrá al menú raíz.
7. Colocar el menú raíz en la barra de menús usando `gtk_menu_bar_append`.

## Menú PopUp (colgante)

Los pasos a seguir para crear un menú popup son muy similares a lo comentado en el apartado anterior. La única diferencia es que el menú popup no se encuentra dentro de una barra de menús, sino que aparece, por ejemplo, porque se ha pulsado el botón izquierdo del ratón.

Para saber cuándo desplegar el menú popup, se necesita crear una función de manejo del evento. Esta función necesita tener el siguiente prototipo:

```
static gint handler (GtkWidget *widget, GdkEvent  *event);
```

El argumento *`event`* especificará cuándo desplegar el menú popup.

Si el evento, en un manejador de eventos, es la pulsación de uno de los botones del ratón, se debe usar como se muestra en el código de ejemplo para pasar información a `gtk_menu_popup`.

Para asociar un manejador de eventos a un widget determinado, se usa la siguiente función:

```
g_signal_connect_swapped (G_OBJECT (widget), "event",
                          G_CALLBACK (handler),
                          G_OBJECT (menu));
```

*widget* es el widget que se va a asociar al menú, mientras que *handler* es la función de manejo del evento, y *menu* es el menú popup creado con la función `gtk_menu_new`.

## Creación de Menús usando GtkItemFactory

Ahora que hemos visto cómo crear menús de forma manual, veamos cómo hacerlo usando un método más simple: usando llamadas a gtk_item_factory.

ItemFactory permite crear un menú a partir de un array de entradas ItemFactory. Esto significa que puedes definir un esquema de tu menú y después crear los widgets menu/menubar con un mínimo de funciones.

### *Entradas ItemFactory*

ItemFactoryEntry es una estructura que permite definir una entrada de menú. Al definir un array de entradas ItemFactoryEntry, habremos creardo un menú completo. La definición de una estructura de elemento ItemFactory tiene el siguiente aspecto:

```
struct _GtkItemFactoryEntry
{
gchar *path;
gchar *accelerator;

GtkItemFactoryCallback callback;
guint                  callback_action;

gchar          *item_type;
};
```

Cada campo define una parte de la entrada del menú. **path* es una cadena que define tanto el nombre como la ruta de un elemento de menú, por ejemplo, "/Fichero/Abrir" sería el nombre de una entrada de menú que vendría bajo la entrada ItemFactory con el path "/Fichero". Obsérvese sin embargo que "/Fichero/Abrir" se mostraría en el menú Fichero como "Abrir". Nótese también que dado que la barra / es el caracter usado para definir la ruta de un menú, no puede ser usada como parte del nombre. Una letra precedida por un caracter de subrayado indica que dicha letra servirá como atajo de teclado (shortcut) cuando dicho menú esté desplegado.

**accelerator* es una cadena que indica la combinación de teclas que puede usarse como atajo de teclado para acceder a ese elemento del menú. La cadena puede estar construída tanto por un simple caracter como por una combinación de teclas de función con un caracter simple. No se diferencian mayúsculas de minúsculas.

Las teclas de función disponibles son: <ALT> - alt <CTL> ó <CTRL> ó <CONTROL> - control <MOD1> a <MOD5> - modn <SHFT> o <SHIFT> - shift

Ejemplos: '<ConTroL>a' '<SHFT><ALT><CONTROL>X'

*callback* es la función a la que se llamará cuando el elemento de menú emita la señal "activate".

El valor de *callback_action* se pasará a la función de callback.

*item_type* es una cadena que describe el tipo de widget que será empaquetado en el menú. Puede ser cualquiera de los siguientes: NULL o "" o "<Item>" - crea un item simple "<Title>" - crea un item de tipo título "<CheckItem>" - crea un item de tipo Check "<ToggleItem>" - crea un item conmutador "<RadioItem>" - crea un item de tipo radial (raíz) "Path" - crea un item de tipo radial hermana "<Tearoff>" - crea un tearoff "<Separator>" - crea un separador "<Branch>" - crea un item para contener submenús (opcional) "<LastBranch>" - crea un submenú justificado a la derecha

(como los submenús ayuda" "<StockItem>" - crea un item simple con una imagen de fábrica (disponible de serie en GTK) véase gtkstock.h para una lista de elementos stock

Nótese que <LastBranch> sólo es útil para un submenú de una barra de menús.

### *Descripción de la retrollamada (callback)*

La función de retrollamada para una entrada ItemFactory puede tomar dos formas. Si `callback_action` es cero:

```
void callback( void )
```

Si no, será la siguiente:

```
void callback( gpointer    callback_data,
        guint       callback_action,
        GtkWidget  *widget )
```

`callback_data` es un puntero a la estructura que queramos, y se define durante la llamada a gtk_item_factory_create_items().

`callback_action` tiene el mismo valor que `callback_action` en la entrada ItemFactory.

`*widget` es un puntero a un widget elemento de menú (descritos en la sección la sección de nombre *Entradas ItemFactory*).

### *Ejemplos de entradas ItemFactory*

Cómo crear una entrada simple de menú:

```
GtkItemFactoryEntry entry = {"/_Fichero/_Abrir...", "<CTRL>A", print_hello, 0, "<Item>"};
```

Esto definirá una entrada simple de menú "/Fichero/Abrir" (mostrada como "Abrir"), bajo la entrada de menú "/Fichero". Se define además el atajo de teclado control+'A' que al ser pinchado, llama a la función print_hello(). print_hello() tiene la forma void print_hello(void) dado que el campo callback_action field está a cero. Cuando se muestre la 'A' de "Abrir", será subrayada y si el elemento de menú es visible en pantalla, al pulsar la 'A' será activado este item. Obsérverse que también se podría haber usado "Fichero/_Abrir" como la ruta en lugar de "/_Fichero/_Abrir".

Cómo crear una entrada con una callback más compleja:

```
GtkItemFactoryEntry entry = {"/_View/Display _FPS", NULL, print_state, 7,"<CheckItem>"}
```

Esto define una nueva entrada de menú mostrada como "Display FPS" bajo la entrada de menú "View". Cuando se pulse, se hará uan llamada a la función print_state(). Dado que `callback_action` no es cero, la función print_state() tendrá el siguiente prototipo:

```
void print_state( gpointer    callback_data,
    guint       callback_action,
```

```
     GtkWidget  *widget )
```

con el parámetro `callback_action` igual a 7.

Crear un conjunto radiobutton:

```
GtkItemFactoryEntry entry1 = {"/_View/_Low Resolution", NULL, change_resolution,
   1, "<RadioButton>"};
GtkItemFactoryEntry entry2 = {"/_View/_High Resolution", NULL, change_resolution,
   2, "/View/Low Resolution"};
```

`entry1` define un botón radial que al conmutar llamará a la función `change_resolution()` con el parámetro `callback_action` igual a 1. `change_resolution()` es de la forma:

```
void change_resolution(gpointer    callback_data,
                       guint       callback_action,
                       GtkWidget  *widget)
```

`entry2` define un botón radial que pertenece al mismo grupo que `entry1`. Al conmutar llamará a la misma función que la anterior entrada, pero con el parámetro `callback_action` igual a 2. Nótese que el item_type de `entry2` es igual a la ruta de `entry1` pero sin el acelerador ('_'). Si se necesitara otro botón radial en el mismo grupo entonces sería definido de la misma forma que `entry2` con su `item_type` de nuevo igual a "/View/Low Resolution".

### Arrays de ItemFactoryEntry

Un ItemFactoryEntry por sí solo no es algo útil. Es necesario crear una array de estos elementos para definir un menú. A continuación mostramos un ejemplo de cómo podemos declarar un array de este tipo:

```
static GtkItemFactoryEntry entries[] = {
  { "/_File",         NULL,      NULL,         0, "<Branch>" },
  { "/File/tear1",    NULL,      NULL,         0, "<Tearoff>" },
  { "/File/_New",     "<CTRL>N", new_file,     1, "<Item>" },
  { "/File/_Open...", "<CTRL>O", open_file,    1, "<Item>" },
  { "/File/sep1",     NULL,      NULL,         0, "<Separator>" },
  { "/File/_Quit",    "<CTRL>Q", quit_program, 0, "<StockItem>", GTK_STOCK_QUIT } };
```

### Cómo crear un ItemFactory

Un array de elementos GtkItemFactoryEntry define un menú. Una vez definido este array podremos crear la factoría ItemFactory. La función que hace esto es:

```
GtkItemFactory* gtk_item_factory_new( GtkType        container_type,
          const gchar   *path,
          GtkAccelGroup *accel_group );
```

`container_type` puede ser uno de los siguientes: GTK_TYPE_MENU GTK_TYPE_MENU_BAR GTK_TYPE_OPTION_MENU

`container_type` define qué tipo de menú queremos, de tal forma que cuando más adelante lo necesitemos, poder saber si estamos hablando de un menú (por ejemplo para los popups), una barra de menús o una opción de un menú en concreto.

*path* define la ruta a la raíz del menú. Básicamente consiste en un nombre único para la raíz del menú, rodeado de "<>". Esto es imporante para definir luego la nomenclatura de los atajo de teclado y debe ser único, tanto a nivel de menús como a nivel de programa. Por ejemplo, en un programa llamado 'foo', el menú principal debería llamarse "<FooMain>", y un menú pop-up del mismo programa "<FooImagePopUp>", ó similar. Lo que es importante es que sean únicos.

*accel_group* es un puntero a gtk_accel_group. La factoría de elementos define la tabla de aceleradores al generar los menús. Pueden generarse nuevos grupos de aceleradores usando la función `gtk_accel_group_new()`.

Pero esto es sólo el primer paso. Para convertir la información del array de GtkItemFactoryEntry en widgets necesitaremos hacer uso de la siguiente función:

```
void gtk_item_factory_create_items( GtkItemFactory      *ifactory,
                                    guint                n_entries,
                                    GtkItemFactoryEntry *entries,
                                    gpointer             callback_data );
```

**ifactory* es un puntero a la factoría de elementos creada anteriormente. *n_entries* es el número de entradas del array GtkItemFactoryEntry. **entries* es un puntero al array GtkItemFactoryEntry. *callback_data* es lo que se le pasa a todas las funciones callback para todas las entradas con el parámetro *callback_action*distinto de 0.

El grupo de aceleradores ya ha sido formado, por lo tenemos que adjuntarlo a la ventana en la que hemos situado nuestro menú:

```
void gtk_window_add_accel_group( GtkWindow     *window,
                                 GtkAccelGroup *accel_group);
```

### Cómo usar el menú y los elementos de menú

Lo último que debemos estudiar es saber cómo usar el menú. La siguiente función extraé los widgets más relevantes de la factoría ItemFactory:

```
GtkWidget* gtk_item_factory_get_widget( GtkItemFactory *ifactory,
                                        const gchar    *path );
```

Por ejemplo, si una factoría de elementos ItemFactory tiene dos entradas como "/File" y "/File/New", usando en la función anterior un path como "/File", nos devolverá un widget menu de la factoría ItemFactory. Usando un path como "/File/New" nos devolverá un widget elemento de menú. Esto nos permite definir el estado inicial de los elementos de un menú. Por ejemplo, para definir el elemento por defecto de un conjunto de botones radiales a aquel con el path "/Shape/Oval", usaríamos el siguiente trozo de código:

```
gtk_check_menu_item_set_active(
GTK_CHECK_MENU_ITEM (gtk_item_factory_get_item (item_factory, "/Shape/Oval")),
TRUE);
```

Finalmente, para obtener la raíz del menú, usaremos la función `gtk_item_factory_get_item()` con el path "<main>" (o cualquier otro path que hubiéramos usado en `gtk_item_factory_new()`). En caso de que ItemFactory hubiera sido creado con el tipo GTK_TYPE_MENU_BAR este devolvería un widget de tipo barra de menú. Con el tipo GTK_TYPE_MENU, un widget de tipo menú. Y

con el tipo GTK_TYPE_OPTION_MENU, devolvería un widget de tipo opción de menú.

Recordemos que para una entrada definida con el path "/_File", el path a utilizar en la función `gtk_item_factory_get_widget()` es realmente "/File".

Ahora que tenemos una barra de menú o un menú completamente definido, podemos manipularlo de la misma forma que vimos en la sección sobre la sección de nombre *Creación Manual de Menús*crear menús de forma manual.

> **Nota:** Podemos ver un completo ejemplo de la creación y uso de menús usando factorías en el programa menuitemfactory.c del directorio de ejemplos.

## GtkToolbar

Las barras de herramientas se usan generalmente para agrupar varios widgets con el objetivo de simplificar la personalización de su aspecto y colocación. Típicamente, una barra de herramientas consta de una serie de botones con iconos, etiquetas y tooltips, pero es posible colocar cualquier otro widget dentro de una barra de herramientas. Para finalizar, diremos que los elementos de una barra pueden ser colocados en horizontal o vertical y que los botones de la misma pueden mostrar iconos, etiquetas o ambos elementos. Podemos crear una barra de herramientas (como era de esperar) con la siguiente función: GtkWidget *gtk_toolbar_new( void ); Tras crear una barra de herramientas es posible añadirle por delante, por detrás o insertar en una posición determinada simples cadenas de texto o cualquier otro tipo de widget. Para describir un elemento de la barra necesitamos una etiqueta de texto, un texto para la sugerencia ("tooltip"), un texto para la sugerencia privada, un icono para el botón y una función callback. Por ejemplo, para añadir un elemento por delante o por detrás, podríamos usar cualquiera de las siguientes funciones:

```
GtkWidget *  gtk_toolbar_append_item (GtkToolbar *toolbar, const char
*text, const char * tooltip_text, const char * tooltip_private_text,
GtkWidget * icon, GtkSignalFunccallback, gpointeruser_data);
```

`GtkWidget *gtk_toolbar_prepend_item( GtkToolbar *toolbar, const char *text, const char *tooltip_text, const char *tooltip_private_text, GtkWidget *icon, GtkSignalFunc callback, gpointer user_data );` Si el lector desea usar `gtk_toolbar_insert_item()`, el único parámetro adicional que debería de ser especificado es la posición en la que el elemento debería de ser insertado, así: `GtkWidget *gtk_toolbar_insert_item( GtkToolbar *toolbar, const char *text, const char *tooltip_text, const char *tooltip_private_text, GtkWidget *icon, GtkSignalFunc callback, gpointer user_data, gint position );` Para simplificar el proceso de añadir espacios entre los elementos de la barra de herramientas, es posible usar las siguientes funciones: `void gtk_toolbar_append_space( GtkToolbar *toolbar );` `void gtk_toolbar_prepend_space( GtkToolbar *toolbar );` `void gtk_toolbar_insert_space( GtkToolbar *toolbar, gint position );` Si así se requiere, es posible cambiar la orientación y el estilo de una barra de herramientas "al vuelo", usando las siguientes funciones: `void gtk_toolbar_set_orientation( GtkToolbar *toolbar, GtkOrientation orientation );` `void gtk_toolbar_set_style( GtkToolbar *toolbar, GtkToolbarStyle style );` `void gtk_toolbar_set_tooltips( GtkToolbar *toolbar, gint enable );` Donde *orientation* puede ser uno de los siguientes GTK_ORIENTATION_HORIZONTAL o GTK_ORIENTATION_VERTICAL. El estilo se usa para asignar una apariencia

a los elementos de la barra de herramientas, mediante uno de lo siguientes : GTK_TOOLBAR_ICONS, GTK_TOOLBAR_TEXT, ó GTK_TOOLBAR_BOTH.

Para mostrar algunas otras cosas que se pueden hacer con una barra de herramientas, tomemos como ejemplo el siguiente programa (cortaremos el flujo normal del mismo para añadir ciertas explicaciones):

```
       #include <gtk/gtk.h>

/* This function is connected to the Close button or
 * closing the window from the WM */
static gboolean delete_event( GtkWidget *widget,
         GdkEvent *event,
         gpointer data )
{
  gtk_main_quit ();
  return FALSE;
}
```

El comienzo anterior será familiar ya al lector si ha seguido este libro desde el principio. Hay una nota adicional que hacer, y es que incluiremos un gráfico XPM como icono de todos nuestros botones.

```
GtkWidget* close_button; /* This button will emit signal to close
                         * application */
GtkWidget* tooltips_button; /* to enable/disable tooltips */
GtkWidget* text_button,
  * icon_button,
  * both_button; /* radio buttons for toolbar style */
GtkWidget* entry; /* a text entry to show packing any widget into
     * toolbar */
```

De hecho, no se necesitan todos los widgets definidos más arriba, pero los hemos puesto para hacer más didáctico el ejemplo.

```
/* that’s easy... when one of the buttons is toggled, we just
 * check which one is active and set the style of the toolbar
 * accordingly
 * ATTENTION: our toolbar is passed as data to callback ! */
static void radio_event( GtkWidget *widget,
    gpointer data )
{
  if (GTK_TOGGLE_BUTTON (text_button)->active)
    gtk_toolbar_set_style (GTK_TOOLBAR (data), GTK_TOOLBAR_TEXT);
  else if (GTK_TOGGLE_BUTTON (icon_button)->active)
    gtk_toolbar_set_style (GTK_TOOLBAR (data), GTK_TOOLBAR_ICONS);
  else if (GTK_TOGGLE_BUTTON (both_button)->active)
    gtk_toolbar_set_style (GTK_TOOLBAR (data), GTK_TOOLBAR_BOTH);
}

/* even easier, just check given toggle button and enable/disable
 * tooltips */
static void toggle_event( GtkWidget *widget,
     gpointer   data )
{
  gtk_toolbar_set_tooltips (GTK_TOOLBAR (data),
       GTK_TOGGLE_BUTTON (widget)->active );
}
```

Lo de arriba son sólo dos funciones callback que serán llamadas cuando se pulse uno de los botones de la barra de herramientas. El lector debería de estar familiarizado con cosas como ésta al haber leído sobre el uso de "toogle button" y botones radiales.

```
int main (int argc, char *argv[])
{
  /* Here is our main window (a dialog) and a handle for the handlebox */
  GtkWidget* dialog;
  GtkWidget* handlebox;

  /* Ok, we need a toolbar, an icon with a mask (one for all of
     the buttons) and an icon widget to put this icon in (but
     we'll create a separate widget for each button) */
  GtkWidget * toolbar;
  GtkWidget * iconw;

  /* this is called in all GTK application. */
  gtk_init (&argc, &argv);

  /* create a new window with a given title, and nice size */
  dialog = gtk_dialog_new ();
  gtk_window_set_title (GTK_WINDOW (dialog), "GTKToolbar Tutorial");
  gtk_widget_set_size_request (GTK_WIDGET (dialog), 600, 300);
  GTK_WINDOW (dialog)->allow_shrink = TRUE;

  /* typically we quit if someone tries to close us */
  g_signal_connect (G_OBJECT (dialog), "delete_event",
      G_CALLBACK (delete_event), NULL);

  /* we need to realize the window because we use pixmaps for
   * items on the toolbar in the context of it */
  gtk_widget_realize (dialog);

  /* to make it nice we'll put the toolbar into the handle box,
   * so that it can be detached from the main window */
  handlebox = gtk_handle_box_new ();
  gtk_box_pack_start (GTK_BOX (GTK_DIALOG (dialog)->vbox),
        handlebox, FALSE, FALSE, 5);
```

Lo anterior debería de ser similar a cualquier otra aplicación GTK. Simplemente inicializar GTK, crear la ventana, etc. Sólo hay una cosa que merece una especial atención: la caja gestora ("handle box"). La caja gestora es otro tipo de caja que puede ser usada para empaquetar widgets. La diferencia entre ésta y otras cajas típicas es que la caja gestora puede ser desenganchada de una ventana padre (o, más exactamente, la caja gestora permanece enganchada al padre, pero éste se reduce a un muy pequeño rectángulo, mientras que todos sus componentes son heredados por una nueva ventana flotante). Generalmente, es algo deseable disponer de una barra de herramientas que permita ser desenganchada, por lo que la aparición conjunta de estos dos widgets es bastante común.

```
  /* toolbar will be horizontal, with both icons and text, and
   * with 5pxl spaces between items and finally,
   * we'll also put it into our handlebox */
 toolbar = gtk_toolbar_new ();
 gtk_toolbar_set_orientation (GTK_TOOLBAR (toolbar), GTK_ORIENTATION_HORIZONTAL);
 gtk_toolbar_set_style (GTK_TOOLBAR (toolbar), GTK_TOOLBAR_BOTH);
 gtk_container_set_border_width (GTK_CONTAINER (toolbar), 5);
 gtk_container_add (GTK_CONTAINER (handlebox), toolbar);
```

Bien, lo hecho en el párrafo anterior es simplemente una inicialización del widget barra de herramientas.

```
  /* our first item is <close> button */
```

```
iconw = gtk_image_new_from_file ("gtk.xpm"); /* icon widget */
close_button =
  gtk_toolbar_append_item (GTK_TOOLBAR (toolbar), /* our toolbar */
                           "Close",               /* button label */
                           "Closes this app",     /* this button's tooltip */
                           "Private",             /* tooltip private info */
                           iconw,                 /* icon widget */
                           GTK_SIGNAL_FUNC (delete_event), /* a signal */
                           NULL);
gtk_toolbar_append_space (GTK_TOOLBAR (toolbar)); /* space after item */
```

En el código anterior podemos ver el caso más sencillo: añador un botón a la barra de herramientas. Justo antes de añadir un nuevo elemento, debemos construir un widget imagen que sirva como icono de este elemento; este paso deberá ser repetido para cada nuevo item. Justo tras colocar el item añadiremos también un espacio, de tal forma que los siguientes items no se toquen entre sí. Como puede observar, `gtk_toolbar_append_item()` devuelve un puntero al widget botón que acabamos de crear, para poder trabajar con él de la forma habitual.

```
/* now, let's make our radio buttons group... */
iconw = gtk_image_new_from_file ("gtk.xpm");
icon_button = gtk_toolbar_append_element (
    GTK_TOOLBAR (toolbar),
    GTK_TOOLBAR_CHILD_RADIOBUTTON, /* a type of element */
    NULL,                          /* pointer to widget */
    "Icon",                        /* label */
    "Only icons in toolbar",       /* tooltip */
    "Private",                     /* tooltip private string */
    iconw,                         /* icon */
    GTK_SIGNAL_FUNC (radio_event), /* signal */
    toolbar);                      /* data for signal */
gtk_toolbar_append_space (GTK_TOOLBAR (toolbar));
```

En las líneas anteriores comenzamos creando un grupo de botones radiales. Para ello, nos valemos de la función `gtk_toolbar_append_element`. De hecho, usando esta función, podríamos añadir simples ítems de texto o incluso espacios (usando los tipos GTK_TOOLBAR_CHILD_SPACE ó GTK_TOOLBAR_CHILD_BUTTON).En el ejemplo anterior, hemos creado un grupo de botones radiales. Al crear otros botones radiales en este grupo, es necesario disponer de un puntero al botón anterior del grupo, de tal forma que sea posible crear fácilmente una lista de botones (para más información, consulte la sección la sección de nombre *GtkRadioButton*)

```
/* following radio buttons refer to previous ones */
iconw = gtk_image_new_from_file ("gtk.xpm");
text_button =
  gtk_toolbar_append_element (GTK_TOOLBAR (toolbar),
                              GTK_TOOLBAR_CHILD_RADIOBUTTON,
                              icon_button,
                              "Text",
                              "Only texts in toolbar",
                              "Private",
                              iconw,
                              GTK_SIGNAL_FUNC (radio_event),
                              toolbar);
gtk_toolbar_append_space (GTK_TOOLBAR (toolbar));

iconw = gtk_image_new_from_file ("gtk.xpm");
both_button =
  gtk_toolbar_append_element (GTK_TOOLBAR (toolbar),
                              GTK_TOOLBAR_CHILD_RADIOBUTTON,
                              text_button,
                              "Both",
```

```
                              "Icons and text in toolbar",
                              "Private",
                              iconw,
                              GTK_SIGNAL_FUNC (radio_event),
                              toolbar);
gtk_toolbar_append_space (GTK_TOOLBAR (toolbar));
gtk_toggle_button_set_active (GTK_TOGGLE_BUTTON (both_button), TRUE);
```

Finalmente, debemos activar el estado de uno de los botones de forma manual (o en caso contrario todos estarían activos al comienzo, impidiéndonos pulsar uno u otro)

```
 /* here we have just a simple toggle button */
iconw = gtk_image_new_from_file ("gtk.xpm");
tooltips_button =
  gtk_toolbar_append_element (GTK_TOOLBAR (toolbar),
                              GTK_TOOLBAR_CHILD_TOGGLEBUTTON,
                              NULL,
                              "Tooltips",
                              "Toolbar with or without tips",
                              "Private",
                              iconw,
                              GTK_SIGNAL_FUNC (toggle_event),
                              toolbar);
gtk_toolbar_append_space (GTK_TOOLBAR (toolbar));
gtk_toggle_button_set_active (GTK_TOGGLE_BUTTON (tooltips_button), TRUE);
```

Podemos crear un botón conmutador de la forma obvia (si uno ya sabe crear botones radiales).

```
 /* to pack a widget into toolbar, we only have to
  * create it and append it with an appropriate tooltip */
entry = gtk_entry_new ();
gtk_toolbar_append_widget (GTK_TOOLBAR (toolbar),
                           entry,
                           "This is just an entry",
                           "Private");

/* well, it isn't created within the toolbar, so we must still show it */
gtk_widget_show (entry);
```

Como podemos ver, añadir cualquier tipo de widget a una barra de heramientas es algo simple. Lo único que hay que recordar es que este widget debe ser mostrado manualmente (al contrario que otros items que mostraremos más adelante conjuntamente con la barra de herramientas)

```
  /* that's it ! let's show everything. */
  gtk_widget_show (toolbar);
  gtk_widget_show (handlebox);
  gtk_widget_show (dialog);

  /* rest in gtk_main and wait for the fun to begin! */
  gtk_main ();

  return 0;
}
```

Con esto finalizar el capítulo de la barra de herramientas. Sólo añadir que será necesario hacer uso del siguiente icono XPM para apreciar el ejemplo en todo su esplendor:

```
/* XPM */
```

```
static char * gtk_xpm[] = {
"32 39 5 1",
".      c none",
"+      c black",
"@      c #3070E0",
"#      c #F05050",
"$      c #35E035",
"...............+................",
"..............+++++.............",
"............+++++@@++...........",
"..........+++++@@@@@@++.........",
"........++++@@@@@@@@@@++........",
"......++++@@++++++++@@@++.......",
".....+++@@@+++++++++++@@@++.....",
"...+++@@@@+++@@@@@@++++@@@@+....",
"..+++@@@@+++@@@@@@@@+++@@@@@++..",
".++@@@@@@+++@@@@@@@@@@@@@@@@@@++",
".+#+@@@@@@++@@@@+++@@@@@@@@@@@@+",
".+##++@@@@+++@@@+++++@@@@@@@@$@.",
".+###++@@@@+++@@@+++@@@@@++$$$@.",
".+####+++@@@+++++++@@@@@+@$$$$@.",
".+#####+++@@@@+++@@@@++@$$$$$$+.",
".+######++++@@@@@@@++@$$$$$$$$+.",
".+#######+##+@@@@+++$$$$$$@@$$+.",
".+###+++##+##+@@++@$$$$$$++$$$+.",
".+###++++##+##+@@$$$$$$$@+@$$@+.",
".+###++++++#+++@$$@+@$$@++$$$@+.",
".+####+++++++#++$$@+@$$++$$$$+..",
".++####++++++#++$$@+@$++@$$$$+..",
".+#####+++++##++$$++@+++$$$$$+..",
".++####+++##+#++$$+++++@$$$$$+..",
".++####+++####++$$++++++@$$$@+..",
".+#####++#####++$$+++@++++@$@+..",
".+#####++#####++$$++@$$@+++$@@..",
".++####++#####++$$++$$$$$+@$@++.",
".++####++#####++$$++$$$$$$$$+++.",
".+++####+#####++$$++$$$$$$$@+++.",
"..+++#########+@$$+@$$$$$$+++...",
"...+++########+@$$$$$$$$@+++....",
".....+++######+@$$$$$$$+++......",
"......+++#####+@$$$$$@++........",
".......+++####+@$$$$+++.........",
".........++###+$$$@++...........",
"..........++##+$@+++............",
"...........+++++++..............",
".............++++..............."};
```

## GtkCombo

El combo box es otro widget relativamente simple que simplemente consiste en una agrupación de otros widgets. Desde el punto de vista del usuario, el widget consiste en una caja de entrada de texto y un menú desplegable desde el que el usuario puede seleccionar una opción de un conjunto de entradas predefinidas. De manera opcional, el usuario puede teclear una opción diferente directamente en la caja de texto.

El siguiente trozo de código está extraído de la definición de la estructura ComboBox e identifica muchos de sus componentes:

```
 struct _GtkCombo {
        GtkHBox hbox;
        GtkWidget *entry;
        GtkWidget *button;
        GtkWidget *popup;
```

```
        GtkWidget *popwin;
        GtkWidget *list;
...  };
```

Como puede observar el lector, el ComboBox consta de dos partes principales: un widget de entrada de datos y un widget lista.

Para crear un combo box, deberíamos de usar la siguiente instrucción:

```
GtkWidget *gtk_combo_new( void );
```

Si ahora quisiéramos inicializar la cadena de texto en la sección de entrada del combo box, lo podríamos hacer manipulando directamente dicho widget de entrada de datos:

```
gtk_entry_set_text (GTK_ENTRY (GTK_COMBO (combo)->entry), "My String.");
```

Para inicializar los valores en la lista desplegable, deberíamos de usar la siguiente función:

```
void gtk_combo_set_popdown_strings( GtkCombo *combo,
                                   GList    *strings );
```

Antes de poder ejecutar la instrucción anterior, es necesario preparar la lista de strings que queremos visualizar en la lista desplegable dentro de una construcción GList. GList es una implementación de lista ligada que forma parte de GLib, una biblioteca de funciones que da soporte a GTK. Por el momento, la explicación más rápida consiste en decir que es necesario definir un puntero a GList, inicializarlo a NULL y después añadirle strings GList is a linked list implementation that is part of GLib, a library supporting GTK. For the moment, the quick and dirty explanation is that you need to set up a GList pointer, set it equal to NULL, then append strings con la instrucción:

```
GList *g_list_append( GList *glist,
                     gpointer data );
```

Es importante inicializar el puntero inicial GList a NULL. El valor devuelto por la función g_list_append() debe ser usado como el nuevo puntero para GList.

Aquí podemos ver un fragmento de código típico para crear un conjunto de opciones:

```
GList *glist = NULL;

glist = g_list_append (glist, "String 1");
glist = g_list_append (glist, "String 2");
glist = g_list_append (glist, "String 3");
glist = g_list_append (glist, "String 4");

gtk_combo_set_popdown_strings (GTK_COMBO (combo), glist);

/* ahora es posible liberar glist, pues el combo ya tiene una copia de la lista */
```

El widget combo creará una copia de las cadenas de texto que se le pasen como parámetro en la estructura glist. Como resultado, es necesario asegurarse de liberar la memoria usada por la lista si es necesario para la aplicación que estemos construyendo.

En este punto ya disponemos de una configuración funcional de un combo box. Existen algunos aspectos del comportamiento de un combo que podemos cambiar por programación, usando las siguientes funciones:

```
void gtk_combo_set_use_arrows( GtkCombo *combo,
                               gboolean  val );

void gtk_combo_set_use_arrows_always( GtkCombo *combo,
                                      gboolean  val );

void gtk_combo_set_case_sensitive( GtkCombo *combo,
                                   gboolean  val );
```

`gtk_combo_set_use_arrows()` permite al usuario cambiar el valor de la entrada usando las teclas del cursor arriba/abajo. Esto no muestra la lista desplegable, sino que reemplaza el texto actual de la entrada de texto del combo con el siguiente valor de la lista (arriba o abajo, en función de la pulsación de tecla). Esto se consigue buscando en la lista el valor actual de la entrada del combo y seleccionando el siguiente o el anterior elemento, en función de lo pulsado por el usuario. Normalmente, en una entrada de texto, las teclas de movimiento arriba/abajo se usan para cambiar el foco (es posible hacerlo igualmente mediante el TABulador). Como nota adicional, hacer notar que cuando el elemento seleccionado es el último de la lista y el usuario pulsa flecha abajo, éste cambiará el foco (lo mismo ocurre cuando está seleccionado el primer elemento y el usuario pulsa flecha arriba).

Si el valor actual de la entrada no está en la lista, la función `gtk_combo_set_use_arrows()` se deshabilita.

De forma similar, `gtk_combo_set_use_arrows_always()` permite el uso de las flechas arriba/abajo para moverse por las opciones de la lista desplegable. Esta función es idéntica a la anterior, pero permite usar las flechas arriba y abajo para cambiar el elemento de la lista desplegable aunque lo que se haya tecleado en la entrada de texto no coincida con ninguno de los elementos.

La función gtk_combo_set_case_sensitive() permite conmutar entre la posibilidad o imposibilidad de que GTK busque las entradas del combo teniendo en cuenta diferencias entre mayúsculas y minúsculas. Se usa cuando el widget combo necesita buscar un valor de la lista usando el contenido actual de la entrada de texto. El completamiento puede ser realizado, dependiendo del valor de esta función, teniendo en cuenta mayúsculas/minúsculas o no teniéndolas en cuenta. El widget Combo puede también simplemente completar el valor actual de la entrada de texto si el usuario pulsa la combinación de teclas MOD-1 y "Tab". MOD-1 está normalmente mapeado por la utilidad xmodmap a la tecla "Alt". Obsérvese, sin embargo, que algunos gestores de ventanas también podrían usar esta combinación de teclas para otra función, que tendrá preferencias sobre su uso en GTK.

Ahora que disponemos de un combo box, personalizado para funcionar a nuestro gusto, todo lo que queda es saber cómo acceder a los datos que muestra este combo. Esto es relativamente sencillo. La mayor parte de las veces, todo lo que necesitamos hacer para obtener datos del combo es saber acceder a la entrada de texto del mismo. Esta entrada puede ser accedida simplemente a través de GTK_ENTRY (GTK_COMBO (combo)->entry). Las dos cosas principales que vamos a necesitar hacer es conectar esta entrada a la señal "activate", que indica cuándo el usuario ha

pulsado la tecla Return o Enter, y leer el contenido de dicha entrada de texto. Lo primero lo podemos conseguir usando algo como:

```
g_signal_connect (G_OBJECT (GTK_COMBO (combo)->entry), "activate",
                  G_CALLBACK (my_callback_function), (gpointer) my_data);
```

Para acceder al texto de la entrada en cualquier momento, lo podemos hacer simplemente usando la siguiente función:

```
gchar *gtk_entry_get_text( GtkEntry *entry );
```

Como aquí:

```
gchar *string;

string = gtk_entry_get_text (GTK_ENTRY (GTK_COMBO (combo)->entry));
```

Y esto es todo sobre el widget GtkCombo. Existe también una función

```
void gtk_combo_disable_activate( GtkCombo *combo );
```

que deshabilitará la señal activate en el widget de entrada del combo. Personalmente, no encuentro ninguna situación en la que pueda ser interesante usarla, pero existe.

> **Nota:** En el fichero combo.c del directorio de ejemplos GTK se puede estudiar el código completo de una aplicación que hace uso del widget GtkCombo.

## Ventanas de selección

En numerosas ocasiones nos encontraremos con la necesidad de implementar ventanas de selección comunes a muchas aplicaciones, por ejemplo, ventanas para seleccionar un color, un archivo o un tipo de letra. Construir estas ventanas o cuadros de diálogo cada vez, usando los elementos básicos vistos hasta ahora, puede ser muy costoso y aburrido. Los programadores de GTK+ se dieron cuenta de esto muy pronto y programaron algunos cuadros de diálogo comunes para que todo desarrollador pudiera usarlos, ahorrándose tiempo por una parte y manteniendo una consistencia entre todas las aplicaciones GKT+ por otra. Entre estos cuadros de diálogo se encuentran las típicas funcionalidades de seleccionar un color, un fichero o un tipo de fuente.

### GtkFileSelection

La ventana de selección de fichero es quizás el cuadro de diálogo más usado de aquellos que ofrece GTK+ de serie.

Esta ventana de selección tiene el siguiente aspecto:

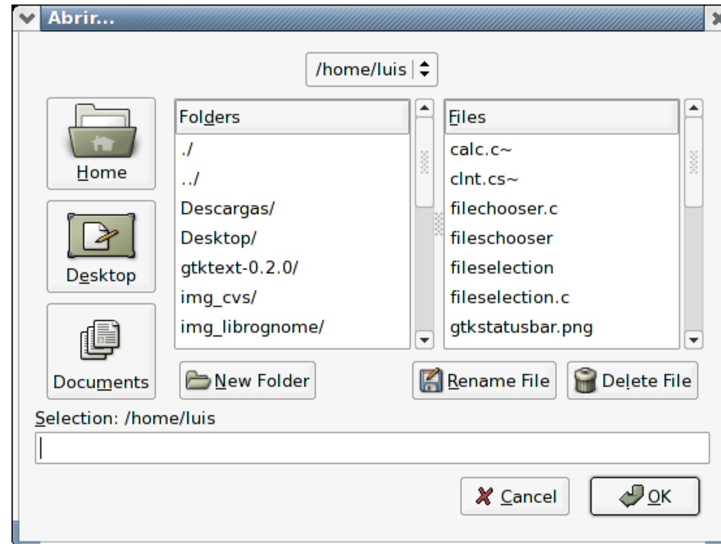



**Figura 7-13. Ventana de selección de archivo**

Consiste en una lista de ficheros, una lista de directorios, un botón con una lista desplegable que nos permite seleccionar un directorio de la ruta actual, y una caja de entrada de datos que muestra el filtro aplicado o el nombre del fichero seleccionado una vez que pinches sobre él. Además, vemos una serie de botones disponibles en la parte superior de la ventana, con los que podremos no sólo seleccionar un archivo, sino también borrarlos y renombrarlos, así como crear nuevos directorios. Cuando creemos una ventana de selección de archivo, podremos también deshabilitar esta fila de botones.

Es posible navegar por los directorios de manera tradicional, pulsando con el ratón sobre los nombres de los mismos o bien usando la lista desplegable con el path actual. Un truco para navegar por el sistema de archivos rápidamente con el teclado, consiste en el uso del tabulador (TAB), que nos permitirá teclear las primeras letras de un nombre de directorio al que queremos entrar, pulsar TAB y la ventana de selección nos colocará directamente en ese directorio, sin tener que teclear el nombre completo. Es lo que se conoce como autocompletamiento - una funcionalidad común en editores como emacs y jed.

Crearemos una ventana de selección de fichero llamando a la función `gtk_file_selection_new()`.

```
GtkWidget * gtk_file_selection_new ( const gchar * title );
```

Una vez creada la ventana de selección de fichero, podemos indicar un nombre de fichero a abrir por defecto usando la función gtk_file_selection_set_filename.

```
void gtk_file_selection_set_filename ( GtkFileSelection * filesel,
     const gchar * filename );
```

Cuando el usuario abra una ventana de selección de archivo, podemos hacer que sólo se le muestren determinados tipos de archivo, aplicando un filtro sobre el nombre. Por ejemplo, podemos hacer que sólo muestre los archivos gráficos .png us-

ando la función `gtk_file_selection_set_complete()` pasándole como segundo parámetro la cadena "*.png".

```
void gtk_file_selection_complete ( GtkFileSelection * filesel,
      const gchar * pattern );
```

Finalmente, para obtener el nombre del fichero que el usuario ha seleccionado, llamaremos a la función `gtk_file_selection_get_filename()`.

```
gchar * gtk_file_selection_get_filename ( GtkFileSelection * filesel );
```

Esta función devuelve un puntero a una cadena que contiene el nombre del fichero seleccionado (con el path absoluto).

Veamos un ejemplo completo de creación de un cuadro de diálogo de selección de archivo:

```
#include <stdlib.h>
#include <gtk/gtk.h>

void OnButton(GtkWidget *pWidget);

int main(int argc,char **argv)
{
    GtkWidget *pWindow;
    GtkWidget *pButton;

    gtk_init(&argc,&argv);

    pWindow = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(pWindow), "GtkFileSelection");
    gtk_window_set_default_size(GTK_WINDOW(pWindow), 320, 200);

    pButton=gtk_button_new_with_mnemonic("_Examinar...");
    gtk_container_add(GTK_CONTAINER(pWindow), pButton);

    g_signal_connect(G_OBJECT(pWindow), "destroy", G_CALLBACK(gtk_main_quit), NULL);
    g_signal_connect(G_OBJECT(pButton), "clicked", G_CALLBACK(OnButton), NULL);

    gtk_widget_show_all(pWindow);
    gtk_main();
    return EXIT_SUCCESS;
}

void OnButton(GtkWidget *pWidget)
{
    GtkWidget *pFileSelection;
    GtkWidget *pDialog;
    const gchar *sPath;

    /* Creación de la ventana de selección */
    pFileSelection = gtk_file_selection_new("Abrir...");
    /* Limitar las acciones a esta ventana */
    gtk_window_set_modal(GTK_WINDOW(pFileSelection), TRUE);

    /* Gestionar ventana */
    switch(gtk_dialog_run(GTK_DIALOG(pFileSelection)))
    {
        case GTK_RESPONSE_OK:
            /* Obtención del path */
            sPath = gtk_file_selection_get_filename(GTK_FILE_SELECTION(pFileSelection))
```

```
            pDialog = gtk_message_dialog_new(GTK_WINDOW(pFileSelection),
                GTK_DIALOG_MODAL,
                GTK_MESSAGE_INFO,
                GTK_BUTTONS_OK,
                "Path del fichero :\n%s", sPath);
            gtk_dialog_run(GTK_DIALOG(pDialog));
            gtk_widget_destroy(pDialog);
            break;
        default:
            break;
    }
    gtk_widget_destroy(pFileSelection);
}
```

## GtkFileChooser

GtkFileChooser es el nuevo conjunto de APIs (dese GTK+ 2.4) para los widgets y cuadros de diálogo de selección de archivos. Las versiones anteriores de GTK+ usaban GtkFileSelection, que como hemos visto, tenía varias deficiencias y opciones a mejorar.

GtkFileChooser es un interfaz abstracto que puede ser implementado por los widgets que realicen tareas de selección de archivos. Hay dos widgets en GTK+ que implementan este interfaz: GtkFileChooserDialog y GtkFileChooserWidget. La mayoría de las aplicciones necesitan únicamente hacer uso de GtkFileChooserDialog, que es una caja de diálogo que permite al usuario seleccionar un archivo entre todos los existentes para abrirlo, o guardar y nombrar nuevos documentos y archivos. GtkFileChooserWidget es un widget preparado para aplicaciones especiales que requieran empotrar el widget de selección de archivos dentro de una ventana con mayores funcionalidades. Dentro del contexto GTK+, un GtkFileChooserDialog es simplemente una caja GtkDialog con un widget GtkFileChooserWidget dentro de la misma.

### Cómo crear un GtkFileChooserDialog

Para crear un GtkFileChooserDialog, simplemente llamaremos a la función gtk_file_chooser_dialog_new(). Esta función es similar a gtk_dialog_new(), tomando como parámetros el titulo del cuadro de diálogo y la ventana padre, así como sus botones. Usa también otro parámetro adicional, que determina si la ventana para la selección de archivos será usada para abrir un fichero existente o para guardar un nuevo archivo.

Véase el siguiente ejemplo, que muestra un uso típico de una ventana de selección de archivos y cómo obtener el nombre de fichero seleccionado:

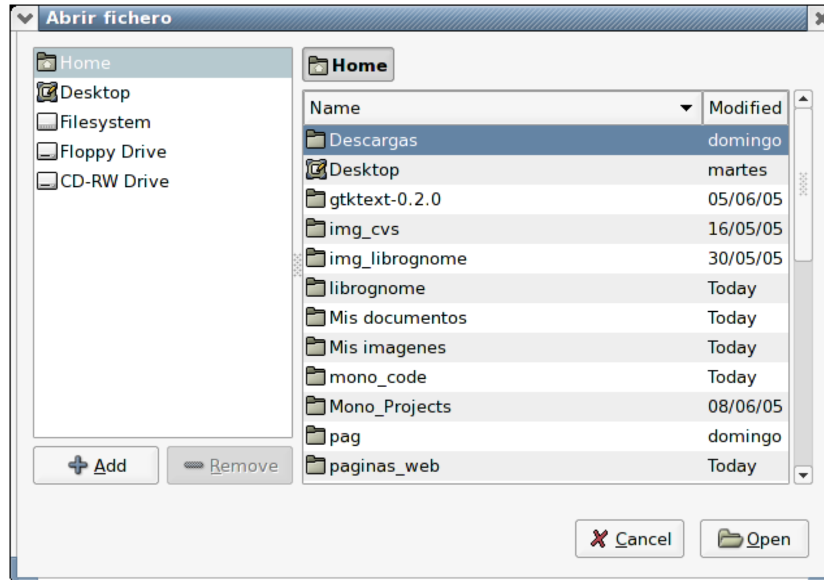



**Figura 7-14. GtkFileChooser en acción**

```
#include <stdlib.h>
#include <gtk/gtk.h>


void OnButton(GtkWidget *pWidget);

int main(int argc,char **argv)
{
    GtkWidget *pWindow;
    GtkWidget *pButton;

    gtk_init(&argc,&argv);

    pWindow = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(pWindow), "GtkFileChooser");
    gtk_window_set_default_size(GTK_WINDOW(pWindow), 320, 200);

    pButton=gtk_button_new_with_mnemonic("_Examinar...");
    gtk_container_add(GTK_CONTAINER(pWindow), pButton);

    g_signal_connect(G_OBJECT(pWindow), "destroy", G_CALLBACK(gtk_main_quit), NULL);
    g_signal_connect(G_OBJECT(pButton), "clicked", G_CALLBACK(OnButton), NULL);

    gtk_widget_show_all(pWindow);
    gtk_main();
    return EXIT_SUCCESS;
}

void OnButton(GtkWidget *pWidget)
{
    GtkWidget *pFileSelection;
    GtkWidget *pDialog;
    const gchar *sChemin;

    /* Creación de la ventana de selección */

    pFileSelection = gtk_file_chooser_dialog_new("Abrir fichero",
```

```
                                     NULL,
                                     GTK_FILE_CHOOSER_ACTION_OPEN,
                                     GTK_STOCK_CANCEL, GTK_RESPONSE_CANCEL,
                                     GTK_STOCK_OPEN, GTK_RESPONSE_ACCEPT,
                                     NULL);
    /* Limitar las acciones a esta ventana */
    gtk_window_set_modal(GTK_WINDOW(pFileSelection), TRUE);

        if (gtk_dialog_run (GTK_DIALOG (pFileSelection)) == GTK_RESPONSE_ACCEPT)
          {
            char *filename;

            filename = gtk_file_chooser_get_filename (GTK_FILE_CHOOSER (pFileSelection)
            g_print ("%s\n", filename);
            g_free (filename);
          }

    gtk_widget_destroy(pFileSelection);
}
```

### Modos de selección

GtkFileChooser puede ser usado en dos modos, para seleccionar un fichero único o para seleccionar múltiples ficheros a la vez. Para fijar el comportamiento deseado, usaremos la `gtk_file_chooser_set_select_multiple()`. En el modo de selección individual, podemos usar la función `gtk_file_chooser_get_filename()` para obtener el nombre del fichero del sistema de archivos local o `gtk_file_chooser_get_uri()` para obtener la URI completa del mismo. En el modo de selección múltiple, usaremos `gtk_file_chooser_get_filenames()` para obtener una lista GSList con los nombres (strings) de los ficheros, o `gtk_file_chooser_get_uris()` para obtener una lista de las URIs de los mismos.

También es posible configurar GtkFileChooser para seleccionar ficheros o carpetas. Por ejemplo, imaginemos un programa de backup que debe permitir al usuario seleccinar una carpeta de la que hacer una coia de seguridad (incluyendo todos sus subdirectorios). Para determinar si GtkFileChooser será usado para seleccionar ficheros o carpetas, usaremos la función `gtk_file_chooser_set_action()`. Esta función nos permitirá además configurar si el selector de fichero será usado para seleccionar archivos o carpetas existentes (ej. para implementar "Archivo/Abrir..."), ó para teclear el nombre de un nuevo fichero (ej. para implementar "Archivo/Guardar como...").

### Cómo instalar un widget de previsualización

Muchas aplicaciones necesitan tener una facilidad que permita la previsualización de los ficheros a seleccionar dentro de la ventana de selección de archivos. Antes de la versión GTK+ 2.4, el programador debía de acceder directamente a la jerarquía del widget GtkFileSelection para enganchar el código de un widget de previsualización. Con GtkFileChooser, existe un API encargado de facilitar esta acción.

En la siguiente sección de código podemos ver un ejemplo de creación de un widget de previsualización:

```
{
  GtkImage *preview;

  ...

  preview = gtk_image_new ();
```

```
  gtk_file_chooser_set_preview_widget (my_file_chooser, preview);
  g_signal_connect (my_file_chooser, "update-preview",
                    G_CALLBACK (update_preview_cb), preview);
}

static void
update_preview_cb (GtkFileChooser *file_chooser, gpointer data)
{
  GtkWidget *preview;
  char *filename;
  GdkPixbuf *pixbuf;
  gboolean have_preview;

  preview = GTK_WIDGET (data);
  filename = gtk_file_chooser_get_preview_filename (file_chooser);

  pixbuf = gdk_pixbuf_new_from_file_at_size (filename, 128, 128, NULL);
  have_preview = (pixbuf != NULL);
  g_free (filename);

  gtk_image_set_from_pixbuf (GTK_IMAGE (preview), pixbuf);
  if (pixbuf)
    gdk_pixbuf_unref (pixbuf);

  gtk_file_chooser_set_preview_widget_active (file_chooser, have_preview);
}
```

## Cómo instalar widgets adicionales

Algunas aplicaciones necesitarán instalar funcionalidades adicionales (widgets extras) en la ventana de selección de archivos. Por ejemplo, una aplicación podría querer ofrecer un botón conmutador para dar la usuario la opción de abrir un fichero como sólo-lectura o lectura-escritura.

Veamos un ejemplo de código que muestra cómo crear widgets extra en la ventana de selección de archivos:

```
{
  GtkWidget *toggle;

  ...

  toggle = gtk_check_button_new_with_label ("Open file read-only");
  gtk_widget_show (toggle);
  gtk_file_chooser_set_extra_widget (my_file_chooser, toggle);
}
```

## Nuevas características

El widget GtkFileChooser incluye nuevas características (respecto a GtkFileSelection) como la siguientes:

- Posibilidad de seleccionar URIs en vez de simples ficheros locales. Deberás usar una implementación de GtkFileSystem que soporte esto, por ejemplo el backend gnome-vfs.

- Presentar una lista de accesos directos a carpetas (en función de cada aplicación). Por ejemplo, un programa de diseño gráfico podría querer añadir su propio acceso directo a la carpeta /usr/share/paint_program/Clipart.
- Definir filtros personalizados de tal forma que no se muestren todos los ficheros de una carpeta. Por ejemplo, podrías querer filtrar los ficheros temporales (de backup) que crean algunas aplicaciones, o mostrar sólo ficheros gráficos.

Para obtener más información sobre cómo usar estas características, podemos consultar la documentación de referencia de GtkFileChooser.

## GtkColorSelection

GtkColorSelection es un widget usado para seleccionar un color. Consta de una rueda de color y varias escalas y cajas de entrada de datos para teclear valores que definen un color, como tonalidad, saturación, valor, rojo, verde, azul y opacidad. Este widget podemos encontrarlo en la ventana de selección de color estándar GtkColorSelection-Dialog. En las siguientes páginas veremos cómo usar este cuadro de diálogo.

## GtkColorSelectionDialog

GtkColorSelectionDialog es el cuadro de diálogo estándar usado en GKT+ para seleccionar un color, al igual que el cuadro de diálogo GtkFileSelection nos ofrecía la posibilidad de seleccionar un fichero.

La función `GtkWidget* gtk_color_selection_dialog_new (const gchar *title)` toma como parámetro de entrada el texto del título de la ventana que queremos crear y devuelve el propio GtkColorSelectionDialog. La estructura interna de éste widget contiene los siguientes elementos:

- GtkWidget *colorsel : widget GtkColorSelection que contiene el cuadro de diálogo. Usaremos este widget y su función `gtk_color_selection_get_current_color()` para acceder al color seleccionado por el usuario. Debemos conectar un manejador para la señal color_changed de este widget, para ser notificados cuando el color cambie.
- GtkWidget *ok_button : el widget de botón OK contenido en la caja de diálogo. Debemos conectar un manejador al evento clicked.
- GtkWidget *cancel_button: el widget de botón OK contenido en la caja de diálogo. Debemos conectar un manejador al evento clicked.
- GtkWidget *help_button: el widget del botón AYUDA contenido en la caja de diálogo. Debemos conectar un manejador al evento clicked.

Por ejemplo, para acceder al botón OK de un cuadro de diálogo GtkColorSelectionDialog lo haríamos así GTK_COLOR_SELECTION_DIALOG (colorseldialog)->ok_button

El widget de selección de color soporta ajustar la opacidad de un color (también conocido como ajuste del canal alpha). Por defecto, esto está deshabilitado. Si llamamos a la función

```
void gtk_color_selection_set_has_opacity_control( GtkColorSelection *colorsel,
                                                  gboolean           has_opacity );
```

con *`has_opacity`* a TRUE, habilitamos la opción de opacidad. De la misma forma, si has_opacity es FALSE, se deshabilitará.

Es posible establecer el color actual explícitamente mediante la llamada a la función `gtk_color_selection_set_current_color()` con un puntero a una estructura GdkColor. Se puede establecer la opacidad (canal alpha) mediante `gtk_color_selection_set_current_alpha()`. El valor de alpha debería de estar entre 0 (totalmente transparente) y 65636 (totalmente opaco). El prototipo de las funciones anteriores es el siguiente:

```
void gtk_color_selection_set_current_color( GtkColorSelection *colorsel,
                                            GdkColor          *color );


void gtk_color_selection_set_current_alpha( GtkColorSelection *colorsel,
                                            guint16            alpha );
```

Cuando lo que queremos es obtener el color actual, algo típico cuando recibimos la señal color_changed, usaremos estas funciones:

```
void gtk_color_selection_get_current_color( GtkColorSelection *colorsel,
                                     GdkColor *color );


void gtk_color_selection_get_current_alpha( GtkColorSelection *colorsel,
                                            guint16           *alpha );
```

Lo siguiente es un ejemplo que demuestra el uso del widget ColorSelectionDialog. El programa muestra una ventana que contiene un área de dibujo. Pulsando sobre ella, se abre un cuadro de diálogo de selección de color, donde al seleccionar un color conseguiremos cambiar también el color de fondo del área de dibujo.

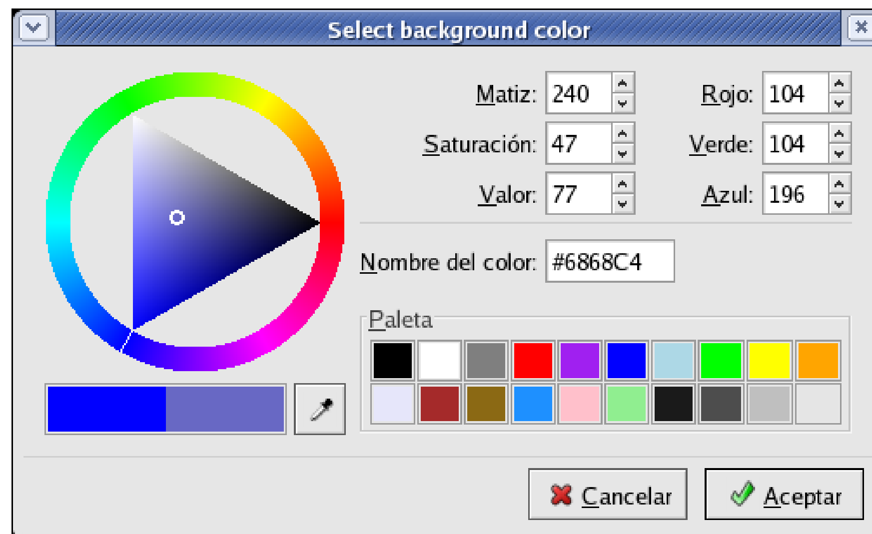



**Figura 7-15. Ejemplo de GtkColorSelectionDialog en ejecución**

```
#include <glib.h>
#include <gdk/gdk.h>
#include <gtk/gtk.h>
```

```
GtkWidget *colorseldlg = NULL;
GtkWidget *drawingarea = NULL;
GdkColor color;

/* Color changed handler */

static void color_changed_cb( GtkWidget         *widget,
                              GtkColorSelection *colorsel )
{
  GdkColor ncolor;

  gtk_color_selection_get_current_color (colorsel, &ncolor);
  gtk_widget_modify_bg (drawingarea, GTK_STATE_NORMAL, &ncolor);
}

/* Drawingarea event handler */

static gboolean area_event( GtkWidget *widget,
                            GdkEvent  *event,
                            gpointer   client_data )
{
  gint handled = FALSE;
  gint response;
  GtkColorSelection *colorsel;

  /* Check if we’ve received a button pressed event */

  if (event->type == GDK_BUTTON_PRESS)
    {
      handled = TRUE;

       /* Create color selection dialog */
      if (colorseldlg == NULL)
        colorseldlg = gtk_color_selection_dialog_new ("Select background color");

      /* Get the ColorSelection widget */
      colorsel = GTK_COLOR_SELECTION (GTK_COLOR_SELECTION_DIALOG (colorseldlg)->colorse

      gtk_color_selection_set_previous_color (colorsel, &color);
      gtk_color_selection_set_current_color (colorsel, &color);
      gtk_color_selection_set_has_palette (colorsel, TRUE);

      /* Connect to the "color_changedsignal, set the client-data
       * to the colorsel widget */
      g_signal_connect (G_OBJECT (colorsel), "color_changed",
                        G_CALLBACK (color_changed_cb), (gpointer) colorsel);

      /* Show the dialog */
      response = gtk_dialog_run (GTK_DIALOG (colorseldlg));

      if (response == GTK_RESPONSE_OK)
        gtk_color_selection_get_current_color (colorsel, &color);
      else
        gtk_widget_modify_bg (drawingarea, GTK_STATE_NORMAL, &color);

      gtk_widget_hide (colorseldlg);
    }

  return handled;
}

/* Close down and exit handler */

static gboolean destroy_window( GtkWidget *widget,
                                GdkEvent  *event,
                                gpointer   client_data )
```

```
{
  gtk_main_quit ();
  return TRUE;
}

/* Main */

gint main( gint   argc,
           gchar *argv[] )
{
  GtkWidget *window;

  /* Initialize the toolkit, remove gtk-related commandline stuff */

  gtk_init (&argc, &argv);

  /* Create toplevel window, set title and policies */

  window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  gtk_window_set_title (GTK_WINDOW (window), "Color selection test");
  gtk_window_set_policy (GTK_WINDOW (window), TRUE, TRUE, TRUE);

  /* Attach to the "deleteand "destroyevents so we can exit */

  g_signal_connect (GTK_OBJECT (window), "delete_event",
                    GTK_SIGNAL_FUNC (destroy_window), (gpointer) window);

  /* Create drawingarea, set size and catch button events */

  drawingarea = gtk_drawing_area_new ();

  color.red = 0;
  color.blue = 65535;
  color.green = 0;
  gtk_widget_modify_bg (drawingarea, GTK_STATE_NORMAL, &color);

  gtk_widget_set_size_request (GTK_WIDGET (drawingarea), 200, 200);

  gtk_widget_set_events (drawingarea, GDK_BUTTON_PRESS_MASK);

  g_signal_connect (GTK_OBJECT (drawingarea), "event",
                    GTK_SIGNAL_FUNC (area_event), (gpointer) drawingarea);

  /* Add drawingarea to window, then show them both */

  gtk_container_add (GTK_CONTAINER (window), drawingarea);

  gtk_widget_show (drawingarea);
  gtk_widget_show (window);

  /* Enter the gtk main loop (this never returns) */

  gtk_main ();

  /* Satisfy grumpy compilers */

  return 0;
}
```

## GtkFontSelection

GtkFontSelection es un widget para selección de fuentes (tipos de letra). Permite listar todos los tipos disponibles, sus estilos y tamaños, permitiendo al usuario seleccionar una fuente. Usado en conjunto con el widget GtkFontSelectionDialog, que ofrece una ventana ya preparada (cuadro de diálogo) para la selección de fuentes. Para definir qué fuente estará inicialmente seleccionada, usaremo la función `gtk_font_selection_set_font_name()`. Para obtener la fuente seleccionada, usaremos la función `gtk_font_selection_get_font()` o `gtk_font_selection_get_font_name()`. Para cambiar el texto que se muestra en el área de previsualización usaremos la función `gtk_font_selection_set_preview_text()`.

## GtkFontSelectionDialog

El widget GtkFontSelectionDialog es el cuadro de diálogo que usaremos normalmente para seleccionar un tipo de fuente.

Para definir desde este cuadro de diálogo qué fuente estará inicialmente seleccionada, usaremos una función con nombre similar a la vista en la sección anterior: `gtk_font_selection_dialog_set_font_name()`. Para obtener la fuente seleccionada, usaremos la función `gtk_font_selection_dialog_get_font()` o `gtk_font_selection_dialog_get_font_name()`. Y finalmente, siguiendo el esquema anterior, para cambiar el texto que se muestra en el área de previsualización usaremos la función `gtk_font_selection_dialog_set_preview_text()`.

La estructura GtkFontSelectionDialog se define de la siguiente forma:

```
      typedef struct {
       GtkWidget *ok_button;
 GtkWidget *apply_button;
 GtkWidget *cancel_button;
 } GtkFontSelectionDialog;
```

Donde:

```
GtkWidget *ok_button: es el botón OK de la ventana


GtkWidget *apply_button: el botón APLICAR de la ventana.
Este botón está oculto por defecto, pero es posible mostrarlo/ocultarlo por progr


GtkWidget *cancel_button;  The Cancel button of the dialog
```

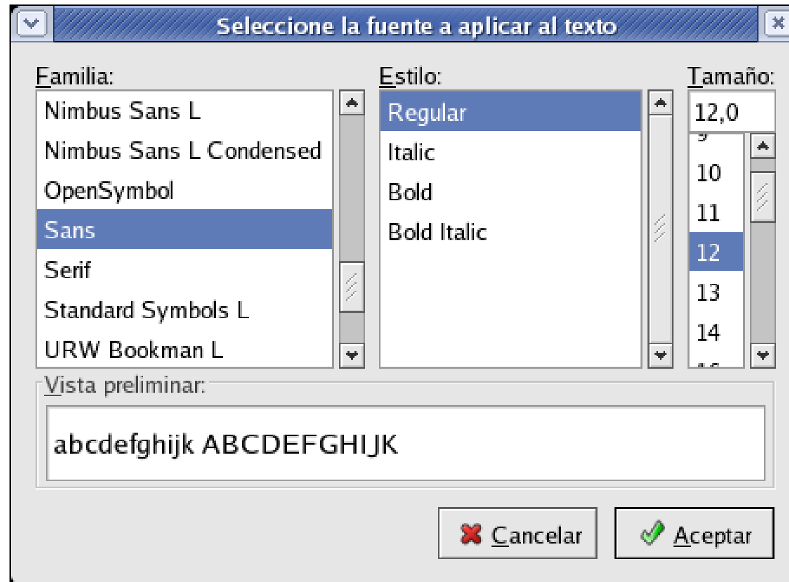



**Figura 7-16. Ejemplo de GtkFontSelectionDialog en ejecución**

En el ejemplo de código siguiente, hemos seleccionado únicamente una sección de código donde se muestra cómo lanzar una ventana de selección de fuente, obtener la fuente que seleccione el usuario y cambiar el tipo de letra que se estaba usando en un área de texto consecuentemente.

```
static void OkClicked(GtkButton * button, gpointer data)
{
   chosenfont = pango_font_description_from_string( gtk_font_selection_dialog_get_font_
   gtk_widget_destroy (GTK_WIDGET(data));
}

PangoFontDescription * ChooseFont ( gchar * title )
{
  GtkWidget * fontdialog;

  fontdialog = gtk_font_selection_dialog_new ( title);

  gtk_signal_connect ( GTK_OBJECT (fontdialog), "destroy",
        GTK_SIGNAL_FUNC(Close), NULL);

  gtk_signal_connect (GTK_OBJECT (GTK_FONT_SELECTION_DIALOG (fontdialog ) ->
                        ok_button), "clicked", GTK_SIGNAL_FUNC (OkClicked),
                        fontdialog);

  gtk_signal_connect (GTK_OBJECT (GTK_FONT_SELECTION_DIALOG ( fontdialog) ->
                        cancel_button), "clicked", GTK_SIGNAL_FUNC(CancelClicked),
                        fontdialog);

  chosenfont = NULL;
  gtk_widget_show(fontdialog);
  gtk_window_set_modal(GTK_WINDOW(fontdialog),TRUE);
  gtk_main();

  return chosenfont;
 }


void ChangeFont (GtkButton * button, gpointer data)
```

```
{
        PangoFontDescription * font;

        font = ChooseFont ("Seleccione la fuente a aplicar al texto");
        if (font!=NULL)
           gtk_widget_modify_font(entryarea,font);

        pango_font_description_free(font);
}
```

## Widgets de desplazamiento

### GtkHScrollbar/GtkVSrollbar

Véase la la sección de nombre *Widgets de selección de rango*

### GtkScrolledWindow

GtkScrolledWindow es una subclase de GtkBin: es decir, es un contenedor que permite albergar a un único widget hijo. GtkScrolledWindow añade barras de desplazamiento al widget albergado y opcionalmente, permite dibujar un marco biselado alrededor de dicho widget hijo. La ventana con scroll puede funcionar de dos formas diferentes. Algunos widgets tienen soporte de scrolling nativo; estos widgets tienen ranuras para insertar objtos de tipo GtkAdjustment (ver la sección de nombre *Ajustes*). Algunos de los widgets con soporte nativo de scroll son GtkTreeView, GtkTextView y GtkLayout. Para los widgets sin soporte nativo de scroll, se debe usar un widget adaptador previamente, el GtkViewport, antes de poder insertarlo en una ventana de scroll. Por ejemplo, deberemos usar GtkViewport para darle funcionalidad de scroll a widgets como GtkTable, GtkBox, etc.

Por tanto, si un widget tiene soporte nativo de scrolling, puede ser añadido a un GtkScrolledWindow directamente con la función `gtk_container_add()`. Si un widget carece de soporte de scrolling de forma nativa, debemos añadirlo primero a un GtkViewport y luego añadir éste último a la ventana de scroll. Para hacer esto, la forma más directa es usar la función `gtk_scrolled_window_add_with_viewport()` .

La posición de las barras de scroll se controla con los ajustes de scroll (ver la sección de nombre *Ajustes* para conocer los campos disponibles de un ajuste - para GtkScrollbar, widget usado internamente por GtkScrolledWindow, el campo "value" representa la posición de la barra de scroll, que debe tomar un valor entre el campo "lower" y la diferencia "upper - page_size". El campo "page_size" representa el tamaño del área visible de scroll. Los campos "step_increment" y "page_increment" se usan cuando el usuario pincha en "bajar" (usando las flechas de de bajar) o en página abajo (usando por ejemplo la tecla PageDown).

Si el widget GtkScrolledWindow no se comporta tal y como nos gustaría, o no se muestra en pantalla tal y como deseamos, existe la posibilidad de personalizar nuestras propias barra de desplazamiento a más detalle, usando GtkScrollbar y por ejemplo un widget GtkTable.

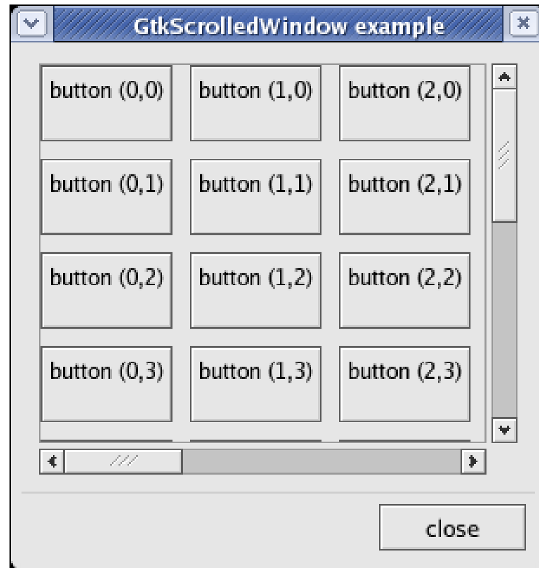



**Figura 7-17. Ejemplo de GtkScrolledWindow, extraído de la sección de ejemplos de la documentación de GTK+**

En el ejemplo de código siguiente, crearemos una tabla de botones de dimensiones 10x10, y la meteremos dentro de un contenedor GtkScrolledWindow. El scroll vertical se dará siempre, mientras que el horizontal sólo se mostrará en caso necesario (si redimensionamos con el ratón la ventana padre, haciéndola lo suficientemente ancha como para que entren las 10 columnas de la tabla, el scroll horizontal no aparecerá).

```
/* la política es ó GTK_POLICY AUTOMATIC ó GTK_POLICY_ALWAYS.
 * GTK_POLICY_AUTOMATIC decidirá automáticamente cuando es necesario pintar
 * barras de scroll, mientras que GTK_POLICY_ALWAYS pintará siempre
 * dichas barras.  El primer parámetro se refiere a la barra horizontal,
 * el segundo a la barra vertical. */
gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW (scrolled_window),
                                GTK_POLICY_AUTOMATIC, GTK_POLICY_ALWAYS);
/* se crea el cuadro de diálogo, empaquetando un vbox dentro del mismo */
gtk_box_pack_start (GTK_BOX (GTK_DIALOG(window)->vbox), scrolled_window,
                    TRUE, TRUE, 0);
gtk_widget_show (scrolled_window);

/* creamos la tabla de 10 x 10 celdas */
table = gtk_table_new (10, 10, FALSE);

/* espaciado de 10 en vertical y 10 en horizontal  */
gtk_table_set_row_spacings (GTK_TABLE (table), 10);
gtk_table_set_col_spacings (GTK_TABLE (table), 10);

/* empaquetamos la tabla en una ventana de scroll */
gtk_scrolled_window_add_with_viewport (
               GTK_SCROLLED_WINDOW (scrolled_window), table);
gtk_widget_show (table);
```

# Capítulo 8. GtkTreeView: Árboles y listas en GTK+

Es habitual que se necesiten mostrar estructuras de datos relativamente complejas dentro de las interfaces gráficas de las aplicaciones. Estos datos en muchos casos están relacionados al pertenecer a un mismo conjunto y se pueden mostrar al usuario como una lista. Pero en ocasiones, los elementos de esta lista a su vez son estructuras de datos necesitando mostrarse a su vez como nuevas listas.

Para este tipo de situaciones dentro de GTK2 se ha creado el potente y flexible control (widget) GtkTreeView/Data, un control que sigue el diseño MVC (modelo/vista/controlador) y que como veremos, nos permite mostrar una estructura compleja al usuario de forma tal que para éste, su manejo sea sencillo y rápido.

La idea básica es proporcionar un control que permita representar datos en forma de árbol, donde cada nodo a su vez puede ser un nuevo árbol. Las listas son un caso especial de este caso general ya que son sólo un nivel de nodos simples sin ramificaciones.

## Ejemplos básicos

Nada mejor para romper el hielo que un ejemplo simple de una lista basada en GtkTreeView. Para ello hemos cogido dos ejemplos del tutorial de GTK2 y los hemos simplificado. La idea es mostrar al usuario una lista cerrada de la compra, es decir, un conjunto de productos que el usuario tendrá que comprar. En el segundo ejemplo veremos como se puede mostrar información organizada en forma anidada, es decir, donde un elemento de datos puede a su vez tener otros elementos que dependan de él.

### Ejemplo de lista de datos

Vamos a ver en el ejemplo varias partes bien diferenciadas. Dentro del `main()` se crea la ventana principal de la aplicación, la cual contiene una etiqueta y nuestra lista.

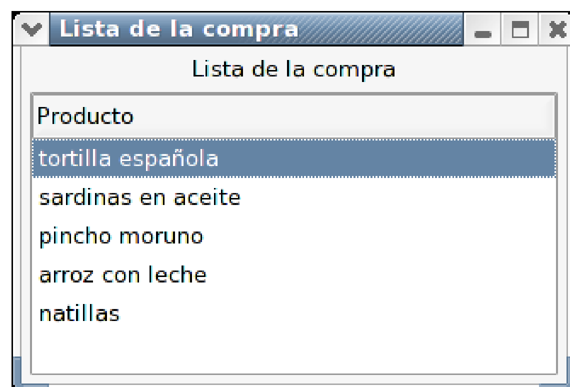



**Figura 8-1. Lista de la compra**

En la parte específica del uso de GtkTreeView nos encontramos con una llamada a `create_model()`, la cual se encarga de crear el modelo de datos de la lista, el cual será representado a través de la vista, que creamos utilizando este modelo de datos. Tras crear la vista, añadimos las columnas que queremos que el usuario vea y añadimos la vista al contenedor desde el que se visualiza en la pantalla principal.

Lo más importante de este ejemplo es ver como están perfectamente desacoplados los datos y la vista. Vemos que se asocian cuando se crea la vista con el modelo de datos, pero a la hora de visualizar, podemos elegir que parte de los datos se van a mostrar. En este caso los datos sólo tienen un campo, que es el que se muestra, pero como iremos viendo esto puede no ser así.

```
#include <gtk/gtk.h>
#include <string.h>
#include <stdlib.h>

static GtkWidget *window = NULL;

typedef struct
{
        gchar   *product;
} Item;

enum
{
        COLUMN_PRODUCT,
        NUM_COLUMNS
};

static GArray *articles = NULL;

static void
add_items (void)
{
        Item foo;

        g_return_if_fail (articles != NULL);

        foo.product = g_strdup ("tortilla espa\303\261ola");

        g_array_append_vals (articles, &foo, 1);

        foo.product = g_strdup ("sardinas en aceite");
        g_array_append_vals (articles, &foo, 1);

        foo.product = g_strdup ("pincho moruno");
        g_array_append_vals (articles, &foo, 1);

        foo.product = g_strdup ("arroz con leche");
        g_array_append_vals (articles, &foo, 1);

        foo.product = g_strdup ("natillas");
        g_array_append_vals (articles, &foo, 1);
}

/* Creamos el modelo de datos asociado a la vista */
static GtkTreeModel *
create_model (void)
{
        gint i = 0;
        GtkListStore *model;
        GtkTreeIter iter;

        /* crear matriz */
        articles = g_array_sized_new (FALSE, FALSE, sizeof (Item), 1);

        add_items ();

        /* crear lista de almacen de datos list */
        model = gtk_list_store_new (NUM_COLUMNS, G_TYPE_STRING);

        /* añadir elementos */
```

```
        for (i = 0; i < articles->len; i++) {
                gtk_list_store_append (model, &iter);

                gtk_list_store_set (model, &iter,
                                    COLUMN_PRODUCT,
                                    g_array_index (articles, Item, i).product,
                                    -1);
        }

        return GTK_TREE_MODEL (model);
}

static void
add_columns (GtkTreeView *treeview)
{
        GtkCellRenderer *renderer;
        GtkTreeModel *model = gtk_tree_view_get_model (treeview);

        /* columna de producto */
        renderer = gtk_cell_renderer_text_new ();
        gtk_tree_view_insert_column_with_attributes (GTK_TREE_VIEW (treeview),
                                                     -1, "Producto", renderer,
                                                     "text", COLUMN_PRODUCT,
                                                     NULL);
}

GtkWidget *
do_cells (void)
{
        if (!window) {
                GtkWidget *vbox;
                GtkWidget *hbox;
                GtkWidget *sw;
                GtkWidget *treeview;
                GtkTreeModel *model;

                /* creamos ventana principal y demás elementos */
                window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
                gtk_window_set_title (GTK_WINDOW (window), "Lista de la compra");
                gtk_container_set_border_width (GTK_CONTAINER (window), 5);
                g_signal_connect (G_OBJECT (window), "destroy",
                                  G_CALLBACK (gtk_widget_destroyed), &window);

                vbox = gtk_vbox_new (FALSE, 5);
                gtk_container_add (GTK_CONTAINER (window), vbox);

                gtk_box_pack_start (GTK_BOX (vbox),
                                    gtk_label_new ("Lista de la compra"),
                                    FALSE, FALSE, 0);

                sw = gtk_scrolled_window_new (NULL, NULL);
                gtk_scrolled_window_set_shadow_type (GTK_SCROLLED_WINDOW (sw),
                                                     GTK_SHADOW_ETCHED_IN);
                gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW (sw),
                                                GTK_POLICY_AUTOMATIC,
                                                GTK_POLICY_AUTOMATIC);
                gtk_box_pack_start (GTK_BOX (vbox), sw, TRUE, TRUE, 0);

                /* crear el modelo */
                model = create_model ();

                /* crear tree view */
                treeview = gtk_tree_view_new_with_model (model);
                g_object_unref (G_OBJECT (model));

                add_columns (GTK_TREE_VIEW (treeview));
```

```
                gtk_container_add (GTK_CONTAINER (sw), treeview);

                gtk_window_set_default_size (GTK_WINDOW (window), 320, 200);
        }

        if (!GTK_WIDGET_VISIBLE (window))
                {
                        gtk_widget_show_all (window);
                } else {
                        gtk_widget_destroy (window);
                        window = NULL;
                }

        return window;
}

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

        gtk_init (&argc, &argv);

        window = GTK_WIDGET (do_cells ());
        g_signal_connect (G_OBJECT (window), "destroy",
                    G_CALLBACK (gtk_main_quit),
                    NULL);
        gtk_widget_show_all (window);

        gtk_main ();
}
```

Ya habiendo visto cada parte resumidamente, iremos desglosando el ejemplo en varias partes de las cuales explicaremos paso a paso que se va a haciendo, por ello que mejor que empezar que con nuestro querido `main()` :-)

```
int
main (int argc, char *argv[]) {

        gtk_init (&argc, &argv);

        window = GTK_WIDGET (do_cells ());
        gtk_widget_show_all (window);

        gtk_main ();
}
```

Aquí inicializamos la librería GTK y luego llamamos al método `do_cells()` que es el que se encargará de crear las ventana donde se dispondrá nuestro GtkTreeView. Posteriormente se visualizará nuestra ventana (mediante la llamada a `gtk_widget_show_all()`) y se ejecutará el bucle `gtk_main()`.

Ya visto que se llama a la función `do_cells()` podemos ver que ahí lo único que se hace es crear la ventana y los consecutivos contenedores, además de crear nuestro modelo de datos, nuestro GtkTreeModel a partir de la función `create_model()` que veremos a continuación. Una vez hecho esto lo cogerá nuestra vista de árbol y se embeberá dentro del sw que está dentro del hbox. Esta parte no es necesaria ver de forma exhaustiva ya que está perfectamente explicada en la parte relativa a GTK por lo que seguro que lo entendéis :)

La parte de `create_model()` es la que se encarga de crear el modelo de datos de nuestro árbol en el widget `model`, posteriormente se creará el modelo de vistas del árbol además de la columna que le añadiremos, llamada con la función `add_columns()`.

Ya visto que llama a `create_model()`, analizaremos dicha función. Esta se encarga de recorrer el arreglo donde hemos metido todos los elementos de la lista de compra y los agrega al final del widget GtkListStore, que para efectos prácticos puede ser entendido como una simple lista a la cual se le agregan nuevos nodos. Para agregar un nuevo elemento a un GtkListStore utilizamos la función `gtk_list_store_append()`, la cual almacena en `iter` una referencia al elemento creado, el cual se encuentra momentáneamente vacío. Para almacenar los datos en éste se utiliza `gtk_list_store_set()`. Una vez completo nuestro modelo lo retornamos para que nuestra función `do_cells()` pueda crear el widget GtkTreeView referenciando inmediatamente nuestro modelo con este mediante:

```
treeview = gtk_tree_view_new_with_model (model);
```

A partir de este momento, cuando necesitemos trabajar con los datos lo haremos únicamente con el GtkListStore, y dichos cambios se reflejarán de forma automática dentro de la vista. Por ejemplo, para añadir un nuevo producto al treeview solo debemos añadirlo al GtkListStore y la vista lo cargará de forma inmediata.

Sólo queda ver el añadido de una celda productos con la función `add_columns()` sobre nuestra vista de árbol GtkTreeView.

En la función `add_columns()` utilizaremos un tipo GtkCellRenderer con el texto "producto" el cual insertaremos en el árbol de la siguiente manera:

```
renderer = gtk_cell_renderer_text_new ();
gtk_tree_view_insert_column_with_attributes (GTK_TREE_VIEW (treeview),
                                              -1, "Producto", renderer,
                                              "text", COLUMN_PRODUCT,
                                              NULL);
```

Como podéis ver es bastante sencillo, la única parte no comentada es la de la introducción de datos dentro del array, ya correspondiente a la parte de glib, no por ello complicada, únicamente vamos añadiendo valores sobre el array de la siguiente forma:

```
    foo.product = g_strdup ("sardinas en aceite");
    g_array_append_vals (articles, &foo, 1);
```

Luego estos datos serán los que utilicemos en la lista para almacenar los datos y conseguir nuestro modelo de árbol GtkTreeModel.

Como podéis ver no es complicado, únicamente hay que diferenciar entre el modelo de datos que es el que utilizamos para toda la manipulación del contenido del árbol y luego el GtkTreeView que es el que utilizaremos para la parte correspondiente a la muestra por pantalla de ese modelo de datos ya creado.

Para concluir este primer ejemplo, resumir que hemos utilizado un modelo de datos GtkListStore que hemos podido utilizar de forma sencilla y asociarlo a la vista pero, ¿y si nuestros datos tienen una estructura más compleja? ¿Y si no son simplemente cadenas de caractéres si no que son completos objetos con mucha información? ¿Es capaz GtkTreeView de visualizar estos objetos según nuestras necesidades?

La respuesta es que sí. Tenemos el modelo de datos GtkListStore para los habituales casos de tratamiento de cadenas de caracteres en una lista y GtkTreeStore para las cadenas de caractéres que se organizan en árbol. Es muy útil tenerlos ya predefinidos ya que los utilizaremos muy a menudo. Pero nada nos impide crearnos nuestros propios modelos de datos, tal y como vamos a mostrar en el siguiente apartado.

## Ejemplo de árbol de datos

A continuación veremos un ejemplo que nos permitirá explicar el uso básico del modelo GtkTreeStore. Este modelo es una versión genérica de lo que un programador podría querer hacer con un GtkTreeModel que, al igual que GtkListStore, simplifica mucho las cosas. Las simplifica, básicamente, porque siendo la estructura de árbol tan común y utilizada dentro de las aplicaciones, no existe ninguna razón para dejar los detalles del diseño del módelo a cualquiera que los necesite. Un ejemplo típico podría ser una estructura de directorios, un árbol de dependencias, o una lista de países y ciudades de éstos, que en esencia es el ejemplo que veremos a continuación.

Implementaremos una pequeña ventana con un GtkTreeView en su interior que muestre en una sola columna una lista de países expandible que nos muestre algunas ciudades de estos. Una pantalla de éste ejemplo es:

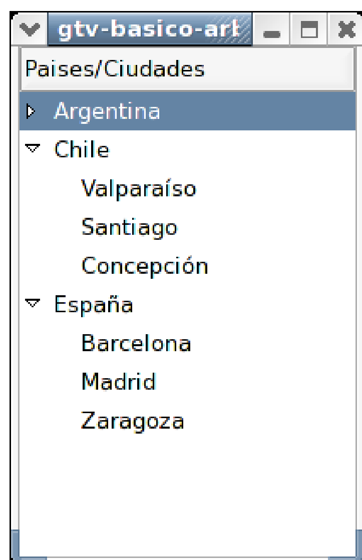



**Figura 8-2. Países y algunas de sus ciudades**

Como puede apreciarse, existe una jerarquía clara entre los elementos del GtkTreeView. Tenemos tres elementos (los países) en un mismo nivel. Si obviamos por un instante las ciudades, tendremos simplemente una lista. Ahora bien, considerando que cada ciudad está asociada a algún país, es lógico ver a éstas como *hijos* de los países. Cada ciudad o país (en general, cualquier fila en un GtkTreeStore) no es más que un nodo dentro de un árbol, que contiene los mismos datos que el resto de éstos (en el ejemplo, sólo el nombre de la ciudad o país), pero que se relaciona con los otros mediante una relación jerárquica que se especifica al momento de llenar el modelo.

Sin más preámbulo, demos un vistazo al código completo.

```
#include <gtk/gtk.h>

enum {
        COL_DATA,
        COL_TOTAL
};

GtkTreeModel *
create_model (void)
```

```
{
        GtkTreeStore *model;
        GtkTreeIter top;
        GtkTreeIter child;

        model = gtk_tree_store_new (COL_TOTAL, G_TYPE_STRING);

        gtk_tree_store_append (model, &top, NULL);
        gtk_tree_store_set (model, &top,
                            COL_DATA, "Argentina",
                            -1);

        gtk_tree_store_append (model, &child, &top);
        gtk_tree_store_set (model, &child,
                            COL_DATA, "Buenos Aires",
                            -1);

        gtk_tree_store_append (model, &child, &top);
        gtk_tree_store_set (model, &child,
                            COL_DATA, "Mendoza",
                            -1);

        gtk_tree_store_append (model, &child, &top);
        gtk_tree_store_set (model, &child,
                            COL_DATA,
                            "C\303\263rdoba",
                            -1);

        gtk_tree_store_append (model, &top, NULL);
        gtk_tree_store_set (model, &top,
                            COL_DATA, "Chile",
                            -1);

        gtk_tree_store_append (model, &child, &top);
        gtk_tree_store_set (model, &child,
                            COL_DATA,
                            "Valpara\303\255so",
                            -1);

        gtk_tree_store_append (model, &child, &top);
        gtk_tree_store_set (model, &child,
                            COL_DATA, "Santiago",
                            -1);

        gtk_tree_store_append (model, &child, &top);
        gtk_tree_store_set (model, &child,
                            COL_DATA,
                            "Concepci\303\263n",
                            -1);

        gtk_tree_store_append (model, &top, NULL);
        gtk_tree_store_set (model, &top,
                            COL_DATA,
                            "Espa\303\261a",
                            -1);

        gtk_tree_store_append (model, &child, &top);
        gtk_tree_store_set (model, &child,
                            COL_DATA, "Barcelona",
                            -1);

        gtk_tree_store_append (model, &child, &top);
        gtk_tree_store_set (model, &child,
                            COL_DATA, "Madrid",
                            -1);
```

```
        gtk_tree_store_append (model, &child, &top);
        gtk_tree_store_set (model, &child,
                            COL_DATA,
                            "Zaragoza",
                            -1);

        return GTK_TREE_MODEL (model);

}

GtkWidget *
create_view_and_model (void)
{
        GtkWidget *view;
        GtkTreeModel *model;
        GtkTreeViewColumn *col;
        GtkCellRenderer *renderer;

        view = gtk_tree_view_new ();

        col = gtk_tree_view_column_new ();
        gtk_tree_view_column_set_title (col, "Paises/Ciudades");

        gtk_tree_view_append_column (GTK_TREE_VIEW (view), col);

        renderer = gtk_cell_renderer_text_new ();
        gtk_tree_view_column_pack_start (col, renderer, TRUE);
        gtk_tree_view_column_add_attribute  (col, renderer, "text",
                                            COL_DATA);

        model = create_model ();

        gtk_tree_view_set_model (GTK_TREE_VIEW (view), model);

        g_object_unref (model);

        return view;
}

gboolean
app_quit (GtkWidget *widget, GdkEvent *event,
          gpointer data)
{
        gtk_main_quit ();
        return TRUE;
}

int
main (int argc, char **argv)
{
        GtkWidget *window;
        GtkWidget *view;

        gtk_init (&argc, &argv);

        window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

        g_signal_connect (window, "delete_event",
                          G_CALLBACK (app_quit), NULL);

        view = create_view_and_model ();

        gtk_container_add (GTK_CONTAINER (window), view);

        gtk_widget_show_all (window);
```

```
        gtk_main ();

        return 0;
}
```

Como se puede apreciar en el código, no hemos seguido el mismo estilo del ejemplo anterior con respecto a la creación del modelo y la vista. En el ejemplo anterior, creamos el modelo, luego lo asociamos a una vista mientras creamos ésta mediante `gtk_tree_view_new_with_model()` y por último creamos sus columnas. En este ejemplo creamos primero que nada la vista y sus columnas, luego creamos y llenamos el modelo para finalmente relacionar vista y modelo mediante `gtk_tree_view_set_model()`. Hemos hecho esto para hacer énfasis en la independencia entre el modelo y la vista. Del mismo modo, debemos mencionar que podemos relacionar un sólo modelo con varias vistas distintas.

En nuestra función `main()` hemos creado dos widgets, la ventana y la vista de árbol, para luego embeber una dentro de la otra. La creación concreta de la vista se realiza en la función `create_view_and_model()` mediante `gtk_tree_view_new()`. Luego procedemos a crear la columna que contendrá el nombre de la ciudad o país y la integramos a la vista. La creación específica del modelo la hacemos en la función `create_model()` que analizaremos más adelante. Ahora solo queda integrar el modelo a la vista. Hacemos esto mediante `gtk_treeview_set_model()`. La llamada a `g_object_unref()` es necesaria para que cuando destruyamos la vista se haga lo mismo con el modelo. En caso contrario, el modelo tendrá todavía una referencia al destruir la vista y seguirá en el entorno.

Como es lógico, nos enfocaremos en la función `create_model()` que es la de mayor relevancia para el ejemplo. En ésta función comenzamos creando el modelo, con:

```
model = gtk_tree_store_new (COL_TOTAL, G_TYPE_STRING);
```

Donde el primer parámetro es el total de columnas en el modelo (en nuestro ejemplo, sólo una) y a continuación especificamos el tipo de dato que contendrá cada una de dichas columnas.

Para agregar un nuevo elemento al modelo GtkTreeStore utilizamos la función `gtk_tree_store_append()`. Esta función necesita tres parametros, el modelo al cual queremos agregar un elemento, un GtkTreeIter para almacenar una referencia a este elemento y otro que indica el elemento al cual *anidaremos* el nuevo. Este segundo GtkTreeIter debe haber sido creado previamente, y si se desease añadir un elemento en el nivel principal (como es el caso de los países en el ejemplo) basta con pasar NULL como padre. He ahí la diferencia radical entre las siguientes líneas de código:

```
    gtk_tree_store_append (model, &top, NULL);
    ...
    gtk_tree_store_append (model, &child, &top);
```

La primera llamada a `gtk_tree_store_append()` crea un nuevo elemento de nivel cero y la siguiente crea un elemento que será *hijo* de ésta.

Llenar un elemento en los modelos GtkTreeModel y GtkListStore es una tarea prácticamente idéntica. Se utiliza para esto la función `gtk_tree_model_set()` que análogamente a `gtk_list_store_set()` necesita como parámetros el modelo, el iterador, y una lista variable de los valores a definir. En nuestro ejemplo, podemos ver una de las llamadas a dicha función en la siguientes líneas:

```
    gtk_tree_store_set (model, &child,
                        COL_DATA, "Mendoza",
                        -1);
```

## Modelos de datos estándares

Los datos son la parte que nos permite modificar la información que se asocia al GtkTreeView. En realidad el modelo de datos es una interfaz que se debe de implementar por parte de los modelos de datos que utilicemos, aunque los desarrolladores de GTK nos proporcionan como ejemplo dos modelos de datos estándar que pasamos a analizar. En la mayoría de las aplicaciones basta con usar uno de estos modelos de datos, pero como veremos en la próxima sección, a veces estos modelos de datos son insuficientes.

### Modelos de datos GtkListStore

El primero modelo, GtkListStore, ya lo hemos podido ver y es el que nos permite gestionar de forma sencilla una lista de elementos. Este modelo es muy útil en casos en los que tengamos una lista simple de elementos a mostrar, como en la lista de la compra.

A continuación exploraremos

### Modelos de datos GtkTreeStore

Para cubrir el habitual caso de tener que mostrar una información organizada en más de un nivel, es decir en una estructura de árbol como la del ejemplo de los países y ciudades, disponemos también de un modelo de datos estándar: GtkTreeStore.

### Generalidades

Un modelo contiene columnas y filas. Si bien cada fila en el modelo será desplegada de alguna manera en la vista, debe tenerse en cuenta que no necesariamente todas las columnas del modelo lo harán. De hecho, una columna en el modelo *no guarda relación directa las columnas de la vista*. Una columna del modelo se puede considerar como un campo en una estructura que tiene alguna relevancia para los datos que se quiere representar en la vista, ya sea información a ser desplegada, o atributos de ésta. Por ejemplo, se puede tener una columna con un texto a desplegar y otra con un valor que represente el color de la fuente del texto que se desplegará. Es muy importante tener claro que información se desea almacenar en el modelo, ya que no se pueden agregar o quitar columnas una vez creado el modelo.

Cada columna del modelo almacena elementos de un mismo tipo. Los tipos de datos usan el sistema de tipos de datos de GLib (GType) y los más comúnmente usados son:

- `G_TYPE_BOOLEAN`
- `G_TYPE_INT` - Para almacenar enteros.
- `G_TYPE_LONG`, `G_TYPE_ULONG`, `G_TYPE_FLOAT`, `G_TYPE_DOUBLE` - Varios tipos numéricos.
- `G_TYPE_STRING` - Para almacenar un string.
- `GDK_TYPE_PIXBUF` - Para almacenar una imagen.

Debe tenerse claro que no es necesario entender el sistema de tipos de GLib para poder usar un GtkTreeView -- Basta con conocerlos. Una de las ventajas del uso de

GType es que los usuarios avanzados podrán derivar sus propios tipos de datos, aunque por lo general, esto no es necesario.

Existen dos modos de crear un modelo de datos GtkListStore o GtkTreeStore. El primero es útil cuando la cantidad de columnas que se necesitan es un fija. Para esto utilizamos la función

```
GtkListStore * gtk_list_store_new ( gint n_columns, ... );
```

si se trata de un GtkListStore o `gtk_tree_store_new()` en el caso de un GtkTreeStore. Los prototipos de ambas funciones son análogos. `n_columns` es el número de columnas que se desea almacenar en el modelo y la lista variable de parámetros representa una lista ordenada de los `n_columns` GTypes que se desean almacenar. Por ejemplo, para crear un modelo de lista de datos con 3 columnas, la primera para texto, la segunda para una imagen y la tercera para un número entero, tendremos que llamar a `gtk_list_store_new()` de la siguiente manera:

```
GtkListStore * list_store = gtk_list_store_new (3, G_TYPE_STRING,
                                                    GDK_TYPE_PIXBUF,
                                                    G_TYPE_INT);
```

Por otro lado, si desconocemos la cantidad de columnas que necesitará el modelo, ie. éste debe crearse de manera dinámica, existe una función que recibe como parámetro la cantidad de columnas y un vector de punteros a GType. De este modo, podemos crear dicho vector en tiempo de ejecución y luego utilizarlo para crear el modelo. Esta función, para el caso de GtkTreeStore es:

```
GtkTreeStore * gtk_tree_store_newv ( gint n_columns, GType *types );
```

Una vez más, se debe mencionar que existe una función análoga para GtkListStore, `gtk_list_store_newv()`. Un ejemplo de uso de esta función sería:

```
/* crea y retorna un modelo con n_columns columnas */
GtkTreeModel *
create_model (gint n_columns)
{

        GtkListStore *list_store;
        GType *types;
        gint i;

        types = g_new (GType, n_columns);

        /* creamos las columnas del modelo alternando un pixbuf y un
    string */
        for (i = 0; i < n_columns; i++)
                types[i] = (i%2 == 0) ? GDK_TYPE_PIXBUF : G_TYPE_STRING;

        list_store = gtk_list_store_newv (n_columns, types);
 g_free (types);

 return GTK_TREE_MODEL (list_store);

}
```

Las columnas son enumeradas a partir de 0, y por lo general se acostumbra a usar enumeraciones para hacer más clara la referencia a las columnas. Usando enumeraciones, podemos agregar un elemento extra a ésta para referirnos a la cantidad de columnas que tendrá el modelo, evitando así el uso de *números mágicos*. Se puede notar esto en el ejemplo de la la sección de nombre *Ejemplo de lista de datos*.

## Refiriendose a las filas

A esta altura ya sabemos como crear un modelo, pero no tenemos interes alguno en contar con un modelo que no tiene datos. Pero antes de estudiar conceptos como la añadidura, eliminación, modificación o selección de filas revisaremos los elementos que hacen posibles la referencia a una fila. Como ya se mencionó previamente, una fila es a la vez un nodo dentro de un árbol, a la cual debemos ser capaces de *apuntar* si queremos modificarla. Es por esto que GTK brinda tres elementos, todos de distinta naturaleza, que nos permitirán referirnos a cada *nodo* sin complicaciones y de manera transparente. Estos elementos son GtkTreeIter, GtkTreePath y GtkTreeRowReference.

### GtkTreeIter: Una referencia directa.

Un GtkTreeIter es una estructura que se utiliza para apuntar a una fila en un modelo y que es soportada tanto por GtkListStore como por GtkTreeStore. Cualquier implementación personalizada de un GtkTreeModel debe ser capaces de soportar las funciones que actuan sobre GtkTreeIter. La composición interna de ésta estructura no es en absoluto relevante para un desarrollador y jamás debe accederse directamente a ésta ni modificarse sin usar las funciones destinadas para ésto.

Enumeraremos algunas de éstas funciones para dar la idea sobre su utilidad.

- `gtk_tree_model_get_iter_first()` - Obtiene un GtkTreeIter al primer elemento del nivel superior de un módelo. En un GtkTreeStore éste elemento sería la raíz y en un GtkListStore la primera fila del modelo.
- `gtk_tree_model_get_iter_next()` - Obtiene el siguente GtkTreeIter al nivel del Iter dado, si este existe.
- `gtk_tree_model_get_iter_child()` - Obtiene un GtkTreeIter apuntando al primer hijo del GtkTreeIter actualmente referenciado. Obviamente, esta función es util para árboles, no para listas.
- `gtk_tree_model_iter_parent()` - Dado un GtkTreeIter obtiene el padre de la fila referenciada por éste.

Para detalles sobre los prototipos de éstas y otras funciones, recomendamos ver la API de Referencia de GtkTreeModel[1].

Los GtkTreeIter son utilizados para almacenar y obtener datos hacia y desde un modelo. Cuando se agregan filas a un modelo se obtiene un GtkTreeIter como resultado. La validez de esta estructura de datos es temporal - tan pronto como el modelo cambie o emita una señal un GtkTreeIter referenciando a una fila en él puede transformarse en inválido, por lo que no debe pensarse jamás en un iter más allá que para acción directa e inmediata sobre un modelo.

Si se observa con cuidado la API de GtkTreeModel se podrá ver que las únicas funciones que actuan única y exclusivamente sobre GtkTreeIter de manera directa son `gtk_tree_iter_copy()` y `gtk_tree_iter_free()` y además *no se recomienda usarlas en aplicaciones*. Esto no hace más que mostrarnos que la utilidad de los iters es solo temporal y no son utiles para, por ejemplo, ser utilizados como datos de usuario en callbacks. Considerando que ni siquiera existe una función constructora, ¿debemos nosotros, desarrolladores de aplicaciones, reservar dinamicamente la memoria para almacenar un GtkTreeIter? La respuesta es no. GtkTreeIter debe ser utilizado solo en el ámbito de la función donde se obtuvo, y es por eso que basta con declararlos de manera estática y usarlos como tal:

```
void
get_data_from_model (GtkTreeModel *model)
{
        GtkTreeIter iterator;
        gint age;

        if (gtk_tree_model_get_iter_first (model, &iterator)
            != FALSE) {
                gtk_tree_model_get (model, &iterator,
                                    COLUMN_AGE, &age,
                                    -1);
        }
        ...
}
```

### GtkTreePath: Una referencia absoluta.

### GtkTreeRowReference: Una referencia persistente.

## Modelos de datos ordenados

Hasta ahora hemos visto la flexibilidad de GtkTreeView a la hora de visualizar datos. Hemos visto que pueden estar organizados los datos en cualquier tipo de estructura de datos, que podemos crearnos nuestros modelos de datos y con ellos, visualizar los datos de forma sencilla.

Ahora nos toca analizar como facilitar el acceso a dichos datos en el caso de que el volumen a mostrar sea muy grande. Imaginemos que mostramos una lista de nombres, y que dicha lista tenga cientos de entradas. Y ahora pensemos que el usuario quiere localizar un nombre en dicha lista. Si los nombres no aparecen ordenados de alguna forma, el usuario puede tardar mucho tiempo si quiere localizar en dicha lista un nombre. O por ejemplo, pensemos que hay otro campo que sea la edad. Si el usuario quiere localizar por edad a ciertos nombres, lo que realmente va a necesitar es la posibilidad de ordenar todos los campos por edad.

Resumiendo, que necesitamos la funcionalidad que le permita al usuario seleccionar una columna (campo) de los datos y ordenar por ella la visualización de los datos. Y como no podía ser de otra forma, GtkTreeView soporta esta funcionalidad. Para ello tan sólo tenemos que asegurarnos de proporcionar la función o funciones de ordenación para nuestros datos.

Vamos a ver que en los casos más sencillo, de ordenar cadenas u ordenar enteros, la propia GTK se encarga de hacer el trabajo por nosotros. En caso de que queramos ordenar de formas especiales, entoces sí será necesario proporcionar una función de ordenación.

Para convertir un modelo en un modelo ordenado basta con pasar el modelo de datos a la función `gtk_tree_model_sort_new_with_model()`, que es parte de `gtk-2.0/gtk/gtktreemodelsort.h`. Para acceder luego al contenido del modelo ordenado, hay que tener cuidado de llamar a las funciones de esta clase a la hora de acceder a los contenidos del modelo de datos.

Por último, ¿cómo indica un usuario que quiere ordenar por una columna? A la hora de crear la columna y visualizarla, como ya veremos en la

sección siguiente, llamamos sobre la columna ya creada a la función `gtk_tree_view_column_set_sort_column_id()`.

Vamos a mostrar todo lo hasta ahora expuesto con el ejemplo de la lista de la compra, pero modificado para que soporte ordenación. En nuestro caso, el modelo de datos GtkListStore, basta con indicar que la columna se puede ordenar para pasar a tener la posibilidad de pulsar sobre la cabezera de la columna y que se ordenen los datos. El propio modelo GtkListStore ya incorpora la funcionalidad de ordenación de datos.

Por lo tanto, los únicos cambios a realizar son:

```
static void
add_columns (GtkTreeView *treeview)
{
        GtkCellRenderer   *renderer;
        GtkTreeModel      *model = gtk_tree_view_get_model (treeview);
        GtkTreeViewColumn *col;

        /* columna de producto */
        renderer = gtk_cell_renderer_text_new ();
        col = gtk_tree_view_column_new_with_attributes ("Producto",
                                                         renderer,
                                                         "text", COLUMN_PRODUCT,
                                                         NULL);
        gtk_tree_view_column_set_sort_column_id (col, COLUMN_PRODUCT);
        gtk_tree_view_append_column (treeview, col);
}
```

Si ahora compilamos y ejecutamos el programa de la lista de la compra:

Que duda cabe que si nuestro modelo de datos es más complejo, por ejemplo en árbol, deberemos de profundizar mucho más en la metodología de ordenación de datos. Si utilizamos el modelo de datos GtkTreeStore de nuevo tendremos ordenación gratis, al igual que en los casos con modelos de datos sencillos.

Pero si nuestros modelos de datos deben de ordenarse por ejemplo por objetos, tendremos que crear las funciones que especifiquen como se ordenan los objetos.

## Modelos de datos a medida

Está claro que, a pesar del gran número de casos que podemos cubrir con los dos modelos estándar, no siempre nuestros datos serán cadenas de caracteres. Es aquí donde aparece la gran potencia de GtkTreeView ya que el modelo de datos que podemos usar es cualquier que nosotros definamos. Tan sólo deberemos de implementar las interfaces adecuadas.

Y nada mejor que un ejemplo de un desarrollo real donde se tuvo esa necesidad para mostrar como se resolvió: el gestor de proyectos MrProject. En este proyecto se ha utilizado de varias formas el widget de GtkTreeView y nos va a servir para analizar las posibles combinaciones de su uso. En el siguiente diagrama UML podemos ver cómo se implementa la interfaz de los modelos de datos de un GtkTreeView siguiendo diferentes aproximaciones en función de la necesidad que se tenga. Por ejemplo, en muchas ocasiones se necesitaba mostrar datos en forma de lista, estando dichos datos contenidos dentro de las propiedades de objetos. Por ello se definió la clase MgListModel que facilitó mucho la implementación de este tipo de modelos de datos.

En otra ocasión, se necesitó mostrar en forma de árbol datos de objetos, relacionados entre sí, por lo que se implementó de forma directa la interfaz de modelos de datos.

Y en otras ocasiones, se utilizó los modelos de datos predefinidos GtkTreeStore y GtkListStore. Todo ello lo vemos resumido en el siguiente diagrama UML.

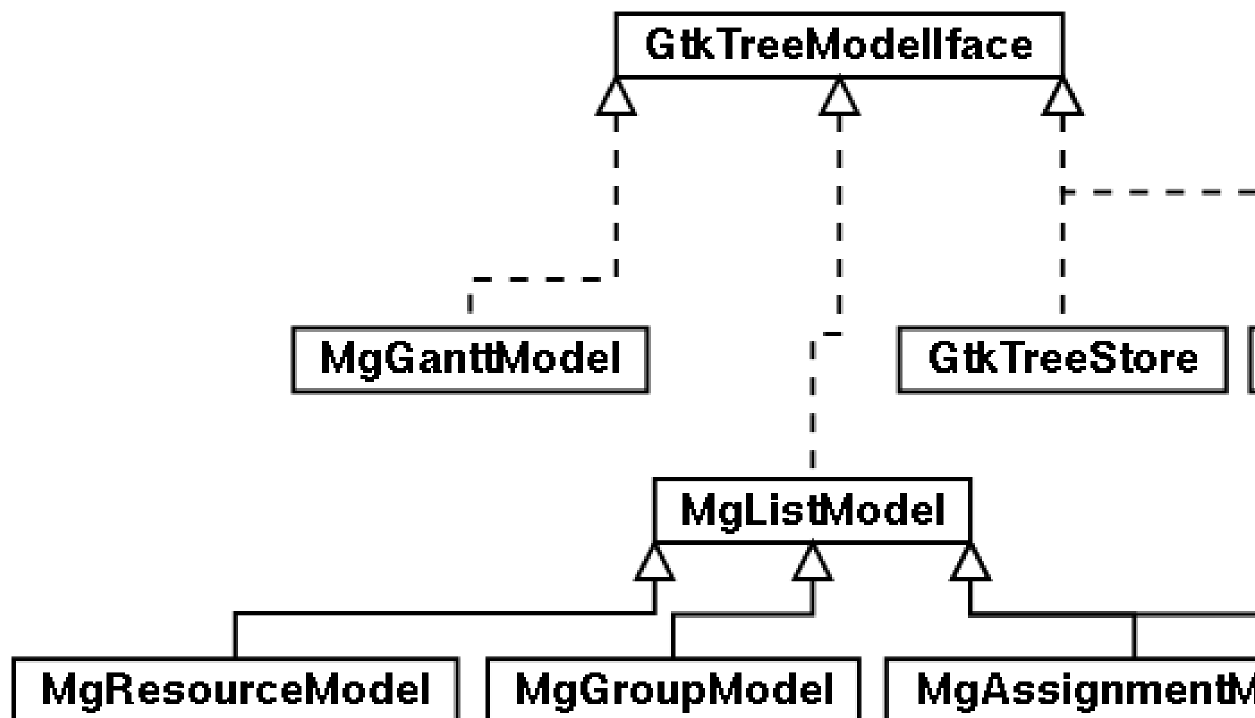



**Figura 8-3. UML de interfaces para modelos de datos**

Vamos a comenzar analizando MgListModel y como simplifica en gran medida la creación de los modelos de datos que se usan en MrProject, que serán bastante similares a los que se puede encontrar en otro tipo de proyectos.

El modelo de datos lo constituyen una serie de objetos que son los que se van a visualizar dentro de la interfaz gráfica. Estos objetos tienen mucha información y no queremos mostrar toda, si no únicamente parte de la misma.

Para poder implementar un nuevo modelo de datos debemos de implementar la interfaz GtkTreeModelIface que se puede consultar dentro del fichero de cabeceras de GTK `/usr/include/gtk-2.0/gtk/gtktreemodel.h` que es precisamente lo que hace la clase `MgListModel`, en la que se basarán muchos de los modelos de datos posteriores en MrProject. Vamos con el análisis de esta clase. Vamos a centrarnos en las partes del código que nos interesan. Si el lector quiere consultar los fuentes completos, no tiene mas que obtener el código fuente de MrProject e ir a los ficheros que vamos a ir indicando. En primer lugar comenzamos con `mg-list-model.h`.

## La interfaz `MgListModel`

Ha llegado el momento analizar esta interfaz, que implementa gran parte de GtkTreeModelIface, pero dejando sin implementar los métodos fundamentales dentro de las necesidades de los modelos de datos de MrProject.

```
struct _MgListModel
{
        GObject          parent;

        MgListModelPriv *priv;
};

struct _MgListModelClass
{
```

```
        GObjectClass parent_class;

        gint  (*get_n_columns)   (GtkTreeModel *tree_model);
        GType (*get_column_type) (GtkTreeModel *tree_model,
                                  gint          column);
        void  (*get_value)       (GtkTreeModel *tree_model,
                                  GtkTreeIter  *iter,
                                  gint          column,
                                  GValue       *value);
};

GType            mg_list_model_get_type      (void);
void             mg_list_model_append        (MgListModel      *model,
                                              MrpObject        *object);
void             mg_list_model_remove        (MgListModel      *model,
                                              MrpObject        *object);
void             mg_list_model_update        (MgListModel      *model,
                                              MrpObject        *object);
GtkTreePath *    mg_list_model_get_path      (MgListModel      *model,
                                              MrpObject        *object);
MrpObject *      mg_list_model_get_object    (MgListModel      *model,
                                              GtkTreeIter      *iter);
void             mg_list_model_set_data      (MgListModel      *model,
                                              GList            *data);
GList *          mg_list_model_get_data      (MgListModel      *model);
```

Vemos que la clase implementa parte de la interfaz GtkTreeModelIface aunque deja tres métodos sin implementar, que serán los únicos que tendrán que implementar las clases que utilicen como interfaz de `MgListModel`: `get_n_columns()`, `get_column_type()` y `get_value()`.

La interfaz GtkTreeModelIface es mucho más amplia, pero `MgListModel` se encarga de realizar una implementación por defecto de todos los demás métodos. Para verlo vamos a analizar el fichero `mg-list-model.c` y en concreto, el inicializador de la clase y la construcción propiamente dicha de la clase.

```
GtkType
mg_list_model_get_type (void)
{
        static GType type = 0;

        if (!type) {
                static const GTypeInfo info =
                {
                        sizeof (MgListModelClass),
                        NULL,                /* base_init */
                        NULL,                /* base_finalize */
                        (GClassInitFunc) mlm_class_init,
                        NULL,                /* class_finalize */
                        NULL,                /* class_data */
                        sizeof (MgListModel),
                        0,
                        (GInstanceInitFunc) mlm_init,
                };

                static const GInterfaceInfo tree_model_info =
                {
                        (GInterfaceInitFunc) mlm_tree_model_init,
                        NULL,
                        NULL
                };

                type = g_type_register_static (G_TYPE_OBJECT,
                                               "MgListModel",
                                               &info, 0);
```

```
                g_type_add_interface_static (type,
                                             GTK_TYPE_TREE_MODEL,
                                             &tree_model_info);
        }

        return type;
}
```

La parte que nos interesa del constructor de la clase es donde indicamos que nuestra clase implementa la interfaz GTK_TYPE_TREE_MODEL, algo que vamos a ver en el inicializador de la clase, `mlm_tree_model_init()` y, si nos fijamos en el constructor de clase `mlm_class_init()`, vamos a ver los tres métodos que se dejan sin implementar de la interfaz de GtkTreeModelIface.

```
static void
mlm_class_init (MgListModelClass *klass)
{
        GObjectClass *object_class;

        parent_class = g_type_class_peek_parent (klass);
        object_class = (GObjectClass*) klass;

        object_class->finalize = mlm_finalize;

        klass->get_n_columns   = NULL;
        klass->get_column_type = NULL;
        klass->get_value       = NULL;
}
```

Estos tres métodos son los que dejamos obligatorios para que sean implementados por cada uno de los modelos de datos que se basen en esta interfaz.

Veamos ahora como nuestra clase implementa los demás métodos de la interfaz.

```
mlm_tree_model_init (GtkTreeModelIface *iface)
{
        iface->get_iter        = mlm_get_iter;
        iface->get_path        = mlm_get_path;
        iface->iter_next       = mlm_iter_next;
        iface->iter_children   = mlm_iter_children;
        iface->iter_has_child  = mlm_iter_has_child;
        iface->iter_n_children = mlm_iter_n_children;
        iface->iter_nth_child  = mlm_iter_nth_child;
        iface->iter_parent     = mlm_iter_parent;

        /* Llama al método class->function por lo que las subclases pueden
  poner sus propios métodos */
        iface->get_n_columns   = mlm_get_n_columns;
        iface->get_column_type = mlm_get_column_type;
        iface->get_value       = mlm_get_value;
}
```

Comencemos precisamente con estos tres últimos métodos, por ejemplo con el que nos permite obtener el número de columnas de un modelo de datos.

```
static gint
mlm_get_n_columns (GtkTreeModel *tree_model)
{
        MgListModelClass *klass;

        klass = MG_LIST_MODEL_GET_CLASS (tree_model);
```

```
        if (klass->get_n_columns) {
                return klass->get_n_columns (tree_model);
        }

        g_warning ("Tienes que implementar get_n_columns!");

        return -1;
}
```

Vemos como en este método, se obtiene la clase real del objeto que está siendo utilizado, por ejemplo MgGroupModel, y se intenta invocar al método `get_n_columns()` de esta clase. En el caso de que la clase que dice implementar `MgListModel` no lo haga, se saca un mensaje de aviso y se devuelve -1.

Veamos ahora un ejemplo de uno de los métodos que si que implementa esta interfaz, clase abstracta en terminología de C++, y que evita que los modelos de datos basados en esta interfaz tenga que realizar esta labor. Para ser capaces de realizar esta implementación, nos basamos en que el modelo en realidad es una lista de objetos.Este es el único requisito. Si quisieramos una estructura de datos en árbol, no nos valdrá esta interfaz para ayudarnos en nuestra labor.

```
void
mg_list_model_update (MgListModel *model, MrpObject *object)
{
        MgListModelPriv *priv;
        GtkTreePath     *path;
        GtkTreeIter      iter;
        gint             i;

        g_return_if_fail (MG_IS_LIST_MODEL (model));
        g_return_if_fail (MRP_IS_OBJECT (object));

        priv = model->priv;

        i = g_list_index (priv->data_list, object);

        path = gtk_tree_path_new ();
        gtk_tree_path_append_index (path, i);

        gtk_tree_model_get_iter (GTK_TREE_MODEL (model), &iter, path);

        gtk_tree_model_row_changed (GTK_TREE_MODEL (model), path,
                                    &iter);

        gtk_tree_path_free (path);
}
```

Vemos que para implementar el método que nos permite actualizar un elemento del modelo de datos, lo localizamos en la lista de objetos (`g_list_index()`) y utilizando los métodos que nos proporciona el propio GtkTreeModel, provocamos el cambio en el modelo de datos. Este cambio se propagará a todas las interfaces gráficas que utilicen este modelo de datos y se actualizarán para reflejar los cambios.

## MgGroupModel: El modelo de datos completo

Hasta el momento no hemos creado mas que una clase que nos facilita implementar modelos de datos específicos basados en la idea de que son una lista de objetos de los que queremos mostrar ciertas características. Por ello nos basta con que la parte que cambia de un modelo de datos a otro es el número de columnas a mostrar, el

tipo de cada columna y el valor que almacena. En el caso de MgGroupModel dichos métodos se implementan en:

```
mgm_class_init (MgGroupModelClass *klass)
{
        GObjectClass     *object_class;
        MgListModelClass *lm_class;

        parent_class = g_type_class_peek_parent (klass);
        object_class = G_OBJECT_CLASS (klass);
        lm_class     = MG_LIST_MODEL_CLASS (klass);

        object_class->finalize = mgm_finalize;

        lm_class->get_n_columns   = mgm_get_n_columns;
        lm_class->get_column_type = mgm_get_column_type;
        lm_class->get_value       = mgm_get_value;
}
```

La implementación de `mgm_get_n_columns()` y `mgm_get_column_type()` es la siguiente:

```
static gint
mgm_get_n_columns (GtkTreeModel *tree_model)
{
        return NUMBER_OF_GROUP_COLS;
}

static GType
mgm_get_column_type (GtkTreeModel *tree_model,
                     gint          column)
{
        switch (column) {
        case GROUP_COL_NAME:
        case GROUP_COL_MANAGER_NAME:
        case GROUP_COL_MANAGER_PHONE:
        case GROUP_COL_MANAGER_EMAIL:
                return G_TYPE_STRING;

        case GROUP_COL_GROUP_DEFAULT:
                return G_TYPE_BOOLEAN;

        default:
                return G_TYPE_INVALID;
        }
}
```

que se basan en que tenemos una unión que contiene todas las columnas que son visibles de un grupo. El añadir nuevas columnas sería muy sencillo ya que bastaría con añadirlas a la unión de columnas y actualizar los métodos que realizan el tratamiento de dichas columnas. De forma automática las interfaces gráficas se verían también actualizadas. De igual forma se podría eliminar una columna, o permitir desde algún diálogo especificar que columnas se quieren ver en la interfaz y cuales no. Veamos esa unión con las columnas:

```
enum {
        GROUP_COL_NAME,
        GROUP_COL_GROUP_DEFAULT,
        GROUP_COL_MANAGER_NAME,
        GROUP_COL_MANAGER_PHONE,
        GROUP_COL_MANAGER_EMAIL,
        NUMBER_OF_GROUP_COLS
};
```

El método quizá más complejo es el que se encarga de obtener los valores de cada una de las columnas. Estos valores se obtienen del objeto que se está mostrando actualmente, y en nuestro caso, veremos que en su mayoría son propiedades del objeto.

```
static void
mgm_get_value (GtkTreeModel *tree_model,
               GtkTreeIter  *iter,
               gint          column,
               GValue       *value)
{
        gchar           *str = NULL;
        MrpGroup        *group, *default_group;
        MgGroupModelPriv *priv;
        gboolean         is_default;

        g_return_if_fail (MG_IS_GROUP_MODEL (tree_model));
        g_return_if_fail (iter != NULL);

        priv = MG_GROUP_MODEL (tree_model)->priv;
        group = MRP_GROUP (mg_list_model_get_object
                           (MG_LIST_MODEL (tree_model), iter));
```

Aquí es donde hemos obtenido de la lista de objetos (`mg_list_model_get_object()`) el objeto, en nuestro caso un MrpGroup, del que queremos mostrar la información. Vemos que hacemos uso de uno de los métodos públicos de la clase `MgListModel` para obtenerdicho objeto de la lista. Ahora, en función de la columna que nos estén pidiendo de dicho MrpGroup, devolvemos un valor u otro.

```
        switch (column) {
        case GROUP_COL_NAME:
                mrp_object_get (group, "name", &str, NULL);
                g_value_init (value, G_TYPE_STRING);
                g_value_set_string (value, str);
                g_free (str);
                break;

        case GROUP_COL_GROUP_DEFAULT:
                g_object_get (priv->project,
                              "default-group", &default_group,
                              NULL);

                is_default = (group == default_group);

                g_value_init (value, G_TYPE_BOOLEAN);
                g_value_set_boolean (value, is_default);
                break;

        case GROUP_COL_MANAGER_NAME:
                mrp_object_get (group, "manager_name", &str, NULL);
                g_value_init (value, G_TYPE_STRING);
                g_value_set_string (value, str);
                g_free (str);
                break;

        case GROUP_COL_MANAGER_PHONE:
                mrp_object_get (group, "manager_phone", &str, NULL);
                g_value_init (value, G_TYPE_STRING);
                g_value_set_string (value, str);
                g_free (str);

                break;
```

```
                case GROUP_COL_MANAGER_EMAIL:
                        mrp_object_get (group, "manager_email", &str, NULL);
                        g_value_init (value, G_TYPE_STRING);
                        g_value_set_string (value, str);
                        g_free (str);

                        break;

                default:
                        g_assert_not_reached ();
                }
        }
```

y estos valores que hemos obtenido aquí serán los que se muestren dentro de la interfaz gráfica. Y ha llegado el momento de mostrar como aparece todo esto dentro de la interfaz gráfica de MrProject.

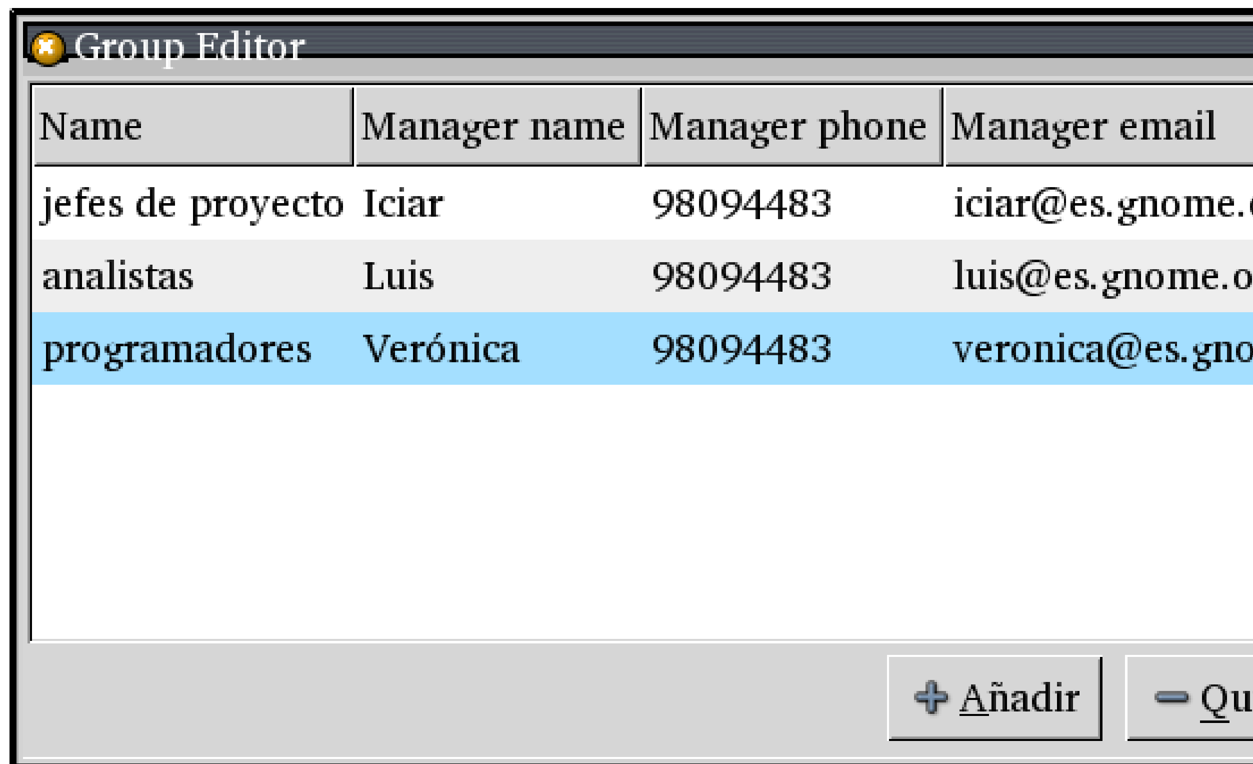



**Figura 8-4. Grupos en MrProject**

## Implementación completa de modelo de datos

Lo que hemos analizado hasta ahora es aplicable a los casos en los que tengamos una lista de datos a visualizar pero no siempre es así, lo que por ejemplo dentro de MrProject ha obligado a que algunos modelos de datos, como el de la vista Gantt, sean implementados por completo desde cero.

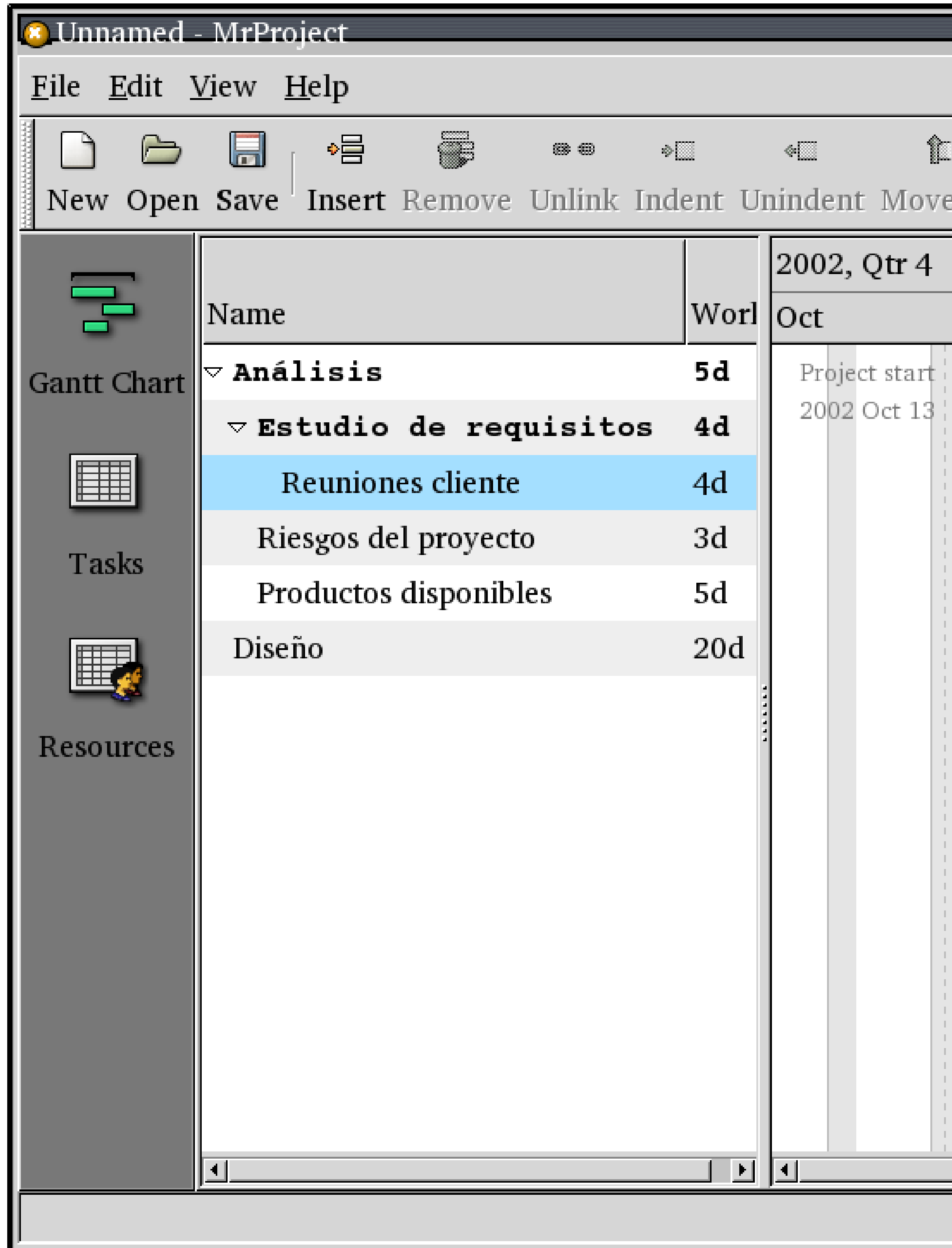



**Figura 8-5. Vista Gantt con tareas anidadas**

Para resolver este modelo de datos, se ha tenido que crear un modelo dentro de `mg-gantt-model.c/h` que implementa por completo todos los métodos de la interfaz GtkTreeModelIface. Veámoslo:

```
GType
mg_gantt_model_get_type (void)
{
        static GType type = 0;
        ...
        static const GInterfaceInfo tree_model_info = {
                (GInterfaceInitFunc) gantt_model_tree_model_init,
                NULL,
                NULL
        };
        ...
}
```

que nos indica que el método que utilizamos para inicializar la interfaz GInterfaceInfo que implementa esta clase es `gantt_model_tree_model_init()`.

```
static void
gantt_model_tree_model_init (GtkTreeModelIface *iface)
{
        iface->get_n_columns = gantt_model_get_n_columns;
        iface->get_column_type = gantt_model_get_column_type;
        iface->get_iter = gantt_model_get_iter;
        iface->get_path = gantt_model_get_path;
        iface->get_value = gantt_model_get_value;
        iface->iter_next = gantt_model_iter_next;
        iface->iter_children = gantt_model_iter_children;
        iface->iter_has_child = gantt_model_iter_has_child;
        iface->iter_n_children = gantt_model_iter_n_children;
        iface->iter_nth_child = gantt_model_iter_nth_child;
        iface->iter_parent = gantt_model_iter_parent;
}
```

Vemos que aquí implementamos por completo todas las funciones de la interfaz GtkTreeModelIface y que, entre otras funciones, permiten que los datos se organicen en estructuras de árbol.

[FIXME] Destacar las partes más importantes de la implementación en árbol

## GtkTreeView: Visualización

Hasta ahora hemos hablado mucho de modelos de datos pero en ningún momento hemos indicado mas que con código como se muestran dichos datos al usuario. Ha llegado el momento de ver lo sencilla que es la visualización de dichos datos y como, gracias al patrón modelo/vista, podemos añadir contenidos a los datos y estos se visualizarán de forma automática al usuario. O como podemos ordenar los datos y de nuevo, la vista del usuario se actualizará de forma automática en función de como hayamos ordenado los datos. O quizá, como unos mismos datos se visualizan en más de una vista, y se actualizan de forma sincronizada todas las vistas de los mismos.

### Listas

### Árboles

### Celdas

## Cortar y pegar en GtkTreeView

## Notas

1. http://developer.gnome.org/doc/API/2.0/gtk/GtkTreeModel.html

# Capítulo 9. Editor de texto multilínea

GTK+ provee un estructura para el trabajo de edición multilínea bastante poderoso y es muy diferente con la forma de trabajar en Gtk+ 1.2. La forma básica de trabajo consiste en GTkTextBuffer y GtkTextView. GtkTextBuffer representa el texto editado, mientras que GtkTextView es el widget encargado de mostrar un GtkTextBuffer, representando una vista.

Cada GtkTextBuffer puede tener múltiples vistas. El manejo de texto de GTK+ emplea UTF-8 como juego de caracteres, lo cual permite disponer de las mismas aplicaciones para idiomas tan disímiles como japonés, árabe y castellano; inclusive dentro del mismo texto. Sin embargo, esta característica cambia algunos conceptos tradicionalmente empleados en el manejo de texto, y esto es por que en UTF-8 los acaracteres no se codifican en un sólo byte, por lo tanto, un caracter deja de ser un byte.

De esta forma, para referirnos al número de caracteres lo haremos llamándolos «offsets», mientras que a la cantidad de bytes le llamaremos «índices». Es muy importante mantener clara la idea de su funcionamiento de lo contrario lograremos un buen desempeño con ASCII, pero obténdremos errores cuando aparezcan carateres multibyte. Si ejecutamos nuestra aplicación desde un terminal, veremos mensajes [invalid UTF-8] o similar, además de un posible comportamiento errático.

El manejo de texto multilínea de GTK+ 1.2 se limita sólo a texto de 8 bits; en cambio, en GTK+ 2.x es posible definir atributos para trozos de texto a través de la definición de marquillas, así es posible crear texto coloreado, emplear distinas tipografías, utilizar negrita, cursiva e incluso controlar atributos no visibles tal como reaccionar en forma distinta si el cursor se encuentra sobre el texto. Las marquillas se representan a través del objeto GtkTextTag el cual puede aplicarse desde un rango de texto hasta cualquier número del búfer. Las marcas se alcamenan dentro de un GtkTextTagTable y es posible utilizar varias marcas en un mismo trozo de texto. Existe como restricción, que un GtkTextBuffer sólo puede disponer de un sólo GtkTextTagTable; pero un GtkTextTagTable puede ser compartido entre varios GtkTextBuffer.

## Manipulación de texto

La manipulación del texto se hace a través de los iteradores, usando un GtkTextIter. Un iterador representa la posición entre dos caracteres en el búfer de texto y no son válidos indefinidamente. Cuando se modifica el búfer de tal forma que el número de caracteres disminuye, todos los iteradores que apuntan a posiciones mayores al tamaño dejan de ser válidos. En realidad, dejan de ser válidos, toda vez que el texto alrededor del iterador se modifique. Por ejemplo, el proceso de borrar 5 caracteres e insertar otros 5 caracteres, inválida a un iterador, independientemente que sigan con el mismo número de caracteres. Es por esta razón que los iteradores no se pueden emplear para preservar las posiciones entre modificaciones del búfer.

Si se desea preservar una posición, se debe emplear el objeto GtkTextMark el cual puede ser visto como un cursor invisible que flota en el búfer, guardando su posición. Si se borra el texto alrededor de la marca, ésta mantiene su posición una vez que la posición se vuelve a ocupar.

En resumen, tenemos GtkTextView que es el widget que representará visualmente el texto, GtkTextBuffer que es el texto en sí, GtkTextIter que nos ayuda a desplazarnos a través del texto, GtkTextMark para colocar marcadores o referencias en el texto, GtkTextTag que permite manejar las marquillas a través de una tabla GtkTextTagTable; todo esta división de roles a la hora de manipular texto es por que se sigue el patrón de diseño MVC (Model View Controller).

## Mini editor de texto

Para la compresión de los temas tratados con anterioridad se presenta un ejemplo

de un mini editor de etxto que tendrá las funciones de limpiar, cortar, copiar, pegar, negrita, subrayado, tachado y coloreado, el cual tendrá el siguiente aspecto:

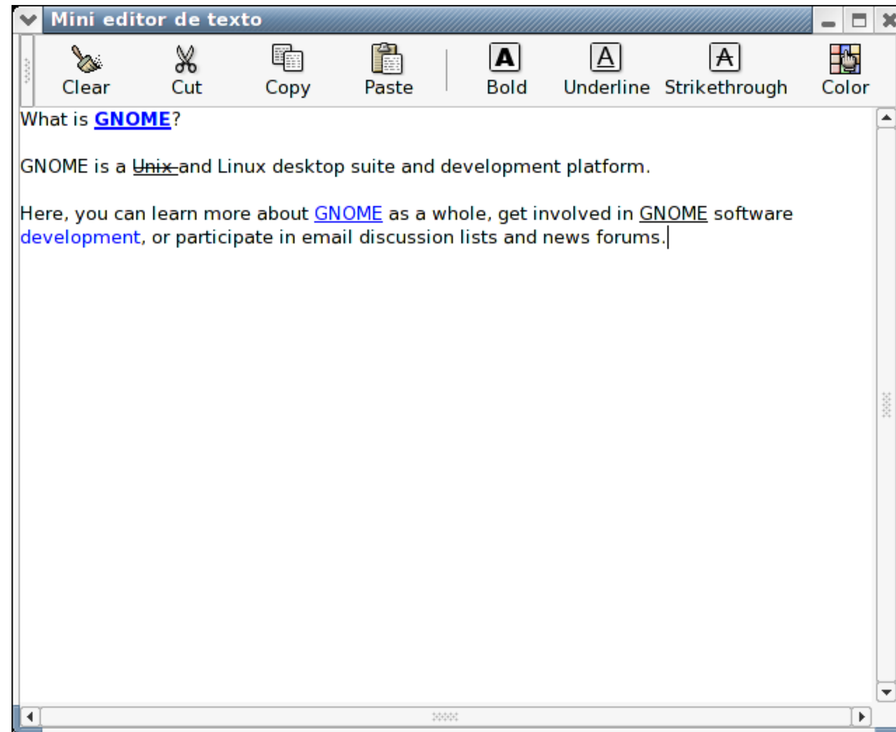



**Figura 9-1. Mini editor de texto**

Ahora veamos el código de esta aplicación:

```
#include <gtk/gtk.h>

GtkWidget *create_window ();
void create_tags (GtkTextBuffer * buffer);
void on_button_clear_clicked (GtkButton * button, gpointer user_data);
void on_button_cut_clicked (GtkButton * button, gpointer user_data);
void on_button_copy_clicked (GtkButton * button, gpointer user_data);
void on_button_paste_clicked (GtkButton * button, gpointer user_data);
void on_button_bold_clicked (GtkButton * button, gpointer user_data);
void on_button_underline_clicked (GtkButton * button, gpointer user_data);
void on_button_strike_clicked (GtkButton * button, gpointer user_data);
void on_button_color_clicked (GtkButton * button, gpointer user_data);

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

    gtk_init (&argc, &argv);

    window = create_window ();
    gtk_widget_show_all (window);

    gtk_main ();
    return 0;
```

```
}

GtkWidget *
create_window ()
{
    GtkWidget *window;
    GtkWidget *vbox_main;
    GtkWidget *handlebox;
    GtkWidget *toolbar;
    GtkWidget *button_clear;
    GtkWidget *button_cut;
    GtkWidget *button_copy;
    GtkWidget *button_paste;
    GtkWidget *button_bold;
    GtkWidget *button_underline;
    GtkWidget *button_strike;
    GtkWidget *button_color;
    GtkWidget *scrolledwindow;
    GtkWidget *textview;
    GtkTextBuffer *buffer;

    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title (GTK_WINDOW (window), "Mini editor de texto");
    gtk_window_set_default_size (GTK_WINDOW (window), 400, 500);

    vbox_main = gtk_vbox_new (FALSE, 0);
    gtk_container_add (GTK_CONTAINER (window), vbox_main);

    handlebox = gtk_handle_box_new ();
    gtk_box_pack_start (GTK_BOX (vbox_main), handlebox, FALSE, FALSE,
                0);

    toolbar = gtk_toolbar_new ();
    gtk_container_add (GTK_CONTAINER (handlebox), toolbar);

    button_clear = gtk_toolbar_insert_stock (GTK_TOOLBAR (toolbar),
                        "gtk-clear",
                        NULL,
                        NULL, NULL, NULL, -1);
    button_cut = gtk_toolbar_insert_stock (GTK_TOOLBAR (toolbar),
                          "gtk-cut",
                          NULL, NULL, NULL, NULL, -1);
    button_copy = gtk_toolbar_insert_stock (GTK_TOOLBAR (toolbar),
                       "gtk-copy",
                       NULL,
                       NULL, NULL, NULL, -1);
    button_paste = gtk_toolbar_insert_stock (GTK_TOOLBAR (toolbar),
                        "gtk-paste",
                        NULL,
                        NULL, NULL, NULL, -1);

    gtk_toolbar_append_space (GTK_TOOLBAR (toolbar));

    button_bold = gtk_toolbar_insert_stock (GTK_TOOLBAR (toolbar),
                       "gtk-bold",
                       NULL,
                       NULL, NULL, NULL, -1);
    button_underline = gtk_toolbar_insert_stock (GTK_TOOLBAR (toolbar),
                            "gtk-underline",
                            NULL,
                            NULL, NULL, NULL, -1);
    button_strike = gtk_toolbar_insert_stock (GTK_TOOLBAR (toolbar),
                         "gtk-strikethrough",
                         NULL,
                         NULL, NULL, NULL, -1);
    button_color = gtk_toolbar_insert_stock (GTK_TOOLBAR (toolbar),
```

```
                         "gtk-select-color",
                         NULL,
                         NULL, NULL, NULL, -1);
    scrolledwindow = gtk_scrolled_window_new (NULL, NULL);
    gtk_box_pack_start (GTK_BOX (vbox_main), scrolledwindow, TRUE,
                TRUE, 0);

    textview = gtk_text_view_new ();
    gtk_container_add (GTK_CONTAINER (scrolledwindow), textview);

    buffer = gtk_text_view_get_buffer (GTK_TEXT_VIEW (textview));
    create_tags (buffer);

    g_signal_connect ((gpointer) window, "delete_event",
              G_CALLBACK (gtk_main_quit), NULL);
    g_signal_connect ((gpointer) button_clear, "clicked",
              G_CALLBACK (on_button_clear_clicked),
              (gpointer) textview);
    g_signal_connect ((gpointer) button_cut, "clicked",
              G_CALLBACK (on_button_cut_clicked),
              (gpointer) textview);
    g_signal_connect ((gpointer) button_copy, "clicked",
              G_CALLBACK (on_button_copy_clicked),
              (gpointer) textview);
    g_signal_connect ((gpointer) button_paste, "clicked",
              G_CALLBACK (on_button_paste_clicked),
              (gpointer) textview);
    g_signal_connect ((gpointer) button_bold, "clicked",
              G_CALLBACK (on_button_bold_clicked),
              (gpointer) textview);
    g_signal_connect ((gpointer) button_underline, "clicked",
              G_CALLBACK (on_button_underline_clicked),
              (gpointer) textview);
    g_signal_connect ((gpointer) button_strike, "clicked",
              G_CALLBACK (on_button_strike_clicked),
              (gpointer) textview);
    g_signal_connect ((gpointer) button_color, "clicked",
              G_CALLBACK (on_button_color_clicked),
              (gpointer) textview);
    return window;
}

void
on_button_clear_clicked (GtkButton * button, gpointer user_data)
{
    GtkTextBuffer *textbuffer = NULL;

    g_assert (GTK_IS_TEXT_VIEW (user_data));

    textbuffer = gtk_text_view_get_buffer (GTK_TEXT_VIEW (user_data));

    gtk_text_buffer_delete_selection (textbuffer, TRUE, TRUE);
}

void
on_button_cut_clicked (GtkButton * button, gpointer user_data)
{
    GtkTextBuffer *textbuffer = NULL;

    g_assert (GTK_IS_TEXT_VIEW (user_data));

    textbuffer = gtk_text_view_get_buffer (GTK_TEXT_VIEW (user_data));

    gtk_text_buffer_cut_clipboard (textbuffer,
                       gtk_clipboard_get (GDK_NONE), TRUE);
}
```

```
void
on_button_copy_clicked (GtkButton * button, gpointer user_data)
{
    GtkTextBuffer *textbuffer = NULL;

    g_assert (GTK_IS_TEXT_VIEW (user_data));

    textbuffer = gtk_text_view_get_buffer (GTK_TEXT_VIEW (user_data));

    gtk_text_buffer_copy_clipboard (textbuffer,
                    gtk_clipboard_get (GDK_NONE));
}

void
on_button_paste_clicked (GtkButton * button, gpointer user_data)
{
    GtkTextBuffer *textbuffer = NULL;

    g_assert (GTK_IS_TEXT_VIEW (user_data));

    textbuffer = gtk_text_view_get_buffer (GTK_TEXT_VIEW (user_data));

    gtk_text_buffer_paste_clipboard (textbuffer,
                     gtk_clipboard_get (GDK_NONE),
                     NULL, TRUE);
}

void
on_button_bold_clicked (GtkButton * button, gpointer user_data)
{
    GtkTextBuffer *textbuffer = NULL;
    GtkTextIter start, end;

    g_assert (GTK_IS_TEXT_VIEW (user_data));

    textbuffer = gtk_text_view_get_buffer (GTK_TEXT_VIEW (user_data));

    gtk_text_buffer_get_selection_bounds (textbuffer, &start, &end);

    gtk_text_buffer_apply_tag_by_name (textbuffer, "bold", &start,
                       &end);
}

void
on_button_underline_clicked (GtkButton * button, gpointer user_data)
{
    GtkTextBuffer *textbuffer = NULL;
    GtkTextIter start, end;

    g_assert (GTK_IS_TEXT_VIEW (user_data));

    textbuffer = gtk_text_view_get_buffer (GTK_TEXT_VIEW (user_data));

    gtk_text_buffer_get_selection_bounds (textbuffer, &start, &end);

    gtk_text_buffer_apply_tag_by_name (textbuffer, "underline", &start,
                       &end);
}


void
on_button_strike_clicked (GtkButton * button, gpointer user_data)
{
    GtkTextBuffer *textbuffer = NULL;
```

```
    GtkTextIter start, end;

    g_assert (GTK_IS_TEXT_VIEW (user_data));

    textbuffer = gtk_text_view_get_buffer (GTK_TEXT_VIEW (user_data));

    gtk_text_buffer_get_selection_bounds (textbuffer, &start, &end);

    gtk_text_buffer_apply_tag_by_name (textbuffer, "strike", &start,
                       &end);
}

void
on_button_color_clicked (GtkButton * button, gpointer user_data)
{
    GtkTextBuffer *textbuffer = NULL;
    GtkTextIter start, end;

    g_assert (GTK_IS_TEXT_VIEW (user_data));

    textbuffer = gtk_text_view_get_buffer (GTK_TEXT_VIEW (user_data));

    gtk_text_buffer_get_selection_bounds (textbuffer, &start, &end);

    gtk_text_buffer_apply_tag_by_name (textbuffer, "color", &start,
                       &end);
}

void
create_tags (GtkTextBuffer * buffer)
{
    g_assert (GTK_IS_TEXT_BUFFER (buffer));

    gtk_text_buffer_create_tag (buffer, "bold",
                    "weight", PANGO_WEIGHT_BOLD, NULL);
    gtk_text_buffer_create_tag (buffer, "underline",
                    "underline", PANGO_UNDERLINE_SINGLE,
                    NULL);
    gtk_text_buffer_create_tag (buffer, "strike", "strikethrough",
                    TRUE, NULL);
    gtk_text_buffer_create_tag (buffer, "color", "foreground", "blue",
                    NULL);
}
```

# Capítulo 10. GTK+ avanzado

## El portapapeles GTK

## Drag and Drop

### Introducción

GTK+ proporciona una serie de funciones para facilitar el intercomunicaicón entre procesos usando *drag-and-drop*. Los protocolos que reconoce GTK+ son Xdnd y el de Motif.

Para usar DND primero hay que definir qué widgets se usaran como fuentes y/o como destino. Estos widgets tienen que tener asociada una ventana real, lo cual se puede comprobar con la macro GTK_WIDGET_NO_WINDOW(widget))

El envío y la recepción de datos se hace a través de señales. El widget fuente debe controlar la señal *drag_data_get*, que será usada para pedir los datos a la fuente. Mientras, el widget destino usará *drag_data_received* para saber cuándo se le están enviado datos. Hay otras señales, que permiten saber cuándo el DND comienza y cuándo acaba.

### Definiendo el widget destino

Para que un widget pueda recibir eventos DND basta con invocar a la función `gtk_drag_dest_set`

```
void gtk_drag_dest_set(GtkWidget* widget, GtkDestDefaults flags, const
GtkTargetEntry* targets, gint n_targets, GdkDragAction actions);
```

Con esta función, el widget podrá recibir la señal *drag_data_received*, que le indicará que se han soltado elementos sobre él.

```
void "drag_data_received"(GtkWidget* widget, GdkDragContext* dc, gint
x, gint y, GtkSelectionData* selection_data, guint info, guint t,
gpointer data);
```

En `gtk_drag_dest_set`, el argumento *`widget`* se refiere al widget que recogerá las señales de DND.

*`flags`* indica las acciones a tomar cuando un elemento se suelte sobre el widget. Los valores que puede tomar son

**Tabla 10-1. Acciones que se pueden tomar**

| | |
|---|---|
| GTK_DEST_DEFAULT_MOTION | Esto hará que cuando el objeto pase sobre el widget, GTK+ comprobará si se puede soltar sobre él. Si es posible, se llamará a gdk_drag_status. |

| GTK_DEST_DEFAULT_HIGHLIGHT | Con esto, GTK+ cambiará el estado del widget a *highlight* cuando un objeto esté sobre él y este objeto pueda soltarse sobre el widget. |
|---|---|
| GTK_DEST_DEFAULT_DROP | Éste hará que cuando un objeto se suelte (es decir, se libere el botón del ratón) se compruebe si el objeto puede ser recibido por el widget. Si es así, GTK+ llamará a gtk_drag_get_data para obtener los datos del widget fuente. |
| GTK_DEST_DEFAULT_ALL | >Éste es una combinación de los tres anteriores. |

`targets` y `n_targets` se usan para especificar qué objetivos (*targets*) puede recibir. `targets` es una array de GtkTargetEntry, y `n_targets` define el número de entradas que tiene ese array. GtkTargetEntry se define como

```
struct GtkTargetEntry {
 gchar *target;
 guint  flags;
 guint  info;
};
```

El miembro `target` define el nombre del objetivo. Se pueden usar nombres *estándares*, como text/uri-list o text/plain, y también usar nombre propios específicos para la aplicación, como GIMP_IMAGE o GIMP_LAYER de gimp.

El miembro `flags` puede tomar cualquier de estos tres valores.

**Tabla 10-2. Posibles valores para `flags`**

| 0 | Se puede recibir objetos de cualquier lugar |
|---|---|
| GTK_TARGET_SAME_APP | Sólo se puede recibir objetos desde otros widgets de la propia aplicación |
| GTK_TARGET_SAME_WIDGET | Sólo se reciben objetos del mismo widget. |

El último miembro, `actions`, define qué acciones reconoce esta fuente. Las acciones disponibles son

**Tabla 10-3. Acciones cuando se suelta**

| GDK_ACTION_DEFAULT | Acción por defecto |
|---|---|
| GDK_ACTION_COPY | Copiar los datos |
| GDK_ACTION_MOVE | Moverlos. Es decir, primero copiarlos de la fuente y después borrarlos de ella. |
| GDK_ACTION_LINK | Hacer un enlace. Esto es útil sólo si la fuente y el destino entienden lo mismo por *enlazar*. |
| GDK_ACTION_PRIVATE | Un acción especial que la fuente no conocerá. |
| GDK_ACTION_ASK | Preguntar al usuario qué quiere hacer |

## Definiendo el widget fuente

Para definir un widget y usarlo como fuente usamos la función `gtk_drag_source_set`

```
void gtk_drag_source_set(GtkWidget* widget, GdkModifierType
start_button_mask, const GtkTargetEntry* targets, gint n_targets,
GdkDragAction actions);
```

Los argumentos `widget`, `targets`, `n_targets` y `actions` son los mismo que los usados en `gtk_drag_dest_set`.

El argumento `start_button_mask` define qué botones será válidos para comenzar a arrastrar objetos. Algunos de los valores típicos para este argumento son GDK_BUTTON1_MASK, GDK_BUTTON2_MASK, etc.

### Señales

Existen seis señales relacionadas con el DND para el widget fuente, aunque sólo es necesario implementar *drag_data_get* para poder enviar datos.

#### *drag_data_get*

Ésta es la señal principal, a través de la cual se envían datos hacia el cliente.

```
void "drag_data_get"(GtkWidget* widget, GdkDragContext* dc,
GtkSelectionData* selection_data, guint info, guint t, gpointer
data);
```

En esta señal, `widget` es el wiget del que se quieren extraer datos. `dc` contiene información sobre el proceso de DND. `selection_data` se usará para guardar la información que se enviará al widget destino. `info` es el tipo de objetivo que hemos definido en el miembro `info` de GtkTargetEntry.

`dc` es una estructura GdkDragContext, que se define como

```
 struct GdkDragContext {

  GObject parent_instance;


  GdkDragProtocol protocol;

  gboolean is_source;

  GdkWindow *source_window;
  GdkWindow *dest_window;

  GList *targets;
  GdkDragAction actions;
  GdkDragAction suggested_action;
  GdkDragAction action;

  guint32 start_time;

 };
```

Y `selection_data` una estructura GtkSelectionData. Vea el capítulo de selecciones para saber más sobre ellas.

*drag_data_delete*

Cuando un elemento se suelta sobre el widget destino y la acción realizada es GDK_ACTION_MOVE (por ejemplo), esta señal se enviará para notificar a la fuente que elimine el objeto que se ha arrastrado.

```
void* "drag_data_delete"(GtkWidget* widget, GdkDragContext* dc,
gpointer data);
```

*drag_begin*

Esta señal se envía cuando se inicia el arrastre

```
void "drag_begin"(GtkWidget* widget, GdkDragContext* dc, gpointer
data);
```

*drag_motion*

Esta otra se envía cada vez que el cursor se mueve arrastrando un objeto

```
gboolean "drag_motion"(GtkWidget* widget, GdkDragContext* dc, gint x,
gint y, guint t, gpointer data);
```

*drag_drop*

Esta se enviará cuando el objeto se suelte sobre el widget destino

```
gboolean "drag_drop"(GtkWidget* widget, GdkDragContext* dc, gint x,
gint y, guint t, gpointer data);
```

*drag_end*

Y esta última se enviará cuando el proceso del DND haya acabado.

```
void "drag_end"(GtkWidget* widget, GdkDragContext* dc, gpointer
data);
```

## Imágenes, botones, menús de stock

# Ficheros de recursos

## Introducción

Los ficheros de recursos dentro de gnome sirven para modificar opciones de los widgets, por ejemplo la visualización de ellos, es decir, su color de fondo, su fuente o bien si van a tener un pixmap sobre ellos. Todo esto lo podemos indicar en ficheros adjuntos a la aplicación de tal manera que cada usuario podra personalizar la apariencia de los widgets solo modificando este fichero.

## Estructura de un fichero rc

Los ficheros de recursos contienen dos clases de instrucciones, las de definición de estilos y las de asignación de estilos. La combinación de las dos instrucciones permiten asignar un estilo a un determinado widget.

Para definir un estilo vamos a usar la orden style, para controlar la apariencia del widget existe varios identificadores, estos son

bg: para el fondo del widget
fg: para el color de frente del widget
bg_pixmap: para el pixmap del fondo
font: para la fuente del widget

Junto con estos identificadores existen otros que indican el estado del widget, que son los siguientes

NORMAL: Estado normal del widget
PRELIGHT: Cuando el puntero del ratón esta sobre el widget
ACTIVE: Cuando el se hace click con el ratón sobre el widget
INSENSITIVE: Cuando el widget no esta activo
SELECTED: Cuando el widget esta seleccionado

Con estos dos identificadores se puede definir la forma que van a tener los identificadores

```
bg/fg[ESTADO] = {Rojo, Verde, Azul }
bg_pixmap[ESTADO] = pixmap
font = nombre_fuente
```

La primera de todas establece el valor para los colores de fondo y frente para un estado determinado a los valores que se pasan en entre las llaves. Estos valores están en el rango 0.0 - 1.0 siendo el 1.0 el mas fuerte. Es importante destacar que estos valores siempre tienen que llevar el decimal, esto es, si ponemos 1 sin el .0 la aplicación considerará como 0; por tanto es importante poner todos los dígitos indicando que es un numero decimal.

La segunda instrucción establece el pixmap de fondo para un estado determinado al pixmap que se indica. Se debe tener en cuenta que la ruta para buscar estos pixmaps se puede establecer dentro del fichero mediante la orden *pixmap_path* que contendrá una lista de directorios separados por el carácter ":"

Por ultimo la ultima instrucción asigna la fuente "nombre_fuente" al widget, el nombre de la fuente es del tipo *-adobe-lucida-medium-*-normal--*-*-*-*-*-*-*-** por lo que se aconseja usar una aplicación como puede ser xfontsel para obtener el nombre.

Estas instrucciones se meten dentro de un estilo para definirlo, un ejemplo de definición de estilo podría ser

```
style "boton"
{
  fg[ACTIVE] = {1.0,0.5,0}
  bg[PRELIGHT] = {0,0,1.0}
  font =  "-sony-fixed-medium-*-*-*-12-120-*-*-c-*-*-*"
}
```

Con estas lineas definimos el estilo "boton" que podremos asignar a uno o varios widgets. Antes de continuar hasta la asignación de estilos, cabe destacar que los estilos tienen una especie de herencia, esto es, se puede crear un nuevo estilo partiendo de uno base y definiendo nuevos parámetros o bien redefiniendo estados que ya estaban declarados en el estilo base. Un ejemplo podría ser

```
style "boton_especial" = "boton"
{
  fg[NORMAL] = {1.0,1.0,0}
  font = "-adobe-times-medium-r-*-*-12-*-*-*-*-*-iso8859-15"
}
```

Con estas instrucciones se crea un nuevo estilo llamado "boton_especial" pero que toma como base el estilo "boton", es decir lo que ya estaba definido en el estilo "boton" permanecerá en "boton_especial". Por tanto el estilo "boton_especial" tendrá las características de "boton" mas la nueva, que es fg[NORMAL]. Además de todo esto el estilo "boton_especial" redefine el tipo de fuente por lo que no se usará el de "boton".

Una vez creado un estilo, el siguiente paso es asignar dicho estilo a un widget o conjunto de ellos, para esto existe dos sentencias.

Una de ellas es widget que asigna un estilo a un widget o a un grupo que se identifica con un nombre determinado. Este nombre se especifica dentro de la aplicación, como se vera en el siguiente capitulo. Esto permite la agrupación de widgets para asignar estilos y no limitar la asignación a clases de widgets, como sería aplicar un estilo a todos los botones de la aplicación. El prototipo de estas sentencias es.

```
widget "nombre.*Clase*" style "estilo"
```

Donde nombre es el nombre que asignamos dentro de la aplicación con `gtk_widget_set_name()` y Clase es el tipo de widget a los que vamos a aplicar el estilo; por tanto este puede ser GtkButton o bien GtkText, por ejemplo. Por ultimo estilo es el estilo que vamos a aplicar al widget en cuestión. Un ejemplo de uso sería.

```
widget "ventana_principal.*GtkButton*" style "boton"
```

En el ejemplo se asigna a todos los botones que están agrupados bajo el nombre ventana_principal el estilo que esta definido anteriormente llamado "boton"

La ultima instrucción que vamos a tratar es widget_class que asigna un estilo a un widget sin necesidad que este agrupado bajo un nombre especial. La sintaxis de esta orden es.

## Como usar los ficheros de recursos

En gtk2 se dispone de varias funciones para el manejo de estos ficheros. La función principal es

```
void  gtk_rc_parse  (char *filename);
```

Cuando llamamos a esa función , con el nombre del fichero rc como parámetro, la aplicación analiza el archivo dando a los widgets que figuran en el archivo la apariencia definida en el.

Ahora bien, para crear grupos de widgets para luego usarlos dentro del archivo de recursos, existe la función

```
void  gtk_widget_set_name  (GtkWidget * widget , gchar * name);
```

Esta función agrupa el widget que se pasa por parámetro con el nombre que se pasa en el segundo parámetro. Este nombre va a servir para identificar el grupo dentro del fichero de rc de tal forma que la instrucción

```
gtk_widget_set_name(boton,"boton_aceptar");
```

agrupa el widget boton ( que se supone que es un botón ya creado anteriormente ) con el nombre "boton_aceptar" de tal manera que el fichero rc se podrá referir a el como *boton_aceptar.GtkButton*.

## Selecciones

### Resumen

Las selecciones son un tipo de comunicación entre procesos, que esta soportado por las X y GTK. Una selección identifica una porción de datos, por ejemplo, una porción de texto, que ha sido seleccionado dentro de una tarea que esta ejecutando un usuario, por ejemplo una forma de seleccionar un contenido podria ser arrastrando el ratón, sobre él. Solo una aplicación ( la propietaria ) puede tener una selección al mismo momento, por tanto cuando otra aplicación quiere poseer dicha selección, la que poseia dicha selección en un primer momento debe indicar al usuario que esta, ha sido abandonada. Otras aplicaciones pueden pedir el contenido de una selección de diferentes formas, que se conocen con el nombre de *objetivos.* Las aplicaciones pueden manejar cualquier numero de selecciones, pero la mayoria solo tienen en cuenta una, la llamada *selección primaria*.

En la mayoria de los casos, no es necesario que la propia aplicación GTK maneje las selecciones. Los Widgets estandar, como por ejemplo la caja de texto, tienen la capacidad de solicitar la selección cuando sea necesario ( por ejemplo cuando el usuario arrastra el raton sobre el texto ), y obtener los contenidos de la selección que pertenecen a otro widget o incluso otra aplicacion ( por ejemplo cuando el usuario pulsa el segundo botón del ratón ). Sin embargo, puede darse el caso en el cual se quiera tener la posibilidad para que un widget pueda proveer un objetivo que no este diseñado por defecto.

Un concepto fundamental a la hora de entender el manejo de selecciones es el significado de un *átomo*. Un átomo es un entero que identifica inequivocamente una cadena ( en un determinado display). Algunos átomos estan predefinidos por el servidor X y en algunos casos en constantes , que se corresponden con los átomos, dentro de gtk.h. Por ejemplo la constante GDK_PRIMARY_SELECTION corresponde con la cadena PRIMARY. En otros casos se debe usar las funciones `gdk_atom_intern()`, para obtener el átomo correspondiente a una cadena, y `gdk_atom_name()`, para obtener el nombre de un átomo. Ambas cosas, selecciones y objetivos estan identificados por átomos.

## Obteniendo la selección

Obtener la selección es un proceso asincrono. Para comenzar el proceso, se llama a:

```
gboolean gtk_selection_convert (GtkWidget *widget, GdkAtom selection,
GdkAtom target, guint32 time);
```

Esta llamada convierte la selección en el tipo especificado por el parametro `target`. Si es posible , el parametro `time` deberia ser el tiempo del el evento que ha lanzado la selección. Esto ayuda para asegurarse que los eventos ocurren en el orden que el usuario los ha lanzado. Sin embargo si esto no es posible ( por ejemplo , si la conversión fue disparada por una señal `clicked` ), entonces se puede usar la constante GDK_CURRENT_TIME.

Cuando la aplicacion propietaria de la selección responde a la petición, la señal *"selection_received"* se manda a la aplicación que ha realizado la petición. El manejador para esta señal recibe un puntero a una estructura *GtkSelectionData*, que esta definida como:

```
struct _GtkSelectionData
{
GdkAtom selection;
GdkAtom target;
GdkAtom type;
gint    format;
guchar *data;
gint    length;
};
```

*selection* y *target* son los valores que se dieron en la llamada a `gtk_selection_convert()`. *type* es un átomo que identifica el tipo de datos que va a ser devuelto por el propietario de la selección. Algunos valores posibles son *"STRING"* para una cadena de caracteres, *"ATOM"*, una serie de átomos, *"INTEGER"*, un entero, etc. La mayoria de objetivos solo pueden devolver un tipo.*format* nos indica la longitud de las unidades ( por ejemplo los caracteres ) en bits. Normalmente, no hay que preocuparse de esto cuando se recibe los datos. *data* es un puntero a los datos devueltos, en bytes. Si la longitud es negativa la seleccion no puede ser recuperada ya que se ha producido un error. Esto podria pasar si ninguna aplicación posee la selección, o si se solicita un objetivo que la aplicación no tiene definido. La longitud del buffer se garantiza que sera un byte mas grande que lo que se indica en *length*; este byte extra sera siempre cero, por lo que si es una cadena no nos tenemos que preocupar de terminarla en este caracter nulo.

En el siguiente ejemplo, Vamos a obtener el objetivo especial *"TARGETS"*, el cual es una lista de todos los objetivos en los que una selección puede ser convertida

## Suministrando la selección

Suministrar la selección es un poco mas complicado, Se debe registrar manejadores que seran llamados cuando se solicite la selección. Para cada pareja selección/objetivo que se quiera controlar, se debe hacer una llamada a:

```
void gtk_selection_add_target (GtkWidget *widget, GdkAtom selection,
GdkAtom target, guint info);
```

`widget, selection` y `target` identifica las peticiones que el manejador puede gestionar. Cuando una petición de una selección es recibida, se llamara a la señal *"selection_get"*. `info` puede ser usado como una enumeración para identificar el objetivo especifico que acompaña a la función de callback.

La función de callback tiene la forma:

```
void "selection_get" (GtkWidget *widget, GtkSelectionData *
selection_data , guint info, guint time);
```

El elemento *GtkSelectionData* es el mismo que antes, solo que ahora, Debemos rellenar los campos *type*, *format*, *data* y *legth*. ( El campo *format* es importante aqui - el servidor X lo usa para saber si los datos necesitan ser intercambiados a nivel de byte o no. Normalmente sera 8, para un caracter, o 32 para un entero) Esto se hace llamando a la función:

```
void gtk_selection_data_set (GtkSelectionData * selection_data ,
GdkAtom type, gint format, guchar *data, gint lenght);
```

Esta función se encarga de hacer una copia de el parametro `data` por lo que no hay que preocuparse de mantenerlo. ( No se deberia rellenar a mano los campos de `selection_data`

Cuando sea necesario, se establecera la propiedad de la selección llamando a la función

```
gboolean gtk_selection_ownet_set (GtkWidget *widget, GdkAtom selection,
guint32 time);
```

Si otra aplicación reclama la propiedad de la selección, se recibira una señal del tipo*selection_clear_event*

Como el ejemplo anterior de obtener una selección, la siguiente aplicación añade la funcionalidad de selección a un toggle button. Cuando el toggle button esta no activo, el programa reclamara la selección primaria. El unico objetivo disponible ( junto con otros objetivos, como *TARGET*, suministrados por la propia GTK ), es *STRING*. Cuando se hace una petición a este objetivo, se retorna una cadena que contiene la hora actual

## Otros widgets

GtkAdjustment GtkArrow GtkCalendar GtkDrawingArea GtkEventBox GtkHandleBox GtkIMContext GtkIMContextSimple GtkIMMulticontext GtkSizeGroup GtkTooltips GtkViewport

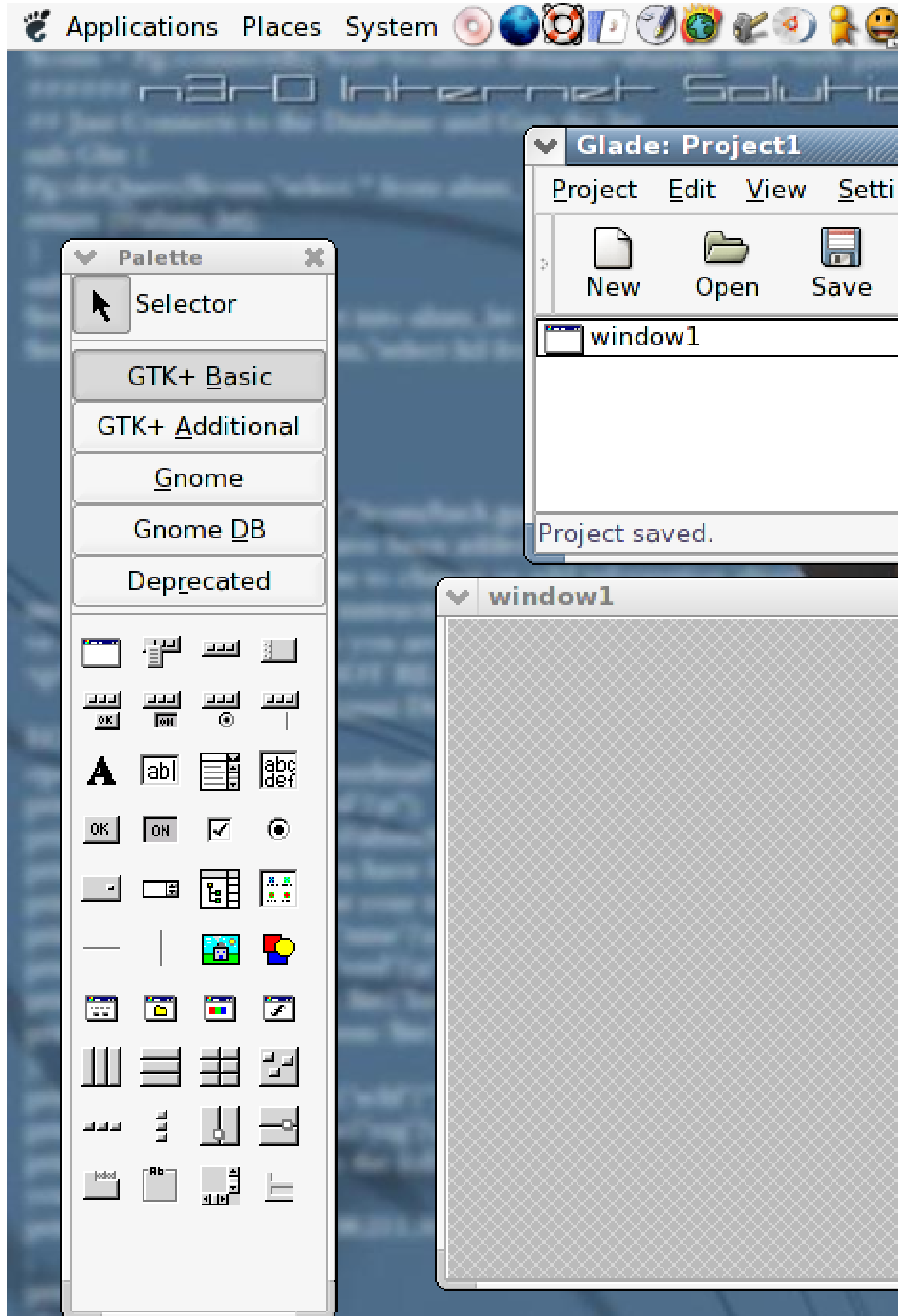
# Capítulo 11. Accesibilidad en GNOME

Accesibilidad en GNOME

# Capítulo 12. Interfaces de usuario con Glade y libglade

## Introducción

Una pequeña introducción del equipo y como llegó a mis manos.


Applications Places System
Glade: Project1
Project Edit View Setti
New Open Save
window1
Project saved.
Palette
Selector
GTK+ Basic
GTK+ Additional
Gnome
Gnome DB
Deprecated
window1

## Diseño de interfaces de usuario con Glade

### Ventana principal

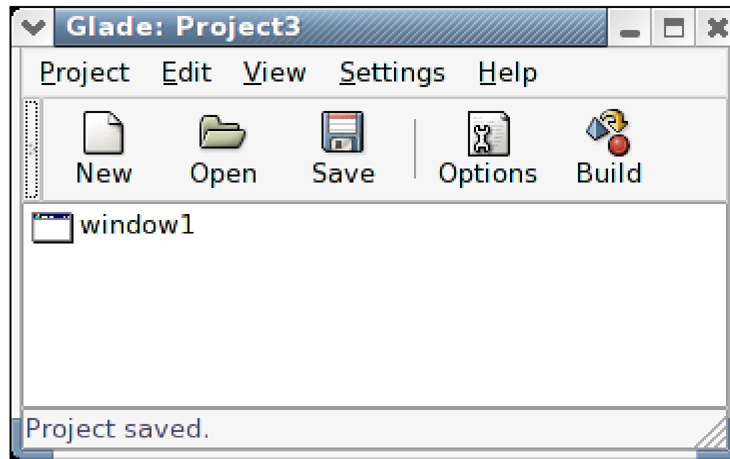



**Figura 12-2. Ventana principal**

## Paleta de herramientas

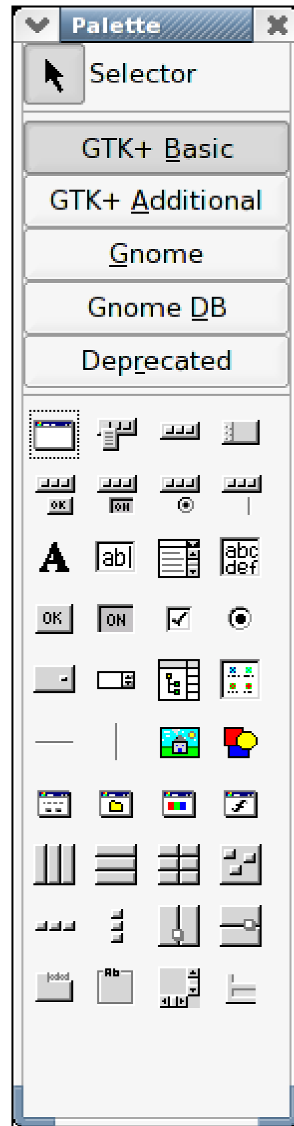
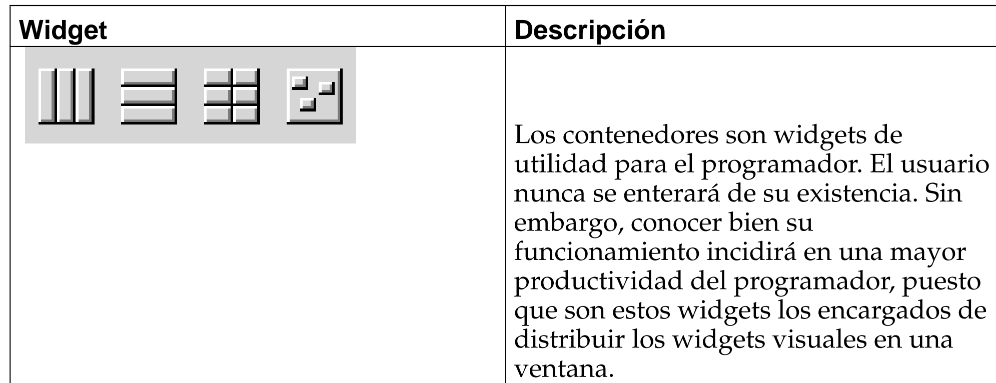



**Figura 12-3. Paleta de herramientas**

**Tabla 12-1. Principales widgets**

| Widget | Descripción |
| --- | --- |

| Widget | Descripción |
| --- | --- |
|  | Los contenedores son widgets de utilidad para el programador. El usuario nunca se enterará de su existencia. Sin embargo, conocer bien su funcionamiento incidirá en una mayor productividad del programador, puesto que son estos widgets los encargados de distribuir los widgets visuales en una ventana. |

## Ventana de propiedades

propiedades

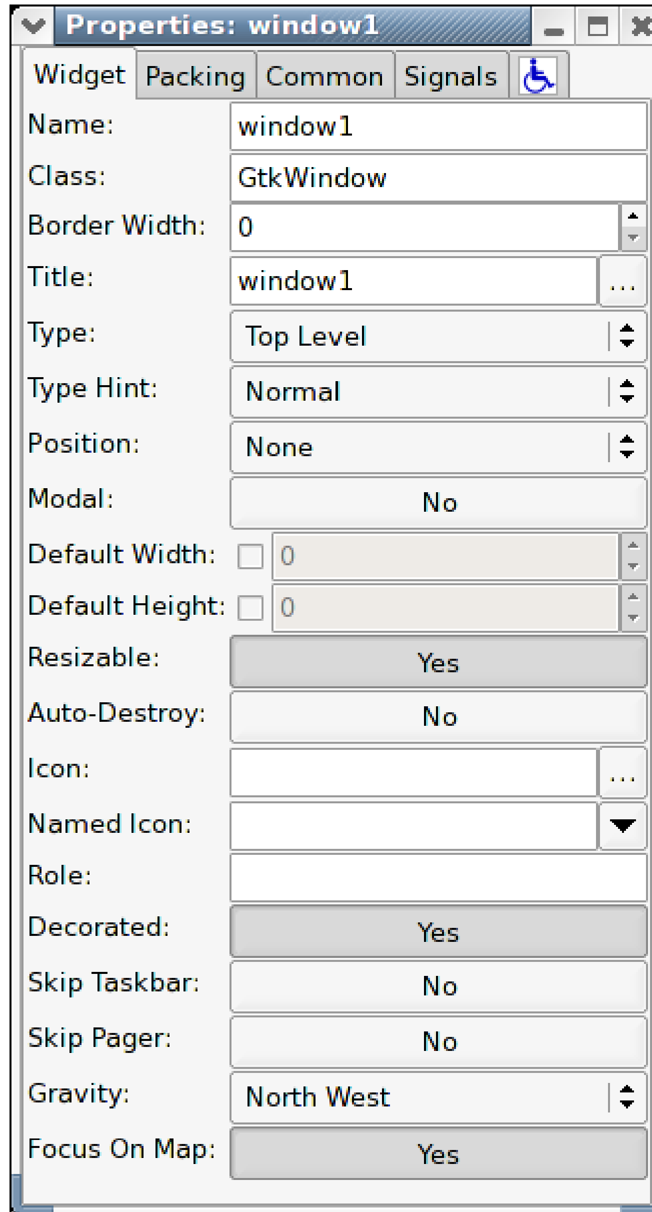



**Figura 12-4. Ventana de propiedades**

## Ventana de trabajo

trabajo

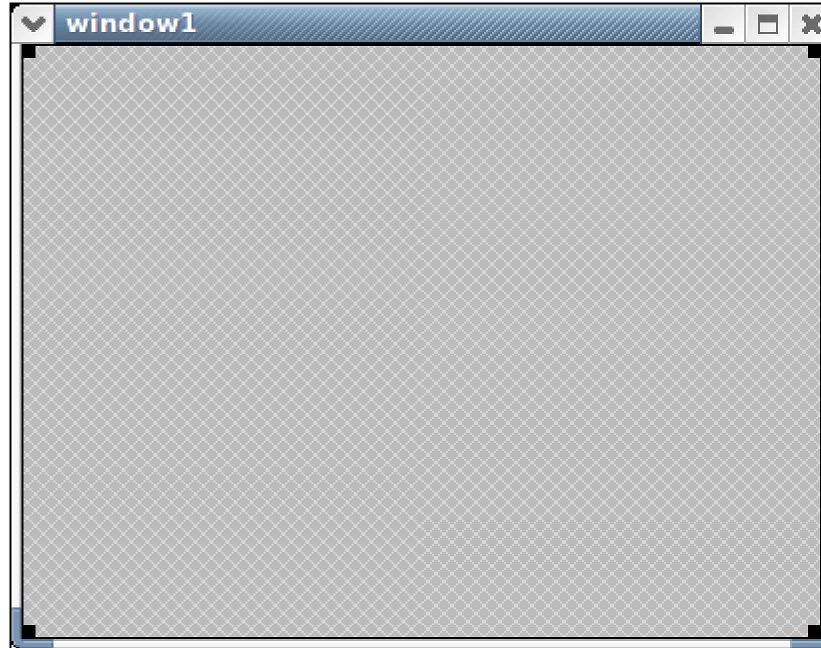



**Figura 12-5. Ventana de trabajo**

## La biblioteca libglade

Glade permite generar código a partir de las interfaces diseñadas, sin embargo su práctica no es recomendada y es por eso que se omite en este libro. La alternativa es emplear las funcionalidades provistas por la biblioteca libglade, la cual se encarga de cargar y procesar los archivos XML de las interfaces durante la ejecución del programa.

A través de libglade se logra la separación entre la interfaz de usuario y la lógica del programa.

### Ejemplo básico de uso de libglade

**Ejemplo 12-1. Primer ejemplo con GTK**

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

void some_signal_handler_func(GtkWidget *widget, gpointer user_data) {
  /* Algún código útil */
}

int main(int argc, char *argv[]) {
 GladeXML *xml;

 gtk_init(&argc, &argv);

 /* Cargar la interfaz de usuario */
 xml = glade_xml_new("filename.glade", NULL, NULL);
```

```
  /* Conectar las señales de la interfaz */
  glade_xml_signal_autoconnect(xml);

  /* Iniciar el ciclo principal */
  gtk_main();

  return 0;
}
```

## Compilación

### Desde el shell

```
$ gcc -o main main.c `pkg-config --cflags --libs libglade-2.0`
```

### Usando autoconf

```
PKG_CHECK_MODULES(MYPROG, libglade-2.0 libgnomeui-2.0 >= 1.110.0)
AC_SUBST(MYPROG_CFLAGS)
AC_SUBST(MYPROG_LIBS)
```

## Un ejemplo más elaborado

El siguiente ejemplo constituye una elaboración del anterior. El ejercicio consiste en disponer de 3 widgets GtkEntry, GtkLabel y GtkButton llamados entry, label y button respectivamente, de tal forma que el usuario modifique el texto de entry y al presionar el botón se modifique el widget label.

El ejercicio permitirá mezclar varios conceptos ya aprendidos, a saber:

- Paso de otros widget a las señales
- Validación de `g_assert`
- Diseño de intefaz con Glade
- Emplear distintos caminos para llegar a una misma interfaz con Glade

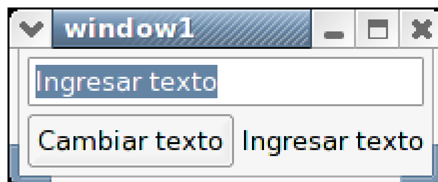



**Figura 12-6. Ventana resultante para el ejericicio**

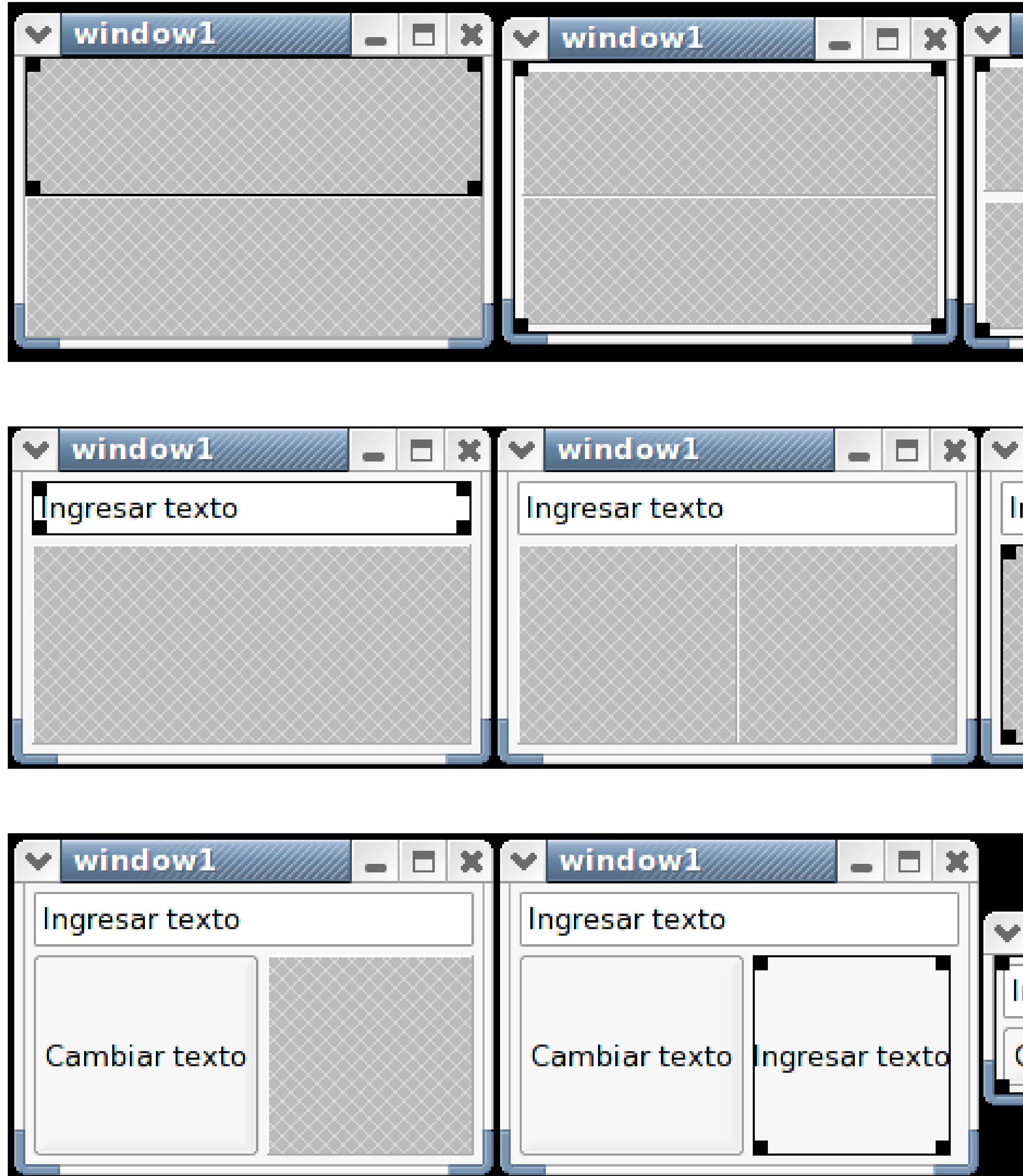



**Figura 12-7. Construcción paso a paso con cajas**

1. Crear una nueva ventana.
2. Insertar una caja vertical con 2 filas.

a. Definir ancho de borde 4.
b. Definir espaciado 4.

3. Insertar una entrada de texto en la primera fila del vbox.
   a. Renombrar a entry (en vez de entry1).
   b. Definir Texto como "Ingresar texto".
   c. Activar como predeterminado (Activates default). Esto dejará el foco en este widget al momento de cargar la ventana.

4. Insertar una caja horizontal con 2 columnas en la segunda fila del vbox y definir espaciado 4.
5. Insertar un boton en la primera columna.
   a. Renombrarlo a button (en vez de button1).
   b. Definir la etiqueta como "Cambiar texto".
   c. Definir como predeterminado (en la paleta comunes). Esto habilitará al botón para ser ejecutado cuando se presione la tecla Return en la ventana (acción predeterminada).
   d. Definir una función para la señal clicked En la paleta "Señales" elegir "clicked" para Señal. El manejador es el nombre de la función dentro del programa que se ejecutará al momento de efectuar click sobre el botón. En esta ocasión dejaremos el nombre propuesto por Glade: "on_button_clicked". Para que los cambios tengan efecto es necesario presionar el botón "Añadir". De la misma forma, si se modifica alguno de esos valores, es necesario presionar el botón "Actualizar", de los contrario los cambios no se verán reflejados.

6. Insertar una etiqueta en la segunda columna de hbox1.
   a. Renombrar a label (en vez de label1).
   b. Definir Texto como "Ingresar texto".

7. Redimensionar la ventana para que cuadre con los controles. Al momento de ejecutar la ventana, se ajustará automáticamente al tamaño de los controles, independientemente de la forma en que se muestra en Glade. Sin embargo, es buena práctica realizar el redimensionamiento para trabajar de la misma forma en que se verá en la aplicación. Si se desea modificar a un tamaño distinto y mayor al tamaño mínimo dado por los widgets, entonces es necesario sobrescribir los valores de los campos Ancho y Altura para el widget ventana (ver paleta "Comunes").

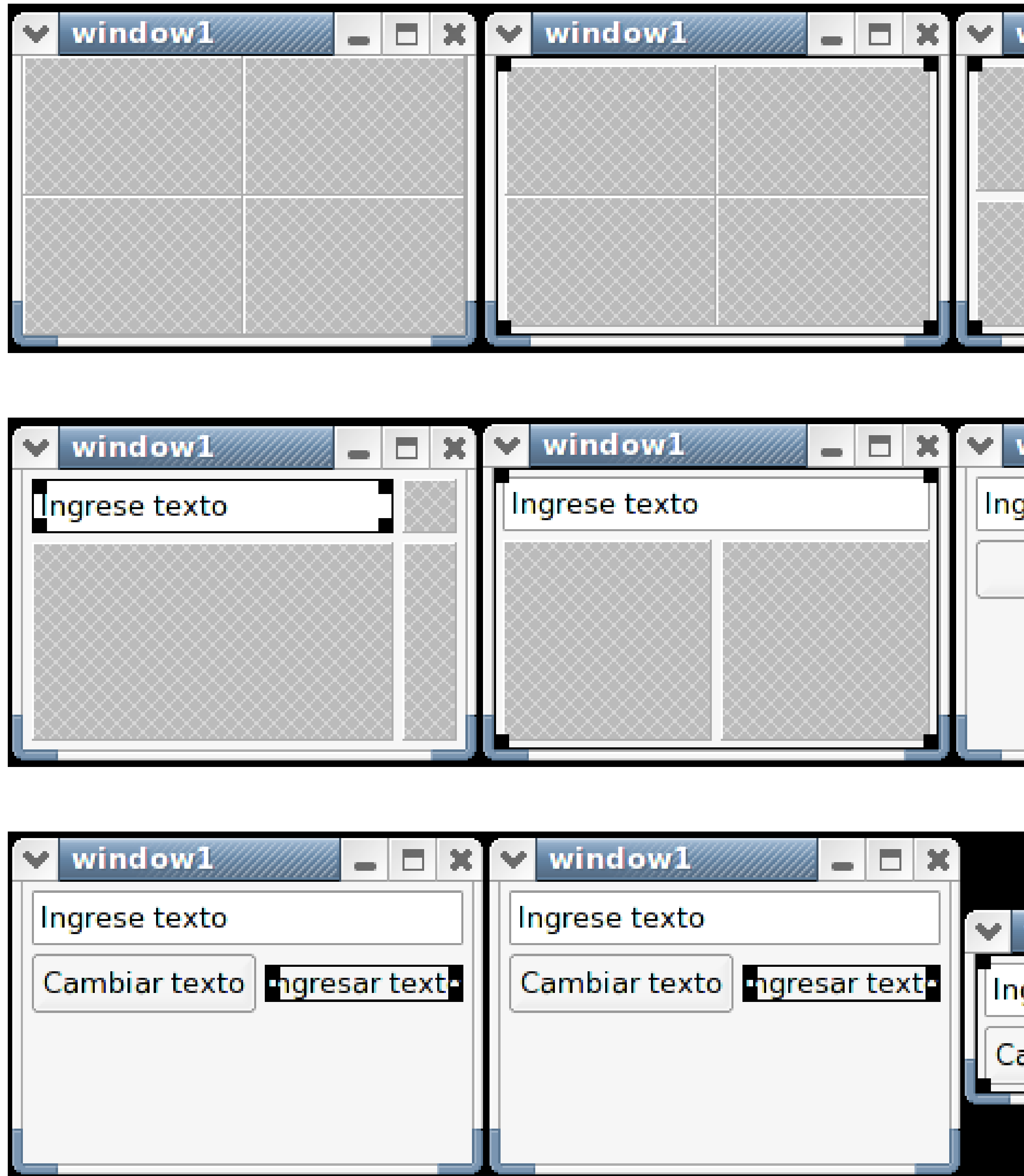



**Figura 12-8. Construcción paso a paso con tabla**

1. Crear una nueva ventana.
2. Insertar una tabla con 2 filas y 2 columnas.

a. Definir ancho de borde 4.
b. Definir espaciado de filas 4.
c. Definir espaciado de columnas 4.

3. Insertar una entrada de texto en la primera fila y primera columna de la tabla.
    a. Renombrar a entry (en vez de entry1).
    b. Definir Texto como "Ingresar texto".
    c. Activar como predeterminado (Activates default). Esto dejará el foco en este widget al momento de cargar la ventana.
    d. Definir Interlineado de columnas 2 (paleta Empaquetado). Las opciones de interlinado permiten que el widget ocupe mas de una "celda" dentro de la tabla.

4. Insertar un boton en la primera columna de la seguna de fila de la tabla.
    a. Renombrarlo a button (en vez de button1).
    b. Definir la etiqueta como "Cambiar texto".
    c. Definir como predeterminado (en la paleta comunes). Esto habilitará al botón para ser ejecutado cuando se presione la tecla Return en la ventana (acción predeterminada).
    d. Definir una función para la señal clicked En la paleta "Señales" elegir "clicked" para Señal. El manejador es el nombre de la función dentro del programa que se ejecutará al momento de efectuar click sobre el botón. En esta ocasión dejaremos el nombre propuesto por Glade: "on_button_clicked". Para que los cambios tengan efecto es necesario presionar el botón "Añadir". De la misma forma, si se modifica alguno de esos valores, es necesario presionar el botón "Actualizar", de los contrario los cambios no se verán reflejados.
    e. Deshabilitar la opción "Rellenar en X" de la paleta Empaquetado. Las opciones Rellenar X/Y y Expandir en X/Y son similares a las encontradas en los hbox y vbox, sin emgargo, en una tabla es necesario si se desea rellenar/expandir respecto a la columna y/o fila.

5. Insertar una etiqueta en la segunda fila y segunda columna de la tabla.
    a. Renombrar a label (en vez de label1).
    b. Definir Texto como "Ingresar texto".
    c. Habilitar la opción "Expandir en X" en la paleta Empaquetado.

6. Redimensionar la ventana para que cuadre con los controles. Al momento de ejecutar la ventana, se ajustará automáticamente al tamaño de los controles, independientemente de la forma en que se muestra en Glade. Sin embargo, es buena práctica realizar el redimensionamiento para trabajar de la misma forma en que se verá en la aplicación. Si se desea modificar a un tamaño distinto y mayor al tamaño mínimo dado por los widgets, entonces es necesario sobrescribir los valores de los campos Ancho y Altura para el widget ventana (ver paleta "Comunes").

**Ejemplo 12-2. Un ejemplo más elaborado con GTK**

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

void on_button_clicked (GtkButton *button, gpointer user_data) {
 GtkWidget *label, *entry;
 gchar *text;

 label = g_object_get_data (G_OBJECT (button), "label");
 entry = g_object_get_data (G_OBJECT (button), "entry");

 text = gtk_editable_get_chars (GTK_EDITABLE (entry), 0, -1);
 gtk_label_set_text (GTK_LABEL (label), text);
}

int main(int argc, char *argv[]) {
 GladeXML *xml;
 GtkWidget *label, *button, *entry;

 gtk_init(&argc, &argv);

 /* Cargar la interfaz de usuario */
 xml = glade_xml_new("hola_mundo.glade", "window1", NULL);

 button = glade_xml_get_widget (xml, "button");
 label = glade_xml_get_widget (xml, "label");
 entry = glade_xml_get_widget (xml, "entry");

 g_assert (button != NULL);
 g_assert (label != NULL);
 g_assert (entry != NULL);

 g_object_set_data (G_OBJECT (button), "label", label);
 g_object_set_data (G_OBJECT (button), "entry", entry);

 /* Conectar las señales de la interfaz */
 glade_xml_signal_autoconnect(xml);

 g_object_unref (G_OBJECT (xml));

 /* Iniciar el ciclo principal */
 gtk_main();

 return 0;
}
```

### libglade y las bibliotecas de GNOME

Para integrar los widgets que se construyen sobre GTK+, sólo es necesario realizar la inicialización a través de `gnome_program_init` en vez de `gtk_init`.

## Internacionalización de las interfaces

## Ejemplos más avanzados

### Integración de GtkTreeView

### Integración de Bonobo

### Trabajo sólo con widgets específicos

# Capítulo 13. El canvas de GNOME

El canvas de GNOME es un *widget* que ofrece un método fácil y potente para presentar los datos de una aplicación. Este objeto es una simple área en blanco, donde se pueden insertar objectos GnomeCanvasItem, los cuales representan los elementos mostrados en el GnomeCanvas. De este modo es posible trabajar con gráfico en términos de objetos.

## Bases

### Modos del `GnomeCanvas`

El `GnomeCanvas` puede trabajar en dos modos: el modo *RGB* y el modo *GDK*. En el modo *RGB*, los objetos en el `GnomeCanvas` dibujan su contenido en un búfer RGB, y una vez que todos están actualizados, el búfer se muestra en el *widget*. En el modo *GDK*, los objetos dibujan directamente sobre un `GdkPixmap` usando las primitivas de dibujo de GDK.

El modo RGB también se llama *antialiased*, ya que todos los objetos estándares que vienen en GnomeCanvas usan la librería libart_lgpl, que tiene funciones para dibujar figuras con anti-aliasing.

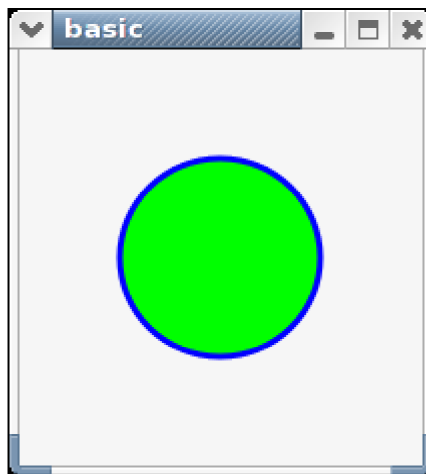



**Figura 13-1. Modo RGB**

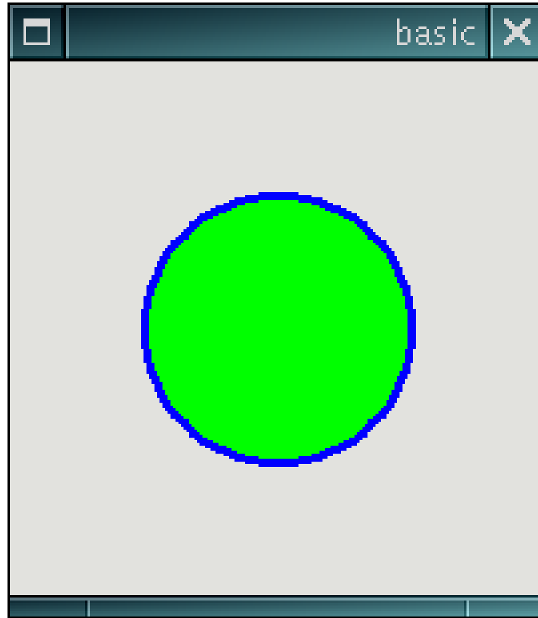



**Figura 13-2. Modo GDK**

## Grupos

Los grupos se crean con el objecto `GnomeCanvasGroup`. Todos los objetos que están dentro de un grupo funcionan como una unidad. Por ejemplo, si un grupo es destruido, todos los objetos contenidos en él también serán destruidos.

Siempre hay un grupo principal, en el cual se sitúan todos los objetos del *canvas*. Este grupo se puede obtener usando la función `gnome_canvas_root`.

## Crear un canvas

Para crear el widget se usan las funciones `gnome_canvas_new` (modo GDK) o `gnome_canvas_new_aa` (modo RGB).

```
GtkWidget*  gnome_canvas_new (void);
```

```
GtkWidget*  gnome_canvas_new_aa (void);
```

Cuando se crea el *widget*, se crea un grupo, conocido como *Root*, en el cual se insertarán todos los objetos `GnomeCanvasItem`. Para crear nuevos elementos es necesario conocer ese grupo, que se puede obtener con `gnome_canvas_root`.

Todos los objetos que se crean en el canvas deben pertenecer al grupo devuelto por `gnome_canvas_root`, o a alguno de sus grupos hijos. Los objetos se introducen en los grupos con `gnome_canvas_item_new`.

```
GnomeCanvasGroup * gnome_canvas_root (GnomeCanvas *canvas);
```

```
GnomeCanvasItem * gnome_canvas_item_new (GnomeCanvasGroup *parent,
GtkType *type, const gchar *first_arg_name, ...);
```

En este sencillo ejemplo se ve cómo crear un círculo en el canvas.

**Ejemplo 13-1. Canvas Básico**

```
#include <gnome.h>

int
main(int argc, char** argv)
{
        GtkWidget *window;
        GtkWidget *canvas;
        GnomeCanvasGroup *root;

        /* inicializamos las librer&iacute;as */
        gnome_init("basic-canvas", "0.1", argc, argv);

        /* crear una ventana que contenga al canvas */
        window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
        g_signal_connect(G_OBJECT(window), "destroy", gtk_main_quit, 0);

        /* canvas en modo RGB */
        canvas = gnome_canvas_new_aa();
        gtk_container_add(GTK_CONTAINER(window), canvas);

        root = gnome_canvas_root(GNOME_CANVAS(canvas));

        /* poner un c&iacute;rculo, con fondo verde y borde azul */
        gnome_canvas_item_new(root,
                        gnome_canvas_ellipse_get_type(),
                        "x1", 0.0,
                        "y1", 0.0,
                        "x2", 100.0,
                        "y2", 100.0,
                        "fill_color_rgba", 0x00ff00ff,
                        "outline_color_rgba", 0x0000ffff,
                        "width_units", 3.0,
                        NULL);

        /* mostrar los widgets creados y entrar en el bucle gtk_main */
        gtk_widget_show_all(window);
        gtk_main();
}
```

Del ejemplo, lo más significativo es la llamada a `gnome_canvas_item_new`, a la cual se pasan los valores que tendrá el círculos dibujado en el canvas.

## Zoom

Una de las principales ventajas de `GnomeCanvas` es la posibilidad de hacer zoom.

Para alterar el zoom del canvas se usa la función `gnome_canvas_set_pixels_per_unit`. Cuando cambiamos el zoom, los objetos se actualizan para adoptar la nueva escala.

```
void gnome_canvas_set_pixels_per_unit(GnomeCanvas* canvas, double n);
```

**Ejemplo 13-2. Cambiar el zoom**

```
/* Poner un zoom del 200 % */
gnome_canvas_set_pixels_per_unit(canvas, 2.0);

/* Poner un zoom del 50 % */
gnome_canvas_set_pixels_per_unit(canvas, 0.5);
```

## Región del canvas

El canvas tiene una región conceptualmente infinita, sólo limitada por el rango de *`double`*. Sin embargo, la mayoría de las veces sólo interesa trabajar sobre un área concreta, e incluso disponer de barras de desplazamiento para recorrerla.

Para poner una región en el canvas se usa `gnome_canvas_set_scroll_region`. Esta función define un rectángulo que nos servirá, por ejemplo, para definir los valores de las barras de desplazamiento del canvas.

```
void gnome_canvas_set_scroll_region(GnomeCanvas* canvas, double x1,
double y1, double x2, double y2);
```

La forma más sencilla de usar barras de desplazamiento para el canvas es añadirlo a un `GtkScrolledWindow`.

**Ejemplo 13-3. GnomeCanvas y GtkScrolledWindow**

```
GtkWindow *scrolled;
GtkWindow *canvas;

/* GtkScrolledWindow para el canvas */
scrolled = gtk_scrolled_window_new(NULL, NULL);

/* canvas en modo RGB */
canvas = gnome_canvas_new_aa();
gtk_container_add(GTK_CONTAINER(scrolled), canvas);
gnome_canvas_set_scroll_region(GNOME_CANVAS(canvas), x1, x2, y1, y2);
```

Para desplazar las barras a una posición determinada se puede usar `gnome_canvas_scroll_to`

```
void gnome_canvas_scroll_to(GnomeCanvas* canvas, int cx, int cy);
```

# Objetos por defecto

`GnomeCanvas` trae consigo una serie de objetos por defecto para crear elipses, rectángulos, polígonos, imágenes e incluso incrustar otros *widgets* en el canvas.

Todos los objetos tienen una serie de parámetros que modifican su representación en el canvas. Estos parámetros se pueden poner en la llamada a `gnome_canvas_item_new`, y pueden ser cambiados en cualquier momento con `gnome_canvas_item_set`. La asignación de valores a los parámetros es igual que en `GObject` con `g_object_set`.

```
GnomeCanvasItem* gnome_canvas_item_new(GnomeCanvasGroup* parent, GType
type, const gchar* first_arg_name, ...);
```

```
void gnome_canvas_item_set(GnomeCanvasItem* item, const gchar*
first_arg_name, ...);
```

## Elipses y cuadrados

A través de los objetos `GnomeCanvasRect` y `GnomeCanvasEllipse` es posible dibujar rectángulos y elipses en el canvas. Para ambos objetos, la lista de parámetros es la misma. Los más importantes son

**Tabla 13-1. Parámetros de GnomeCanvasRect y GnomeCanvasEllipse**

| Nombre | Tipo | Descripción |
|---|---|---|
| x1 | gdouble | Lado izquierdo de la figura |
| y1 | gdouble | Lado superior de la figura |
| x2 | gdouble | Lado izquierdo de la figura |
| y2 | gdouble | Lado derecho de la figura |
| outline_color_rgba | guint32 | Color RGBA del borde |
| outline_color | gchar* | Nombre del color del borde |
| fill_color_rgba | guint32 | Color RGBA de relleno |
| fill_color | gchar* | Nombre del color de relleno |
| width_units | gdouble | Anchura del borde de la figura |

Para crear una elipse se usa el valor devuelto por `gnome_canvas_ellipse_get_type` como segundo argumento de `gnome_canvas_item_new`. Para un rectángulo se usa `gnome_canvas_rect_get_type`.

```
GType  gnome_canvas_ellipse_get_type (void);
```

```
GType  gnome_canvas_rect_get_type (void);
```

## Polígonos

Con `GnomeCanvasPolygon` es posible crear polígonos. Los puntos que definen la imagen del polígono se introducen con `GnomeCanvasPoints`.

`GnomeCanvasPoints` es una estructura que debe ser creada con `gnome_canvas_points_new` y, una vez pasada a `GnomeCanvasPolygon`, se libera con `gnome_canvas_points_unref`. Esta estructura posee un miembro *`double* coords`*, que se usa para guardar las coordenadas de los puntos.

```
typedef struct {
      double *coords;
      int num_points;
      int ref_count;
} GnomeCanvasPoints;
```

```
GnomeCanvasPoints* gnome_canvas_points_new(int num_points);
```

```
void gnome_canvas_points_unref(GnomeCanvasPoints* points);
```

Cuando se introducen puntos en *`coords`*, se alternan las coordenadas X e Y en cada punto, de modo que las *`x`* ocuparán los índices pares del array *`coords`*, mientras que las *`y`* ocuparán los índices impares.

**Ejemplo 13-4. Uso de GnomeCanvasPoints**

```
/*
 * Ejemplo: crear un polígono de 5 puntos e introducirlo en el
 * objeto polygon cuando se está creando.
 *
 * Las coordenadas son (x1, y1), (x2, y2), (x3, y3), (x4, y4) y (x5, y5)
 */

GnomeCanvasPoints* points = gnome_canvas_points_new(5); /* 5 puntos */

points->coords[0] = x1;
points->coords[1] = y1;
points->coords[2] = x2;
points->coords[3] = y2;
points->coords[4] = x3;
points->coords[5] = y3;
points->coords[6] = x4;
points->coords[7] = y4;
points->coords[8] = x5;
points->coords[9] = y5;

gnome_canvas_item_new(root_group,
        gnome_canvas_polygon_get_type (),
        "points", points,
        "fill_color", "black",
        "outline_color", "white",
        "width_units", 3.0,
        NULL);

/* Una vez pasados al polígno ya no son necesarios */
```

```
gnome_canvas_points_unref(points);
```

Los parámetros que definen `GnomeCanvasPolygon` son bastante parecidos a los objetos de las secciones anteriores. Los principales son

**Tabla 13-2. Parámetros de GnomeCanvasPolygon**

| Nombre | Tipo | Descripción |
|---|---|---|
| points | GnomeCanvasPoints* | Lista de puntos que definen el polígono |
| outline_color_rgba | guint32 | Color RGBA del borde |
| outline_color | gchar* | Nombre del color del borde |
| fill_color_rgba | guint32 | Color RGBA de relleno |
| fill_color | gchar* | Nombre del color de relleno |
| width_units | gdouble | Anchura del borde de la figura |

## Texto

`GnomeCanvasText` permite insertar texto en el canvas. El dibujo del texto se hace a través de *Pango*, por lo que no se explicarán aquí los detalles sobre las fuentes.

Todo el control sobre el texto se hace a través de sus parámetros.

**Tabla 13-3. Parámetros GnomeCanvasText**

| Nombre | Tipo | Descripción |
|---|---|---|
| text | gchar* | La cadena de texto |
| markup | gchar* | Una cadena en el formato *Pango markup* |
| x | double | Coordenada X donde dibujar el texto |
| y | double | Coordenada Y donde dibujar el texto |
| font | gchar* | Una cadena para describir la fuente |
| font_desc | PangoFontDescription* | Puntero a PangoFontDescriptor |
| attributes | PangoAttrList* | Puntero a la lista de atributos de Pango |
| style | PangoStyle | Estilo Pango |
| variant | PangoVariant | Variante Pango |
| weight | int | Grosor de la fuente |
| stretch | PangoStretch | Estiramiento de la fuente |
| size | int | Tamaño (en pixels) de la fuente |

| Nombre | Tipo | Descripción |
| --- | --- | --- |
| size_points | double | Tamaño (en puntos) de la fuente |
| scale | double | Escala |
| anchor | GtkAnchorType | Forma de encuadrar el texto en el rectángulo del texto |
| justification | GtkJustification | Justificación para texto multilínea |
| clip_width | double | Anchura del rectángulo de recorte |
| clip_height | double | Alto del rectángulo de recorte |
| clip | boolean | Determina si usar o no un rectángulo de recorte |
| x_offset | double | Distancia desde el valor de la coordenada X |
| y_offset | double | Distancia desde el valor de la coordenada Y |
| fill_color | gchar* | Nombre del color de relleno |
| fill_color_gdk | GdkColor* | Puntero a un GdkColor para relleno |
| fill_color_rgba | guint | Valor RGBA para rellenar el texto |

Para crear un objeto `GnomeCanvasText` hay que usar el valor devuelto por `gnome_canvas_text_get_type`.

```
GType gnome_canvas_text_get_type(void);
```

## Líneas

Crear líneas con `GnomeCanvasLine` es muy similar a crear polígonos, ya que ambos usan `GnomeCanvasPoints` para definir las rutas, y comparten muchos parámetros en común.

**Tabla 13-4. Parámetros de GnomeCanvasLine**

| Nombre | Tipo | Descripción |
| --- | --- | --- |
| points | GnomeCanvasPoints* | Lista de puntos que definen las trayectorias de las líneas |
| fill_color_rgba | guint32 | Color RGBA de relleno |
| fill_color | gchar* | Nombre del color de relleno |
| width_pixels | uint | Anchura de la línea en pixels |

| Nombre | Tipo | Descripción |
|---|---|---|
| width_units | gdouble | Anchura de la línea en unidades del canvas |

Para crear un objeto `GnomeCanvasLine` hay que usar el valor devuelto por `gnome_canvas_line_get_type`.

```
GType gnome_canvas_line_get_type(void);
```

## Imágenes

`GnomeCanvasPixbuf` permite añadir imágenes al canvas. Las imágenes se deben cargar en un `GdkPixbuf`, y luego pasarse a `GnomeCanvasPixbuf`.

Al igual que todos los objetos por defecto de `GnomeCanvas`, la configuración de `GnomeCanvasPixbuf` se hace por sus parámetros.

**Tabla 13-5. Parámetros GnomeCanvasPixbuf**

| Nombre | Tipo | Descripción |
|---|---|---|
| pixbuf | GdkPixbuf* | Imagen a mostrar |
| width | double | Anchura del rectángulo donde encuadrar la imagen |
| width_set | boolean | Anchura de la imagen cuando se muestre en el canvas |
| height | double | Altura del rectángulo donde encuadrar la imagen |
| height_set | boolean | Altura que tendrá que la imagen cuando se muestre en el canvas |
| x | double | Coordenada X donde poner la imagen |
| y | double | Coordenada X donde poner la imagen |
| anchor | GtkAnchorType | Forma de encuadrar el texto en el rectángulo |

La diferencia entre *`width_set`* y *`width`* es que *`width_set`* cambia la anchura de la imagen, de modo que la imagen se escala para adoptar el nuevo valor, mientras que *`width`* respeta las dimensiones de la imagen. Lo mismo pasa con *`height_set`* y *`height`*.

En el caso de *`width`* y *`height`*, *`anchor`* se usa para encajar la imagen cuando la anchura y la altura de la imagen es menor que la del rectángulo formado por los parámatros *`x`*, *`y`*, *`width`* y *`height`*.

## Widgets

El `GnomeCanvasWidget` permite insertar *widgets* dentro del canvas. Los *widgets* son

colocados en el propio `GnomeCanvas`, ya que éste deriva de `GtkLayout`.

**Tabla 13-6. Parámetros de GnomeCanvasWidget**

| Nombre | Tipo | Descripción |
|---|---|---|
| widget | GtkWidget* | Puntero al widget |
| x | double | Coordenada X donde situar el widget |
| y | double | Coordenada Y donde situal el widget |
| width | double | Anchura del widget |
| height | double | Altura del widget |
| anchor | GtkAnchorType | Colocación del widget |
| size_pixels | boolean | Si es TRUE, los valores de width y height serán medidos en pixels |

El valor de *`size_pixels`* determina el tipo de unidad a usar. Si es FALSE, los valores de *`width`* y *`height`* se medirán en unidades del canvas, por lo que si el canvas cambia de zoom, el widget se redimensionará. Si es TRUE, el widget no se redimensionará, manteniendo su aspecto aunque cambie el zoom del canvas.

## Ejemplo sobre los objetos

En este ejemplo se muestra cómo poner varios objetos sobre el canvas. Para la imagen se ha escogido el fichero `gnome.png` y para el *widget* se ha creado un botón.

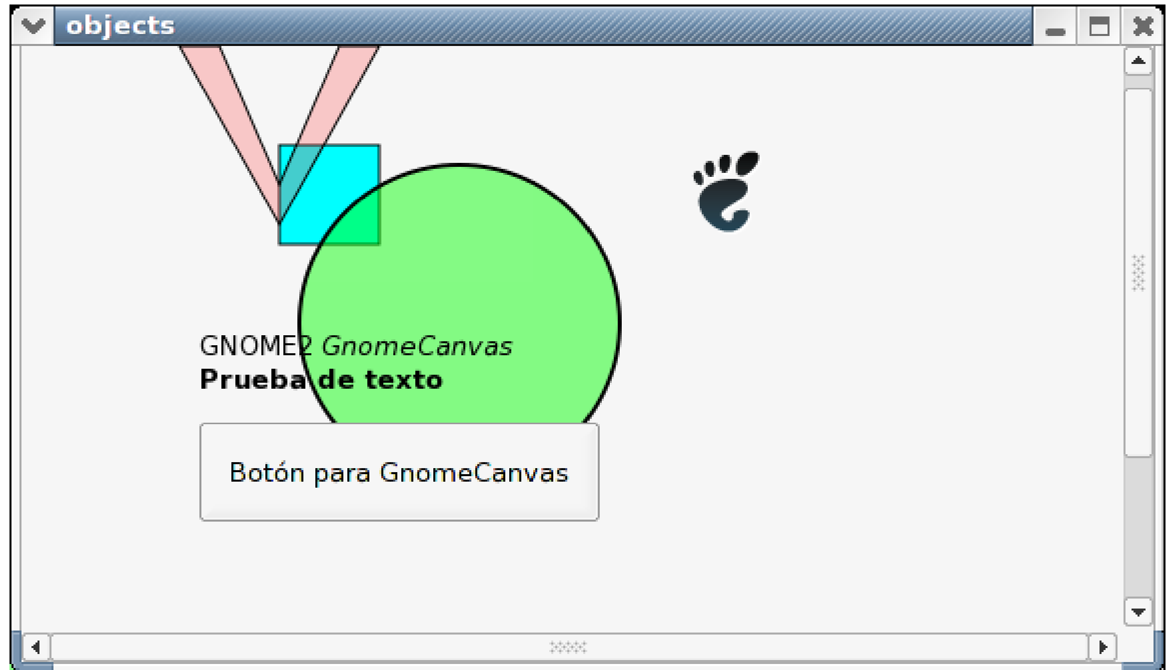



**Figura 13-3. Objetos en el canvas**

**Ejemplo 13-5. Objetos en el canvas**

```
#include <gnome.h>

int
main(int argc, char** argv)
{
        GtkWidget *window;
        GtkWidget *canvas;
        GtkWidget *button;
        GtkWidget *scrolled;
        GdkPixbuf *pixbuf;
        GnomeCanvasGroup *root;
        GnomeCanvasPoints *points;

        /* inicializamos las librerías */
        gnome_init("objects-canvas", "0.1", argc, argv);

        /* crear una ventana que contenga al canvas */
        window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
        g_signal_connect(G_OBJECT(window), "destroy", gtk_main_quit, 0);

        /* ventana para tener barras de desplazamiento */
        scrolled = gtk_scrolled_window_new(NULL, NULL);
        gtk_container_add(GTK_CONTAINER(window), scrolled);

        /* canvas en modo RGB */
        canvas = gnome_canvas_new_aa();
        gtk_container_add(GTK_CONTAINER(scrolled), canvas);
        gnome_canvas_set_scroll_region(GNOME_CANVAS(canvas),
                        -10.0, -10.0, 400.0, 400.0);

        root = gnome_canvas_root(GNOME_CANVAS(canvas));
```

```
/* un rectángulo simple */
gnome_canvas_item_new(root,
                gnome_canvas_rect_get_type(),
                "x1", 50.0,
                "y1", 50.0,
                "x2", 100.0,
                "y2", 100.0,
                "fill_color_rgba", 0x00ffffff,
                "outline_color", "black",
                "width_units", 1.0,
                NULL);

/* círculo, con fondo semitransparente */
gnome_canvas_item_new(root,
                gnome_canvas_ellipse_get_type(),
                "x1", 60.0,
                "y1", 60.0,
                "x2", 220.0,
                "y2", 220.0,
                "fill_color_rgba", 0x00ff0077,
                "outline_color", "black",
                "width_units", 2.0,
                NULL);

/* polígono con forma de V */
points = gnome_canvas_points_new(6); /* 6 puntos */

points->coords[0]  = 0;
points->coords[1]  = 0;
points->coords[2]  = 20;
points->coords[3]  = 0;
points->coords[4]  = 50;
points->coords[5]  = 70;
points->coords[6]  = 80;
points->coords[7]  = 0;
points->coords[8]  = 100;
points->coords[9]  = 0;
points->coords[10] = 50;
points->coords[11] = 90;

gnome_canvas_item_new(root,
                gnome_canvas_polygon_get_type(),
                "points", points,
                "fill_color_rgba", 0xff444444,
                "outline_color", "black",
                "width_units", 1.0,
                NULL);

/* borrar los puntos */
gnome_canvas_points_unref(points);

/* Crear un objeto de texto sobre la figura */
gnome_canvas_item_new(root,
                gnome_canvas_text_get_type(),
                "font", "Sans 10",
                "markup", "GNOME2 <i>GnomeCanvas</i>\n<b>Prueba de texto</b>",
                "fill_color", "black",
                "x", 10.0,
                "y", 160.0,
                "anchor", GTK_ANCHOR_WEST,
                NULL);

/* Crear un botón con la etiqueta "Botón para GnomeCanvas" */
button = gtk_button_new_with_label("Botón para GnomeCanvas");
gnome_canvas_item_new(root,
```

```
                    gnome_canvas_widget_get_type(),
                    "widget", button,
                    "x", 10.0,
                    "y", 190.0,
                    "width", 200.0,
                    "height", 50.0,
                    NULL);

        /* Cargar la imagen del archivo gnome.png y ponerla en el canvas */
        pixbuf = gdk_pixbuf_new_from_file("gnome.png", NULL);
        gnome_canvas_item_new(root,
                    gnome_canvas_pixbuf_get_type(),
                    "pixbuf", pixbuf,
                    "x", 250.0,
                    "y", 50.0,
                    "width", 200.0,
                    "height", 200.0,
                    NULL);
        gdk_pixbuf_unref(pixbuf);

        /* mostrar los widgets creados y entrar en el bucle gtk_main */
        gtk_widget_show_all(window);
        gtk_main();
}
```

# Trabajar con los objetos

`GnomeCanvas` ofrece diversas funciones para manipular la posición de los objetos en el canvas.

Las tranformaciones geométricas necesarias para los objetos se hacen usando matrices *affine*. Estas matrices definen valores de escala, rotación y *shearing*. Las matrices se pueden manipular fácilmente usando las funciones de libart_lgpl. En la documentación de libart_lgpl[1] se encuentra una descripción detallada sobre estas matrices.

## Manejando objetos

### Mostrar y ocultar

Para mostrar y ocultar disponemos de `gnome_canvas_item_show` y `gnome_canvas_item_hide`.

```
void gnome_canvas_item_show(GnomeCanvasItem* item);
```

```
void gnome_canvas_item_hide(GnomeCanvasItem* item);
```

Los grupos formados por `GnomeCanvasGroup` también pueden ocultarse o mostrarse usando estas funciones, ya que `GnomeCanvasGroup` deriva de `GnomeCanvasItem`. Cuando un grupo se oculta, todos los objetos que hay dentro de él también son ocultados.

### Mover

Para poder mover objetos sin cambiar sus parámetros se puede usar `gnome_canvas_item_move`

```
void gnome_canvas_item_move(GnomeCanvasItem* item, double dx, double
dy);
```

Los valores de *dx* y *dy* indican la cantidad de espacio que tiene que moverse desde la posición en la que estén en el momento de llamar a la función. Esta función es especialmente útil cuando no es conocido el tipo de objeto que se quiere mover, ya que no todos usan los mismos parámetros para ubicarse en el canvas.

Cuando esta función se aplica sobre un grupo, todos los objetos de ese grupo son movidos por igual.

### Transformar

Para hacer transformaciones sobre los objetos (como rotar o escalar) es necesario crear una matriz `affine` y aplicarla sobre el objeto.

Cuando la matriz esté lista, se deberá llamar a `gnome_canvas_item_affine_relative` o a `gnome_canvas_item_affine_absolute` para aplicarla.

```
void gnome_canvas_item_affine_relative(const double affine[6]);
```

```
void gnome_canvas_item_affine_absolute(const double affine[6]);
```

La diferencia entre estas dos funciones es que `gnome_canvas_item_affine_absolute` elimina cualquier otra transformación que se haya aplicado anteriormente sobre ese mismo objeto, mientras que `gnome_canvas_item_affine_relative` combina la matriz enviada como argumento con la que ya exista.

### Reagrupar

Cambiar el grupo al que pertenece un elemento es posible con `gnome_canvas_item_reparent`. Crear grupos facilita trabajar con varios objetos a la vez como si fueran uno solo, lo que permite ocultarlos, moverlos o incluso destruirlos con una sola llamada.

```
void gnome_canvas_item_reparent(GnomeCanvasItem* item,
GnomeCanvasGroup* new_group);
```

## Recibiendo eventos

Los objetos que se encuentren en el canvas reciben eventos del mismo modo que lo hace un *widget*, es decir, a través de señales. Cualquier objeto o grupo puede recibir eventos del teclado o del ratón.

La propagación de eventos se hace a través de grupos. Primero el evento se envía al grupo principal (obtenido con `gnome_canvas_root`). Éste lo propaga por los objetos que le pertenezcan. Si el evento llega a otro grupo, se vuelve a propagar entre los objetos de ese nuevo grupo.

Es posible conectar una señal a un grupo, de modo que la función conectada (pasada como tercer argumento de `g_signal_connect`) será llamada por cada evento que ocurra en alguno de los objetos pertenecientes a ese grupo.

### Señal *event*

Todos los eventos que pueden recibir los objetos se mandan por la señal *event*

```
int item_event(GnomeCanvasItem* item, GdkEvent* event, gpointer
data);
```

La estructura *event* se rellena con la misma información que recibe el canvas, con las coordenadas ya transformadas en unidades del canvas, teniendo en cuenta el zoom y el desplazamiento de las barras (si las hubiera).

### Capturando eventos

Un objeto sólo recibe eventos del cursor cuando éste está sobre el objeto. Para cambiar este comportamiento y hacer que el objeto reciba los eventos del cursor aunque no esté sobre el objeto, se puede usar `gnome_canvas_item_grab`.

```
int gnome_canvas_item_grab(GnomeCanvasItem* item, unsigned int
event_mask, GdkCursor* cursor, guint32 etime);
```

Los argumentos *event_mask* y *etime* son los mismos que recibe `gdk_pointer_grab`. El argumento *cursor* es el cursor que se motrará mientras el objeto tenga el control sobre los eventos del cursor. El valor devuelto es el mismo que `gdk_pointer_grab`, que será *GDK_GRAB_SUCCESS* en caso de éxito.

Cuando se quiera liberar el cursor se debe llamar a `gnome_canvas_item_ungrab`, que recibe como argumentos el objeto que tiene el control de los eventos del cursor y el valor pasado en el argumento *etime* del `gnome_canvas_item_grab`.

```
void gnome_canvas_item_ungrab(GnomeCanvasItem* item, guint32 etime);
```

El canvas también ofrece la posibilidad de enviar los eventos que reciba del teclado a un determinado objeto. Para ello es necesario llamar a `gnome_canvas_item_grab_focus`. Al contrario que `gnome_canvas_item_grab`, ésta sólo redirige los eventos que le lleguen al *widget*, no todos los del sistema.

```
void gnome_canvas_item_grab_focus(GnomeCanvasItem* item);
```

## Ejemplo con señales

El siguiente ejemplo muestra cómo interpretar los eventos del ratón de modo que se puedan mover los objetos que hay en el canvas, arrastrándolos manteniendo pulsado cualquier botón del ratón.

**Ejemplo 13-6. Capturando eventos**

```
/*
 * Este ejemplo muestra cómo conectar una función para que reciba los eventos
 * que llegan de los objetos del canvas.
 *
 * La función mueve el objeto cuando se pulsa con el botón izquierdo y se
 * mueve el cursor.
 */

#include <gnome.h>

/*
 * Esta función hace que el objeto se arrastre si se pulsa el botón izquierdo
 * del ratón y se mueve. Es decir, se arrastra el objeto :-)
 */
int
item_event(GnomeCanvasItem* item, GdkEvent* event, gpointer data)
{
        static gboolean dragging = FALSE;
        static gdouble x;
        static gdouble y;
        gdouble new_x, new_y;
        GdkCursor *cursor;

        new_x = event->button.x;
        new_y = event->button.y;

        switch(event->type)
        {
                case GDK_BUTTON_PRESS:
                        dragging = TRUE;
                        x = new_x;
                        y = new_y;

                        cursor = gdk_cursor_new(GDK_FLEUR);
                        gnome_canvas_item_grab(item,
                                        GDK_POINTER_MOTION_MASK |
                                        GDK_BUTTON_RELEASE_MASK |
                                        GDK_BUTTON_PRESS_MASK,
                                        cursor, GDK_CURRENT_TIME);
                        gdk_cursor_destroy(cursor);
                        break;

                case GDK_MOTION_NOTIFY:
                        if(dragging)
                        {
                                gnome_canvas_item_move(item, new_x - x, new_y - y);
                                x = new_x;
                                y = new_y;
                        }
                        break;

                case GDK_BUTTON_RELEASE:
                        if(dragging)
                        {
                                dragging = FALSE;
                                gnome_canvas_item_ungrab(item, GDK_CURRENT_TIME);
                        }
```

```
                        break;

                default:
                        break;
        }
}

/*
 * Para cada objeto en el canvas, se conectará la señal "event" a la función item_event
 * para que se mueva junto con el cursor cuando se pulse con el ratón
 */
void
set_signals(GnomeCanvasItem* item)
{
        g_signal_connect(G_OBJECT(item), "event", G_CALLBACK(item_event), 0);
}

int
main(int argc, char** argv)
{
        GtkWidget *window;
        GtkWidget *canvas;
        GnomeCanvasGroup *root;

        /* inicializamos las librerías */
        gnome_init("events-canvas", "0.1", argc, argv);

        /* crear una ventana que contenga al canvas */
        window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
        g_signal_connect(G_OBJECT(window), "destroy", gtk_main_quit, 0);

        /* canvas en modo RGB */
        canvas = gnome_canvas_new_aa();
        gtk_container_add(GTK_CONTAINER(window), canvas);

        root = gnome_canvas_root(GNOME_CANVAS(canvas));

        /* poner un rectángulo */
        set_signals(gnome_canvas_item_new(root,
                        gnome_canvas_rect_get_type(),
                        "x1", 0.0,
                        "y1", 0.0,
                        "x2", 100.0,
                        "y2", 100.0,
                        "fill_color_rgba", 0xffff0077,
                        "outline_color_rgba", 0x0000ffff,
                        "width_units", 1.0,
                        NULL));

        /* poner un círculo */
        set_signals(gnome_canvas_item_new(root,
                        gnome_canvas_ellipse_get_type(),
                        "x1", 20.0,
                        "y1", 25.0,
                        "x2", 80.0,
                        "y2", 75.0,
                        "fill_color_rgba", 0x00ff0077,
                        "outline_color_rgba", 0xff00ffff,
                        "width_units", 1.5,
                        NULL));

        /* mostrar los widgets creados y entrar en el bucle gtk_main */
        gtk_widget_show_all(window);
        gtk_main();
}
```

# Crear objetos propios

Lo más interesante del `GnomeCanvas` es poder crear objetos propios, ya que con ello se puede ajustar el canvas a las neceidades de cada aplicación.

Todos los objetos que se creen deben derivar de `GnomeCanvasItem` o de alguna clase que derive de `GnomeCanvasItem`.

**Ejemplo 13-7. Objeto propio**

```
typedef struct _MyItem MyItem;

struct _MyItem
{
        GnomeCanvasItem item_class;

        /* datos propios */
};
```

## Métodos

Todos los objetos deben dar una serie de métodos para que el canvas pueda interactuar con ellos, bien para que realicen alguna acción (como dibujarse) o para que den información (como su posición en el canvas).

Para dar tal funcionalidad a los objetos se deben sobrescribir los valores de los miembros de `GnomeCanvasItem`, los cuales se explican a continuación

### point

El método *point* se usa para calcular qué objeto está en un determinado punto. Este método devolverá la distancia que hay entre el objeto y el punto y, si el punto está dentro del objeto, devolverá cero. El valor exacto no es muy importante, ya que el principal uso de este método es ver si un punto está dentro del objeto (devuelve cero) o no (devuelve un valor distinto de cero).

```
struct _GnomeCanvasItemClass {
        GtkObjectClass parent_class;

/* ... */

        double (* point) (GnomeCanvasItem *item, double x, double y,
                int cx, int cy, GnomeCanvasItem **actual_item);
};
```

Los argumentos *`x`* e *`y`* corresponden al punto en unidades del canvas, mientas que *`cx`* y *`cy`* corresponden al mismo punto, pero en pixels del canvas.

El último argumento, *`actual_item`*, debe rellenarse siempre.

**Ejemplo 13-8. Crear un evento point**

### bounds

Este no lo usa internamente el canvas, y sólo es requerido por `gnome_canvas_item_get_bounds`. Este evento calcula el rectángulo que encuadra al objeto.

```
struct _GnomeCanvasItemClass {
        GtkObjectClass parent_class;

/* ... */

        double (* point) (GnomeCanvasItem *item, double x, double y,
                int cx, int cy, GnomeCanvasItem **actual_item);

        void (* bounds) (GnomeCanvasItem *item, double *x1, double *y1,
                double *x2, double *y2);
};
```

```
void gnome_canvas_item_get_bounds(GnomeCanvasItem* item, double* x1,
double* y1, double* x2, double* y2);
```

Puesto que no es usado internamente por el canvas, no es necesario implementar este evento.

### event

Este evento hace lo mismo que la señal *event*. La única diferencia es que con este evento no es necesario conectar ninguna señal

```
struct _GnomeCanvasItemClass
{
        GtkObjectClass parent_class;

/* ... */

        gboolean (* event)  (GnomeCanvasItem *item, GdkEvent *event);
};
```

En el apartado Recibiendo eventos y los siguientes hay una descripción de event.

### *realizing* y *mapping*

Los métodos *realize*, *unrealize*, *map* y *unmap* tienen el mismo significado que las señales equivalentes en los widgets.

*realize* se usa para crear todos los recursos GDK que el objeto del canvas quiera usar, y *unrealize* es llamado cuando esos recursos deben ser liberados.

*map* y *unmap* son usados sólo por objetos que tengan una `GdkWindow` asociada con ellos. Esto es poco habitual, aunque hay excepciones como `GnomeCanvasWidget`. *map* es invocado cuando su `GdkWindow` es mostrada, y *unmap* cuando se oculta.

**Ejemplo 13-9. realize y unrealize**

```
GnomeCanvasItemClass * my_item_parent_class;

typedef struct {
        GnomeCanvasItem item;

        GdkDC *bg_gc;
} MyItem;

typedef struct {
        GnomeCanvasItemClass parent_class;

} MyItemClass;

/* ... */

void
my_item_realize(GnomeCanvasItem* item)
{
        MyItem* my_item = MY_ITEM(item);

        /* Llamar a la función de la clase GnomeCanvasItemClass.
         * Para obtener la dirección de la clase se llama a g_type_class_peek_parent
         * durante la inicialización de la clase MyItemClass;
         */
        if (my_item_parent_class->realize)
                (* my_item_parent_class->realize)(item);

        /* Crear el GC cuando el canvas no esté en modo RGB */
        if(!item->canvas->aa)
                my_item->bg_gc = gdk_gc_new (item->canvas->layout.bin_window);
}

void
my_item_unrealize(GnomeCanvasItem* item)
{
        MyItem* my_item = MY_ITEM(item);

        if (my_item_parent_class->unrealize)
                (* my_item_parent_class->unrealize)(item);

        /* Crear el GC cuando el canvas no esté en modo RGB */
        if(!item->canvas->aa)
                gdk_gc_unref (my_item->bg_gc);
}
```

## Dibujar en el canvas

Para poner contenido en el canvas hay que implementar al menos dos métodos. Uno de ellos es *update*, y el segundo puede ser *draw* o *render*, aunque es posible tener los dos.

## render y draw

La diferencia entre *draw* y *render* es que *draw* es llamado cuando el canvas está en modo GDK, mientras que *render* es llamado cuando el canvas está en modo RGB.

```
struct _GnomeCanvasItemClass {
        GtkObjectClass parent_class;

/* ... */

        double (* draw) (GnomeCanvasItem *item, GdkDrawable *drawable,
                         int x, int y, int width, int height);

        void (* render) (GnomeCanvasItem *item, GnomeCanvasBuf *buf);
}
```

En *draw*, el argumento *item* indica qué objeto se solicita dibujar en el `GdkDrawable` indicado por el argumento *drawable*. Los cuatro últimos argumentos indican la zona que se pidea actualizar.

En *render*, la información se pasa a través de una estructura de tipo `GnomeCanvasBuf`.

```
/* Data for rendering in antialiased mode */
typedef struct {
        /* 24-bit RGB buffer for rendering */
        guchar *buf;

        /* Rectangle describing the rendering area */
        ArtIRect rect;

        /* Rowstride for the buffer */
        int buf_rowstride;

        /* Background color, given as 0xrrggbb */
        guint32 bg_color;

        /* Invariant: at least one of the following flags is true. */

        /* Set when the render rectangle area is the solid color bg_color */
        unsigned int is_bg : 1;

        /* Set when the render rectangle area is represented by the buf */
        unsigned int is_buf : 1;
} GnomeCanvasBuf;
```

- *buf* es un puntero al búfer RGB donde se debe dibujar.
- *rect* indica la zona donde se quiere dibujar.
- *buf_rowstride* es el número de bytes que tiene cada fila de pixels
- *bg_color* es el color que debe tener el fondo del canvas.
- *is_bg* e *is_buf* indican cuándo se ha dibujado en el búfer RGB y cuándo se debe usar el color del fondo.

Siempre hay que comprobar el valor de los miembros *is_buf* e *is_bg*.

- Si queremos poner contenido en el búfer RGB, *is_buf* debe ser TRUE y *is_bg* FALSE

- Si *is_bg* es TRUE, el canvas usará el valor del miembro *bg_color* para dibujar un rectángulo sólido.
- Cuando se quiere dibujar sobre el búfer RGB es preferible llamar a `gnome_canvas_buf_ensure_buf` para evitar que en el búfer hayan residuos.

```
void gnome_canvas_buf_ensure_buf(GnomeCanvasBuf* buf);
```

**Ejemplo 13-10. Verificar is_buf y is_bg**

```
void my_item_render (GnomeCanvasItem *item, GnomeCanvasBuf *buf)
{
        gnome_canvas_buf_ensure_buf(buf)
        buf->is_bg = 0;

        /* Dibujar en el búfer */

}
```

### update

Los métodos *render* y *draw* son llamados cada vez que el canvas necesita actualizar el widget.

El canvas puede llamar a estos eventos varias veces en una sola actualización, por lo que es necesario que los eventos tengan la menor carga posible, ya que, de lo contrario, actualizar el contenido del canvas puede ser muy lento.

Para evitar cargar los métodos *render* o *draw* se usa el método *update*. *update* es llamado sólo cuando los datos del objeto cambian, y hace falta actualizarse. Casi siempre, después de *update*, el canvas llamará a *draw* o *render*.

```
struct _GnomeCanvasItemClass
{
        GtkObjectClass parent_class;

/* ... */

        void (* update) (GnomeCanvasItem *item, double *affine,
                ArtSVP *clip_path, int flags);
}
```

El canvas usa este método para indicarle al objeto que se actualice él mismo y guarde su estado, para que cuando sea necesario pintar se haga más rápidamente, sin tener que calcular repetidamente los datos del objeto.

Es muy importante usar este evento para hacer los cálculos que se necesiten para dibujar en el canvas, y usar *render* o *draw* únicamente para mostrar los datos sin tener que calcular nada (o lo menos posible).

## El objeto genérico GnomeCanvasShape

El canvas tiene un objeto propio que de por sí no hace nada, pero sirve de base para otros como `GnomeCanvasRect`, `GnomeCanvasPolygon` o `GnomeCanvasEllipse`. Este objeto es `GnomeCanvasShape`

Este objeto implementa los eventos necesarios para dibujar cualquier figura sobre el canvas. Para definir la figura que contiene el objeto se usan rutas.

## Rutas

Las rutas son similares a las instrucciones PostScript de dibujo: definen figuras con funciones como LINETO, MOVETO o CURVETO.

Para enviar una ruta a `GnomeCanvasShape` hay que crearla antes con `gnome_canvas_path_def_new`. Esta función devolverá un puntero a la estructura `GnomeCanvasPathDef`, que será usada por funciones como `gnome_canvas_path_def_moveto`, `gnome_canvas_path_def_lineto` o `gnome_canvas_path_def_curveto`.

Cuando las rutas no son más necesarias hay que liberarlas con `gnome_canvas_path_def_unref`.

```
GnomeCanvasPathDef* gnome_canvas_path_def_new(void);
```

```
void gnome_canvas_path_def_moveto(GnomeCanvasPathDef* path, gdouble x,
gdouble y);
```

```
void gnome_canvas_path_def_lineto(GnomeCanvasPathDef* path, gdouble x,
gdouble y);
```

```
void gnome_canvas_path_def_curveto(GnomeCanvasPathDef* path, gdouble
x0, gdouble y0, gdouble x1, gdouble y1, gdouble x2, gdouble y2);
```

```
void gnome_canvas_path_def_unref(GnomeCanvasPathDef* path);
```

Para asegurar que una figura está cerrada es posible llamar a `gnome_canvas_path_def_closepath` o `gnome_canvas_path_def_closepath_current`. Con `gnome_canvas_path_def_closepath` se añadirá una línea entre el último punto creado y el primero (si están separados), y con `gnome_canvas_path_def_closepath_current` se moverán las coordenadas del último punto para que esté encima del primero. Con ambas funciones se garantiza que la figura definida por la ruta está cerrada.

## Pasar una ruta a GnomeCanvasShape

Una vez que la ruta esté creada es necesario mandarla a `GnomeCanvasShape`. Esto se hace con `gnome_canvas_shape_set_path_def`. Una vez que la ruta se haya mandado a `GnomeCanvasShape` hay que liberarla con `gnome_canvas_path_def_unref`, a menos que se vaya a volver a emplear en otra llamada a `gnome_canvas_shape_set_path_def`.

```
void gnome_canvas_shape_set_path_def(GnomeCanvasShape* shape,
GnomeCanvasPathDef* path);
```

## Uso

Para usar `GnomeCanvasShape` hay que crear un nuevo objeto que derive de `GnomeCanvasShape` en lugar de `GnomeCanvasItem`.

El único evento que se debe implementar es el *update*, ya que `GnomeCanvasShape` implementa los demás métodos necesarios.

**Ejemplo 13-11. Cabecera para usar GnomeCanvasShape**

```
typedef struct _MyItem MyItem;
typedef struct _MyItemClass MyItemClass;

struct _MyItem {
        GnomeCanvasShape shape_item;

        /* datos propios de este objeto */
};

struct _MyItemClass {
        GnomeCanvasShapeClass parent_class;
};
```

**Ejemplo 13-12. Funciones para usar GnomeCanvasShape**

```
static GnomeCanvasShapeClass* my_item_parent_class;

void
my_item_class_init(MyItemClass* class)
{
        GnomeCanvasItemClass *item_class;

        item_class = (GnomeCanvasItemClass *) class;

        item_class->update = my_item_update;
        my_item_parent_class = g_type_class_peek_parent (class);
}

void
my_item_update (GnomeCanvasItem *item, gdouble *affine,
        ArtSVP *clip_path, gint flags)
{
        GnomeCanvasPathDef* path_def;

        /* crear una ruta */
        path_def = gnome_canvas_path_def_new();

        /*
         * llamadas a gnome_canvas_path_def_* para crear una figura
         */

        /* meter la ruta en el GnomeCanvasShape */
        gnome_canvas_path_def_closepath_current(path_def);
        gnome_canvas_shape_set_path_def (GNOME_CANVAS_SHAPE (item), path_def);
        gnome_canvas_path_def_unref(path_def);

        /* Llamar al update de GnomeCanvasShape para terminar */
        if (my_item_parent_class->update)
                (* my_item_parent_class->update) (item, affine, clip_path, flags);
}
```

Al final de la función de *update* es necesario llamar al *update* de `GnomeCanvasShape`. Éste creará a partir de la ruta que hemos definido con `gnome_canvas_shape_set_path_def` los datos que necesite para responder a los demás evento del canvas.

## Un objeto propio

Este objeto, `MyItem`, deriva de `GnomeCanvasShape`. Sólo implementa el método *update*, y deja todos los demás a `GnomeCanvasShape`.

Además, en el ejemplo se usa un widget `GtkScale` para poder cambiar el zoom del canvas.

**Ejemplo 13-13. MyItem**

```
#include <gnome.h>

/*
 * Definición de MyItem
 */

typedef struct _MyItem MyItem;
typedef struct _MyItemClass MyItemClass;

struct _MyItem {
        GnomeCanvasShape shape;

        gdouble x0;
        gdouble y0;

        gdouble x1;
        gdouble y1;
};


struct _MyItemClass {
        GnomeCanvasShapeClass parent_class;
};

#define MY_ITEM_TYPE (my_item_get_type())
#define MY_ITEM(obj) (GTK_CHECK_CAST ((obj), MY_ITEM_TYPE, MyItem))


/*
 * "update" para calcular la ruta que se enviará a GnomeCanvasShape
 */
void my_item_update (GnomeCanvasItem *item, gdouble *affine,
                ArtSVP *clip_path, gint flags);

/*
 * Guardar la dirección de la clase padre de MyItem para llamar a su update
 */
static GnomeCanvasItemClass *my_item_parent_class;

/*
 * Inicializar el objeto. &Eacute;sta es llamada cada vez que se crea un objeto
 */
void
```

```
my_item_init(MyItem* item)
{
        item->x0 = 100.0;
        item->y0 = 100.0;
        item->x1 = 200.0;
        item->y1 = 200.0;
}

/*
 * Inicializar la clase. &Eacute;sta es llamada sólo una vez, cuando se crea el primer
 * objeto y se crea la clase
 */
void
my_item_class_init(MyItemClass* class)
{
        GnomeCanvasItemClass *item_class;

        item_class = (GnomeCanvasItemClass *) class;
        item_class->update = my_item_update;

        my_item_parent_class = g_type_class_peek_parent (class);
}

/*
 * Usada para crear la clase y poder enviar un GType a gnome_canvas_item_new
 */
GType
my_item_get_type ()
{
        static GType item_type = 0;

        if(item_type == 0)
        {
                static const GTypeInfo object_info = {
                        sizeof (MyItemClass),
                        (GBaseInitFunc) NULL,
                        (GBaseFinalizeFunc) NULL,
                        (GClassInitFunc) my_item_class_init,
                        (GClassFinalizeFunc) NULL,
                        NULL,
                        sizeof (MyItem),
                        0,
                        (GInstanceInitFunc) my_item_init,
                        NULL
                };

                item_type = g_type_register_static (GNOME_TYPE_CANVAS_SHAPE,
                                    "MyItem", &object_info, 0);
        }

        return item_type;
}

void my_item_update (GnomeCanvasItem *item, gdouble *affine,
                ArtSVP *clip_path, gint flags)
{
        GnomeCanvasPathDef* path_def;
        MyItem* my_item;
        gdouble cx, cy;

        /*
         * crear una nueva ruta
         */

        path_def = gnome_canvas_path_def_new();
        my_item = MY_ITEM(item);
```

```
        cx = (my_item->x1 - my_item->x0) * 0.25;
        cy = (my_item->y1 - my_item->y0) * 0.25;

        gnome_canvas_path_def_moveto(path_def, my_item->x0, my_item->y0);
        gnome_canvas_path_def_lineto(path_def, my_item->x0 + cx, my_item->y0 - cy );
        gnome_canvas_path_def_lineto(path_def, my_item->x1, my_item->y1);
        gnome_canvas_path_def_lineto(path_def, my_item->x1 - cx, my_item->y1 + cy);
        gnome_canvas_path_def_lineto(path_def, my_item->x0, my_item->y0);

        /* mandar la ruta a GnomeCanvasShape */
        gnome_canvas_path_def_closepath(path_def);
        gnome_canvas_shape_set_path_def (GNOME_CANVAS_SHAPE (item), path_def);
        gnome_canvas_path_def_unref(path_def);

        if (my_item_parent_class->update)
                (* my_item_parent_class->update)(item, affine, clip_path, flags);
}


gboolean
button_press_event(GtkWidget* canvas, GdkEventButton* event, MyItem* item)
{
        gdouble x, y;
        gnome_canvas_window_to_world(GNOME_CANVAS_ITEM(item)->canvas, event->x, event->

        switch(event->button)
        {
                case 1: /* botón izquierdo */
                        item->x0 = x;
                        item->y0 = y;
                        gnome_canvas_item_request_update(GNOME_CANVAS_ITEM(item));
                        break;

                case 3: /* botón derecho */
                        item->x1 = x;
                        item->y1 = y;
                        gnome_canvas_item_request_update(GNOME_CANVAS_ITEM(item));
                        break;

                default:
                        break;
        }

        /*
         * Devolver TRUE finaliza la propagación del evento.
         * FALSE lo sigue propagando
         */
        return FALSE;
}

void
on_zoom_changed(GtkAdjustment* adjs, GnomeCanvas* canvas)
{
        gnome_canvas_set_pixels_per_unit(canvas, adjs->value);
}

int
main(int argc, char** argv)
{
        GtkWidget *window;
        GtkWidget *canvas;
        GtkWidget *scroll_window;
        GtkWidget *vbox;
        GtkWidget *scrool_zoom;
        GnomeCanvasGroup *root;
```

```
        GnomeCanvasItem* item;
        GtkAdjustment *adjustment;

        /* inicializamos las librerías */
        gnome_init("my_item", "0.1", argc, argv);

        /* crear una ventana que contenga al canvas */
        window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
        g_signal_connect(G_OBJECT(window), "destroy", gtk_main_quit, 0);

        vbox = gtk_vbox_new(FALSE, 1);
        gtk_container_add(GTK_CONTAINER(window), vbox);

        /* Usar una barra de desplazamiento para el zoom */
        adjustment = GTK_ADJUSTMENT(gtk_adjustment_new(1.0, 0.25, 4.0, 0.1, 0.5, 1.0));
        scrool_zoom = gtk_hscale_new(adjustment);
        gtk_scale_set_draw_value(GTK_SCALE(scrool_zoom), TRUE);
        gtk_box_pack_start(GTK_BOX(vbox), scrool_zoom, FALSE, TRUE, 1);

        /* Usar GtkScrolledWindow para disponer de barras de desplazamiento
         * en el canvas */
        scroll_window = gtk_scrolled_window_new(NULL, NULL);
        gtk_box_pack_end(GTK_BOX(vbox), scroll_window, TRUE, TRUE, 1);

        /* canvas en modo RGB */
        canvas = gnome_canvas_new_aa();
        gtk_container_add(GTK_CONTAINER(scroll_window), canvas);
        gnome_canvas_set_scroll_region(GNOME_CANVAS(canvas), 0, 0, 1000, 1000);

        root = gnome_canvas_root(GNOME_CANVAS(canvas));

        /* poner un círculo, con fondo verde y borde azul */
        item = gnome_canvas_item_new(root,
                        my_item_get_type(),
                        "fill_color", "blue",
                        "outline_color", "black",
                        "width_units", 2.0,
                        NULL);

        /* recibir los eventos de pulsación con el ratón para cambiar el item y la seña
         * de cambio de valor para el zoom*/
        g_signal_connect(G_OBJECT(window), "button-press-event", G_CALLBACK(button_pres
        g_signal_connect(G_OBJECT(adjustment), "value_changed", G_CALLBACK(on_zoom_chan

        /* mostrar los widgets creados y entrar en el bucle gtk_main */
        gtk_widget_show_all(window);
        gtk_main();

        return 0;
}
```

## Notas

1. http://www.gnome.org/~mathieu/libart/libart-affine-transformation-matrices.html

# Capítulo 14. CORBA

Los escritorios modernos están compuestos por multitud de pequeños programas que deben comunicarse los unos con los otros para que el usuario tenga la sensación de estar trabajando en un entorno coherente, un entorno en el que todo encaja y trabaja de forma coordinada.

Tómese como ejemplo el Panel, el Panel, este es un contenedor en el cual se pueden colocar diferentes "applets", estos "applets" son programas autónomos que deben comunicarse con el Panel para que el usuario tenga la sensación de que el Panel y los "applets" están perfectamente integrados.

Otros ejemplos mucho más complejos son también posibles, por ejemplo, el caso en cual un programa toma datos de un sensor y los introduce de forma automática en Gnumeric (la hoja de cálculo de GNOME Office) sin necesidad de que el usuario intervenga.

Para poder conseguir esta integración y la posibilidad de controlar las aplicaciones de esta forma es necesario un sistema que permita a los diferentes programas comunicarse entre ellos.

En este capítulo se presentará la solución que la comunidad GNOME a adoptado para este importante problema, sus características, sus ventajas y sus limitaciones.

## Introducción a CORBA

CORBA es una tecnología que oculta la programación a bajo nivel de aplicaciones distribuidas, de tal forma que el programador no se tiene que ocupar de tratar con sockets, flujos de datos, paquetes, sesiones etc. CORBA oculta todos estos detalles de bajo nivel. No obstante CORBA también brinda al programador una tecnología orientada objetos, las funciones y los datos se agrupan en objetos, estos objetos pueden estar en diferentes máquinas, pero el programador accederá a ellos a través de funciones normales dentro de su programa.

Veamos un ejemplo:

```
...
GNOME_Evolution_Calendar_Cal__setMode (object, MODE_LOCAL, &ev);
...
```

Esta función ejecutaría el método setMode sobre el objeto object, para el programador esta llamada es como una operación local, no hay más complejidad.

Los métodos y datos CORBA se agrupan formando lo que se demoninan interfaces, los interfaces pueden ser interpretados como objetos que grupan datos y métodos para acceder a estos. Todos estos interfaces se definen usado un lenguaje IDL (Interface Definition Language), que es precisamente esto, un lenguaje para la definición de interfaces. Este lenguaje es estandar y lo soportan todas las implementaciones CORBA.

es algo más que una abstracción que oculta la complejidad de red, -> TODAS LAS LLAMAS PARA EL PROGRAMADOR SON IGUALES -> SE DEFINEN OBJETOS Y METODOS -> LOS OBJETOS SON REMOTOS -> NO PORQUE CORBA OCULTE COSAS DEBEMOS DEJAR DE PENSAR EN LA EFICIENCIA DE RED Hay una multitud de sistemas con un propósito muy similar a CORBA circulando por el mundo, los más usados son: el RPC (Remote Procedure Call) de Sun Microsystems y DCOM (Distributed que los desarrolladores de GNOME han adoptado es individuales CORBA -> muy importante especificación, OMG

## La Historia de CORBA en el proyecto GNOME

El método de comunicación entre las aplicaciones que se seleccione es muy importante, esta decisión puede limitar a los programadores de forma grave durante el desarrollo de las aplicaciones, esto se notará sobre todo cuando las aplicaciones que se construyan alcancen un alto nivel de complejidad.

La solución adoptada por los arquitectos del GNOME recibe el nombre de CORBA (Common Object Request Broker Arquitecture).

CORBA es una especificación del OMG (Object Management Group) que está respaldada por importantes empresas de todos los ámbitos, desde IBM a Telefónica. CORBA permite transparentemente hacer peticiones y recibir respuestas en un entorno distribuido. Es la base para construir aplicaciones distribuidas basadas en objetos y de integrar entornos heterogéneos.

Hay que diferenciar entre una especificación y su implementación el OMG solo define especificaciones, no realiza ninguna implementación de ellas, son las empresas y/o grupos de desarrolladores los que se deben ocupar de esto.

CORBA ha sido usado en GNOME desde los primeros días de su existencia, aunque es ahora, en GNOME 2.0 cuando la introducción de CORBA en el GNOME ha alcanzado su máximo uso.

En principio el GNOME usó una implementación de CORBA denominada MICO, esta implementación también era usada por el proyecto KDE, no obstante, pronto se vieron sus problemas, MICO era muy lento y pesado, así pues, no era una solución viable para un escritorio, se precisaba de una implementación rápida y ligera. La solución fue escribir ORBit, que hoy por hoy es la implementación más rápida de CORBA existente. Lamentablemente el equipo del KDE tomó la decisión de abandonar CORBA por completo e implementar un nuevo sistema de comunicación mucho más limitado que CORBA.

En el GNOME 1.2 y anteriores CORBA se usaba solo para tareas muy sencillas y concretas, por ejemplo para integrar el Panel y los "applets", el Centro de Control y los "capplets", etc. CORBA era usado en sitios puntuales y de forma poco estándar. Para cada aplicación se hacía uso de una biblioteca que encapsulaba el uso de CORBA. Para simplificar el uso de CORBA se creó libGnorba, presente hasta el GNOME 2.0, esta biblioteca hacía poco más que inicializar el sistema CORBA. Desde el principio se tuvo la intención de crear un sistema de componentes sobre el modelo de comunicación que CORBA aportaba, así comenzó el trabajo en Bongo, más tarde se deshecho todo este trabajo y se comenzó a implementar Bonobo (del cual se habla más adelante).

Con la versión 1.4 del GNOME se liberó la primera versión estable de Bonobo así como la primera aplicación que hacía un uso extensivo de este, Nautilus que era el gestor de archivos destinado a sustituir a GMC.

Una vez liberado Bonobo han comenzado a aparecer programas que hacen uso de este sistema: Evolution (el gestor de información personal desarrollado por Ximian), Gnumeric (la hoja de cálculo), Guppi3 (programa de representación de datos), gIDE (el futuro entorno de desarrollo), Gnome-DB (aplicación de acceso a bases de datos), etc.

Con esta liberación y la aparición de estas aplicaciones se comenzaron a detectar problemas importantes en Bonobo, problemas de rendimiento y el aumento del consumo de memoria que suponía su uso, todos estos problemas y otros, han sido solucionados en GNOME 2.0.

Como se puede ver Bonobo es un sistema de componentes que está alcanzando rápidamente su madurez, en GNOME 1.4 era usado en una única aplicación, pero en GNOME 2.0 se usa de forma extensiva, desde el Panel a un conjunto significativo de aplicaciones. No obstante al no ser aún una tecnología del todo madura estará expuesta a modificaciones importantes. Aunque esto no debe asustarnos, pues el interfaz de programación no variará demasiado entre unas versiones y otras.

## El lenguaje IDL

A la hora de desarrollar aplicaciones CORBA, uno de los primeros pasos que tenemos que hacer es escribir los interfaces IDL de nuestra aplicación. IDL es un lenguaje que permite definir una serie de interfaces para la comunicación entre dos o más aplicaciones. El IDL se escribe en un fichero, normalmente con la extensión IDL, en el cual se definen los interfaces y los métodos (funciones) de esos interfaces.

Un fichero IDL podría ser el equivalente a un fichero de cabecera en C++, en el que se definen las clases (el equivalente a los interfaces CORBA) y las propiedades (variables de la clase) y métodos de esas clases. De hecho, IDL NO ES un lenguaje para implementar interfaces CORBA, sino que es un lenguaje PARA DEFINIR interfaces CORBA, por ello la equivalencia con un fichero de cabecera de C++, donde, normalmente, no se incluye ninguna implementación, sino simplemente definiciones.

Antes de seguir adelante con las explicaciones, veamos un ejemplo de un fichero IDL, en este caso, `String.idl`. En este caso, definimos un solo interfaz, `String`, que incluye dos métodos, `toUpper` y `toLower`.

```
   module MyModule {
interface String {
   string toUpper (in string str);
   string toLower (in string str);
};
   };
```

Cualquier persona que conozca algun lenguaje de programación orientado a objetos, podrá fácilmente deducir que lo que estamos haciendo aquí es definir un interfaz (o sea, una clase), con dos métodos. Efectivamente, no hay más misterio en la escritura de los interfaces IDL, excepto, por supuesto, que las palabras clave no siempre son iguales que en C/C++, aunque podemos comprobar la similitud que existe entre estos dos lenguajes y el IDL. La diferencia es, simplemente, que, con CORBA, estamos definiendo objetos dentro de un sistema de objetos distribuido, que podrá ser usado desde cualquier aplicación que soporte CORBA, independientemente del lenguaje, plataforma o sistema operativo para el que se implemente. Esto parece una clara ventaja frente al uso de objetos en entornos limitados (un solo programa, una librería que sólo puede ser usada en determinado lenguaje/sistema operativo, etc).

Este fichero IDL es el que establece el contrato entre nuestros objetos y las aplicaciones que hagan uso de ellos. Así, este fichero es lo único que necesita una aplicación cliente para acceder a nuestros objetos (aparte, por supuesto, de una implementación de CORBA), pues a partir de él puede generar los cabos y esqueletos (ver siguiente sección) para el lenguaje que vaya a ser usado para la implementación de los clientes. Es decir, el cliente no necesita de unas librerías específicas desarrolladas por nosotros para acceder a nuestros objetos, sino que simplemente necesita la definición de los interfaces implementados por nuestros objetos. CORBA se encarga del resto, como veremos a continuación.

Los tipos de datos básicos que podemos usar en IDL son: long, string, float, double, boolean. En cuanto a los tipos de datos compuestos, tenemos: struct, enum, sequence.

Aparte de métodos, tambien podemos definir propiedades para nuestros objetos. Para ello, se usa la palabra clave `attribute`, como se muestra a continuación:

```
interface MiInterfaz {
        attribute string atributo_cadena;
        readonly attribute atributo_solo_lectura;
};
```

Como vemos en el ejemplo, tambien podemos definir atributos de sólo lectura. Por cada atributo que definamos en el interfaz, el compilador IDL (que veremos a continuación) genera dos funciones/métodos en el lenguaje que vayamos a usar. Un

método es para establecer (set) el valor del atributo, mientras que el otro es para obtener (get) dicho valor. Para los atributos de sólo lectura, sólo se genera una función/método: para obtener el valor, pero no la función para modificarlo.

## ORBit, la implementación de CORBA de GNOME

ORBit cumple con los dos requisitos del proyecto GNOME: es libre, y es uno de los ORBs más rápidos que existen, aparte de que es muy ligero (sólo añade 70 KB a los programas que lo usan), lo cual lo hace ideal para ser usado en un escritorio como GNOME. Además, consigue una velocidad casi idéntica a una simple llamada a función cuando se usa en local, pues detecta automáticamente que las comunicaciones se están realizando en la misma máquina, desactivando en ese caso gran parte de la complejidad del protocolo de comunicaciones usado en CORBA (GIOP/IIOP).

Como el resto de partes básicas de la arquitectura de GNOME, ORBit está implementado en C, y el lenguaje que soporta es C precisamente. Por tanto, en este artículo vamos a hablar de cómo usar ORBit desde C, pero hay que destacar que existe soporte para otros lenguajes de programación, como C++, Perl, PHP. Información sobre esto la podemos encontrar en la sección de referencias.

### Cabos y esqueletos

Estamos hablando de independencia de lenguaje, de sistema operativo, etc. Pero, Â¿cómo se consigue esto? La respuesta es muy sencilla: una vez que nos hagamos con un ORB de CORBA (Object Request Broker, o sea, una implementación del estándar CORBA, como es el caso de ORBit), simplemente usamos un programa especial que suelen incluir los ORBs, que se conoce como compilador IDL, y que genera una serie de ficheros en determinado lenguaje de programación, que luego compilaremos en nuestro sistema operativo/plataforma. Una vez hecho esto, toda la responsabilidad de realizar las comunicaciones entre las aplicaciones haciendo uso de los objetos y los propios objetos, pasa a ser responsabilidad de nuestro ORB. Éste, una vez que sabe qué interfaces IDL se van a usar, y conocida la forma de acceder a los objetos, simplemente tiene que hacer uso del protocolo de comunicaciones de CORBA (GIOP/IIOP) para hacer las invocaciones de los métodos de los objetos.

Por tanto, nuestro siguiente paso va a consistir en generar los cabos y los esqueletos a partir de los interfaces IDL, todo ello, mediante el compilador IDL que incluye ORBit, llamado `orbit-idl`, como se muestra a continuación:

```
orbit-idl --skeleton-impl String.idl
```

Este comando generará cinco ficheros:

- `String.h`: definiciones usadas tanto en la implementación de los objetos como en las aplicaciones que hagan uso de ellos
- `String-common.c`: funciones usadas tanto en la implementación de los objetos como en las aplicaciones que hagan uso de ellos
- `String-stubs.c`: cabos para las aplicaciones que hacen uso de los objetos
- `String-skels.c`: esqueletos para la implementación de los objetos
- `String-skelimpl.c`: implementación básica de los objetos definidos en el fichero IDL. Este fichero será el que usemos como base para la implementación de los objetos. Contiene una serie de funciones vacías (no implementadas) que simplemente tendremos que rellenar con nuestra implementación.

En este ejemplo, hemos generado los cabos y esqueletos de nuestros interfaces para el lenguaje C con ORBit. Pero este paso podríamos haberlo realizado perfectamente con otro ORB en otro sistema operativo o plataforma, y así, por ejemplo, implementar los clientes (o sea, la parte que accede a los objetos) en dicho sistema operativo/plataforma, y dejar la parte de implementación de los objetos en GNU/Linux+ORBit.

## Accediendo a los objetos

Para acceder a los objetos que implementemos, lo primero que tenemos que hacer es activar dichos objetos, o sea obtener una instancia de un objeto que implemente el interfaz que queramos usar. Para hacer las cosas más sencillas, vamos a usar Bonobo Activation, que, como vimos en los artículos dedicados a Bonobo, facilita muchísimo la activación de objetos. Así, sin más dilaciones, vamos a ver el código de nuestro programa, y luego lo comentamos:

```
#include <orb/orbit.h>
#include <bonobo-activation/bonobo-activation.h>
#include "String.h"

int main (int argc, char *argv[])
{
      CORBA_Environment ev;
      MyModule_String corba_string;

              /* inicializamos Bonobo-Activation */
              bonobo_activation_init (argc, argv);

              /* activamos el objeto a través de Bonobo-Activation */
              CORBA_exception_init(&ev);
              corba_string = bonobo_activation_activate_from_id("OAFIID:MyModule_stri
              if (ev._major == CORBA_NO_EXCEPTION
                      && corba_string != CORBA_OBJECT_NIL) {
                      gchar *up;
                      gchar *lw;
                      gchar word[32];

                      /* aquí, ya tenemos una instancia de MyModule::String en corba_

                      /* ejecutamos un bucle en el que le pedimos al usuario que intr
                      /* una palabra, que mostraremos seguidamente tanto en mayúscula
                      /* en minúsculas */
                      do {
                              printf("Introduzca palabra: ");
                              scanf("%s", word);

                              /* ahora, hacemos las llamadas a los objetos CORBA */
                              up = MyModule_String_toUpper(corba_string, word, &ev);
                              if (ev._major != CORBA_NO_EXCEPTION) {
                                      printf("se ha producido un error en la llamada
                                      break;
                              }

                              lw = MyModule_String_toLower(corba_string, word, &ev);
                              if (ev._major != CORBA_NO_EXCEPTION) {
                                      printf("se ha producido un error en la llamada
                                      break;
                              }

                              printf("Tu palabra '%s' es '%s' en mayúsculas y '%s' en
                                     word, up, lw);
                      } while (strlen(word) > 0);
```

```
                        /* desactivamos el objeto antes de terminar */
                        CORBA_Object_release(corba_string, &ev);
                }
                return 0;
}
```

Como se puede apreciar en este ejemplo, usamos una serie de tipos de datos y funciones con unos nombres similares a los que usamos en el fichero IDL. Â¿Qué es esto? Â¿nos han plagiado nuestro interfaz antes de implementarlo siquiera? No, simplemente, son nombres de funciones, estructuras, etc que están declarados (e implementadas, en el caso de las funciones) en los ficheros generados por el compilador IDL en el punto anterior. Si observamos con detalle, podremos apreciar que estos nombres siguen unas reglas fácilmente visibles, que consiste en incluir el nombre del módulo y del interfaz en todos los identificadores. Así, por ejemplo, los métodos `toUpper` y `toLower` que definíamos en el fichero IDL pasan a convertirse, en el caso del lenguaje C, en `MyModule_String_toUpper` y `MyModule_String_toLower`. Estas reglas de nomenclatura permiten evitar posibles conflictos de nombres con otros proyectos, aunque, a pesar de todas las precauciones, es más que probable que, si no usamos un espacio de nombres para nuestros módulos que realmente sea único, entremos tarde o temprano en conflicto con algun otro proyecto. Así, la dirección del proyecto GNOME decidió en su día usar GNOME como módulo de más alto nivel, y dentro de él, que cada aplicación usara su propio módulo (pues en CORBA es posible definir módulos dentro de otros módulos. Así, lo correcto para el ejemplo que estamos mostrando en este artículo, sería usar el siguiente IDL:

```
module GNOME {
                module MyModule {
                        interface String {
                                string toUpper (in string str);
                                string toLower (in string str);
                        };
                };
};
```

Como vemos, la única diferencia con respecto al IDL que escribimos al principio es que en este último, hemos incluido nuestro módulo dentro de un módulo de más alto nivel, GNOME. Por supuesto, en aras de la sencillez de los ejemplos, usaremos el primer IDL, aunque la única diferencia es que los elementos generados por el compilador IDL usarían el prefijo `GNOME_MyModule_` en vez de `MyModule_`.

Por último, sólo nos queda compilar el cliente. Para ello, usaremos el siguiente comando:

```
gcc -o client `pkg-config --cflags --libs ORBit-2.0` client.c String-common.c String
```

Como vemos, tenemos que incluir dos de los ficheros generados por `orbit-idl` en nuestra fase de compilación/enlazado.

## Implementación de nuestro objeto

Bien, pues ya en el último paso de este pequeño tutorial sobre el uso de ORBit, nos queda lo más complicado, que es todo lo relativo a la implementación real del objeto CORBA que definimos en nuestro fichero IDL. Para ello, lo mejor es que abramos, con nuestro editor preferido, uno de los ficheros generados por el compilador IDL en los pasos anteriores. En el que estamos interesados en este caso es `String-skelimpl.c`

que, como comentábamos anteriormente, contiene la implementación base del objero definido en el fichero IDL, y que, como tambien comentábamos, simplemente tenemos que rellenar.

Nada más ver el contenido del fichero, es probable que el lector pase al siguiente artículo de la revista, pues hay que reconocer que no es un código muy legible que digamos. Pero no desesperemos, en este caso, simplemente estamos interesados en dos funciones, que podremos encontrar justo al final de dicho fichero, como puede observarse en la figura 1.

Ambas funciones tienen el mismo aspecto, con una variable de tipo `CORBA_char` declarada al principio de la función, y una línea que devuelve el contenido de dicha variable como resultado de la función. Como vemos, este código, lo primero, es que no hace absolutamente nada, y lo segundo, que puede que nuestro programa termine con un error grave (acceso a ilegal a memoria) si tratamos de usar las funciones en su estado actual, pues estamos devolviendo un puntero que no ha sido inicializado. Pero todo tiene una razón, y la razón para este código tan estúpido es que es simplemente un esqueleto de las funciones que tenemos que implementar generado amablemente por `orbit-idl`.

Así, lo que vamos a hacer, es implementar dichas funciones:

```
static CORBA_char *
impl_MyModule_String_toUpper(impl_POA_MyModule_String * servant,
                             CORBA_char * str, CORBA_Environment * ev)
{
             int cnt;
             CORBA_char *retval = CORBA_string_dup(str);

             for (cnt = 0; cnt < strlen(retval); cnt++)
                     retval[cnt] = toupper(retval[cnt]);
             return retval;
}

static CORBA_char *
impl_MyModule_String_toLower(impl_POA_MyModule_String * servant,
                             CORBA_char * str, CORBA_Environment * ev)
{
             int cnt;
             CORBA_char *retval = CORBA_string_dup(str);

             for (cnt = 0; cnt < strlen(retval); cnt++)
                     retval[cnt] = tolower(retval[cnt]);
             return retval;
}
```

No necesitan mucha explicación estas funciones, que simplemente convierten una cadena a mayúsculas o minúsculas. La única diferencia es que, en el caso de CORBA, a la hora de asignar memoria para cadenas y otros tipos (como estructuras, secuencias), tendremos que usar una serie de funciones específicas en vez de usar las que usaríamos en un programa "normal" de C. Por ejemplo, en este caso, en vez de usar `strdup` (o `g_strdup`), usamos `CORBA_string_dup`.

Si observamos el contenido de todos los ficheros generados por el compilador IDL, veremos que no hay ninguna función `main`. Por tanto, el último paso en la implementación de nuestro objeto será escribir dicha función, en la que inicializaremos todo lo necesario para activar nuestro objeto. Esta función, lo normal sería incluirla en un fichero aparte, pero tambien podemos añadirla al fichero `String-skelimpl.c`. Así, éste sería su aspecto:

```
int main (int argc, char argv[])
{
```

```
                CORBA_Environment ev;
                CORBA_ORB orb;
                CORBA_char *objref;
                PortableServer_POA root_poa;
                MyModule_String corba_string;
                PortableServer_POAManager pm;

                CORBA_exception_init(&ev);
                orb = bonobo_activation_init(argc, argv);

                /* obtenemos referencia al "RootPOA" */
                root_poa = (PortableServer_POA)
                        CORBA_ORB_resolve_initial_references(orb, "RootPOA", &ev);

                /* creamos instancia de nuestro objeto */
                corba_string = impl_MyModule_String__create(root_poa, &ev);
                objref = CORBA_ORB_object_to_string(orb, corba_string, &ev);
                /* registramos el objeto en Bonobo-Activation */
                if (bonobo_activation_register_active_server ("OAFIID:MyModule_String",
                    != Bonobo_ACTIVATION_REG_SUCCESS) {
                        printf("No se pudo registrar el objeto en Bonobo\n");
                        return -1;
                }

                /* ejecutamos el objeto */
                pm = PortableServer_POA__get_the_POAManager(root_poa, &ev);
                PortableServer_POAManager_activate(pm, &ev);

                CORBA_ORB_run(orb, &ev);

                /* cuando el programa salga de CORBA_ORB_run, es hora de terminar */
                CORBA_ORB_shutdown(orb, TRUE, &ev);
                bonobo_activation_unregister_active_server ("OAFIID:MyModule_String", c

                return 0;
}
```

Aquí ya vemos que las cosas se complican, aunque esto no debe preocuparnos, pues la mayor parte de los pasos son siempre iguales, por los que no es ni siquiera necesario que entendamos lo que hacen (de hecho, su explicación la reservamos para otro posible artículo sobre uso más avanzado de CORBA). Simplemente tenemos que preocuparnos de la llamada a la función `impl_MyModule_String__create` y al identificador de implementación OAF (OAFIID) usado en las funciones `bonobo_activation_register_active_server` y `bonobo_activation_unregister_active_server`. En la primera función, `impl_MyModule_String__create`, generada por el compilador IDL, creamos una instancia de nuestro objeto. El valor que devuelve es una referencia al objeto creado. Con este valor (*corba_string*) hacemos las llamadas a Bonobo-Activation, para indicarle que queremos registrar un objeto ya activo. De esta forma, Bonobo tiene constancia de la existencia de nuestro objeto, algo que es esencial para que otra aplicación pueda acceder a él.

El resto del código realiza todas las inicializaciones y tareas necesarias para la ejecución del ORB (ORBit en este caso). En la mayor parte de los casos, simplemente nos limitaremos a copiar y pegar este código entre proyectos, aunque recomendamos encarecidamente al lector que intente por su cuenta aprender qué hace ese código, pues conocer CORBA/ORBit a fondo ofrece características realmente interesantes, especialmente el uso de POAs, que sin duda querremos aprovechar en nuestras aplicaciones si vamos a usar CORBA "de verdad".

Para compilar nuestro objeto, usaremos un comando muy similar al que usamos anteriormente para compilar el cliente. Lo único que cambia es la lista de ficheros a

compilar, que en el caso del servidor añade `String-skelimpl.c` y sustituye `String-stubs.c` por `String-skels.c`, que es el fichero adecuado, como explicábamos antes, para compilar la implementación de nuestro objeto. Así:

```
gcc -o server `pkg-config --cflags --libs ORBit-2.0` String-skelimpl.c String-common
```

## Ejecutando la aplicación

Llegados a este punto, sólo nos queda probar el código que hemos escrito y, si funciona y nos ha llamado la atención el uso de CORBA, comenzar a hacer pruebas uno mismo. Para lo primero, ejecutar la aplicación, lo primero que vamos a necesitar es un fichero `.server` (descritos en los artículos referentes a Bonobo publicados anteriormente); este fichero es necesario para que la activación de nuestro objeto mediante Bonobo se lleve a cabo satisfactoriamente. Seguidamente, lo mejor es que abramos dos terminales, y en uno de ellos ejecutemos el programa servidor (`server`) y, acto seguido, en el otro terminal, ejecutemos el cliente. Si todo ha ido bien, deberíamos ser capaces de enviar, a través del programa cliente, cadenas al servidor que nos son seguidamente mostradas tanto en mayúsculas como en minúsculas.

Para lo segundo, investigar por nuestra cuenta sobre el uso de ORBit/CORBA en GNOME, lo mejor es que visitemos las páginas especificadas en la sección de referencias, donde podremos obtener enlaces a otros lugares en Internet donde encontrar ejemplos y más documentación. Hay que destacar que, desgraciadamente, la documentación sobre todo esto es bastante escasa, y en algunos casos, la existente está bastante desactualizada. Por ello, recomendamos al lector que procure contrastar la documentación que lea con código fuente. Para obtener código fuente, sin duda el mejor sitio es el servidor CVS de GNOME.

# Capítulo 15. Activación de componentes

Una de las lagunas del estándar CORBA actual es la forma en la que se activan los objetos. Si bien con ORBit tenemos la posibilidad de "conectarnos" a un objeto ya inicializado (mediante su IOR), y si bien tenemos a nuestra disposición, en ORBit, el servicio de nombres CORBA, en el que los objetos se registran, queda una pequeña laguna en cuanto a activación de objetos no inicializados (no arrancadas, y no registrados en el servicio de nombres CORBA).

Por esta razón, y para cubrir esta pequeña laguna del estándar CORBA, el proyecto GNOME ha tenido siempre a disposición de los desarrolladores, un sistema para el registro, en tiempo de instalación, de los componentes, de forma que, más adelante, una aplicación pueda activar un componente sin necesidad de que éste esté ya registrado en el sistema.

## libgnorba

En un primer momento, la solución del proyecto GNOME para este problema fue `libgnorba`, que estaba compuesta por una librería muy fácil de usar que permitía tanto activar componentes como registrarlos en el servicio de nombres CORBA, y por un formato de ficheros, el cual se usaba para que los distintos componentes pudieran especificar distintos parámetros en tiempo de instalación, de forma que esos componentes quedaran disponibles para las aplicaciones que usaran `libgnorba` para la activación de componentes. Sin embargo, a pesar de su sencillez y su reducido tamaño, `libgnorba` contenía algunas limitaciones que aconsejaban su sustitución. Entre estas limitaciones, la principal era su fuerte dependencia del sistema X Window, de forma que no se podía usar desde aplicaciones sin interfaz gráfico de usuario. Tambien pesaba mucho su incapacidad de ampliación, de forma que, por ejemplo, no se podían añadir propiedades personalizadas a los componentes, simplemente había un conjunto de propiedades, y eso era todo.

Con todo esto en mente, el proyecto GNOME decidió la sustitución de `libgnorba` por `OAF`, que, a partir de GNOME 2.0 pasó a llamarse `bonobo-activation`.

## Instalación de componentes en el sistema

Todas las aplicaciones que incluyen componentes instalan, en el directorio `$prefix/lib/bonobo/servers` (donde $prefix es el prefijo de instalación de GNOME), una serie de ficheros, con extensión `.server`, que tienen un cometido muy parecido al de los ficheros `.gnorba` en `libgnorba`. La diferencia es que en `bonobo-activation` se han resuelto los problemas de extensibilidad que tenía `libgnorba`.

bonobo-activation mantiene un área dentro del sistema de ficheros, en la que se almacenan unos ficheros en los que se especifican las propiedades de cada componente instalado. Este área suele estar en *`$prefix/lib/bonobo/servers`*, donde *`$prefix`* es el prefijo de instalación de bonobo-activation, que puede saberse mediante la ejecución del siguiente comando:

```
pkg-config --variable=prefix bonobo-activation-2.0
```

Si tenemos bonobo-activation bien instalado, este comando nos mostrará por pantalla el directorio usado para la instalación de bonobo-activation. En mi caso:

```
$ pkg-config --variable=prefix bonobo-activation-2.0
/gnome/head/INSTALL
```

En ese directorio (en mi caso, /gnome/head/INSTALL/lib/bonobo/servers) se almacenan unos ficheros con extensión `.server`. Antes de seguir con la explicación, veamos un ejemplo de fichero `.server`, que necesitamos instalar en ese directorio para que bonobo-activation/Bonobo sepan que hemos instalado un nuevo componente.

```
<oaf_info>
<oaf_server iid="OAFIID:GNOME_MiCalculadora_ControlFactory"
type="exe"
location="control-calculadora">
<oaf_attribute name="repo_ids" type="stringv">
<item value="IDL:GNOME/GenericFactory:1.0">
</oaf_attribute>
<oaf_attribute name="description" type="string"
value="Factoría para crear controles MiCalculadora"/>
</oaf_server>

<oaf_server iid="OAFIID:GNOME_MiCalculadora"
type="factory"
location="OAFIID:GNOME_MiCalculadora_ControlFactory">
<oaf_attribute name="repo_ids" type="stringv">
<item value="IDL:Bonobo/Control:1.0"/>
<item value="IDL:Bonobo/Unknown:1.0"/>
</oaf_attribute>
<oaf_attribute name="description" type="string"
value="Control MiCalculadora"/>
</oaf_server>
</oaf_info>
```

Como se puede observar, este fichero usa XML como formato, y vemos que, dentro del nodo principal del fichero (<oaf_info>) hay dos subnodos, ambos de tipo `oaf_server`. Estos subnodos especifican los servidores (componentes) que queremos exportar, que en este caso son dos: el componente `MiCalculadora` y la factoría que permite crear objetos de tipo `MiCalculadora`.

Seguidamente, podemos observar que para cada servidor se especifican una serie de propiedades que especifican el tipo de componente y la localización del mismo (para la factoría es un simple ejecutable, pero para el otro componente, vemos que se especifica *"factory"* como tipo, y como localización, el `OAFIID` de la factoría, lo que hace que Bonobo sepa qué tiene que hacer para crear un componente `MiCalculadora`: cargar la factoría y pedirle que cree una nueva instancia del componente `MiCalculadora`).

Seguidamente aparecen los `oaf_attribute`, que son uno o más atributos que queremos darle al componente. El más importante de todos es *"repo_ids"*, que especifica la lista de interfaces CORBA IDL implementados por el componente en cuestión. Así, vemos que nuestro componente factoría implementa el interfaz *"IDL:GNOME/GenericFactory:1.0"* y el componente `MiCalculadora` implementa tanto *"IDL:Bonobo/Control:1.0"* como *"IDL:GNOME:Bonobo/Unknown:1.0"* (ver primer artículo de la serie para saber más acerca de Bonobo/Unknown). Este campo es muy importante, pues es el que usa OAF para saber si un determinado componente implementa o no un determinado interfaz. Así que es una buena idea añadir a la lista de *"repo_ids"* todos y cada uno de los interfaces que nuestro componente implementa, empezando, por supuesto, por *IDL:GNOME:Bonobo/Unknown:1.0*.

El resto de `oaf_attribute` no son obligatorios y suelen usarse como medio informativo, como es el caso del atributo *"description"*, que indica una descripción de lo que hace nuestro componente. Este campo es muy útil, por ejemplo, para una ventana en la que se le pida al usuario qué componente quiere cargar: en vez de mostrar el `OAFIID`, podríamos mostrar la descripción del componente. Esto es precisamente lo que hace el objeto `BonoboSelectorWidget`, que ya implementa todo lo necesario

para preguntar a OAF qué componentes hay instalados, y mostrar toda la información sobre ellos en un bonito *"widget"*. Y Bonobo nos facilita aún más las cosas al incluir el `BonoboSelectorDialog`, que a su vez usa el `BonoboSelectorWidget`. Para usarlo:

```
      const gchar* interfaces[] = { "IDL:Bonobo/Control:1.0", NULL };
      gchar* seleccionado = bonobo_selector_select_id("Selecciona control", interfaces)
      if (seleccionado) {
        GtkWidget* control = bonobo_widget_new_control(seleccionado, NULL);
 gtk_container_add(GTK_CONTAINER(ventana), control);
      }
```

Este código abriría una ventana en la que aparecería un listado de todos los controles Bonobo (debido a que hemos especificado sólo `"IDL:Bonobo/Control:1.0"`) instalados en el sistema, y, según lo que devuelva la función `bonobo_selector_select_id` (NULL si el usuario no seleccionó ningún componente, o una cadena que contiene el nombre del componente seleccionado si el usuario así lo hizo), carga el componente en una ventana o no.

**Figura 15-1. El selector de componentes de Bonobo**

# Activación de Componentes

## El lenguaje de consulta de OAF

Cada componente instalado en el sistema tiene un identificador único, que permite encontrar y activar

# Capítulo 16. Bonobo

Dentro del papel estelar de CORBA en la arquitectura de GNOME, uno de los desarrollos más interesantes que se han hecho dentro del proyecto GNOME es, sin duda, Bonobo, el sistema de componentes.

Bonobo significa, para los desarrolladores originales del proyecto GNOME, la consecución del objetivo inicial del proyecto, que era crear un sistema de componentes para sistemas UNIX/Linux, de forma que una tecnología disponible desde hacía tiempo para otros sistemas, lo estuviera tambien para todos los usuarios de estos sistemas. De hecho, GNOME, la palabra, es un acrónimo, que significa *GNU Network Object Model Environment* (Entorno de Modelo de Objetos en Red, más o menos), lo cual demuestra claramente qué esa intención de crear un sistema de componentes (modelo de objetos) era totalmente cierta.

Bonobo, además, significa el tener disponible, para todas las aplicaciones GNOME, una arquitectura para el desarrollo de aplicaciones realmente potente, que permite llevar, a nuestros queridos sistemas *IX, el tipo de aplicaciones modernas que pueden verse en otros sistemas operativos.

Como anécdota contar que, siguiendo la nomenclatura basada en nombres de primates que se usa para nombrar algunas partes del proyecto GNOME (GNOME mismamente), el primer nombre que recibió el sistema de componentes de GNOME fue *Baboon*, que es el nombre, (en inglés) de una especie de monos. Pero luego se pensó que este nombre no era del todo correcto, y se cambió a Bonobo, que es el nombre de otra especie de chimpancés (que, por cierto, está en peligro de extinción) que practican el sexo varias veces al día, y por tanto, su nombre se ajusta más al de un sistema de componentes, en el que unos componentes se *"incrustan"* dentro de otros.

## Â¿Qué es Bonobo?

Como hemos dicho antes, Bonobo es un sistema de componentes. Y, Â¿qué es un sistema de componentes? La respuesta fácil es: un sistema de componentes es una arquitectura que permite la comunicación, tanto visual como no visual, entre aplicaciones, así como la reutilización de software. Esto significa que, mediante el uso de Bonobo, podemos, por ejemplo, insertar en una aplicación la parte visual de otra aplicación, o acceder directamente a las interioridades de otra aplicación, y hacerla funcionar como si se tratara de una extensión de la primera aplicación, o más aún, podemos insertar objetos de otra aplicación en nuestros documentos, algo que se conoce como *"documentos compuestos"*". Esto es algo que los usuarios de MS Windows conocen perfectamente, pues es lo que hacen cuando insertan, por ejemplo, una hoja de cálculo realizada con el MS Excel en un documento creado con MS Word.

Este sistema abre un campo muy amplio de posibilidades, sobre todo a la hora de desarrollar aplicaciones con interfaz gráfico, pues permite, desde el punto de vista de los desarrolladores, la reutilización de software en el más amplio sentido de la palabra, y, desde el punto de vista de los usuarios, una integración total de todas las aplicaciones que componen el escritorio. Además, permite el desarrollo de aplicaciones a gran escala, pues se pueden reutilizar partes ya existentes, pudiendo los desarrolladores centrarse en añadir nuevas funcionalidades a componentes personalizados.

## CORBA y Bonobo

Bonobo, como no podía ser menos, está basado en CORBA, la tecnología estrella del proyecto GNOME. Eso quiere decir que Bonobo define un conjunto de interfaces CORBA IDL para la consecución de los siguientes objetivos:

- la creación de componentes, tanto visuales como no visuales

- la creación de documentos compuestos, donde una aplicación se "inserta" dentro de otra para permitir la edición de una parte determinada de un documento compuesto

El uso de CORBA en Bonobo no se ha escogido única y exclusivamente por ser la tecnología "preferida" en GNOME. El usar CORBA para la implementación de Bonobo trae muchas ventajas, entre las que destacan:

- Todas las características de un componente pueden ser definidas, fácilmente, por medio del lenguaje de definición de interfaces de CORBA, el IDL.
-

La implementación de Bonobo está fuertemente inspirada en la implementación de OLE2, con la diferencia de que éste último está basado en COM, un sistema propietario de Microsoft que, evidentemente, sólo funciona en entornos Windows, mientras que Bonobo está basado en CORBA, hecho este que le abre un mundo mucho más grande, pues CORBA es un estándar avalado por la plana mayor de la industria informática. El estar basado en CORBA va a permitir que se intercambien componentes entre aplicaciones ejecutándose en distintas máquinas, plataformas o sistemas operativos. Otro de los puntos fuertes inherentes en el uso de CORBA como base de la arquitectura, es que, al estar todo definido en interfaces IDL de CORBA, es totalmente posible el hacer una implementación de Bonobo para otro sistema operativo/plataforma.

Bonobo propiamente dicho simplemente consiste en una serie de interfaces CORBA, por lo que en este artículo nos vamos a centrar en la implementación de Bonobo para el proyecto GNOME, que usa GTK+ y X Window. Pero los desarrolladores de Bonobo han puesto un cuidado especial en hacer Bonobo totalmente independiente del toolkit (GTK+) usado, por lo que sería perfectamente posible el realizar una implementación de Bonobo, por ejemplo, para QT, el *toolkit* usado por el proyecto KDE.

Bonobo se divide en distintos tipos de interfaces IDL, cada uno de ellos enfocado a un uso específico. Se pueden crear componentes Bonobo de tres tipos:

- Componentes
- Controles
- *"Embeddables"* (o empotrables)

Los primeros, los componentes, son simples objetos CORBA que implementan la interfaz `Bonobo::Unknown`, que es en la que están basados todos los componentes, controles y empotrables. Es una interfaz CORBA muy sencilla:

```
module Bonobo {
  interface Unknown {
    void ref ();
    void unref ();
    Unknown queryInterface (in string repoid);
  }
}
```

Como puede verse, este interfaz es tan básico que, por si solo, sirve para más bien poco. Realmente, su única utilidad es para que todos los demás tipos de componentes Bonobo estén basados en este interfaz, que lo que implementa es el control de vida de los objetos (los métodos `ref()` y `unref()`) y la posibilidad de *"preguntar"* si un componente implementa un interfaz determinado. Así, por ejemplo, si tuviéramos un componente ya creado, podríamos hacer lo siguiente (NOTA: este código es inventado, no existen esos objetos/funciones en Bonobo):

```
WordProcessor wp = word_processor_new();
```

```
Application app = wp->queryInterface("IDL:Office/Application");
if (app) {
  app.close();
}
```

En este ejemplo, estamos creando un componente de tipo `WordProcessor`, al que, mediante el método `queryInterface` estamos preguntando si, además de este tipo de objeto, implementa el interfaz `Office::Application`. Esto nos sirve para tratar a un mismo objeto bajo distintas personalidades (= interfaces IDL implementados). Esta idea es muy parecida a lo que es la herencia en lenguajes orientados a objetos, como C++, Java, etc.

Este tipo de componentes pueden ser muy útiles para añadir servicios no visuales a nuestras aplicaciones, como por ejemplo podría ser un servidor de red compartido por varios usuarios, un interfaz encapsulando acceso a la configuración del sistema, etc. De todas formas, como hemos comentado anteriormente, TODOS, absolutamente TODOS los objetos Bonobo implementan este interfaz, por lo que, aún pareciendo un interfaz sin ningún sentido, es la base para la implementación del resto de componentes.

## Implementación de interfaces CORBA con Bonobo

Una de las cosas en las que más nos va a ayudar Bonobo es, sin duda, en la facilidad que nos da para la implementación de interfaces CORBA. Como se pudo comprobar en el capítulo dedicado a ORBit, la lista de cosas a hacer, si bien no muy complicada, hace que el acceso al uso de CORBA se limite un poco más. Por ello, en la implementación de Bonobo para el proyecto GNOME se ha puesto un cuidado especial en ocultar todo el código generado por el compilador IDL, de forma que su uso sea mucho más sencillo. Así, Bonobo, aparte de los interfaces CORBA IDL anteriormente comentados, incluye multitud de funciones que nos van a ayudar enormemente en el uso de CORBA en nuestras aplicaciones, y todo ello probablemente sin enterarnos siquiera de que lo que estamos usando, por debajo, es CORBA.

Una de las cosas más útiles que tenemos en Bonobo es el objeto BonoboObject (o BonoboXObject), que nos ayuda enormemente a la hora de implementar nuestros propios interfaces IDL, eso si, basados en Bonobo::Unknown.

El uso de BonoboObject (o BonoboXObject) es muy sencillo. Para empezar, lo primero que tenemos que hacer es definir, en un fichero IDL, nuestros interfaces. Como hemos comentado, estos interfaces estaran basados en Bonobo::Unknown, de forma que, sin ningun esfuerzo añadido, nuestros objetos CORBA se conviertan en componentes Bonobo.

Así, definiriamos el siguiente interfaz:

```
#include <Bonobo.idl>
module GNOME {
  module Libro {
    interface String : Bonobo::Unknown {
      string toUpper (in string str);
      string toLower (in string str);
    };
  };
};
```

Como se puede observar, estamos definiendo un interfaz (GNOME::Libro::String), que deriva de Bonobo::Unknown, y que contiene dos sencillos métodos para convertir cadenas a mayúsculas (`toUpper`) y a minúsculas (`toLower`).

## Implementación del componente con BonoboObject

Una vez que hemos definido nuestro(s) interfaz(es) IDL, llega el momento de pasar a la implementación. Para ello, usaremos el sistema de objetos de `GLib`, de forma que crearemos una nueva clase, que llamaremos LibroString. Esta clase estará basada en BonoboObject, de forma que todo lo necesario para que nuestro interfaz implemente el interfaz CORBA Bonobo::Unknown sea añadido a nuestra nueva clase.

Así pues, crearemos primero el fichero de cabecera para nuestra clase, que llamaremos `libro-string.h`. Este fichero contendrá todas las declaraciones típicas de un fichero de cabecera para un objeto GObject, con algunas pequeñas diferencias, que comentaremos a continuación.

En primer lugar, una de las diferencias es que tendremos que incluir, en este fichero de cabecera, el fichero de cabecera generado por el compilador `orbit-idl` a partir de nuestro fichero IDL comentado anteriormente. Así, al principio de este fichero, junto con el resto de #include's, añadiremos:

```
#include <GNOME_Libro.h>
```

En la definición de los tipos para las instancias y clases de nuestro nuevo objeto, haremos que éste derive de BonoboObject. En la estructura de clase, además, añadiremos algo nuevo:

```
struct _LibroString {
  BonoboObject object;
};

struct _LibroStringClass {
  BonoboObjectClass parent_class;
  POA_GNOME_Libro_String__epv epv;
};
```

Como se puede observar, hemos añadido la estructura epv (comentada en el capítulo sobre CORBA) a la estructura de clase de nuestro objeto LibroString. Esto se debe a que BonoboObject necesita que la epv esté disponible en la estructura de clase, de forma que pueda inicializar el objeto CORBA.

El siguiente paso es escribir la implementación de nuestro objeto, lo cual haremos en el fichero `libro-string.c`. Aquí es donde realmente notaremos una enorme diferencia con respecto a la implementación de objetos CORBA comentada en el capítulo sobre CORBA/ORBit, donde el código que tenemos, más o menos, que mantener/escribir, es realmente extenso, lo cual, por supuesto, nos hace más propensos a posibles errores. Gracias a Bonobo en general, y a BonoboObject en particular, todo esto se facilita enormemente.

Para empezar, como cualquier clase nueva que creamos, tenemos que implementar la función `_get_type` para dicha clase. En el caso de clases derivadas de BonoboObject, la forma de registrar una nueva clase es algo distinta, pero, de lejos, mucho más sencilla. Sólo tenemos que usar una macro (definida con #define) disponible en el fichero `bonobo-object.h`:

```
#define BONOBO_TYPE_FUNC_FULL(class_name, corba_name, parent, prefix)
```

Esta macro, como podremos observar si editamos el fichero `bonobo-object.h`, simplemente genera una función típica `_get_type` usando los parámetros que le pasamos para generar los nombres de las distintas estructuras, funciones, etc. Así, en el caso de nuestro interfaz GNOME::Libro::String, quedaría de la siguiente manera:

```
BONOBO_TYPE_FUNC_FULL (LibroString,
                       GNOME_Libro_String,
```

```
                        BONOBO_TYPE_OBJECT,
                        libro_string)
```

Incluyendo esto en nuestro fichero `libro-string.c`, tendriamos ya creada la función `libro_string_get_type`, que será la encargada de registrar la nueva clase cuando así sea requerido.

La siguiente diferencia con respecto a una clase "normal" reside en la implementación de la función de inicialización de la clase (`libro_string_class_init` en este caso), en la que, como comentábamos anteriormente, tendremos que inicialiar la estructura epv de nuestra clase:

```
static void
libro_string_class_init (LibroStringClass *klass)
{
  POA_GNOME_Libro_String__epv *epv;

  epv = &klass->epv;
  epv->toUpper = impl_String_toUpper;
  epv->toLower = impl_String_toLower;
}
```

BonoboObject se encarga de todo lo necesario para inicializar nuestro objeto CORBA, nosotros sólo tenemos que indicarle la única y exclusivamente la información que necesita para ello. En esta función, lo que estamos haciendo es especificar dónde está la implementación de los métodos de nuestro objeto (`toUpper, toLower`), para lo que usamos la estructura epv generada por el compilador `orbit-idl`, en la que se especifican los punteros a las funciones que implementan los distintos métodos de los objetos CORBA. Con esto, añadiendo además las demás funciones típicas de una clase (`libro_string_init`, `libro_string_new`, etc), tenemos ya todo lo necesario para nuestro objeto, basado en Bonobo::Unknown. Sólo nos falta añadir la implementación de los métodos. En ellos, que tienen la misma forma que los generados por `orbit-idl`, usaremos la función `bonobo_object`, que nos devuelve una referencia a nuestro BonoboObject dado un objeto CORBA:

```
/* Detects the pointer type and returns the object reference - magic. */
BonoboObject *bonobo_object (gpointer p);
```

Como dice el comentario (sacado del fichero `bonobo-object.h`), esta función lo que hace es magia, pues mediante aritmética de punteros y demás artes esotéricas, accede, dado un objeto CORBA (CORBA_Object) al objeto BonoboObject asociado. Por tanto, nuestros métodos quedaran de la siguiente manera:

```
static CORBA_char *
impl_String_toUpper (PortableServer_Servant servant,
                     const CORBA_char *str,
                     CORBA_Environment *ev)
{
  gint n;
  CORBA_char *corba_str;
  LibroString *libstr = bonobo_object (servant);

  bonobo_return_val_if_fail (LIBRO_IS_STRING (libstr), NULL, ev);

  corba_str = CORBA_string_dup (str);
  for (n = 0; n < strlen (corba_str); n++) {
    corba_str[n] = toupper (corba_str[n]);
  }

  return corba_str;
}
```

```
static CORBA_char *
impl_String_toLower (PortableServer_Servant servant,
                     const CORBA_char *str,
                     CORBA_Environment *ev)
{
  gint n;
  CORBA_char *corba_str;
  LibroString *libstr = bonobo_object (servant);

  bonobo_return_val_if_fail (LIBRO_IS_STRING (libstr), NULL, ev);

  corba_str = CORBA_string_dup (str);
  for (n = 0; n < strlen (corba_str); n++) {
    corba_str[n] = tolower (corba_str[n]);
  }

  return corba_str;
}
```

## Factorías de componentes

Una de las cosas que más se usan en el desarrollo de aplicaciones basadas en componentes son las factorías de objetos, que son objetos CORBA que permiten la creación de otros objetos CORBA. Esta es una práctica muy comun en GNOME, donde normalmente los componentes estan implementados en distintos ejecutables. El uso de factorías permite que se pueda usar un solo ejecutable para la creación de distintos componentes. Si no se usaran factorías, probablemente, se arrancaría un proceso distinto por cada componente, algo que, evidentemente, haría que nuestro sistema se cargara innecesariamente a medida que fueramos creando más y más componentes.

El uso de factorías tambien simplifica enormemente la creación de componentes. Para ayudarnos en la creación y uso de factorías, Bonobo incluye dos clases realmente fáciles de usar: BonoboGenericFactory y BonoboShlibFactory. La primera, se usa para la creación de factorías en binarios independientes, mientras que la segunda se usa para crear factorías en librerías compartidas. En cualquiera de los dos casos, Bonobo nos provee de otras dos macros que nos permiten crear una factoría de forma muy sencilla.

## Controles Bonobo

Este tipo de componentes, los controles, son los más usados, pues, aparte de ser de los más fáciles de implementar, ofrecen lo que antes comentábamos: importar la interfaz gráfica de una aplicación en otra. Esto puede observarse en la siguiente captura de pantalla,

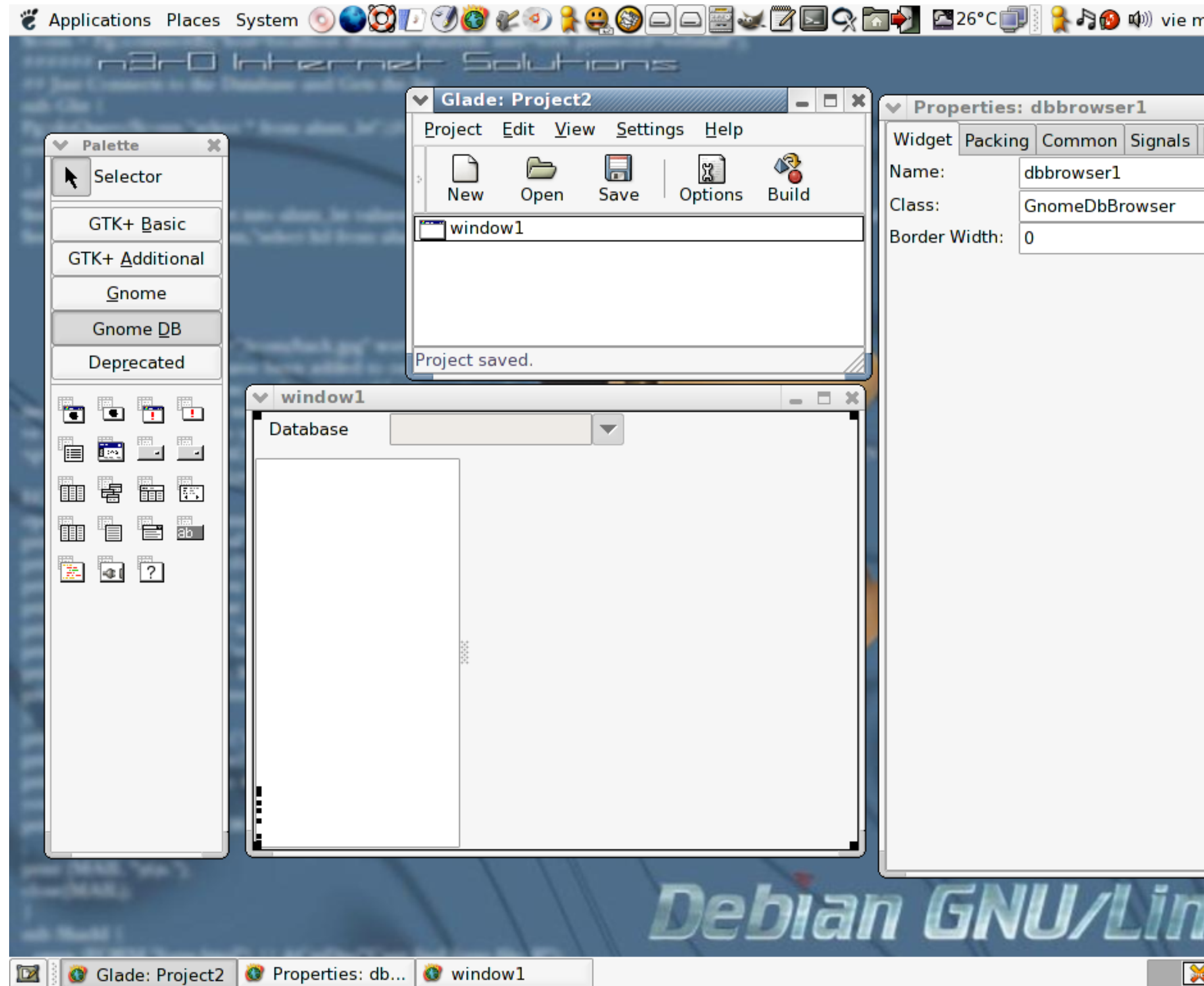



**Figura 16-1. GnomeDbBrowser en Glade**

donde aparece Glade, el programa de diseño de interfaces gráficas del proyecto GNOME, con un ventana en la que se ha insertado el componente `DBBrowser` de GNOME-DB, que aparece, esta vez dentro de la aplicación `gnomedb-fe` (parte de GNOME-DB), en la siguiente captura de pantalla:

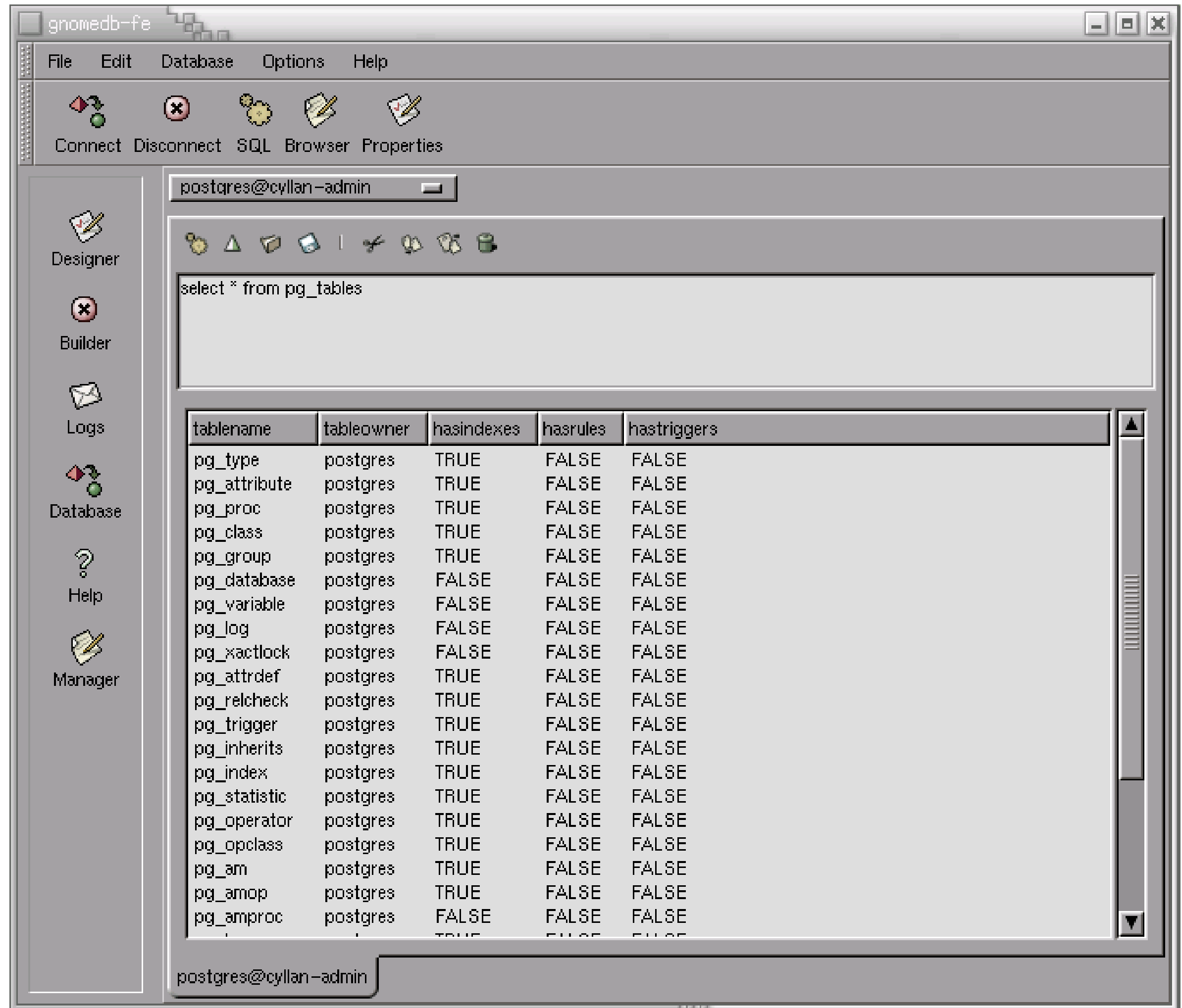



**Figura 16-2. GnomeDbBrowser en el front-end de GNOME-DB**

Como puede verse en este ejemplo *"visual"*, la interfaz gráfica de un programa es exportada para que cualquier otra aplicación Bonobo pueda fácilmente incluirla. Como se comentaba anteriormente, esto (exportar la interfaz gráfica) es algo que las aplicaciones GNOME de 2Âª generación están empezando a hacer, lo que va a posibilitar el desarrollo de aplicaciones a gran escala, pues el tener un sistema de componentes potente como es Bonobo va a permitir a los desarrolladores el centrarse en la implementación de sus componentes espcíficos, que luego interactuarán a las mil maravillas con el resto de aplicaciones Bonobo.

Ejemplos de uso de estos controles podrían ser, por ejemplo, un navegador de internet, que exporte su visualizador HTML, que luego podría ser incluido en cualquier aplicación, dotándola así de acceso a Internet con sólo unas pocas líneas de código:

```
BonoboWidget* browser;
Gtkwidget* window;

window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
browser = bonobo_widget_new_control("OAFIID:GNOME_Web_Browser");
gtk_container_add(GTK_CONTAINER(window), browser);
gtk_widget_show_all(window);
```

Podriamos definir los controles como *"widgets"* convertidos en componentes, es decir, que son simples *"widgets"* que un determinado proceso exporta para ser reutilizados por otras aplicaciones Bonobo. Esta idea puede parecer estúpida, pero no lo es, pues esto nos abre unas posibilidades muy grandes, a la hora de hacer aplicaciones de gran envergadura. Un buen ejemplo es Evolution, el gestor de correo y herramienta de trabajo en grupo que está siendo desarrollado por Ximian: la ventana principal de este programa es un simple contenedor de componentes, es decir, que no tiene mucha funcionalidad por si solo, sino que la funcionalidad la aportan los distintos componentes que se cargan en ella. Esta arquitectura permite, sin ninguna dificultad, fácilmente extender `Evolution` añadiéndole nuevos componentes que se irán cargando. De hecho, se comentó hace poco, en la lista de componentes del proyecto GNOME (ver referencias), que una empresa estaba desarrollando un programa para gestión de proyectos, y que su intención era, en un futuro, liberarlo como un componente Bonobo para que pudiera ser cargado en `Evolution`.

Y, aún más importante, este sistema nos permite reutilizar estos controles de `Evolution` en nuestras aplicaciones. Así, por ejemplo, podríamos tener una opción de menú en nuestro programa para permitirle al usuario el acceso a la libreta de direcciones de `Evolution`. En este caso, simplemente tendríamos que cargar en nuestra aplicación el control (`evolution-addressbook`) suministrado por `Evolution`.

Los controles son, quizás, los componentes más fáciles de implementar. Por ello, y por la utilidad que tienen y que ya hemos comentado, son los más usados. Así que, vamos a entrar ya de lleno en la implementación de aplicaciones Bonobo que usen controles.

## Uso de Controles - clientes

Aunque en Bonobo no existe la noción de cliente/servidor, sino más bien, de contenedor/componente, para hacerlo más sencillo, vamos a utilizar esta nomenclatura. En este caso, cliente sería la aplicación que carga los componentes, y servidores serían todos los procesos que exportan de alguna manera componentes.

Para los clientes, el código es bastante sencillo. Empezando por la inicialización de la aplicación, en la función `main`:

```
int main (int argc, char *argv[])
{
 CORBA_ORB orb;

 /* inicializamos GNOME */
 gnome_program_init ("contenedor", "0.1", argc, argv, oaf_popt_options, 0, NULL);

 /* inicializamos Bonobo/OAF */
 orb = bonobo_activation__init (argc, argv);
        if (bonobo_init(orb, NULL, NULL) == FALSE) {
  g_error("No se pudo inicializar Bonobo\n");
 }
 bonobo_activate();

 g_idle_add (crear_ventana, NULL);
        bonobo_ui_main();
 return 0;
}
```

Como puede observarse, las primeras líneas de la función `main` son las que inicializan todo el sistema, tanto las librerías GNOME, como OAF (el sistema de activación de objetos del proyecto GNOME) y Bonobo. Para la inicialización de GNOME es necesario llamar a la función `gnome_init_with_popt_table` en vez de `gnome_program_init`, debido a que, para el correcto funcionamiento de Bonobo-Activation, es necesario que se procesen los parámetros que pueda

recibir nuestro programa. Los parámetros son procesados por medio de la librería `popt` (usada internamente en las librerías de GNOME) y de la variable *`oaf_popt_options`*, definida en Bonobo-Activation, y que tenemos que usar para que los parámetros que se le puedan pasar a Bonobo-Activation a través de la línea de comandos puedan ser procesados. En el caso de los clientes (aplicación contenedora) este paso no es estrictamente necesario, pero se explica aquí para que luego este mismo código nos sirva para el servidor (implementación del componente).

Seguidamente, tras la inicialización de todas las librerías necesarias, el siguiente paso es activar Bonobo, que se hace mediante la llamada a `bonobo_activate`. Esta función activa internamente todo lo necesario para que Bonobo funcione. Tras esta llamada, Â¡ya estamos dentro del mundo de Bonobo!

El siguiente paso es crear nuestra ventana principal, en la que iremos cargando los componentes. Para ello, hay que tener en cuenta que, una vez activado Bonobo (con la llamada a `bonobo_activate`), no se puede hacer ninguna llamada a Bonobo/CORBA hasta que no estemos en el bucle principal de la aplicación, que se ejecuta en la llamada a la función `bonobo_main`. Así, para poder crear nuestra ventana una vez dentro del bucle de la aplicación, hacemos una llamada a la función de GTK `gtk_idle_add`, cuyo cometido es instalar una función que será llamada cuando el bucle del programa esté inactivo (*"idle"*). En nuestro caso, lo que hacemos es instalar la función `crear_ventana`, que tiene el siguiente aspecto:

```
guint crear_ventana (void)
{
 GtkWidget* ventana;
 GtkWidget* control;

 ventana = gtk_window_new(GTK_WINDOW_TOPLEVEL);
 g_signal_connect (G_OBJECT(ventana), "destroy", gtk_main_quit, NULL);

 /* cargamos el control */
 control = bonobo_widget_new_control ("OAFIID:GNOME_GtkHTML_Editor", NULL);
 if (!BONOBO_IS_WIDGET(control))
  g_error("No se pudo cargar el control Bonobo");
 gtk_container_add(GTK_CONTAINER(ventana), control);
 gtk_widget_show_all(ventana);

 return FALSE; /* para que no se llame más a esta función */
}
```

En esta función, lo primero que hacemos es crear una ventana, que será donde carguemos el control Bonobo. Esto se hace mediante la llamada a `gtk_window_new()`, para seguidamente conectar la función `gtk_main_quit` a la señal *"destroy"* de la ventana, lo que causará, como es de imaginar, que, cuando el usuario cierre la ventana, se termine el programa.

La siguiente línea ya forma parte de lo que realmente nos interesa: la carga del control Bonobo. Esto se hace mediante la función `bonobo_widget_new_control`, a la que tenemos que pasarle como parámetro el *`OAFIID`* (Identificador de la Implementación de un objeto OAF) del componente que queremos cargar. En este caso, lo que estamos cargando (`"OAFIID:GNOME_GtkHTML_Editor"`) es un editor HTML que viene incluido en la librería `gtkhtml`, que es necesaria para el correcto funcionamiento de `Evolution`, por lo que si tenemos instalado `Evolution`, seguro que tenemos este componente instalado en nuestro sistema.

Esta función, `bonobo_widget_new_control`, devuelve un puntero a un `BonoboWidget` que no es otra cosa que un simple `GtkWidget`, por lo que podemos tratarlo como tal e insertarlo en la ventana que hemos creado con las funciones que GTK incluye para ello. En este caso, puesto que un `GtkWindow` es también un `GtkContainer`, usamos la función `gtk_container_add` para insertar el componente Bonobo que hemos cargado en nuestra ventana. En este ejemplo lo hacemos de esta

manera, pues es la más sencilla para mostrar fácilmente la funcionalidad de Bonobo, pero evidentemente, podríamos insertar el componente Bonobo en cualquier `GtkWidget` que lo permita (`GtkContainer`, `GtkTable`, `GtkBox`, `GnomeDialog`, etc), y de hecho es lo que haremos cuando empecemos a hacer aplicaciones más sofisticadas y complejas.

En la siguiente captura de pantalla podemos observar nuestra aplicación contenedora en funcionamiento, con el componente de edición HTML cargado.

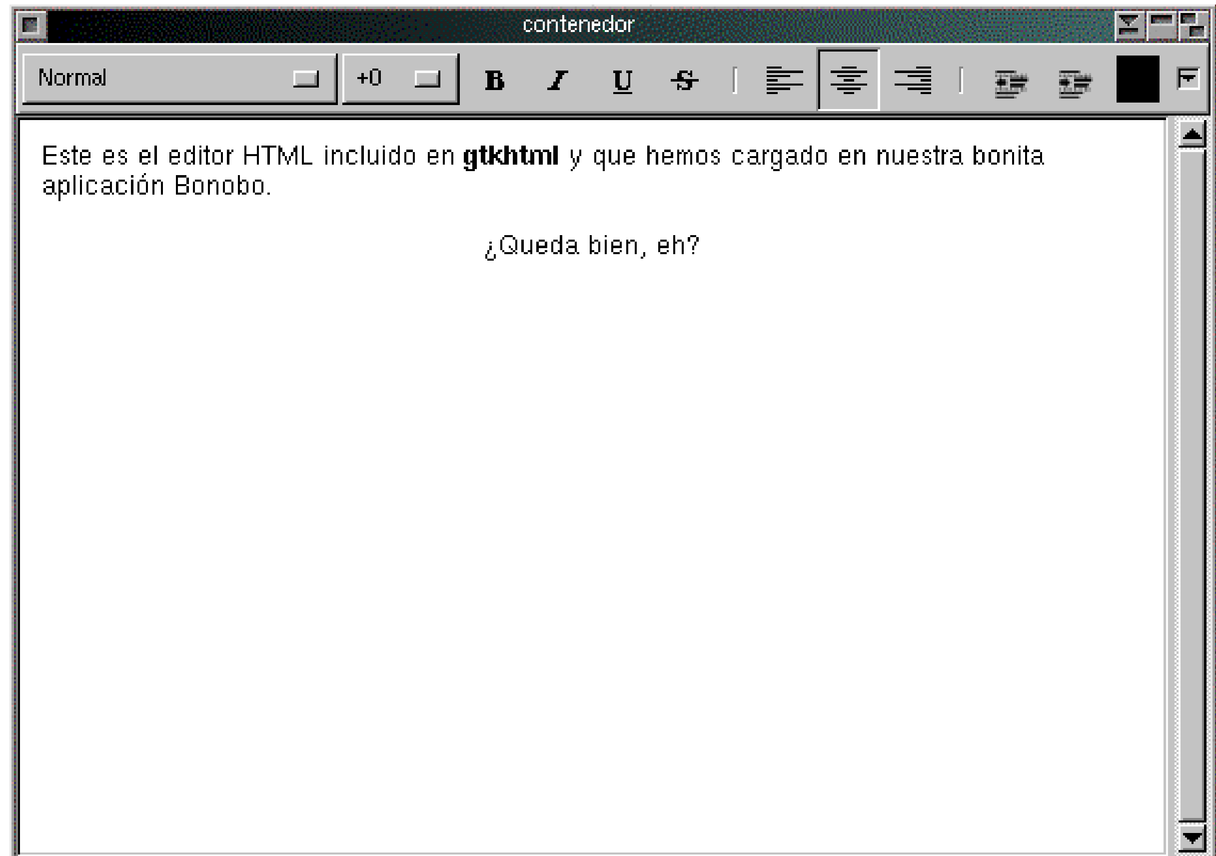



**Figura 16-3. Componente de edición GtkHTML**

Como última nota sobre la función `crear_ventana`, comentar que es muy importante que la función devuelva FALSE, pues si no GTK seguirá llamándola durante toda la duración del programa, cada vez que el bucle principal esté inactivo. Y evidentemente, eso nos es lo que queremos que haga nuestra aplicación, pues eso significaría que se crearía una ventana, con un control de edición HTML insertado, cada Â¡*n* milisegundos!

Bien, pues con esto ya tenemos una bonita aplicación Bonobo, sin mucha funcionalidad, pues le faltaría añadirle menús, barras de tareas y alguna que otra cosa más, pero que nos sirve para empezar a entender todo el funcionamiento de la arquitectura Bonobo. Ahora, pasemos a la creación de nuevos controles.

## Creación de nuevos controles

Si bien la creación de nuevos controles no es tan sencilla como su uso (como hemos visto en el apartado anterior), la verdad es que la dificultad es mínima, más bien ligada a la serie de pasos que hay que seguir que a la implementación en si. Una vez que se entienden los pasos básicos, no es complicado meterse a indagar en las entrañas de Bonobo, y así comenzar a usar características mucho más avanzadas que las que mostramos en este artículo.

Los controles (y todos los demás tipos de componentes) suelen implementarse en procesos independientes, aunque, por supuesto, también pueden implementarse en librerías dinámicas, que son cargadas por OAF según se vayan necesitando, o incluso en el mismo proceso en el que se utilizan. En nuestro caso, vamos a implementar nuestro control en un proceso independiente, pues resulta más sencillo. Y vamos a hacer que nuestro control sea una calculadora, mediante el uso del *"widget"* `GnomeCalculator`, que forma parte de las librerías básicas de GNOME (*gnome-libs*). Este *"widget"* implementa una útil calculadora que podemos fácilmente integrar en nuestro programa. Este *"widget"* es el mismo que utiliza el programa `gcalc`, la calculadora incluida en GNOME.

Así, aquí está nuestra función `main`, bastante parecida a la de los clientes anteriormente mostrada:

```
int main (int argc, char *argv[])
{
 CORBA_ORB orb;

 /* inicializamos GNOME */
 gnome_program_init ("prueba-control", "0.1", argc, argv, oaf_popt_options, 0, NULL);

 /* inicializamos Bonobo/OAF */
 orb = bonobo_activation_init (argc, argv);
 if (bonobo_init(orb, NULL, NULL) == FALSE)
  g_error("No se pudo inicializar Bonobo\n");

 bonobo_activate();
        crear_factoria();
 bonobo_main();

 return 0;
}
```

Aquí se puede ver que es prácticamente igual que la versión *"cliente"*, con la única diferencia de la llamada a `crear_factoria` en lugar de la llamada a `gtk_idle_add` que teníamos en el cliente. Y se preguntará el lector: Â¿factoría? Â¿para qué? La explicación es muy sencilla: en Bonobo, para la implementación de componentes, se usa lo que se llaman factorías, que son objetos CORBA que permiten la creación de nuevos objetos, para así facilitar todo el acceso a los distintos componentes. Así, nuestra función `crear_factoria` tendrá la siguiente forma:

```
void crear_factoria (void)
{
 BonoboGenericFactory* factoria;

 factoria = bonobo_generic_factory_new("OAFIID:GNOME_MiCalculadora",
                                       crear_nuevo_objeto,
                                       NULL);
 if (!BONOBO_IS_GENERIC_FACTORY(factoria))
  g_error("No se pudo crear factoría para nuestro control");
}
```

Como puede verse, lo único que hacemos en esta función es crear la factoría (de tipo `BonoboGenericFactory`). Pero hay un pequeño detalle, que es el segundo parámetro que le pasamos a la función `bonobo_generic_factory_new`, que es un puntero a una función que será la encargada de crear nuestros objetos según lo vayan solicitando. Esta función, quedaría de la siguiente manera.

```
BonoboObject *
crear_nuevo_objeto (BonoboGenericFactory *factoria, const char *id, void *data)
{
        BonoboControl*     control;
 GtkWidget*         calculadora;

 calculadora = gnome_calculator_new();
 gtk_widget_show(calculadora);

 control = bonobo_control_new(calculadora);
 bonobo_control_set_automerge(control, TRUE);
 return BONOBO_OBJECT(control);
}
```

Bien, con esta simple función ya tenemos la implementación de nuestro control iniciada y funcional. Lo que se hace en esta función es, primero, crear el *"widget"* que va a ser incluido en el control, que luego será pasado como parámetro a la función `bonobo_control_new`, que ya se encarga de recubrir nuestro *"widget"* (en este caso, una calculadora) con todo lo necesario para que pueda ser cargado en la aplicación contenedora.

Bueno, y como una imagen vale más que mil palabras, aquí se puede observar, en la siguiente captura de pantalla, nuestro control Bonobo cargado en la aplicación cliente comentada en el punto anterior (para ello, simplemente tenemos que cambiar el identificador del objeto a cargar en la llamada a `bonobo_widget_new_control`. En este caso, sustituiriamos la cadena `"OAFIID:GNOME_GtkHTML_Editor"` por `"OAFIID:GNOME_MiCalculadora"`).

**Figura 16-4. El control calculadora en la ventana contenedora**

Como última nota de esta parte, mencionar la llamada a `bonobo_control_set_automerge`, que lo que hace es mezclar automáticamente los menús y barras de tareas del control con los de la aplicación contenedora. Esta es otra de las características que hacen realmente útiles a los controles. Podemos dividir nuestra aplicación en distintos módulos *"visuales"*, e irlos cargando según se necesite. El usuario no notará absolutamente, simplemente verá que según activa uno u otro de los componentes de la aplicación, algunos menús y botones de la barra de tareas aparecen/desaparecen, algo que agradecerá enormemente, pues de nada le sirve la opción de, por ejemplo, corrección ortográfica incluida en el módulo de edición, mientras está retocando una imagen en el módulo de diseño de la aplicación. Esto es sólo un ejemplo, y de hecho vamos a dejar la explicación detallada de esta parte para el siguiente artículo de la serie, pues por su tamaño y extensión, merece un artículo para sí misma. Pero si el lector quiere ir investigando, puede consultar el código fuente de algunas de las aplicaciones que hacen uso de esta característica de Bonobo, como el mismo `Evolution`, GNOME-DB, `Nautilus`, `Eye Of GNOME`, etc.

### Menus y barras de tareas

## Documentos compuestos

A la hora de crear documentos compuestos utilizando Bonobo, Bonobo Embeddable, Empotrables Bonobo, es la clase de componentes Bonobo a utilizar para lograr una integración óptima entre las distintas partes de un documentos: gráficos, celdas de una hoja de cálculo, registros de una base de datos o cualquier otro tipo de contenidos que nos podamos utilizar dentro de un documento.

A diferencia de los controles Bonobo que son mucho más autónomos en su funcionamiento, los empotrables de Bonobo deben integrarse visualmente dentro del documento sis apreciarse que son objetos diferentes.

La motivación que nos ha llevado a escribir este capítulo del libro ha sido el poder utilizar las gráficas empotrables que nos devuelve Guppi.

A lo largo de todo el documento hablaremos de contenedores, empotrables y controles. Todos estos conceptos se refieren a la arquitectura de Bonobo por lo que evitaremos poner la coletilla Bonobo en todos ellos.

Los componentes empotrables Bonobo nos abren toda la potencia de la integración visual, de persistencia y de impresión de diferentes aplicaciones, desarrolladas incluso en diferentes lenguajes de programación. La potencia de esta plataforma veremos que es enorme y tras leer este completo artículo, se estará en disposición de comenzar a programar utilizando Bonobo. Â¿Preparado para el viaje?

## Introducción a los Documentos Compuestos

Una de las motivaciones principales a la hora de crear la arquitectura de componentes Bonobo fue facilitar el desarrollo de aplicaciones centradas en la creación de documentos.

Para el usuario final de la aplicación, lo que ofrece el sistema es un entorno de creación de documentos. Dichos documentos pueden incluir contenidos de muy diferentes tipos como imágenes, hojas de cálculo, bases de datos, texto, sonido o incluso vídeos, pero el tratamiento de todos los contenidos para el usuario final deberá de ser el mismo: se selecciona un tipo de contenido, se inserta en el documento de trabajo, y este contenido se integra visualmente dentro del documento como una parte más del mismo.

A la hora de trabajar con un determinado objeto, el usuario pulsará sobre él para activarlo, lo que provocará normalmente que la barra de herramientas de la aplicación de creación de documentos se especialice para tratar ese tipo determinado de objetos.

Pongamos un ejemplo que es posible se haga realidad en el futuro. Estamos trabajando dentro de AbiWord, el procesador de textos de GNOME, e introducimos un nuevo empotrable que es un conjunto de celdas de Gnumeric. Estas celdas aparecerán como una parte integrada dentro del documento. Al hacer "click" sobre ellas, la barra de herramientas (toolbar) de AbiWord será sustituida por la de Gnumeric y el usuario podrá utilizar esta barra de herramientas para obtener la funcionalidad específica que proporciona la herramienta Gnumeric.

Un ejemplo ya real es la forma en la que Gnumeric integra gráficos dentro de las hojas de cálculo utilizando empotrables Guppi. En la siguiente imagen podemos ver una captura de pantalla de como se integra la gráfica dentro de la hoja de cálculo.

A lo largo de las secciones siguientes del documento vamos a explicar como interacciona el contenedor (AbiWord en el primer ejemplo y Gnumeric en el segundo) con los componentes empotrados que residen dentro de él.

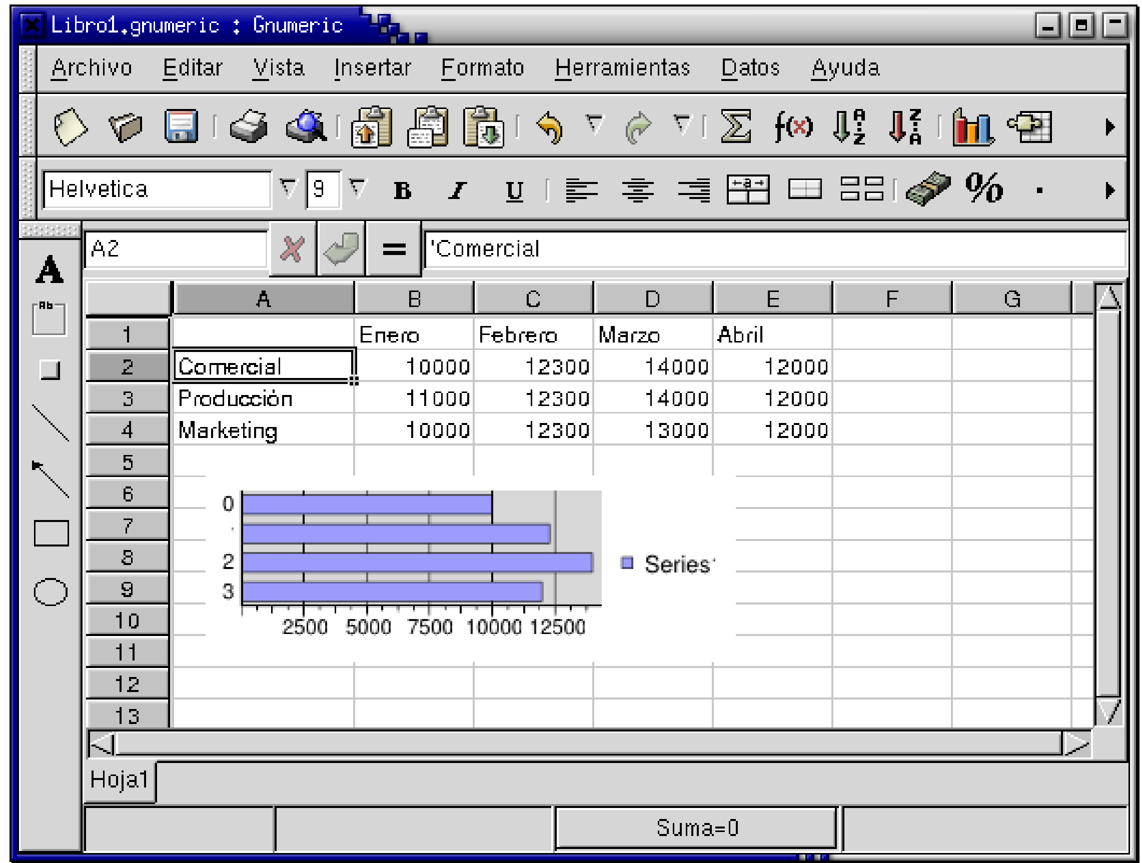



Figura 16-5. Gnumeric con Guppi empotrado

## Interfaces de comunicación entre contenedor y empotrable

En la siguiente figura podemos observar las interfaces de comunicación que se establecen entre el contenedor y el empotrable. Toda la interacción entre ambos se realiza a través de estas interfaces, parte de las IDL de Bonobo.

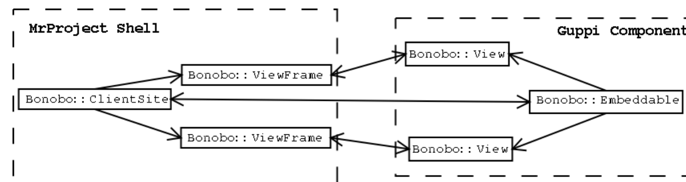



Figura 16-6. Comunicación contenedor y empotrable

Cada empotrable (Bonobo::Embeddable) que se visualiza dentro del contenedor necesita tener un sitio en el contenedor. Este sitio es el "Bonobo::ClientSite". Un

empotrable se puede visualizar en diferentes vistas dentro del mismo contenedor. Todas las vistas mostrarán el mismo empotrable, pero pueden estar situadas en diferentes localizaciones del documento. Para dar soporte a esta funcionalidad, cada empotrable se visualiza dentro del contenedor a través de una vista (Bonobo::View) que está asociada a un lugar de visualización (Bonobo::Frame) dentro del sitio reservado en el contenedor para el empotrable (Bonobo::ClientSite).

Es este un buen ejemplo de la separación entre el modelo de datos, del que hay una copia única en el empotrable, y el modelo de visualización, del que puede haber múltiples copias, con visualizaciones diferentes. Pero los datos se comparten.

## Documentación y ejemplos de documentos compuestos

Ha llegado el momento de ver como se programa un empotrable y como se inserta dentro de un contenedor. Podemos intentar realizar un análisis de como se realiza dentro de Gnumeric o bien, ver los ejemplos que acompañan a Bonobo.

Vamos a seguir este segundo camino inicialmente, intentando utilizar como empotrable a Guppi. Como contenedor nos valdrá uno génerico.

Lo primero es bajarnos las fuentes de bonobo, la implementación en C de las interfaces IDL de Bonobo, ya que es en ellas donde vienen los ejemplos. En muchas ocasiones, los paquetes que instalamos ya no tienen los ejemplos que acompañan a las fuentes.

```
    acs@infra:~/debian-src$ apt-get source bonobo
Reading Package Lists... Done
Building Dependency Tree... Done
Need to get 1560kB of source archives.
0% [Connecting to red-carpet.ximian.com (129.250.184.229)]
Fetched 1560kB in 5m24s (4809B/s)
dpkg-source: extracting bonobo in bonobo-1.0.19
```

Ya dentro de las fuentes, lo mejor suele ser irse a las demos si es que existen, o buscar programas de test que también suelen ser una buena fuente de ejemplos. En el caso de Bonobo tenemos un directorio con ejemplos (samples) e incluso uno para ejemplos de documentos compuestos (compound-doc).

Es conveniente compilar todas las fuentes, aunque no hay que instalarlas porque se supone que ya tenemos Bonobo en nuestra máquina, para poder tener un entorno de desarrollo en el que poder modificar tranquilamente los ejemplos y no tener problemas a la hora de compilarlos.

Junto con los ejemplos de Bonobo en este caso tenemos también la cada vez más completa documentación del API de Bonobo, cuya versión más actualizada he localizado en las páginas de Michael Meeks[1]. En concreto de esta API ahora nos interesa la parte referente a Documentos Compuestos Bonobo[2]. Se divide esta sección en tres grandes grupos: interfaces del modelo, de la vista e impresión.

### Interfaces del modelo (datos)

Aquí se agrupa la interfaz del empotrable BonoboEmbeddable, la del contenedor BonoboClientSite y BononoItemContainer. Sobre esta última decir que esta siendo sustituida por una versión basada en monikers llamada BonoboItemHandler. Esta interfaz es la parte visible de una implementación simple para contenedores de documentos compuestos.

### Interfaces de la vista

## Impresión

## El primer ejemplo de un contenedor

Nos vamos a las fuentes de bonobo y en concreto, al directorio "samples/compound-doc".

En este directorio nos encontramos con un empotrable que es un componente de dibujo (paint-component-simple.c), un ejemplo de uso de BonoboCanvasItem que nos permite crear empotrables a partir de GnomeCanvas, un directorio "container" con un ejemplo de un contenedor y un directorio "bonobo-hello" con un empotrable que muestre el típico "Hello World" pero dentro de un empotrable que reside en un contenedor. Â¿Listo para comenzar la aventura?

Como siempre, empezaremos con el método "main" e iremos siguiendo el flujo de ejecución del programa. Ya en anteriores apartados hemos dibujado el análisis conceptual y más orientado a objetos de los documentos compuestos, por lo que el enfoque de seguir el flujo de ejecución del programa nos dará un nuevo punto de vista que facilitará la compresión.

El método "main" se encuentra en "container.c":

```
   int
main (int argc, char **argv)
{
 setup_data_t init_data;
 CORBA_Environment ev;
 CORBA_ORB orb;

 free (malloc (8));

  CORBA_exception_init (&ev);
 gnome_init_with_popt_table ("container", VERSION,
        argc, argv, oaf_popt_options, 0, &ctx);

 orb = oaf_init (argc, argv);

 if (bonobo_init (orb, NULL, NULL) == FALSE)
  g_error (_("Could not initialize Bonobo!\n"));
```

Hasta el momento, tan sólo hemos inicializado CORBA y Bonobo, algo necesario en cualquier aplicación que haga uso de Bonob.

```
   init_data.app = sample_app_create ();
```

En esta función es donde es previsible que esté la parte de arranque de toda la aplicación, y será la que pasaremos a analizar después de terminar "main".

```
   if (ctx)
  init_data.startup_files = (const char**) poptGetArgs (ctx);
 else
  init_data.startup_files = NULL;
```

Aquí lo único que hacemos es ver si se han pasado argumentos por la línea de comandos, en cuyo caso los procesamos.

```
gtk_idle_add ((GtkFunction) final_setup, &init_data);
```

Como siempre, lanzamos la aplicación de forma retardada para darnos tiempo a entrar en el bucle principal de bonobo. El arranque de la aplicación

```
bonobo_main ();
```

Nos vamos al bucle principal de bonobo a la espera de eventos.

```
   if (ctx)
  poptFreeContext (ctx);

 return 0;
}
```

Cuando salgamos del bucle principal de bonobo por alguna condición de terminación, liberamos el contexto y salimos retornando "0" indicando ejecución correcta.

Bien, vamos a centrarnos ahora en los métodos que realmente hacen el trabajo: sample_app_create y final_setup.

### sample_app_create

```
    static SampleApp *
sample_app_create (void)
{
 SampleApp *app = g_new0 (SampleApp, 1);
```

Este método lo que nos devuelve es un SampleApp el cual esta definido dentro de "container.h" y su estructura es:

```
typedef struct _SampleApp        SampleApp;
struct _SampleApp {
 BonoboItemContainer *container;
 BonoboUIContainer   *ui_container;

 BonoboViewFrame     *curr_view;
 GList               *components;

 GtkWidget           *app;
 GtkWidget           *box;
 GtkWidget           *fileselection;
};
```

Es importante tener en mente esta estructura ya que incluye todos los parámetros que utilizaremos para gestionar los empotrables que vayamos incorporando al contenedor. Sigamos con el método "sample_app_create".

```
 GtkWidget *app_widget;

 /* Create widgets */
 app_widget = app->app = bonobo_window_new ("sample-container",
         _("Sample Bonobo container"));
```

Esta llamada es la que crea el contenedor Bonobo real. Vemos que basta con hacer una llamada para tener una GtkWindow pero con todo el soporte de Bonobo detrás, por lo que el uso de contenedores Bonobo es bastante sencillo.

```
 app->box = gtk_vbox_new (FALSE, 10);

 gtk_signal_connect (GTK_OBJECT (app_widget), "delete_event",
       (GtkSignalFunc) delete_cb, app);

 /* Do the packing stuff */
 bonobo_window_set_contents (BONOBO_WINDOW (app->app), app->box);
 gtk_widget_set_usize (app_widget, 400, 600);
```

Asociamos a los contenidos de la BonoboWindow el GtkWidget box, un contenedor vertical de Gtk en el que vamos a ir poniendo todos los empotrables que vayan incorporando al contenedor. Este es un contenedor simple de Gtk y no tiene nada de Bonobo.

```
 app->container = bonobo_item_container_new ();

 app->ui_container = bonobo_ui_container_new ();
 bonobo_ui_container_set_win (app->ui_container, BONOBO_WINDOW (app->app));
```

De nuevo, otra de las partes claves para tener control sobre todos los empotrables que vayan metiendo en el contenedor Bonobo. En este caso creamos un BonoboUIContainer, que como ya veremos, es el que nos va a permitir controlar detalles de la interfaz gráfica del contenedor Bonobo en función de los empotrables, como la gestión de la barra de herramientas.

```
    /* Crear menu bar */
 sample_app_fill_menu (app);

 gtk_widget_show_all (app_widget);

 return app;
}
```

La interfaz gráfica la completamos con un menú desde el que poder cargar los componentes y una barra de herramientas con menús. Esto es en su mayoría programación normal Gtk. Vemos una captura de pantalla para ver el resultado:

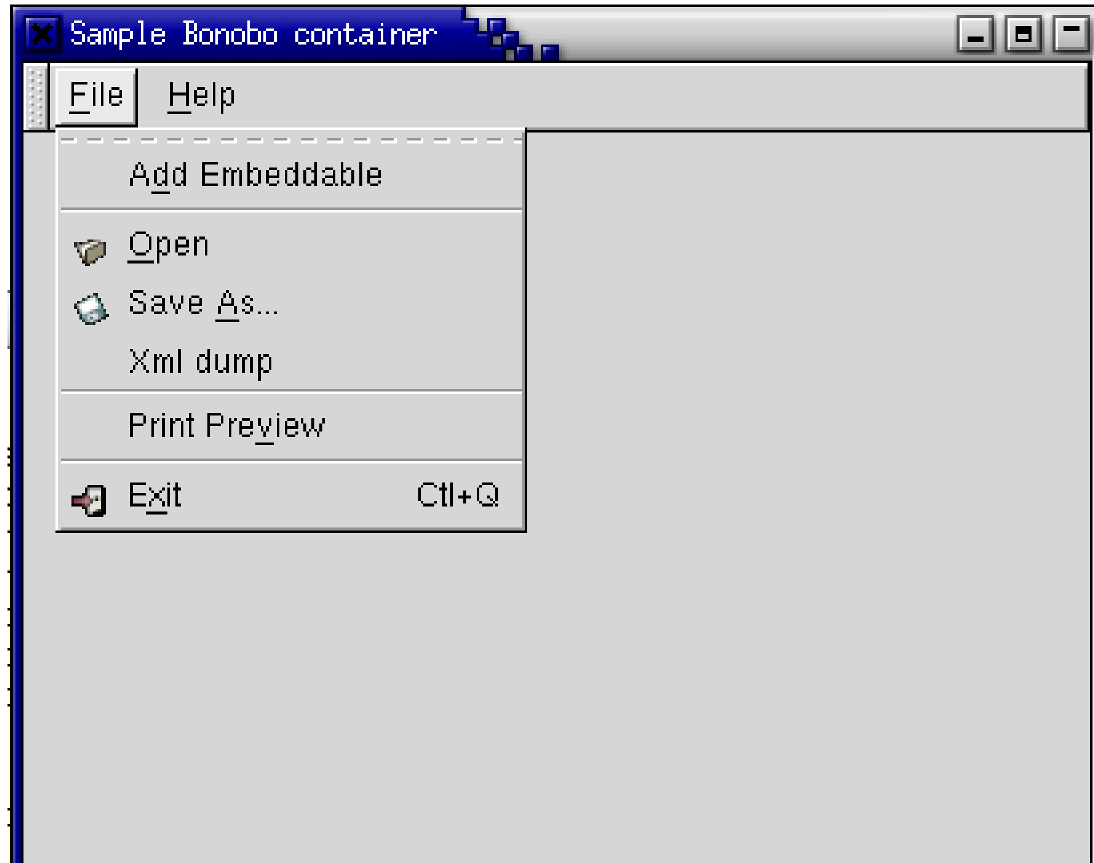



**Figura 16-7. GUI del contenedor Bonobo**

Pero aquí también se incluye el código que permite cambiar la barra de herramientas y menús en función de que componente empotrable este activo. Vamos pues a buscar este código ya que es muy importante para la programación Bonobo. Esta función está implementada en "container-menu":

```
void
sample_app_fill_menu (SampleApp *app)
{
 Bonobo_UIContainer corba_container;
 BonoboUIComponent *uic;

 uic = bonobo_ui_component_new ("sample");
```

Creamos un nuevo componente de UI de Bonobo, el cual va a ser nuestra barra de menú. Esta barra se va a ir cambiando en función de que componente empotrable tengamos activado.

```
 corba_container = BONOBO_OBJREF (app->ui_container);
 bonobo_ui_component_set_container (uic, corba_container);
```

Asociamos el menú Bonobo al contenedor Bonobo donde vamos a ir incorporando los empotrables.

```
 bonobo_ui_component_set_translate (uic, "/", ui_commands, NULL);
 bonobo_ui_component_set_translate (uic, "/", ui_data, NULL);
```

De la API de Bonobo, que viene incluída junto con las fuentes que nos hemos bajado, y que está perfectamente organizada y completada en casi todas sus partes, vemos que estas funciones que son parte de BonoboUIComponent son las responsables de que podamos traducir los contenidos de los menús. No vamos a entrar aquí en más detalles aunque en la API aparece como se debe de utilizar.

```
 bonobo_ui_component_add_verb_list_with_data (uic, sample_app_verbs, app);
}
```

Esta última llamada añade el contenido de los menús. Estos menús son:

```
    static BonoboUIVerb sample_app_verbs[] = {
 BONOBO_UI_VERB ("AddEmbeddable", verb_AddEmbeddable_cb),
 BONOBO_UI_VERB ("FileOpen", verb_FileLoad_cb),
 BONOBO_UI_VERB ("FileSaveAs", verb_FileSaveAs_cb),
 BONOBO_UI_VERB ("PrintPreview", verb_PrintPreview_cb),
 BONOBO_UI_VERB ("XmlDump", verb_XmlDump_cb),
 BONOBO_UI_VERB ("FileExit", verb_FileExit_cb),
 BONOBO_UI_VERB ("HelpAbout", verb_HelpAbout_cb),
 BONOBO_UI_VERB_END
};
```

Vemos que al aplicación se controla en gran medida desde estos menús, que provocan callbacks a funciones según la opción que hayamos seleccionado.

Volvemos de nuevo al método "main" de "container.c" para finalizar con su inicialización, algo que se hace dentro de "final_setup".

### final_setup

Este método realiza cosas interesantes en el caso de que tengamos activadas las pruebas de Monikers, algo que de momento no vamos a realizar, por lo que podemos ignorar este método.

## Añadir empotrables al contenedor

Después de tener ya un contenedor Bonobo totalmente preparado para recibir empotrables e incorporarlos a sus contenidos, vamos a ver como se hace esto. Del menú del contenedor lo podemos deducir de forma sencilla ya que hay un verbo:

```
 BONOBO_UI_VERB ("AddEmbeddable", verb_AddEmbeddable_cb)
```

Es decir, que cuando pulsemos sobre "AddEmbeddable" se va a ejecutar el método de callback "verb_AddEmbeddable_cb". Veamos que es lo que hace este método:

```
static void
verb_AddEmbeddable_cb (BonoboUIComponent *uic, gpointer user_data, const char *cname)
{
 SampleApp *inst = user_data;
 char *required_interfaces [2] =
     { "IDL:Bonobo/Embeddable:1.0", NULL };
```

Aquí indicamos la interfaz IDL requerida por aquellos componentes Bonobo que podemos empotrar en el contenedor. En este caso, sólo aquellos que implementen la interfaz de empotrables, claro.

```
char *obj_id;

/* Ask the user to select a component. */
obj_id = bonobo_selector_select_id (
  _("Select an embeddable Bonobo component to add"),
  (const gchar **) required_interfaces);
```

Esta es una función muy útil de Bonobo que nos abre un dialogo con el usuario final para que seleccione que tipo de empotrable quiere añadir. Esta función localiza todos los empotrables disponibles por lo que si añadimos nuevos empotrables al sistema, los podremos luego cargar utilizando esta función. En realidad, y pensando en el usuario final, lo suyo sería que no se especificara el empotrable si no el objeto que se quiere insertar en el documento de trabajo. En función del tipo de objeto, se consultaría que empotrable lo puede visualizar, y se cargaría el objeto a través de él. Veamos que aspecto tiene este dialogo:

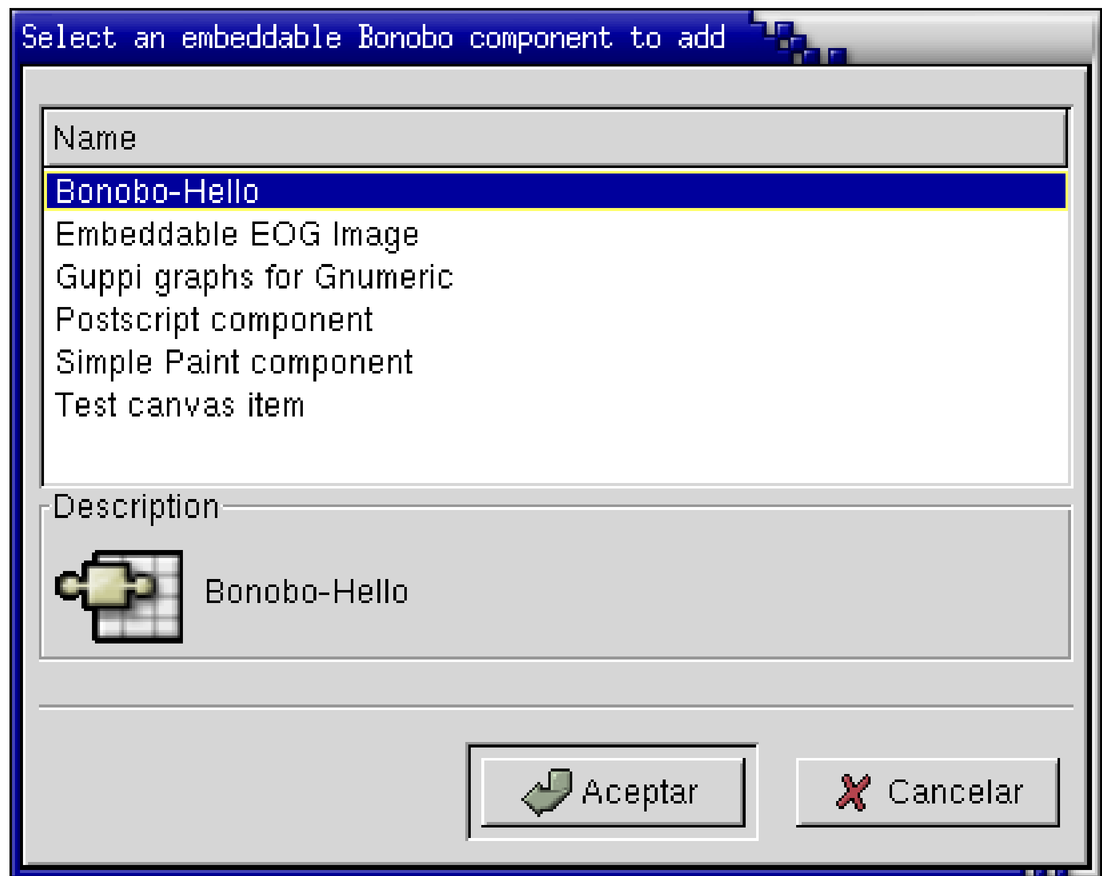



**Figura 16-8. Selección de componente Bonobo empotrable**

Esta función nos devuelve un identificador único para los objetos asociados a este empotrable de forma que podremos utilizarlo a partir de ahora como sinónimo de este tipo de objetos Bonobo.

```
 if (!obj_id)
  return;

 /* Activarlo. */
 sample_app_add_component (inst, obj_id);
```

Vamos con la inserción del empotrable, uno de los procesos claves dentro de esta sección, y para ello, nos toca analizar a fondo esta función.

```
static SampleClientSite *
sample_app_add_embeddable (SampleApp          *app,
      BonoboObjectClient *server,
      char               *object_id)
{
```

De momento ya vemos por aquí la aparición en escena de los ClientSite, en concreto el parámetro de retorno de esta función es SampleClientSite que si analizamos su estructura, que aparece dentro de "component.h":

```
struct _SampleClientSite {
 BonoboClientSite parent;

 SampleApp *app;
 gchar     *obj_id;

 GtkWidget *widget;
 GtkWidget *views_hbox;
 GtkWidget *frame;
};
```

Ya iremos viendo el uso que le damos a los diferentes campos del ClientSite asociado a un componente, en este caso, a componentes Bonobo empotrables. Sigamos con "sample_app_add_embeddable":

```
 SampleClientSite *site;

 /*
  * El ClientSite es el punto de contacto del lado del contenedor
  * para un Embeddable.  Luego hay una correspondencia uno-a-uno
  * entre BonoboClientSites y BonoboEmbeddables.
  */
 site = sample_client_site_new (app->container, app,
           server, object_id);
```

Acabamos de crear el ClientSite en el contenedor por el que se comunicarán el contenedor y el empotrable. Veamos los detalles de esta creación:

```
 SampleClientSite *
sample_client_site_new (BonoboItemContainer *container,
   SampleApp           *app,
   BonoboObjectClient  *embeddable,
   const char          *embeddable_id)
{
 SampleClientSite *site;

 g_return_val_if_fail (app != NULL, NULL);
 g_return_val_if_fail (embeddable_id != NULL, NULL);
 g_return_val_if_fail (BONOBO_IS_OBJECT_CLIENT (embeddable), NULL);
```

```
g_return_val_if_fail (BONOBO_IS_ITEM_CONTAINER (container), NULL);
```

Tratamiento de los parámetros que recibimos para asegurarnos que son todos correctos.

```
site = gtk_type_new (sample_client_site_get_type ());

site = SAMPLE_CLIENT_SITE (bonobo_client_site_construct (
 BONOBO_CLIENT_SITE (site), container));
```

La clase SampleClientSite hereda de BonoboClientSite y para construir un objeto de esta clase, utilizamos la llamada de bonobo "bonobo_client_site_construct", asociando el BonoboClientSite creado al contenedor Bonobo en el que van a residir todos los empotrables y que ya analizamos en el anterior apartado.

```
if (site) {
 bonobo_client_site_bind_embeddable (BONOBO_CLIENT_SITE (site),
         embeddable);
```

Sin duda, la llamada más importante de esta función, cuando se asocia el BonoboClientSite al Embeddable. A partir de este momento se abren las comunicaciones entre el contenedor y el empotrable y podemos comenzar a visualizar el empotrable, crear nuevas vistas ...

```
        bonobo_object_unref (BONOBO_OBJECT (embeddable));
```

El control de referencias a objetos Bonobo es muy importante si no queremos tener agujeros de memoria, aunque en este caso no entiendo porqué tenemos que quitar una referencia al embeddable. Quizá sea debido a que al enlazar el empotrable y el contenedor, el contenedor indica una nueva referencia dentro de él al empotrable. La referencia que teníamos al empotrable fuera del contenedor ya no nos sirve para nada y no la vamos a utilizar, por lo que informamos el objeto empotrable que quite una referencia entre las que tiene.

```
site->app = app;
 g_free (site->obj_id);
 site->obj_id = g_strdup (embeddable_id);

 site_create_widgets (site);
```

De nuevo, una llamada a una función que como vamos a ver, ya sólo trata con Gtk para crear una interfaz gráfica en la que insertar el nuevo empotrable, en realidad, para poder ir insertando las vistas que vayamos creando del empotrable. Aunque veremos que aún en este método, hay algo de Bonobo también:

```
static void
site_create_widgets (SampleClientSite *site)
{
 GtkWidget *frame;
 GtkWidget *vbox, *hbox;
 GtkWidget *new_view_button, *del_view_button;
 GtkWidget *del_comp_button, *fill_comp_button;

 g_return_if_fail (site != NULL);

 /* Display widgets */
 frame = site->frame = gtk_frame_new (site->obj_id);
 vbox = gtk_vbox_new (FALSE, 10);
 hbox = gtk_hbox_new (TRUE, 5);
 new_view_button = gtk_button_new_with_label ("New view");
 del_view_button = gtk_button_new_with_label ("Remove view");
```

```
del_comp_button = gtk_button_new_with_label ("Remove component");
/* The views of the component */
site->views_hbox = gtk_hbox_new (FALSE, 2);
gtk_signal_connect (GTK_OBJECT (new_view_button), "clicked",
      GTK_SIGNAL_FUNC (add_frame_cb), site);
gtk_signal_connect (GTK_OBJECT (del_view_button), "clicked",
      GTK_SIGNAL_FUNC (del_frame_cb), site);
gtk_signal_connect (GTK_OBJECT (del_comp_button), "clicked",
       GTK_SIGNAL_FUNC (del_cb), site);

gtk_container_add (GTK_CONTAINER (hbox), new_view_button);
gtk_container_add (GTK_CONTAINER (hbox), del_view_button);
gtk_container_add (GTK_CONTAINER (hbox), del_comp_button);
```

Hasta el momento hemos creado un nuevo GtkFrame, frame, en que vamos a ir añadiendo las vistas del empotrable. Para poder ir gestionando las vistas y el empotrable, añadimos tres botones con las funcionalidades necesarias. Lo más importante de esta parte son las funciones que se llaman en caso de pulsar cada uno de los botones, funcionalidad que analizaremos en los siguientes apartados.

```
if (bonobo_object_client_has_interface (
 bonobo_client_site_get_embeddable (BONOBO_CLIENT_SITE (site)),
 "IDL:Bonobo/PersistStream:1.0", NULL)) {
```

En el caso de que el componente empotrable implemente la interfaz de persistencia, podemos cargar los datos del empotrable desde un "stream". Si volvemos a pensar en el usuario final, este querrá introducir en su documento de trabajo un objeto, que podrían ser los datos persistentes de un componente empotrable. Los pasos que daríamos serían pues localizar el empotrable que es capaz de cargar dicho objeto y, utilizando la interfaz de persistencia de ese empotrable, cargar los contenidos del objeto dentro del empotrable.

```
fill_comp_button =
     gtk_button_new_with_label ("Fill with stream");
 gtk_container_add (GTK_CONTAINER (hbox), fill_comp_button);

 gtk_signal_connect (GTK_OBJECT (fill_comp_button),
       "clicked", GTK_SIGNAL_FUNC (fill_cb),
       site);
}
```

Creamos un botón que nos permite cargar los datos persistentes del componente empotrable. Lo más importante de esta parte es de nuevo la función a la que se llama "fill_cb", que cubriremos en los próximos apartados.

```
 gtk_container_add (GTK_CONTAINER (vbox),  site->views_hbox);
 gtk_container_add (GTK_CONTAINER (vbox),  hbox);
 gtk_container_add (GTK_CONTAINER (frame), vbox);
}
```

Hemos terminado de construir el widget asociado al empotrable y volvemos el método de creación de un nuevo SampleClientSite.

```
 }

 return site;
}
```

Bueno, visto como se crea el SampleClientSite, continuemos con el método ""sample_app_add_embeddable".

```
app->components = g_list_append (app->components, site);
```

Mantenemos un listado con todos los componentes empotrables que hemos ido añadiendo ya que si luego hay que salir de la aplicación, tenemos que tener cuidado de irlos liberando todos.

```
gtk_box_pack_start (GTK_BOX (app->box),
      sample_client_site_get_widget (site),
      FALSE, FALSE, 0);
```

Vemos lo sencillo que es añadir el nuevo empotrable al contendor GTK que está viendo el usuario final. Aunque esta sencillez es gracias al método "sample_client_site_get_widget" que es parte de la clase "SampleClientSite", implementada en component.c":

```
 GtkWidget *
sample_client_site_get_widget (SampleClientSite *site)
{
 g_return_val_if_fail (site != NULL, NULL);

 return site->frame;
}
```

Esta función es realmente sencilla ya que dentro del constructor de SampleClientSite ya habíamos hecho todo el trabajo de construcción del GUI, por lo que aquí sólo nos queda el acceder a él. Volvemos al método "sample_app_add_embeddable" para terminar de añadir el empotrable al contenedor principal.

```
sample_client_site_add_frame (site);
```

Llegamos a uno de los métodos que más carga de Bonobo tienen ya que tiene que crear una nueva vista del empotrable, lo que significa crear un View dentro del empotrable y un ViewFrame dentro del contenedor. Vamos a analizar el método con cuidado:

```
void
sample_client_site_add_frame (SampleClientSite *site)
{
 BonoboViewFrame *view_frame;
 GtkWidget       *view_widget;

 /*
  * Creamos la View remota del empotrable y el ViewFrame local
  * del contenedor. Esto también estable el BonoboUIHandler para
        * este ViewFrame. De esta forma, el componente empotrado puede
  * tener acceso a nuestro servidor UIHandler de forma que pueda
        * mezclar sus menús y barra de herramientas cuando es activado
  */
 view_frame = bonobo_client_site_new_view (BONOBO_CLIENT_SITE (site),
  BONOBO_OBJREF (site->app->ui_container));

 /*
  * Empotrar la view frame en la aplicación
  */
 view_widget = bonobo_view_frame_get_wrapper (view_frame);
 gtk_box_pack_start (GTK_BOX (site->views_hbox), view_widget,
       FALSE, FALSE, 5);
```

```
 /*
  * La señal "user_activate" se emitirá cuando el usuario
         * haga doble click en la "cubierta" de la vista. Esta
         * cubierta es una ventana transparente que está
         * encima del componente y captura cualquier evento
         * (teclado, ratón) que se produce sobre la cubierta.
         * Cuando el usuario hace doble click sobre la cubierta
         * el contenedor (nosotros en este momento) puede decidir
         * activar el componente
  */
 gtk_signal_connect (GTK_OBJECT (view_frame), "user_activate",
       GTK_SIGNAL_FUNC (activate_request_cb),
       site);

 /*
  * La activación "In-place" de un componente es un proceso
  * en dos pasos. Después de que el usuario haga doble click
         * en el componente, nuestro callback de la señal
         * (component_user_activate_request_cb()) pide al componente
  * que se active (ver  bonobo_view_frame_view_activate()).
  * El componente puede entonces elegir aceptar o rechazar
         * la activación. Cuando un componente empotrado nos informa
         * de su decisión de pasar a estado activo, la señal "activated"
         * se emite desde el view frame. En ese momento quitamos la
  * cubierta del componente para que pueda recibir eventos.
  */
 gtk_signal_connect (GTK_OBJECT (view_frame), "activated",
       GTK_SIGNAL_FUNC (view_activated_cb),
       site);

 /*
  * La señal "user_context" se emite cuando el usuario pulsa el
  * botón derecho en la cubierta. Lo usamos para mostrar un menu verb
  */
 gtk_signal_connect (GTK_OBJECT (view_frame), "user_context",
       GTK_SIGNAL_FUNC (component_user_context_cb),
       site);

 /*
  * Mostrar el component,
  */
 gtk_widget_show_all (view_widget);
}
```

Por fin hemos terminado de añadir el empotrable a nuestro contenedor, proceso en el que hemos aprendido mucho de empotrables, algo que vamos a ir asentando en las próximas secciones de este capítulo.

```
 gtk_widget_show_all (GTK_WIDGET (app->box));

 return site;
}
```

Para terminar el método "verb_AddEmbeddable_cb":

```
 g_free (obj_id);
}
```

Y ojo, no nos olvidemos de no ir dejando memoria sin liberar ;-)

Ya hemos visto todo el código del contenedor necesario para incluir componentes empotrables, asi como el código necesario para introducir dentro del contenedor un componente Bonobo empotrable. Ahora vamos a intentar ir resaltando las partes más importantes para terminar mostrando como crear los empotrables, los componentes que finalmente se empotran en Bonobo.

## Vistas de un Empotrable

En la siguiente figura podemos ver al contenedor Bonobo con varias vistas de un mismo componente empotrable que, como se puede apreciar, tienen todas las vistas el mismo contenido.

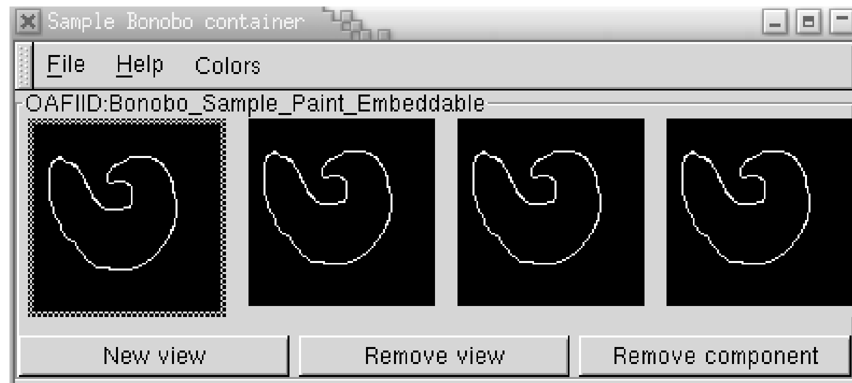



**Figura 16-9. Varias vistas de un mismo empotrado**

Vamos a ir viendo como se logra esta funcionalidad tan potente con Bonobo.

### Las interfaces ClientSite y Embeddable

Como ya hemos visto, para poder introducir un empotrable dentro de un contenedor, algo que se realizaba dentro de la función "sample_app_add_embeddable", lo primero que necesitamos es reservar dentro del contenedor un sitio para el empotrable. Este sitio se crea con un "BonoboClientSite", en nuestro caso SampleClientSite que tiene más información, y la llamada que nos ha permitido crear un "BonoboClientSite" asociado a un contenedor ha sido:

```
site = SAMPLE_CLIENT_SITE (bonobo_client_site_construct (
 BONOBO_CLIENT_SITE (site), container));
```

Ya tenemos un "site" (BonoboClientSite) preparado para recibir un componente empotrable. Vamos a decirle cual es, algo que hacemos con la llamada:

```
bonobo_client_site_bind_embeddable (BONOBO_CLIENT_SITE (site),
         embeddable);
```

De esta forma hemos puesto en comunicación el contenedor con el empotrable, pero esta comunicación aún no ha provocado la visualización del empotrable dentro del contenedor. Aún es necesario que creemos las vistas del empotrable dentro del contenedor.

## Las interfaces ViewFrame y View

Una vez que tenemos asociados un contenedor y un empotrable, podemos insertar tantas vistas (View) del empotrable en el contenedor como queremos. Todas las vistas tendrán los mismos contenidos de origen, aquellos que tenga el empotrable. Para que el contenedor sea capaz de gestionar todas estas vistas, cada una de las "View" de un empotrable es necesaria que esté asociada a un "ViewFrame" dentro del contenedor. Estos "ViewFrame" se crean asociados a un ClientSite, como podemos ver en el siguiente código del ejemplo:

```
 BonoboViewFrame *view_frame;
 view_frame = bonobo_client_site_new_view (BONOBO_CLIENT_SITE (site),
BONOBO_OBJREF (site->app->ui_container));
```

La llamada, junto con el "BonoboClientSite", incluye un Bonobo_UIContainer, que como luego veremos, es el que permite integrar los menús y la barra de herramientas del empotrable con el del contenedor. Esta llamada de forma automática crea una nueva vista (View) del empotrable remoto, y utilizando el ViewFrame, nosotros vamos a poder decidir en que parte de la GUI se muestra el ViewFrame. Para ello, utilizamos la siguiente llamada que nos permite obtener el widget asociado al ViewFrame, que en realidad, es el widget asociado a la vista del empotrable que queremos visualizar.

```
 GtkWidget       *view_widget;
 view_widget = bonobo_view_frame_get_wrapper (view_frame);
```

Ahora ya tenemos un widget Gtk el cual colocamos dentro del GUI utilizando las técnicas habituales de diseño de GUI con Gtk.

```
gtk_box_pack_start (GTK_BOX (site->views_hbox), view_widget,
     FALSE, FALSE, 5);
gtk_widget_show_all (view_widget);
```

donde el contenedor "site->views_hbox" en un contenedor que coloca los widgets en horizontal, y que muestra todas las vistas que vayamos añadiendo. Este contenedor a su vez se ha incluído dentro del contenedor principal.

Es muy importante el tratamiento de las señales que se hace de este widget. Los componentes empotrados están desactivados por defecto. Ello significa que el usuario trabaja únicamente con el documento global o con alguno de los componentes empotrados. Según lo que esté activo, tendremos por ejemplo una barra de menú o una barra de herramientas. Veremos con detalle la importancia de estas señales a la hora de activar y desactivar los componentes empotrables.

## Integración de menús y barras de herramientas

Pongamos en el papel de nuestro usuario final. Tiene abierto un documento de trabajo en el que tiene empotrados varios componentes. Por lo general, al usuario lo que le importa es como se visualizan esos componentes y como salen colocados dentro del documento global. Pero el usuario también quiere poder interactuar con dichos componentes, por ejemplo, para modificar los datos de una gráfica o para modificar una imágen insertada en el documento. Quizá incluso quiera modificar la hoja de cálculo al recibir nuevos datos sobre las ventas del último cuatrimestre.

La idea es que todos estos cambios los pueda hacer el usuario sin necesidad de salir el documento. Para ello, el componente debe de poder presentar una interfaz al usuario capaz de permitirle trabajar y modificar el componente. Ello se logra gracias a la capacidad de los contenedores Bonobo de integrar la barra de menús y de herramientas

de los empotrados dentro del contenedor principal. En la siguiente captura podemos ver varios componentes empotrados dentro del contenedor, pero ninguno de ellos está activado aún. Podemos observar la barra de menús que tenemos disponible en este caso.

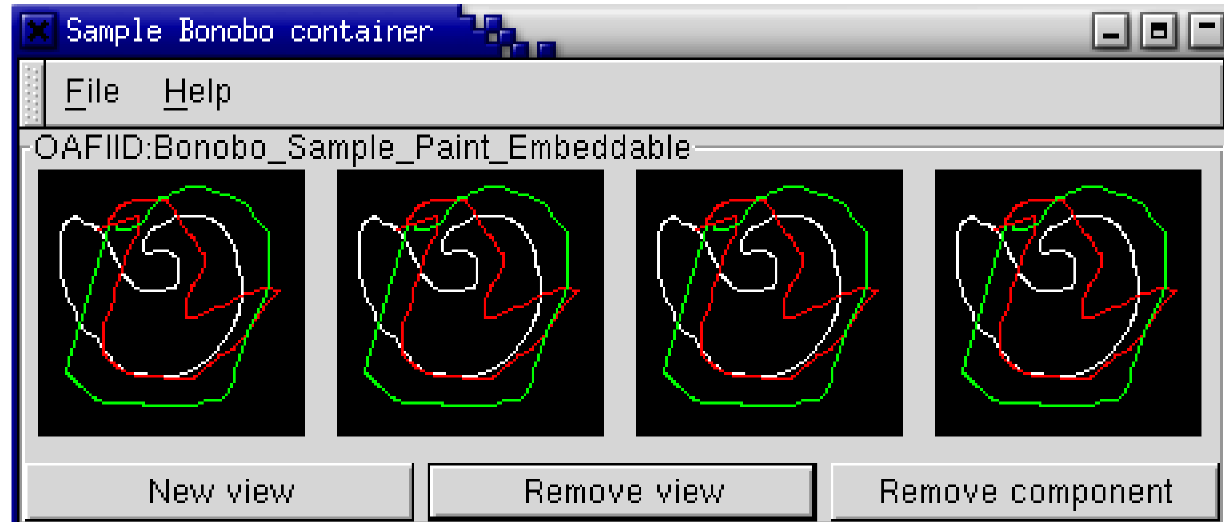



**Figura 16-10. Barra de menús sin componente empotrado activo**

Cuando un usuario activa un componente empotrado, haciendo doble click sobre él normalmente, una de las acciones que lleva a cabo Bonobo es la de cambiar los menús y la barra de herramientas por aquellos que diga el componente. De esta forma el usuario podrá interactuar con el componente y cambiarlo. Cuando finalice, podrá volver a activar el documento global, recuperando los menús y barra de herramientas originales. Podemos observar como cuando tenemos activo un empotrable de dibujo, nos aparece una nueva opción en la barra de menús para seleccionar el color de trazado.

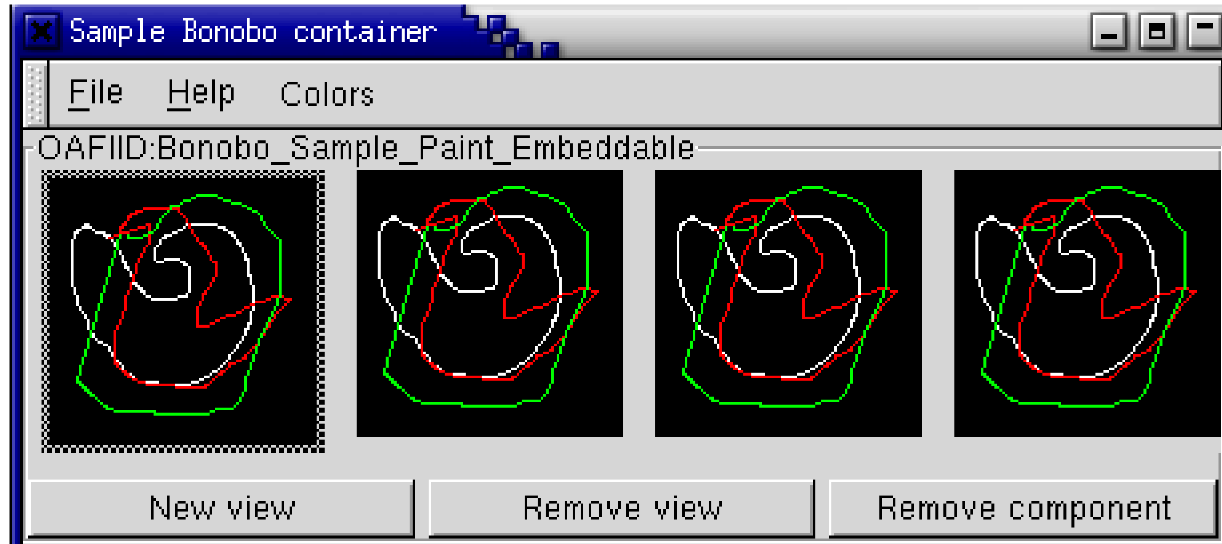



**Figura 16-11. Barra de menús con componente empotrado activo**

La gestión de los menús y la barra de herramientas se lleva a través de un BonoboUIContainer, elemento que hemos utilizado a la hora de crear nuestro contenedor. Dentro de nuestra clase SampleApp, que representa a la aplicación globalmente, junto con el contenedor, la vista actual y algunos widgets, tenemos un BonoboUIContainer, que es el que utilizamos a la hora de crear los menús y barra de herramientas.

```
 struct _SampleApp {
 BonoboItemContainer *container;
 BonoboUIContainer   *ui_container;

 BonoboViewFrame     *curr_view;
 GList               *components;

 GtkWidget           *app;
 GtkWidget           *box;
 GtkWidget           *fileselection;
};
```

Este BonoboUIContainer se crea justo después de crear el contenedor principal:

```
 app->container = bonobo_item_container_new ();

 app->ui_container = bonobo_ui_container_new ();
 bonobo_ui_container_set_win (app->ui_container, BONOBO_WINDOW (app->app));
```

Pero de momento, tenemos el BonoboUIContainer vacío, sin los datos de los menús y de la barra de herramientas. Esta última no existe en nuestro caso, por lo que nos vamos a centrar en los menús. Dentro del método "sample_app_fill_menu" es donde se toman los datos del menú y se llena. Remitimos al lector a la descripción que se dió en el análisis del código del ejemplo para entender como se definen los contenidos de los menús.

## Activación de la vista de un Empotrable

Como ya hemos visto, uno de los momentos más importantes es cuando se activa un empotrable. En ese momento, el usuario tiene que percibir que dicho empotrable ha pasado a estar activo, se deben de sustituir los menús y barras de herramientas por las del empotrable, y este debe de comenzar a recibir los eventos que el usuario provoque sobre él. Podemos ver en la siguiente figura como se muestra que una vista está activada, al aparecer alrededor de ella un borde más grueso.

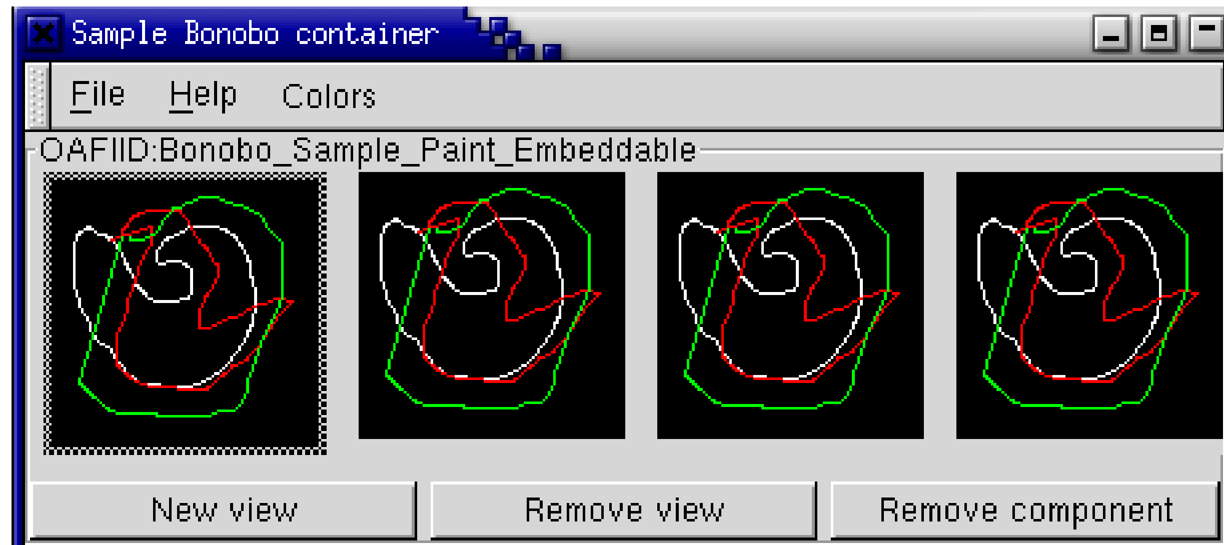



**Figura 16-12. Barra de menús con componente empotrado activo**

Cuando una vistas de un empotrable está desactivada, una capa cubre la zona que ocupa dicha vistas. Esa capa captura todos los eventos que se produzcan en dicha zona. En nuestro caso tratamos dos tipos de eventos. La activación del empotrable al hacer un doble click, o la presentación de un menú si se pulsa el botón derecho sobre el empotrable. Centremos únicamente en la activación:

```
gtk_signal_connect (GTK_OBJECT (view_frame), "user_activate",
       GTK_SIGNAL_FUNC (activate_request_cb),
       site);
```

Vemos que en realidad, la señal se captura utilizando el ViewFrame, el cual es el representante de la vista del empotrable dentro del contenedor. Cuando el usuario provoca "user_activate", que en este caso es hacer doble-click, el componente pasa a ser activado dentro de la función de callback "activate_request_cb". Esta función es parte de "component.c".

```
 static void
activate_request_cb (BonoboViewFrame  *view_frame,
                     SampleClientSite *site)
{
        SampleApp *app;

        g_return_if_fail (site != NULL);
        g_return_if_fail (site->app != NULL);
```

```
        app = site->app;

        if (app->curr_view) {
                bonobo_view_frame_view_deactivate (app->curr_view);
                if (app->curr_view)
                        bonobo_view_frame_set_covered (app->curr_view, FALSE);
        }

        /*
  * Se activa la View sobre la que el usuario ha hecho click.
         * Esto provoca una petición a la View empotrada para que se active.
  * Si la View acepta la activación, se lo notifica al ViewFrame
  * el cual llama a view_activated_cb callback.
         *
         * No descubrimos aquí la View porque puede rechazar la activación.
         * Esperamos a descubrir la vista cuando esta acepta la activación.
         */
        bonobo_view_frame_view_activate (view_frame);
}
```

Bueno, pues a ver el tratamiento que se hace cuando la vista a aceptado ser activada.

```
 static void
view_activated_cb (BonoboViewFrame  *view_frame,
     gboolean          activated,
     SampleClientSite *site)
{
 SampleApp *app = site->app;

 if (activated) {
  /*
         * Si la View que pide ser activada es la actual
                * informamos del hecho y regresamos.
   */
  if (app->curr_view) {
   g_warning (Â¡"La View a activar ya está activa!");
   return;
  }

  /*
   * En otro caso, descubrimos la vista para que pueda recibir
   * eventos, y la ponemos como la View activa.
   */
  bonobo_view_frame_set_covered (view_frame, FALSE);
  app->curr_view = view_frame;
 } else {
  /*
                * Si la View pide ser desactivada, obligar siempre.
         * Podemos haberla desactivado ya (ver user_activation_request_cb),
   * If the View is asking to be deactivated, always
         * pero no ocurre nada malo haciéndolo de nuevo.
                * Siempre está la posibilidad que la vista pida ser
                * desactivada sin que se lo haya pedido el usuario por
                * lo que cubrimos la vista siempre aquí.
   */
  bonobo_view_frame_set_covered (view_frame, TRUE);

  if (view_frame == app->curr_view)
   app->curr_view = NULL;
 }
}
```

De los comentarios del código podemos ver la secuencia de acontecimientos que se producen para poder activar la vista, así como las llamadas que se han de ir haciendo para cubrir y descubrir los componentes empotrables.

## Impresión del contenedor

La impresión del contenedor para el usuario final es la impresión de todo el documento. Ello implica que el contenedor debe de saber como ir imprimiendo cada uno de los empotrados que tiene incluídos dentro de él. Y para llevar a cabo esta tarea, la única forma genérica de lograrlo para cualquier componente empotrable es que estos implementen una interfaz genérica de impresión, que es efectivamente lo que hacen. Implementan la interfaz "IDL:Bonobo/Print". El contenedor puede entonces ir barriendo todos los empotrados y decirles: tu te tienes que imprimir dentro de un contexto de impresión (la página de impresión), en esta posición y con este tamaño. Veamos como efectivamente se sigue este mecanismo.

La localización del código de impresión es sencilla. La impresión se llama desde un menú del contenedor, tal y como podemos ver en la siguiente imagen.

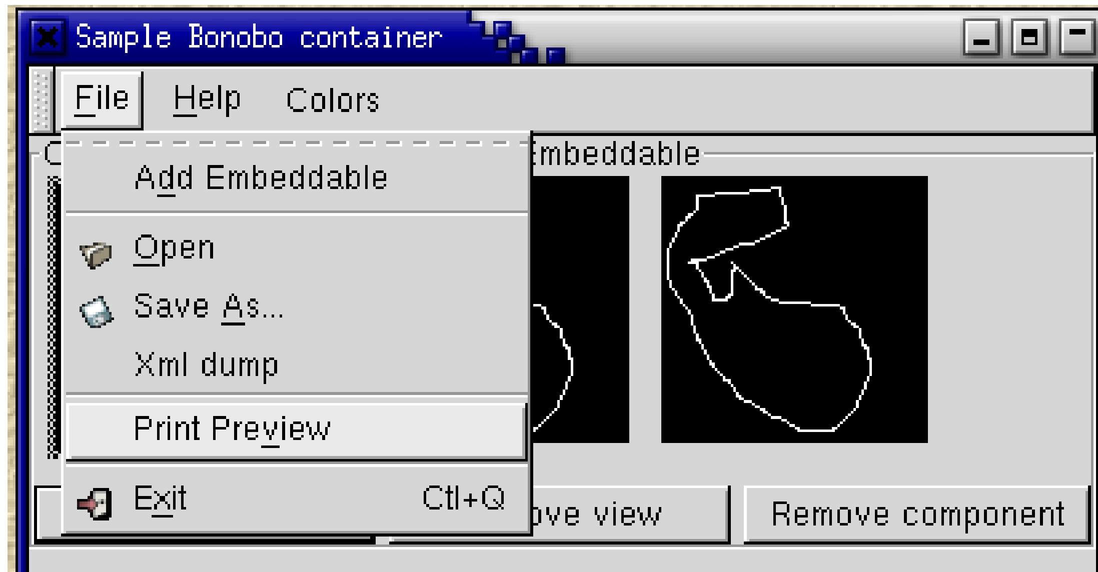



**Figura 16-13. Impresión del contenedor Bonobo**

Por lo tanto, nos vamos a la definición de los menús dentro del contenedor, "container-menu.h", y vemos que la selección de este menú termina provocando la llamada a "verb_PrintPreview_cb". Veamos que hace este método:

```
 static void
verb_PrintPreview_cb (BonoboUIComponent *uic, gpointer user_data, const char *cname)
{
 SampleApp *app = user_data;

 sample_app_print_preview (app);
}
```

lo que nos lleva al método "sample_app_print_preview" que ya está dentro del fichero "container-print.h/c", que las acciones que lleva a cabo son:

```
 void
sample_app_print_preview (SampleApp *app)
{
 GList *l;
 double ypos = 0.0;
 GnomePrintMaster *pm;
 GnomePrintContext *ctx;
 GnomePrintMasterPreview *pv;

 pm = gnome_print_master_new ();
 ctx = gnome_print_master_get_context (pm);

 for (l = app->components; l; l = l->next) {
  BonoboClientSite *site = l->data;

  object_print (bonobo_client_site_get_embeddable (site),
         ctx, 0.0, ypos, 320.0, 320.0);
  ypos += 320.0;
 }

 gnome_print_showpage (ctx);
 gnome_print_context_close (ctx);
 gnome_print_master_close (pm);

 pv = gnome_print_master_preview_new (pm, "Component demo");
 gtk_widget_show  (GTK_WIDGET (pv));
 gtk_object_unref (GTK_OBJECT (pm));
}
```

No nos vamos a parar en este momento a describir el sistema de impresión de GNOME, pero vemos cláramente el bucle en el que se van imprimiendo todos los empotrables con la llamada a "object_print".

Aquí debe ir una captura de pantalla *FIXME*

## Persistencia del contenedor

Al igual que la con la impresión, la persistencia de un contenedor, es decir, el almacenar en un fichero o base de datos o cualquier sistema de almacenamiento persistente a nuestro contenedor Bonobo, incluye que se vayan guardando de forma persistente el estado de cada uno de los empotrables que existan, y para ello de nuevo, todos los empotrables han de implementar la interfaz "IDL:Bonobo/PersistStream".

## Creación de un empotrable

Hasta el momento nos hemos centrado en el contenedor de empotrables, aunque ya hemos visto gran parte de la interacción que se produce entre empotrable y contenedor. Vamos ahora a ver un ejemplo de un empotrable. Y para ello nada mejor que coger el empotrable que pone "Â¡Hola Mundo!". Nos permitirá centrarnos en lo que se necesita para hacer un empotrable sin tener que despistarnos con la funcionalidad propia del empotrable. Este ejemplo va a ser especialmente interesante ya que vamos a mostrar un empotrable que implementa las interfaces:

- IDL:Bonobo/Unknown:1.0 e IDL:Bonobo/Embeddable:1.0, como debe hacer todo componente empotrable.

- IDL:Bonobo/PersistStream:1.0 que nos permite almacenar de forma persistente el empotrable. Esta interfaz, o Bonobo/Persist, debe de ser implementada en general por todos los empotrables ya que, cuando queramos guardar un documento, debemos de poder ir guardando todos los componentes empotrados que incluye, y la única forma de hacer esto es a través de la interfaz Bonobo/PersistStream o Bonobo/Persist. Esta interfaz permite guardar y obtener el estado de un componente utilizando "streams" (flujos) de datos.
- IDL:Bonobo/Persist:1.0 nos permite almacenar el estado de un empotrable a un medio persistente (base de datos, fichero ...).
- IDL:Bonobo/Print:1.0 que nos permite imprimir el empotrable y que al igual que las interfaces de persistencia, debe de ser implementado por los empotrables en general, ya que si queremos poder imprimir un documento completo, debemos de poder imprimir los empotrables, algo que se hace desde este interfaz.

La forma de obtener un empotrable "Hello" es utilizando la factoría de este tipo de empotrables. Esta factoría se haya en el fichero "bonobo-hello.c".

```
#include "config.h"
#include <bonobo.h>

#include "hello-embeddable.h"

static BonoboObject*
hello_embeddable_factory (BonoboGenericFactory *f, gpointer data)
{
        HelloBonoboEmbeddable *embeddable;

        embeddable = gtk_type_new (HELLO_BONOBO_EMBEDDABLE_TYPE);

        g_return_val_if_fail(embeddable != NULL, NULL);

        embeddable = hello_bonobo_embeddable_construct (embeddable);

        return BONOBO_OBJECT (embeddable);
}

BONOBO_OAF_FACTORY ("OAFIID:Bonobo_Sample_Hello_EmbeddableFactory",
                    "bonobo hello", VERSION,
                    hello_embeddable_factory,
                    NULL)
```

Como es tan común que el programa principal de un empotrable sea una factoría que llama a un método que devuelve un nuevo empotrable, ya existe hasta un MACRO que automatiza todo este código, BONOBO_OAF_FACTORY, y al que le pasamos el identificador de la factoría y el método a llamar para obtener nuevos componentes empotrables.

El método "hello_embeddable_factory" nos devuelve el empotrable recién creado. El proceso de creación lo vamos a ver en los siguientes apartados. Vamos a comenzar viendo la definición de la clase "HelloBonoboEmbeddable".

```
struct _HelloBonoboEmbeddable {
 BonoboEmbeddable embeddable;
 char            *text;
};
typedef struct {
 BonoboEmbeddableClass parent_class;
} HelloBonoboEmbeddableClass;
```

Vemos que es una clase muy sencilla, que hereda de BonoboEmbeddable y que sólo añade un campo extra, "text", que quizá se utilice para definir el mensaje a visualizar. Las funciones que exporta esta clase son:

```
GtkType hello_bonobo_embeddable_get_type  (void);

HelloBonoboEmbeddable *
        hello_bonobo_embeddable_construct (HelloBonoboEmbeddable *embeddable);
void    hello_bonobo_embeddable_set_text  (HelloBonoboEmbeddable *embeddable,
        char                  *text);
```

Con ellas podemos crear nuevos empotrables y construirlos, y modificar el texto que visualiza un empotrable. Todo muy sencillo para que nada nos distraiga del objetivo de aprender a programar componentes Bonobo empotrables.

Vamos con la implementación de estos métodos:

```
static void
hello_bonobo_embeddable_class_init (BonoboEmbeddableClass *klass)
{
 GtkObjectClass *object_class = (GtkObjectClass *) klass;

 hello_bonobo_embeddable_parent_class =
  gtk_type_class (bonobo_embeddable_get_type ());

 object_class->destroy = hello_bonobo_embeddable_destroy;
}
```

Este es el constructor de la parte del objeto común a todos los objetos de la clase (métodos y variables estáticos, en terminología Java), que se resume en crear un nuevo objeto de la clase padre, BonoboEmbeddable.

```
 static void
hello_bonobo_embeddable_init (BonoboObject *object)
{
 HelloBonoboEmbeddable *embeddable = HELLO_BONOBO_EMBEDDABLE (object);

 embeddable->text = g_strdup ("Hello World");
}
```

Y aquí tenemos la parte específica por objeto del constructor, en el que vemos que se inicializa la cadena de caracteres a mostrar.

```
BONOBO_X_TYPE_FUNC (HelloBonoboEmbeddable,
        bonobo_embeddable_get_type (),
        hello_bonobo_embeddable);
```

La construcción de este tipo de clases es tan común que se han creado macros para automatizar la labor de creación del tipo (clase). De esta forma, se logra que sea un poco menos laboriosa la creación de nuevas clases, algo pesado a nivel de sintaxis y más después de ver como en lenguajes orientados a objetos como Java o C++, todo esto no es necesario.

Una vez que se creaba un objeto nuevo HelloBonoboEmbeddable dentro del método de la factoría, vimos que se llamaba al método "hello_bonobo_embeddable_construct" el cual es el que termina de construir el empotrable. Vamos con este método:

```
 HelloBonoboEmbeddable *
hello_bonobo_embeddable_construct (HelloBonoboEmbeddable *embeddable)
{
 BonoboPersistStream *stream;
```

```
BonoboPrint          *print;

g_return_val_if_fail (HELLO_BONOBO_IS_EMBEDDABLE (embeddable), NULL);

bonobo_embeddable_construct (BONOBO_EMBEDDABLE (embeddable),
        hello_bonobo_view_factory, NULL);
```

De la descripción de la API leemos que esta rutina se encarga de construir un servidor CORBA Bonobo::Embeddable y activarlo, de forma que cuando se pidan nuevas vistas del empotrable por ejemplo, se redirijan las peticiones al método adecuado.

```
/* Registrar la interfaz Bonobo::PersistStream*/
stream = bonobo_persist_stream_new (hello_object_pstream_load,
        hello_object_pstream_save,
        hello_object_pstream_get_max_size,
        hello_object_pstream_get_types,
        embeddable);
```

Comenzamos a ver lo que significa implementar la interfaz de persistencia, que consiste básicamente en especificar como cargar y guardar la persistencia de empotrable, en concreto, los métodos a ser llamados cuando el contenedor pida por la interfaz Bonobo::PersistStream que se guarde o se obtenga el estado del componente empotrado.

```
if (!stream) {
 bonobo_object_unref (BONOBO_OBJECT (embeddable));
 return NULL;
}
```

Si no logramos crear el objeto de persistencia, eliminamos la referencia que teníamos al empotrable (si nadie más lo está utilizando se destruiría) y devolvemos NULL.

```
bonobo_object_add_interface (BONOBO_OBJECT (embeddable),
        BONOBO_OBJECT (stream));
```

Hemos logrado crear el objeto que implementa la interfaz de persistencia del empotrable, por lo que agregamos la interfaz de este objeto de persistencia a las que soporta el empotrable. Aquí se refleja cláramente la idea de que la funcionalidad de un objeto Bonobo se va logrando por agregación y no por otros mecanismos como la herencia. Esta es una de las claves del modelo de componentes OLE2, y se basa en la idea de que en los sistemas distribuidos en los que la evolución y la flexibilidad son claves para las tecnologías, la agregación es un mecanismo mucho más flexible que la herencia. Por ejemplo, en tiempo de ejecución, las interfaces de los objetos se pueden cambiar por agregación sin tener que tocar para nada la implementación de un objeto. O un caso aún más atractivo: podemos ir cambiando las interfaces de los objetos pero manteniendo también las interfaces antiguas. De esta forma, un mismo objeto puede ir evolucionando pero manteniendo compatibilidad 100% con sus interfaces pasadas. De esta forma, se pueden por ejemplo actualizar sistemas de una forma mucho más gradual.

```
/* Registrar la interfaz the Bonobo::Print */
print = bonobo_print_new (hello_object_print, embeddable);
```

Aquí es donde creamos el objeto que se encargará de las labores de impresión del componente empotrable.

```
if (!print) {
```

```
  bonobo_object_unref (BONOBO_OBJECT (embeddable));
  return NULL;
 }
```

Si no logramos crear el objeto de impresión, eliminamos la referencia que teníamos al empotrable (si nadie más lo está utilizando se destruiría) y devolvemos NULL.

```
 bonobo_object_add_interface (BONOBO_OBJECT (embeddable),
         BONOBO_OBJECT (print));
```

Si el objeto de impresión se ha creado de forma correcta, añadimos la interfaz de impresión entre las soportadas por nuestro empotrable. Y con ello ya tenemos lista nuestro nuevo empotrable para ser utilizado.

```
 return embeddable;
}
```

Devolvemos nuestro nuevo empotrable para que puede ser utilizado dentro de la aplicación.

## Persistencia de un Empotrable

La persistencia del empotrable la creamos en:

```
      /* Registrar la interfaz Bonobo::PersistStream*/
 stream = bonobo_persist_stream_new (hello_object_pstream_load,
         hello_object_pstream_save,
         hello_object_pstream_get_max_size,
         hello_object_pstream_get_types,
         embeddable);
```

La implementación de todos estos método se realiza dentro de "hello-object-io.h/c"

## Impresión de un Empotrable

Como hemos visto, la impresión de un empotrable se logra creando el objeto:

```
 /* Registrar la interfaz the Bonobo::Print */
 print = bonobo_print_new (hello_object_print, embeddable);
```

Por lo tanto, vamos a irnos a "hello-object-print.h/c" para ver como se implementa esta interfaz de impresión, que en nuestro caso tan sólo nos mostrará el texto asociado a la variable "text" de nuestro empotrable. Sólo hay una función pública de esta clase:

```
 void hello_object_print (GnomePrintContext         *ctx,
    double                    width,
    double                    height,
    const Bonobo_PrintScissor *scissor,
    gpointer                  user_data);
```

que lo que hará es pintar dentro del "GnomePrintContext", en la posición indica por "width" y "height" el texto a ser impreso. Vamos a ver la implementación de este método:

```
 void
hello_object_print (GnomePrintContext         *ctx,
      double                     width,
      double                     height,
      const Bonobo_PrintScissor *scissor,
      gpointer                   user_data)
{
 HelloBonoboEmbeddable *embeddable = user_data;
 GnomeFont             *font;
 double                 w, w2, h;
 const char            *str, *descr;

 str = embeddable->text ? embeddable->text : "No hay texto";
```

Si no hay ningún texto, vamos a imprimir "No hay texto".

```
 descr = "Valor:";

 gnome_print_setlinewidth (ctx, 2);
 font = gnome_font_new ("Helvetica", 12.0);
```

Elegimos el tipo de fuente y su tamaño.

```
 g_return_if_fail (font != NULL);
 gnome_print_setrgbcolor (ctx, 0.0, 0.0, 0.0);
 gnome_print_setfont (ctx, font);
```

Ponemos el color y el tipo de fuente al contexto de impresión actual.

```
 w = gnome_font_get_width_string (font, descr);
 w2 = gnome_font_get_width_string (font, str);
 h =
     gnome_font_get_ascender (font) +
     gnome_font_get_descender (font);
```

Obtenemos la anchura de las cadenas a imprimir así como la altura.

```
 gnome_print_moveto (ctx, (width / 2) - (w / 2), (height / 2) + h * 2);
 gnome_print_show (ctx, descr);
```

Nos situamos y pintamos la cadena "Valor:".

```
 gnome_print_moveto (ctx, (width / 2) - (w2 / 2), height / 2 - h);
 gnome_print_show (ctx, str);
```

Nos situamos y pintamos la cadena que resida en "str".

```
 gtk_object_unref (GTK_OBJECT (font));
}
```

Liberamos el objeto fuente que ya no vamos a utilizar más.

Como vemos, la impresión es bastante sencilla en el concepto. Los detalles en concreto de como se lleva a cabo la impresión pueden ser más o menos complejos, pero siempre la metodología es la misma.

## Contenedores (BonoboWindow)

## Monikers

Los "monikers" provienen de la terminología de OLE2, el sistema de componentes de MS Windows. En Bonobo, se han tomado las ideas iniciales de MS a la hora de desarrollarlos, y se han añadido extensiones que nos llevan a lo que es hoy el sistema de "monikers" de Bonobo: un sistema de referenciación de objetos.

Como podremos comprobar cuando hayamos terminado de leer este artículo, el sistema de "monikers" se integra a la perfección en un entorno basado en componentes, como es el caso de Bonobo, y, por consiguiente, de GNOME.

En un entorno basado en componentes, es necesario el poder activar fácilmente objetos desde nuestros programas. Para ello, vimos en artículos anteriores que tenemos la posibilidad de usar OAF directamente (nivel más bajo de la aquitectura, junto con ORBit), y que Bonobo tambien incluía las funciones necesarias para facilitarnos la vida a la hora de activar componentes desde nuestros programas. Sin embargo, este sistema es insuficiente si queremos tener un acceso más detallado a los componentes instalados en nuestro sistema, así como la información manejada por ellos.

### Representación de los monikers

Los "monikers" son representados por una cadena de texto, y son activados mediante la función `bonobo_get_object`. Así, el formato de un moniker es el siguiente:

```
método:representación_del_moniker
```

donde "método" es la cadena que define el moniker de más alto nivel que queremos activar. Â¿Alto nivel? Bien, los "monikers" pueden ser activados "en cadena", es decir, que, dada una cadena de texto representando un moniker, ésta puede activar un moniker, y éste, a su vez, puede activar a su vez otros "monikers" (y estos a su vez, activar otros, etc). Así, observemos varios ejemplos:

- `oafiid:GNOME_Evolution_CalendarFactory:1.0`: representa una instancia del componente cuyo OAFIID (Identificador de la implementación de un objeto OAF) sea `GNOME_Evolution_CalendarFactory:1.0`.
- `oafiid:GNOME_Evolution_CalendarFactory:1.0:new`: representa una NUEVA instancia (al contrario que el anterior moniker, que sólo activaría una nueva instancia si no existiera ninguna) del componente especificado.
- `file:/home/admin/sales.gnumeric!sheet1`: representa la hoja llamada "sheet1" de la hoja de cálculo almacenada en el fichero `/home/admin/sales.gnumeric`.
- `file:/tmp/download.gz`: representa el fichero `/tmp/download.gz`.
- `file:/tmp/download.gz#gunzip`: representa el contenido descomprimido del fichero `/tmp/download.gz`.

En esta lista, se observa que no hay un formato bien definido de cómo se separan los distintos monikers dentro de la cadena de representación de los mismos, excepto con el moniker de más alto nivel, que siempre consiste en el nombre del moniker seguido del carácter ’:’. Esto es completamente intencionado, de forma que cada moniker pueda usar la sintaxis que más le plazca, lo que permite al sistema de "monikers" de Bonobo integrarse a la perfección en otros sistemas de referenciación estándar como pueden ser, por ejemplo, los protocolos HTTP, FTP, etc. De esta forma, podríamos tener, por ejemplo, los siguientes monikers:

- `ftp://ftp.gnome.org`
- `http://www.gnome.org/bonobo.html#Section1`

En este caso, el "moniker" http utiliza el carácter '#' para la separación de los distintos componentes del "moniker" a activar, de esta forma, integrándose perfectamente en la sintaxis usada en el protocolo HTTP.

Todas estas cadenas expuestas anteriormente nos permiten la referenciación del objeto en cuestión que queremos activar. Pero lo que hace realmente interesante el uso de "monikers" en Bonobo es que, a la hora de activar un "moniker", no sólo especificamos la cadena que representa al objeto que queremos activar, sino que además especificamos la "forma" que tiene que tener ese objeto cuando nos sea devuelto. Todo esto lo veremos mejor con algunos ejemplos:

```
objeto = bonobo_get_object("file:/tmp/download.gz#gunzip", "IDL:Bonobo/Stream:1.0");
objeto = bonobo_get_object("http://www.gnome.org", "IDL:Bonobo/Control:1.0");
objeto = bonobo_get_object("config:/gnome/background/image", "IDL:Bonobo/Property:1.
```

Aquí se puede observar claramente la utilidad de los "monikers", como es la posibilidad de especificar el comportamiento exacto que deseamos del objeto a activar, aparte de la posibilidad de tratar a un mismo objeto de distintas maneras. Consideremos el siguiente ejemplo:

```
objeto = bonobo_get_object("http://www.gnome.org", "IDL:Bonobo/Stream:1.0");
```

En este caso, estamos pidiéndole al "moniker" http un Bonobo/Stream (que es un objeto desde el que podemos leer datos) con el contenido de la página referenciada por "www.gnome.org". Esto nos puede ser muy útil si, por ejemplo, en nuestro programa queremos obtener ficheros de servidores HTTP, datos que luego, de alguna manera, procesaremos en nuestro programa. Sin embargo, tambien puede darse el caso de que, en algun momento dado de nuestro programa, queramos mostrar el contenido de una página web. Una solución, tambien proporcionada por Bonobo, sería usar el componente de edición de GtkHTML. Otra, sería intentar activar un "moniker" que nos devuelva un visualizador HTML (un control Bonobo) mostrando el contenido de la página que hayamos pedido en la activación del "moniker". En código, esto se traduciría a:

```
objeto = bonobo_get_object("http://www.gnome.org", "Bonobo/Control:1.0");
```

Con esta línea de código, obtendríamos, en la variable "objeto", un control Bonobo que podría ser perfectamente el componente GtkHTML mencionado anteriormente.

## bonobo-conf

Uno de los ejemplos más significativos de los "monikers" en Bonobo, es, sin duda, bonobo-conf, un proyecto que ha nacido con la intención de incluir en el modelo de componentes en que se está convirtiendo GNOME (gracias a Bonobo) el acceso a toda la configuración del usuario. Bonobo-conf es la implementación del "moniker" config:, del cual veiamos antes algunos ejemplos.

Bonobo-conf soporte la activación de componentes de tres formas (o interfaces) distintas:

- Bonobo:Property:1.0: de esta forma, obtendremos un objeto a través del cual podremos leer y modificar la información asociada con una entrada específica de nuestra configuración:

  ```
  Bonobo_Property prop;
  ```

```
CORBA_Any value;
CORBA_Environment ev;

property = bonobo_get_object("config:/gnome/UI/BackgroundImage", "IDL
value = bonobo_property_get_value(property, &ev);
```

- Bonobo:PropertyBag:1.0: Bonobo::PropertyBag es un interfaz que contiene una lista de Bonobo::Property’s. Por tanto, es especialmente útil en bonobo-conf, pues nos permite obtener una lista de propiedades almacenadas en nuestro sistema de configuración.
- Bonobo:Control:1.0: este interfaz es el más interesante de todos, pues permite activar un control Bonobo que presenta una interfaz gráfica para configurar la opción de configuración seleccionada en el "moniker". Esto nos permitirá, por ejemplo, construir nuestras ventanas de configuración dinámicamente, simplemente reservando un espacio para cada uno de los controles Bonobo que activemos a través de bonobo-conf.

Bonobo-conf es una de las implementaciones de "monikers" más completas a fecha de hoy, por lo que es aconsejable que indaguemos en su código fuente para conocer más a fondo los entresijos de la implementación de "monikers".

## Implementación de monikers

Para aprender cómo se implementan los "monikers" nos vamos a fijar en uno de los "monikers" estándar que se incluyen en Bonobo: el moniker http, que, como hemos visto anteriormente, nos permite referenciar objetos que se crean a partir del contenido de una petición HTTP.

El "moniker" http podemos encontrarlo incluido con el resto de fuentes de Bonobo, en el directorio `monikers`, exactamente en los ficheros `bonobo-moniker-http.h` y `bonobo-moniker-http.c`. Este último es el que más nos interesa, pues en él está el código de implementación del "moniker".

Para empezar, podremos observar, al final de dicho archivo, las siguientes líneas:

```
BONOBO_OAF_FACTORY ("OAFIID:Bonobo_Moniker_http_Factory",
                    "http-moniker", VERSION,
                    bonobo_moniker_http_factory,
                    NULL)
```

Esta línea, que puede parecer un poco extraña en un programa en C, no es más que una macro (definida con #define) que genera la función `main` necesaria para la creación de una factoría. Esta macro está definida en el fichero `bonobo-generic-factory.h` de la siguiente manera:

```
#define BONOBO_OAF_FACTORY(oafiid, descr, version, fn, data)
int main (int argc, char *argv [])
{
             BonoboGenericFactory *factory;
             CORBA_Environment ev;
             CORBA_ORB orb;

             CORBA_exception_init (&ev);
             gnome_init_with_popt_table (descr, version, argc, argv,
                                         oaf_popt_options, 0, NULL);
             orb = oaf_init (argc, argv);
             if (!bonobo_init (orb, CORBA_OBJECT_NIL, CORBA_OBJECT_NIL))
                     g_error (_("Could not initialize Bonobo"));
             factory = bonobo_generic_factory_new (oafiid, fn, data);
```

```
                bonobo_running_context_auto_exit_unref (BONOBO_OBJECT (factory));
                bonobo_main ();
                CORBA_exception_free (&ev);
                return 0;
}
```

Como se puede observar, el código generado es exactamente el mismo que escribíamos en artículos anteriores para crear nuestras factorias.

Así pues, con esta macro hemos creado nuestra factoría de objetos, cuya función (`bonobo_moniker_http_factory`) será llamada cada vez que una aplicación desee crear un "moniker" http. Dicha función tiene la siguiente forma:

```
static BonoboObject *
bonobo_moniker_http_factory (BonoboGenericFactory *this, void *closure)
{
     return BONOBO_OBJECT (bonobo_moniker_simple_new (
                           "http:", http_resolve));
}
```

Como se puede ver, en la función de creación de objetos de nuestra factoría simplemente llamamos a la función `bonobo_moniker_new`, que devuelve un BonoboMoniker. Esta función recibe dos parámetros, el primero es la cadena que representa al "moniker" que estamos creando, y que identificará a nuestro "moniker" a la hora de la activación. El segundo parámetro es un puntero a una función, que será la encargada de devolver un objeto cuando una aplicación esté pidiendo la activación de un "moniker" http. Vamos a ver esta función:

```
static Bonobo_Unknown
http_resolve (BonoboMoniker *moniker,
                   const Bonobo_ResolveOptions *options,
                   const CORBA_char *requested_interface,
                   CORBA_Environment *ev)
{
                const char *url = bonobo_moniker_get_name (moniker);
                char *real_url;

                g_warning ("Going to resolve the http now");

                /* because resolving the moniker drops the "http:" */
                real_url = g_strconcat ("http:", url, NULL);

                if (strcmp (requested_interface, "IDL:Bonobo/Control:1.0") == 0) {
                        BonoboObjectClient *client;
                        Bonobo_Unknown object;

                        client = bonobo_object_activate ("OAFIID:GNOME_GtkHTML_EBrowser
                        if (!client) {
                                /* FIXME: Set a InterfaceNotFound exception here? */
                                return CORBA_OBJECT_NIL;
                        }

                        object = BONOBO_OBJREF (client);

                        if  (ev->_major != CORBA_NO_EXCEPTION)
                                return CORBA_OBJECT_NIL;

                        if  (object == CORBA_OBJECT_NIL) {
                                g_warning ("Can't find object satisfying requirements")
                                CORBA_exception_set  (
                                        ev, CORBA_USER_EXCEPTION,
                                        ex_Bonobo_Moniker_InterfaceNotFound, NULL);
                                return CORBA_OBJECT_NIL;
```

```
                }

                return bonobo_moniker_util_qi_return (object, requested_interfa
        }
        else if (strcmp (requested_interface, "IDL:Bonobo/Stream:1.0") == 0) {
                BonoboStream *stream;

                stream = bonobo_stream_open_full (
                        "http", real_url, Bonobo_Storage_READ, 0644, ev);

                if (!stream) {
                        g_warning ("Failed to open stream '%s'", real_url);
                        g_free (real_url);
                        CORBA_exception_set (
                                ev, CORBA_USER_EXCEPTION,
                                ex_Bonobo_Moniker_InterfaceNotFound, NULL);
                        return CORBA_OBJECT_NIL;
                }

                g_free (real_url);
                return CORBA_Object_duplicate (BONOBO_OBJREF (stream), ev);
        }

        return CORBA_OBJECT_NIL;
}
```

Si leemos detenidamente el código de esta función, veremos que la implementación de "monikers" es realmente sencilla. Con estas pocas líneas, estamos realizando un montón de cosas, todo ello gracias a la enorme funcionalidad del API de Bonobo.

Lo primero que debemos hacer es comparar el parámetro *requested_interface* con los interfaces que nuestro "moniker" maneje. Es decir, este parámetro contendrá el valor que se le haya pasado como segundo parámetro a la función `bonobo_get_object` en la aplicación que está intentando cargar el objeto a través de nuestro "moniker". Por tanto, y puesto que la aplicación que está cargando el objeto espera que dicho objeto implemente determinado interfaz, debemos prestar especial atención a este parámetro, pues deberemos crear el objeto apropiado para cada ocasión. Por tanto, podemos deducir que el "moniker" http permite la creación de objetos (todos ellos basados, de alguna forma, en el contenido de recursos obtenidos via protocolo HTTP) de dos tipos: `IDL:Bonobo/Control:1.0` y `IDL:Bonobo/Stream:1.0`.

El primero de ellos (`IDL:Bonobo/Control:1.0`) realiza una tarea muy útil, que consiste en devolver un control Bonobo que implementa un visualizador HTML. En este caso, lo que hace es crear una instancia del componente `GNOME_GtkHTML_EBrowser`, que viene incluido junto con GtkHTML, y que es usado, por ejemplo, en Evolution. Si usamos de esta forma el "moniker" http desde nuestras aplicaciones, podremos fácilmente cargar un visualizador de HTML e insertarlo en nuestras ventanas. Para realizar eso, en la implementación del "moniker" lo único que hay que hacer es activar el objeto que queremos, mediante la llamada a la función `bonobo_object_activate`, y luego, simplemente obtener una referencia a la implementación del interfaz que queramos (en este caso, `IDL:Bonobo/Control:1.0`) y devolver esa referencia al sistema de "monikers", todo ello mediante la llamada a la función `bonobo_moniker_util_qi_return`. Esto último es debido a que, como vimos en artículos anteriores, `bonobo_object_activate` devuelve un puntero a un `BonoboObjectClient`, y nosotros queremos la referencia a un determinado interfaz implementado por ese objeto, para lo que tenemos que hacer uso del método `queryInterface` del interfaz `Bonobo::Unknown`, que es precisamente lo que hace la función `bonobo_moniker_util_qi_return`.

En el segundo caso, el "moniker" http permite la creación de `BonoboStream` a partir del contenido de una petición HTTP. Con este sistema, se podría implementar fá-

cilmente un servidor de caché, que aceptara peticiones HTTP de otras aplicaciones, y que o bien obtuviera los datos directamente de Internet, o de una copia local, si tuviera una disponible.

Como vemos, para la creación de `BonoboStream`, Bonobo tambien incluye las funciones necesarias para facilitarnos la vida. En este caso, simplemente usamos la función `bonobo_stream_open_full`, que crea un `BonoboStream` a partir de los datos que le pasamos. Luego, simplemente llamamos a la función de ORBit `CORBA_Object_duplicate` para crear un duplicado del objeto CORBA creado con el `BonoboStream`.

Tras la explicación de este código, vemos que la mayor parte del código que tenemos que escribir para implementar un "moniker" está más bien relacionado con la gestión de errores, pues la creación/activación de objetos en Bonobo es de lo más sencilla.

### Otros "monikers"

Desde que se estabilizó el API del sistema de "monikers" de Bonobo, han ido apareciendo implementaciones de distintos "monikers" que ofrecen funcionalidad muy diversa. Forman parte de la distribución de Bonobo los llamados "monikers" estándar, que son: http, gunzip, file, oaf, cache, item y query, algunos de los cuales han sido comentados en este artículo.

Por otro lado, en otros programas basados en Bonobo se están empezando a incluir "monikers" para permitir el acceso a la información gestionada por dichos programas a través de este novedoso sistema de activación de componentes. Entre estos programas, destaca sin duda gPhoto, que implementa el "moniker" camera, que permite el acceso a las fotos de nuestra cámara digital. Tambien destaca el "moniker" database, implementado como parte de GNOME-DB, aunque éste aun no está del todo funcional, por lo que su utilidad de momento es mínima.

## Notas

1. http://primates.ximian.com/~michael/bonobo-docs/book1.html
2. http://primates.ximian.com/~michael/bonobo-docs/bonobo-documents.html

# Capítulo 17. XML en GNOME

XML es un formato, estándar, avalado por el W3C, que permite estructurar la información lógicamente. En contra de otros lenguajes de "marcación", como HTML, por ejemplo, XML no especifica la forma en la que la información debe ser mostrada, sino que sólo se ocupa de estructurarla de una forma lógica.

Esta característica (almacenamiento de información estructurada), añadida a la independencia de la forma de presentación de la información, hace de XML un formato ideal para almacenar datos en las aplicaciones, a la vez que, al ser un estándar reconocido internacionalmente, y, por tanto, soportado en multitud de aplicaciones, es tambien ideal para la compartición de datos entre aplicaciones, incluidas aplicaciones para distintos sistemas operativos.

Por todo esto, el proyecto GNOME, siguiendo con su afán de únicamente utilizar estándares reconocidos, vio en XML la herramienta ideal para el almacenamiento de datos. Y así, ya desde las primeras versiones, multitud de aplicaciones desarrolladas para el proyecto GNOME hacen uso de XML, como por ejemplo, Gnumeric, Dia, GNOME-DB, AbiWord, etc. Estos usos van desde el simple almacenamiento de datos de configuración (GConf), pasando por el formato de ficheros propio (Glade, Dia, Gnumeric), hasta usos más avanzados como la transferencia de datos entre SGBDR (GNOME-DB) o mecanismos de RPC ("Remote Procedure Call", Llamadas a procedimientos remotos) recientemente incorporados con la liberación de Soup, que es una implementación de SOAP desarrollada por Ximian.

## libxml (o GNOME-XML)

Como ha ocurrido con otras partes de la arquitectura, es dentro del propio proyecto GNOME donde se ha implementado todo lo necesario para integrar XML dentro de la arquitecura, y así, está disponible libxml, tambien conocida como GNOME-XML, que es una librería que implementa un analizador sintáctico XML que permite fácilmente hacer uso de XML en aplicaciones. Es de agradecer que, a pesar de haber sido desarrollada en el seno del proyecto GNOME, esta librería no tiene ninguna dependencia con este entorno, por lo que puede ser usada perfectamente en aplicaciones totalmente ajenas a GNOME, si así se desea.

### Características

libxml incluye todo lo que cabría esperar de una librería de estas características. Es decir, con libxml se puede leer, validar y modificar documentos XML, todo ello de una forma bastante clara y sencilla.

Para manejar documentos XML con libxml, se utilizan dos tipos de datos básicos: xmlDocPtr, usado para representar un documento XML, y xmlNodePtr, que se usa para representar cada uno de los nodos contenidos en los documentos XML procesados en los programas. Estos dos tipos de datos son simplemente punteros a dos estructuras que estan definidas en los ficheros de cabecera de libxml, y más concretamente en el fichero `tree.h`. Estas estructuras son xmlDoc y xmlNode, cuyo contenido debe ser conocido, pues a la hora de trabajar con libxml, se usa continuamente para acceder a las propiedades de los objetos XML (documentos, nodos, etc) asociados.

Aunque este libro se centra en la libxml2, que es la usada en todo GNOME 2, es importante conocer que existen una versión anterior de esta librería, conocida como libxml1, que aun es usada por multitud de aplicaciones (muchas de ellas ajenas al proyecto GNOME). Por ello, es indispensable conocer alguna de las diferencias entre las dos versiones, para evitar problemas de compilación segun se use una u otra versión. Una de estas diferencias radica en los nombres de algunos campos de estas estructuras comentadas anteriormente. Así, se deben usar una serie de #define's que se encuentran en los ficheros de cabecera ambas versiones, que permiten usar el mismo código para las dos versiones. Estos #define's son:

- *xmlChildrenNode*, usado para acceder a los nodos hijo de un determinado nodo (xmlNode).
- *xmlRootNode*, usado para acceder al nodo raiz de un documento XML (xmlDoc).

Se deberan usar estos nombres para acceder a los campos especificados de las estructuras, para así asegurar que la aplicación escrita va a poder ser compilada sin cambios con cualquiera de las dos versiones de libxml.

Otra cosa a destacar es que libxml no usa Capítulo 4GLib, por lo que no usa los tipos de datos definidos en esta librería. Más bien, incluso define los suyos propios, como es el caso de xmlChar, que es el equivalente, en libxml del gchar de glib (y del char de C). Por supuesto, ambas librerías son perfectamente compatibles, por lo que se pueden usar las dos al mismo tiempo sin ningun problema. De hecho, eso es lo que se va a hacer en los ejemplos de código de este artículo.

Hechas estas aclaraciones, es tiempo de pasar al uso "real" de la librería libxml, para ello, se usará el siguiente ejemplo:

Hay que almacenar en disco los datos de nuestra aplicación de gestión de libros. Para almacenar los datos de cada libro, se usa una estructura definida de la siguiente manera:

```
typedef struct _libro libro;

struct _libro {
      gchar *titulo;
      gchar *autor;
      gchar *tema;
}
```

Se quiere guardar la información en un documento XML. Para ello, se podría crear el siguiente documento:

```
<listado_libros>
  <libro autor="José Ortega y Gasset" titulo="La rebelión de las masas" tema="Filoso
</listado_libros>
```

o bien este otro:

```
<listado_libros>
  <libro>
    <titulo>La rebelión de las masas</titulo>
    <autor>José Ortega y Gasset</autor>
    <tema>Filosofia</tema>
  </libro>
</listado_libros>
```

A lo largo del capítulo se explicará la forma de hacerlo de las dos maneras.

## Carga de documentos

Un documento XML es, a fin de cuentas, una larga cadena de caracteres. Es decir, que un documento XML no tiene porqué estar asociado a un fichero en el disco, aunque, por supuesto, puede estarlo. Así, libxml ofrece dos funciones para permitir la carga de documentos XML tanto leidos del disco, como desde una posición de memoria previamente inicializada (por ejemplo, si se obtuviera el documento XML a través

de Internet, sería almacenado en una posición de memoria determinada). Estas dos funciones son:

```
xmlDocPtr xmlParseMemory (char *buffer, int size);
xmlDocPtr xmlParseFile (const char *filename);
```

La diferencia entre ellas, como ya se ha comentado anteriormente, es que una (`xmlParseMemory`) carga el documento desde una posición de memoria específica (*buffer*), mientras que la otra (`xmlParseFile`) lo hace desde el fichero especificado como parámetro.

Así, por ejemplo, para cargar un fichero XML, se usaría el siguiente código:

```
xmlDocPtr doc;

doc = xmlParseFile("/home/rodrigo/gastos.xml");
if (!doc) {
  g_print("Error al cargar documento XML\n");
}
```

Tras ejecutar este código, y si no se produce ningun error, se obtiene, en la variable *doc*, una referencia al documento XML. Esta variable, de tipo xmlDocPtr, va a permitir el acceso al contenido del documento XML cargado, en este caso, del fichero `/home/rodrigo/gastos.xml`.

Similarmente, si el documento no está en un fichero, sino que, por ejemplo, se obtiene a través de Internet, como hemos mencionado antes, usariamos el siguiente código:

```
xmlDocPtr doc;
gchar *buffer;

/* llamamos a la función (ficticia) que nos devuelve el documento XML */
buffer = obtener_documento_xml("http://mi.servidor.com/gastos.xml");
doc = xmlParseMemory(buffer, strlen(buffer));
...
```

Una vez que se tiene el documento XML cargado en memoria, es decir, cargado en una variable de tipo xmlDocPtr, ya sólo queda, para terminar de procesar el documento, acceder a su contenido. Para ello, se usan los campos de la estructura xmlDoc y xmlNode.

Así, lo primero que se tiene que hacer es obtener una referencia al nodo raiz del documento. Para ello, se usa la función `xmlDocGetRootElement`:

```
xmlNodePtr root;

root = xmlDocGetRootElement(doc);
```

La función `xmlDocGetRootElement` devuelve un valor de tipo xmlNodePtr, que no es más que un puntero al nodo raiz del documento. A su vez, cada nodo XML puede tener otros nodos colindantes (es decir, al mismo nivel), así como otros nodos dependiendo de él (es decir, nodos hijo). Así, para acceder a toda esta información almacenada en nodos, se usa, como es de esperar, el tipo xmlNodePtr. Por ejemplo, para recorrer la lista de nodos de primer nivel del documento XML que hemos cargado, se usaría el siguiente código:

```
xmlDocPtr doc;
xmlNodePtr root;
```

```
xmlNodePtr node;

/* abrimos documento y obtenemos referencia al nodo raiz */
doc = xmlParseFile ("/home/rodrigo/gastos.xml");
root = xmlDocGetRootElement (doc);
node = root->xmlChildrenNode;
while (node != NULL) {
  g_print ("Encontrado nodo %s\n", node->name);
  node = node->next;
}
```

Como se aprecia en este ejemplo, se accede directamente a los campos de la estructura xmlNode. Más concretamente, primero se usa

```
root->xmlChildrenNode
```

, que devuelve una referencia al primer hijo de este nodo, en este caso el nodo raiz, para luego usar *node->name*, que contiene el nombre de la etiqueta XML del nodo al que se hace referencia. Así, por ejemplo, en el caso de la etiqueta <libro>, *node->name* contendría el valor "libro". Seguidamente, se usa *node->next*, que apunta al siguiente nodo en la lista, es decir, al siguiente nodo colindante con el actual.

Cada nodo puede tener a su vez otros nodos dependiendo de él. Así, si este fuera el caso, se podría cambiar el código anterior por lo siguiente:

```
...
while (node != NULL) {
  xmlNodePtr children;
  g_print("Encontrado nodo %s\n", node->name);

  /* procesamos los hijos de este nodo */
  children = node->xmlChildrenNode;
  while (children != NULL) {
    g_print("\tEncontrado hijo %s\", children->name);
    children = children->next;
  }
  node = node->next;
}
```

Por supuesto, este bucle puede tornarse casi infinito, pues a su vez, los nodos hijos de *node* (*children*) pueden a su vez contener otros nodos, y éstos otros, y así sucesivamente.

Siguiendo con el ejemplo, se va a crear una función que cargue el documento que especificad como parámetro y devuelva un puntero a una estructura xmlDoc.

```
xmlDocPtr xml_open_file (const gchar *filename) {
     xmlDocPtr doc;

     doc = xmlParseFile (filename);

     if (!doc) {
          g_print ("Error al cargar el documento %s \n", filename);
          return NULL;
     }

     return doc;
}
```

Cada entrada del documento se corresponde con una estructura de tipo libro, así que se creará una función que tendrá como parámetro de entrada el nodo a leer y que

devolverá la estructura libro que contiene. Para cargar el primero de los formatos propuestos, se usa el siguiente código:

```
libro *xml_get_entry (xmlNodePtr child)
{
     libro *lb;

     lb = g_new0 (libro, 1);

     lb->titulo = xmlGetProp (child, "titulo");
     lb->autor = xmlGetProp (child, "autor");
     lb->tema = xmlGetProp (child, "tema");

     return lb;
}
```

Para obtener las propiedades del nodo, en este caso *`titulo, autor`* y *`tema`* se usa la función `xmlGetProp` que, como puede observarse en el ejemplo, recibe como parámetros el nodo a leer y la propiedad a recuperar.

Se puede usar esta función para cargar el documento "libros.xml" en un GtkCList (aunque este widget ha quedado obsoleto en GNOME2) de la siguiente manera:

```
xmlNodePtr root, child;
xmlDocPtr doc;

gchar *text[3];
libro lb;

doc = xml_open_file ("libro.xml");

root = xmlDocGetRootElement (doc);

child = root->xmlChildrenNode;

while (child != NULL) {
     lb = xml_get_entry (child);
     text[0] = lb.autor;
     text[1] = lb.titulo;
     text[2] = lb.tema;

     gtk_clist_append (GTK_CLIST (clist_book), text);
     child = child->next;
}
```

Para cargar el segundo formato propuesto, la función `xml_get_entry` será algo diferente, en primer lugar no se usará la función `xmlGetProp` que recupera el valor de una propiedad, si no que se usará la función `xmlNodeGetContent` que devuelve el contenido del nodo. La nueva función quedará así:

```
libro *xml_get_entry (xmlNodePtr child)
{
     xmlNodePtr node;
     libro *lb;

     lb = g_new0 (libro, 1);

     node = child->xmlChildrenNode;

     lb->titulo = xmlNodeGetContent (node);
     node = node->next;
```

```
        lb->autor = xmlNodeGetContent (node);
        node = node->next;

        lb->tema = xmlNodeGetContent (node);

        return lb;
}
```

En primer lugar, el parametro que recibe esta función debe ser una entrada, en este caso el nodo con etiqueta <libro>, a la variable node se le asigna el primer nodo hijo usando child->xmlChildrenNode y se obtiene su contenido mediante la función `xmlNodeGetContent` y con la asignación node = node->next se avanza al siguiente nodo.

Para cargar el documento "libro.xml" sería válido el mismo ejemplo citado anteriormente, ya que son las modificaciones realizadas en la función `xml_get_entry` las que procesarán correctamente el documento.

## Creación de ficheros XML

Cargar documentos XML es muy útil, pero de nada serviría si no se pudiera, paralelamente, crear esos documentos XML. Así, libxml incluye tambien funciones tanto para salvar documentos XML a disco o a memoria, como para crear/modificar el contenido de dichos documentos.

Lo primero que querremos hacer para crear de cero un documento XML es crear la estructura xmlDoc asociada, que posteriormente usaremos, como hemos visto en el apartado anterior, para acceder al contenido del documento. Así, la primera función que usaremos es `xmlNewDoc`:

```
xmlDocPtr doc;

doc = xmlNewDoc("1.0");
```

El parámetro que recibe esta función es el número de versión de la especificación XML que queremos usar para nuestro documento. De momento, simplemente usaremos la 1.0.

Esta función nos devuelve un valor de tipo xmlDocPtr que, como vemos, es el mismo que usamos para cargar el documento desde un fichero/posición de memoria.

Así pues, lo primero que debemos hacer es crear la estructura xmlDoc, para ello creamos una función que nos devuelve un puntero a la estructura xmlDoc y como parametro le pasamos el nombre que queremos darle al nodo raiz:

```
/* crea la estructura xmlDoc con nodo raiz "name" */
xmlDocPtr xml_new_doc (const gchar *name) {

 xmlNodePtr root;
 xmlDocPtr doc;

 doc = xmlNewDoc ("1.0");

 root = xmlNewDocNode (doc, NULL, name, NULL);
 xmlDocSetRootElement (doc, root);

 return doc;
}
```

Como vemos, se usa la función `xmlNewDocNode` para crear el nodo raiz. A esta función, aparte de la referencia a un documento XML (*doc*), le pasamos el nombre de la

etiqueta XML raiz, en este caso, como vemos en el documento XML mostrado anteriormente, "listado_libros". Seguidamente, usamos la función `xmlDocSetRootElement` para asociar el nodo recien creado con el documento XML especificado.

Pero, como es de suponer, en este caso el documento está vacío. Para añadirle contenido, lo que hacemos es crear nuevos nodos y añadirlos a otros nodos, aunque lo primero que tendremos que hacer es crear el nodo raiz de nuestro documento. Para ello, suponiendo que queramos crear un documento con el primer formato propuesto podemos crear la siguiente función:

```
void xml_new_entry (xmlDocPtr doc, libro nuevo) {

 xmlNodePtr root;
 xmlNodePtr node;

 /* nodo raiz */
 root = xmlDocGetRootElement (doc);

 node = xmlNewChild (root, NULL, "libro", NULL);

 xmlSetProp (node, "titulo", nuevo.titulo);
 xmlSetProp (node, "autor", nuevo.autor);
 xmlSetProp (node, "tema", nuevo.tema);
}
```

Como se puede apreciar, el funcionamiento es similar a la creación del nodo raiz. En este caso, usamos la función `xmlNewChild`, a la que le pasamos una referencia al nodo padre (en este caso usamos el nodo raiz, pero podriamos haber usado cualquier otro), así como el nombre de la etiqueta XML del nodo, en este caso "libro". Seguidamente, usamos la función `xmlSetProp` para establecer las propiedades del nodo, que en este caso son *autor*, *titulo* y *tema*. Estas propiedades como hemos visto en la sección de carga de documentos las podemos más tarde recuperar mediante la función `xmlGetProp`, que nos devuelve el valor contenido en la propiedade especificada.

Así, con esta función, nuestro programa para escribir el documento XML mostrado anteriormente quedaría de la siguiente forma:

```
xmlNodePtr root, child;
xmlDocPtr doc;
libro lb;

doc = xml_new_doc ("listado_libros");

lb.autor = "José Ortega y Gasset";
lb.titulo = "La rebelión de las massas";
lb.tema = "Filosofia";

xml_new_entry (doc, lb);

lb.autor = "José Luis Balcazar";
lb.titulo = "Programación metódica";
lb.tema = "Informática";

xml_new_entry (doc, lb);
```

Obteniendo como resultado el siguiente documento:

```
<?xml version="1.0"?>
<listado_libros>
 <libro titulo="La rebelión de las massas" autor="José Ortega y Gasset" tema="Filoso
 <libro titulo="Programación metódica" autor="José Luis Balcazar" tema="Informática"
</listado_libros>
```

Para crear un documento con el segundo formato propuesto, deberemos modificar la función `xml_new_entry` ya que en lugar de añadir propiedades ahora añadiremos contenido al nodo utilizando para ello la función `xmlNewChild` que además de crear un nuevo nodo, nos permite añadirle contenido. La función quedaria de la siguiente manera:

```
/* añade una nueva entrada al documento */
/* una entrada es de tipo libro */
void xml_new_entry (xmlDocPtr doc, libro nuevo) {

 xmlNodePtr root;
 xmlNodePtr libro;

 /* nodo raiz */
 root = xmlDocGetRootElement (doc);

 libro = xmlNewChild (root, NULL, "libro", NULL);

 xmlNewChild (libro, NULL, "titulo", nuevo.titulo);
 xmlNewChild (libro, NULL, "autor", nuevo.autor);
 xmlNewChild (libro, NULL, "tema", nuevo.tema);
}
```

Podemos observar los pasos para crear la entrada en el documento. Primero obtenemos el nodo raiz mediante la función `xmlDocGetRootElement` y después empezamos con la nueva entrada al documento creando primero el nodo hijo usando la función `xmlNewChild`, ha esta función le pasamos como argumentos el nodo padre (root) y la etiqueta de nodo (libro). Para introducir los datos, utilizamos la misma función `xmlNewChild` aunque de forma diferente, en este caso le pasamos como nodo padre (libro) la etiqueta que queremos darle al nuevo nodo (titulo) y el contenido a almacenar en el (nuevo.titulo).

Obteniendo un archivo libro.xml con el siguiente contenido:

```
<?xml version="1.0"?>
<listado_libros>
 <libro>
  <titulo>La rebelión de las masas</titulo>
  <autor>José Ortega y Gasset</autor>
  <tema>Filosofia</tema>
 </libro>
 <libro>
  <titulo>Programación metódica</titulo>
  <autor>José Luis Balcazar</autor>
  <tema>Informática</tema>
 </libro>
</listado_libros>
```

## Salvando documentos XML

Ya sólo nos queda, para terminar esta introducción al uso de libxml, ver cómo almacenar las documentos que modifiquemos en nuestro programas. Para ello, al igual que ocurría para la carga de documentos, libxml nos ofrece un conjunto de funciones que podremos usar para almacenar los documentos XML tanto en disco como en memoria.

En primer lugar, tenemos las funciones:

```
void xmlDocDumpMemory (xmlDocPtr doc, xmlChar **buffer, int *size);
int  xmlSaveFile      (const char *filename, xmlDocPtr doc);
```

que nos permiten salvar un documento XML (referenciado por una variable de tipo xmlDocPtr tanto en memoria (`xmlDocDumpMemory`) como en disco (`xmlSaveFile`).

Junto a estas dos funciones, tenemos otras dos que pueden sernos realmente útiles en algunas ocasiones:

```
void xmlDocDump  (FILE *f, xmlDocPtr doc);
void xmlElemDump (FILE *f, xmlDocPtr doc, xmlNodePtr elem);
```

Como vemos, ambas funciones tienen en comun que tienen un parámetro de tipo FILE (un tipo usado en las funciones de E/S de C), por lo que pueden sernos útiles, como comentábamos anteriormente, en el caso de que ya tengamos un fichero abierto. Además, la segunda de ellas (`xmlElemDump`) es especialmente útil si lo que deseamos es almacenar en disco simplemente un nodo XML y todos los nodos que cuelguen de él, es decir, sin necesidad de almacenar todo el documento, sólo se almacena una parte de él.

Otra función interesante es `xmlFreeDoc (xmlDocPtr cur)` que nos permite liberar la memoria utilizada por un documento.

Una característica muy interesante de libxml es que permite almacenar los documentos XML comprimidos, de forma que ocupen menos espacio en disco. Para ello, podemos usar las siguientes funciones:

```
int xmlGetDocCompressMode (xmlDocPtr doc);
void xmlSetDocCompressMode (xmlDocPtr doc, int mode);
int xmlGetCompressMode (void);
void xmlSetCompressMode (int mode);
```

Las dos primeras nos permiten establecer/obtener el modo de compresión de un documento previamente cargado, mientras que las dos últimas nos permiten establecer/obtener el modo de compresión por defecto de la librería, es decir, el que se usará por defecto si no es especificado por cada documento.

# Capítulo 18. GConf, el sistema de configuración

Hasta hace muy poco (GNOME 1.4), el proyecto GNOME usaba, como medio de almacenamiento de la configuración de las aplicaciones, un formato de ficheros muy parecido a los famosos .INI de MS Windows. Este sistema, si bien cubre las necesidades de la mayor parte de aplicaciones (configuración local para un usuario local), no cubre las expectativas de un escritorio moderno en UNIX, donde las aplicaciones, mediante el sistema X Window, pueden ser ejecutadas en red.

Para solventar esta situación, era necesario que el sistema de configuración de las aplicaciones fuera fácilmente accesible desde cualquier máquina dentro de una red, de forma que un usuario tenga la misma configuración tanto en su ordenador personal, como cuando abre una sesión con su usuario en el servidor de la oficina. Esto sólo se podía solucionar con una herramienta de configuración distribuida, y eso es lo que hace GConf.

Según la propia definición del autor, GConf "es un sistema para almacenar información de configuración, lo que comunmente se conoce por parejas clave/valor". Aparte de esta funcionalidad básica de almacenamiento, GConf ofrece muchos adelantos con respecto a su antecesor (gnome_config), como por ejemplo su sistema de notificación, que permite a una aplicación permanecer a la escucha de los cambios que se hagan en determinadas partes de la configuración. Además, destaca tambien la flexibilidad de su arquitectura, que permite cambiar fácilmente el almacen de datos (es decir, la manera y el lugar en que se almacena la información), o incluso usar varios a la vez, o la posibilidad de que cada entrada en la base de datos (o sea, cada clave), tenga asociado un texto descriptivo, lo que facilita enormemente la labor del usuario, pues cada entrada está perfectamente documentada.

GConf, como no podía ser menos al formar parte del proyecto GNOME, está basado en CORBA, lo cual permite el acceso totalmente transparente a la configuración de las aplicaciones, tanto en entornos locales como distribuidos. Su arquitectura está basada en tres elementos principalmente:

- "backends": son librerías dinámicas que se insertan en GConf (plugins) para permitir el almacenamiento de datos en distintos formatos. En el momento de escribir este artículo hay dos disponibles: XML, que es el formato por defecto en el que se almacenan los datos, y Berkeley DB, que aun está en desarrollo.
- gconfd: este el demonio de GConf, que se activa cuando un cliente (ver siguiente apartado) se conecta al sistema. Este demonio es un servidor CORBA que implementa las interfaces definidas por GConf, y que accede a los datos a través de los diferentes "backends". Normalmente, habrá una instancia de este demonio por cada usuario que esté haciendo uso de GConf.
- Clientes: son aplicaciones que hacen uso de las librerías provistas por GConf para acceder al demonio gconfd.

La información en GConf se guarda en forma de árbol, de la misma forma en que se organizan los discos con sus ficheros y directorios. Así, las siguientes cadenas son entradas válidas en GConf:

```
/aplicaciones/evolution/correo/servidor
/aplicaciones/gnome/escritorio/tapiz
```

Siguiendo con la analogía con los discos, se podría decir que la base de datos de GConf se divide en ficheros (una clave que contiene un valor) y directorios (una lista de otros ficheros y directorios).

GConf sólo admite un conjunto limitado de tipos de datos, principalmente numéricos (entero, coma flotante), cadenas y valores lógicos (booleanos). Hay otros tipos especiales, como las listas, que permiten almacenar una lista de valores bajo una clave

única, o los esquemas, que son tipos especiales que permiten almacenar información acerca de una clave de la base de datos, como su documentación asociada, etc

## Almacenes de datos

Como comentábamos antes, GConf permite fácilmente cambiar el modo de almacenamiento de los datos. El demonio gconfd, cuando arranca, lee el fichero `/etc/gconf/path`, en el que se almacena una lista de las fuentes de configuración que queremos tener disponibles para GConf. Así, un ejemplo de este fichero podría ser:

```
# GConf configuration path file with an include statement
xml:readonly:/etc/gconf.xml.mandatory
include "$(HOME)/.gconf.path"
xml:readonly:/etc/gconf.xml.defaults
# imaginary, no LDAP backend exists right now
ldap::/foo/bar/whatever/ldap/address
```

En este fichero se especifican todas las fuentes de datos que queremos tener disponibles, de forma que cuando pidamos el valor correspondiente a una clave, GConf buscará por todas las fuentes especificadas en este fichero hasta que encuentre una. Cuando establecemos un valor, GConf usa la primera fuente que encuentre en la que tengamos permiso de escritura. Así, en el ejemplo que hemos visto, se puede observar que como primera fuente ponemos un fichero XML, de sólo lectura, en el que están almacenadas las claves que no queremos que puedan ser cambiadas por los usuarios (pues siempre serán leidas antes que las personales del usuario, por lo que serán esos valores los que se usen siempre). Seguidamente, aparece la directiva `include` que, en este caso, incluye el fichero `.gconf.path` que pudiera tener el usuario que activa GConf en su directorio `$HOME`. Finalmente, vemos que aparece otra fuente de datos XML, tambien de sólo lectura, en la que se almacenan los valores por defecto, de forma que los usuarios tuvieran un conjunto de valores por defecto la primera vez que usen GConf. Tambien vemos una fuente de datos LDAP, que, como dice el comentario en el fichero, aun no puede usarse (más que nada porque aun no ha sido implementado un "backend" que permita el acceso a servidores LDAP). Pero es un buen ejemplo para mostrar las posibilidades que nos ofrece GConf a la hora de distribuir la configuración de nuestros usuarios, permitiéndonos de este modo almacenar la información donde más nos guste, sin que esto afecte a las aplicaciones, pues éstas seguirán usando un único interfaz (GConf) para el acceso a los datos. Igual que se añade en este ejemplo una fuente de datos LDAP, se podría estar usando cualquier otra que estuviera disponible en GConf, como una base de datos SQL, un fichero remoto, etc. De momento, como decíamos antes, sólo está disponible el "backend" XML, aunque ya está en desarrollo otro basado en Berkeley DB.

La escritura de nuevos "backends" es muy sencilla, por lo que esperamos que, según se vaya asentando GConf en la plataforma de desarrollo GNOME, irán apareciendo nuevos "backends" que nos permitirán usar todo tipo de almacenes de datos para nuestra configuración, permitiendo así la adaptación del sistema a cualquier entorno.

## Clientes GConf

El interfaz para clientes que ofrece GConf puede ser usado de dos formas distintas. Una es usando directamente el API, y otra usando una capa que se integra perfectamente con aplicaciones escritas para GTK/GNOME. Sea cual sea el modo que usemos, lo primero que tenemos que hacer en una aplicación que vaya a usar GConf, es

inicializar la librería cliente de GConf, que se consigue con una llamada a la función `gconf_init`:

```
gboolean gconf_init (int argc, char *argv[])
```

Esta función devuelve `TRUE` si tuvo éxito, o `FALSE` si hubo algun error. Los parámetros que recibe suelen ser típicamente los mismos que recibamos en la función `main`, pues `gconf_init` los necesita para procesar cualquier parámetro relacionado con CORBA/GConf que hubiera recibido nuestro programa.

Una vez que hemos inicializado la librería cliente GConf, ya podemos empezar a usar las funcionalidades de GConf. Pero para ello, lo primero que necesitamos es obtener una conexión con el sistema GConf, para lo que se usa el objeto *`GConfEngine`*. Este objeto representa una conexión con el motor de GConf (el demonio y los almacenes de datos manejados por éste), y lo necesitaremos para el resto de funciones de la librería.

Para obtener dicho objeto, podemos usar las siguientes dos funciones:

```
GConfEngine *gconf_engine_get_default (void);
GConfEngine *gconf_engine_get_default_with_address (const gchar *address);
```

La primera de estas funciones devuelve la conexión por defecto, mientras que la segunda nos permite activar una fuente de datos específica. En la mayoría de los casos, usaremos la primera función.

Ya con la conexión establecida, podemos comenzar a leer/escribir información a través de GConf. Para obtener valores de la base de datos, se usa la estructura `GConfValue`, que se obtienen mediante una llamada a la función `gconf_engine_get`:

```
GConfEngine *engine;
GConfValue *value;
GError *error;

engine = gconf_engine_get_default();
value = gconf_engine_get(engine, "/aplicaciones/gnome/fondo", &error);
switch (value->type) {
 case GCONF_VALUE_STRING :
  printf("%s\n", gconf_value_get_string(value));
  break;
 case GCONF_VALUE_INT:
  printf("%d\n", gconf_value_get_int(value));
  break;
 case GCONF_VALUE_FLOAT:
  printf("%g\n", gconf_value_get_float(value));
  break;
 case GCONF_VALUE_BOOL:
  printf("%s", gconf_value_get_bool(value) ? "true" : "false");
  break;
}
```

La estructura `GConfValue` contiene toda la información necesaria sobre el valor obtenido de la base de datos, de forma que podemos, en nuestro programa, descubrir todas sus propiedades, como por ejemplo el tipo de datos, que es lo que se hace en este ejemplo de código.

Como vemos en el ejemplo, una variable de tipo `GConfValue` contiene los datos extraidos de la base de datos, sea cual sea su formato. Por ello, es importante que podamos preguntar cuáles son las características de esos datos, y así poder actuar en consecuencia. Se puede observar tambien en el ejemplo cómo tenemos varias funciones que nos permiten obtener el valor almacenado en la

estructura en el formato asociado. Estas funciones son `gconf_value_get_string`, `gconf_value_get_int`, `gconf_value_get_bool` y `gconf_value_get_float`.

Pero la librería de GConf ofrece unas funciones de más alto nivel que nos permiten saltarnos el manejo de los `GConfValue`, y que son especialmente útiles cuando conocemos de antemano el tipo de datos asociado a los datos que queremos obtener a través de GConf. Estas funciones son las siguientes:

```
gdouble gconf_engine_get_float (GConfEngine *conf, const gchar *key, GError **err);
gint gconf_engine_get_int (GConfEngine *conf, const gchar *key, GError **err);
gchar* gconf_engine_get_string (GConfEngine *conf, const gchar *key, GError **err);
gboolean gconf_engine_get_bool (GConfEngine *conf, const gchar *key, GError **err);
```

Estas funciones son atajos directos para evitarnos el uso (a veces algo engorroso) de `GConfValue` y compañía. Estas funciones, internamente, hacen la conversión de `GConfValue` que veiamos antes en el ejemplo.

Análogamente a estas funciones, tenemos sus correspondientes funciones para escribir una entrada en la base de datos de configuración. Estas funciones son:

```
gboolean gconf_engine_set_float (GConfEngine *conf, const gchar *key, gdouble val, GEr
gboolean gconf_engine_set_int (GConfEngine *conf, const gchar *key, gint val, GError *
gboolean gconf_engine_set_string (GConfEngine *conf, const gchar *key, const gchar *va
gboolean gconf_engine_set_bool (GConfEngine *conf, const gchar *key, gboolean val, GEr
```

Todas estas funciones (tanto `*_get_*` como `*_set_*`) tienen un parámetro llamado *`key`*, que es una cadena que representa la entrada a la que nos estamos refiriendo (sería el camino completo del fichero).

Hay otro par de funciones que nos serán especialmente útiles cuando estemos leyendo datos de GConf. Son las siguientes:

```
GSList *gconf_engine_all_entries (GConfEngine *engine, const gchar *dir, GError **erro
GSList *gconf_engine_all_dirs (GConfEngine *engine, const gchar *dir, GError **error);
```

La primera de ellas, `gconf_engine_all_dirs` devuelve una lista de cadenas que representan el nombre de todas las secciones encontradas en el directorio (o sección) especificado en el parámetro *`dir`*.

```
slist = gconf_engine_all_dirs(get_conf_engine(), path, NULL);
if (slist) {
 GSList* node;

 for (node = slist; node != NULL; node = g_slist_next(node)) {
  printf("%s\n", (gchar *) node->data;
 }
 g_slist_free(slist);
}
```

La segunda función, `gconf_engine_all_entries`, hace lo mismo, pero con dos "pequeñas" diferencias: devuelve una lista de claves y no de secciones, y cada uno de los elementos de la lista devuelta, en vez de ser una cadena de texto (de tipo *`gchar *`*), es de tipo *`GConfEntry`*.

```
slist = gconf_engine_all_entries(get_conf_engine(), path, NULL);
if (slist) {
 GSList* node;

 for (node = slist; node != NULL; node = g_slist_next(node)) {
```

```
  GConfEntry* entry = (GConfEntry *) node->data;
  printf("%s\n", gconf_entry_get_key(entry));
 }
 g_slist_free(slist);
}
```

## Notificaciones

Como comentábamos en la introducción, una de las características extra que GConf ofrece es la de la notificación de cambios, que permite que una aplicación sea notificada por el demonio `gconfd` cada vez que se produzca un cambio en determinada(s) entrada(s) de la base de datos de configuración.

El sistema de notificación de GConf actúa bajo demanda. Es decir, sólo notifica a los clientes que piden ser notificados, y además, sólo notifica de cambios en las entradas en las que estén interesados esos clientes. Para registrar nuestra aplicación en el sistema de notificación, usamos la función `gconf_engine_notify_add`, que tiene el siguiente formato:

```
guint gconf_engine_notify_add (GConfEngine *engine,
                               const gchar *section,
                               GConfNotifyFunc func,
                               gpointer user_data,
                               GError **error);
```

Esta función recibe como parámetros, aparte de los típicos *GConfEngine* y *GError* que aparecen en muchas de las funciones que hemos visto anteriormente, una cadena que especifica la sección de la configuración que estamos interesados en monitorizar, y un puntero a una función (*GConfNotifyFunc*), que será la que será llamada cada vez que hay algo que notificar.

La función de respuesta a la notificación, de tipo *GConfNotifyFunc*, debe tener la siguiente forma:

```
void notify_func (GConfEngine *engine, guint cnx_id, GConfEntry *entry, gpointer user_
```

donde *cnx_id* es el identificador devuelto por la llamada a `gconf_engine_notify_add`, *entry* es un puntero a una estructura de tipo *GConfEntry* que contiene toda la información asociada a la entrada de la base de datos que especificamos en la llamada a `gconf_engine_notify_add`, y *user_data* es el mismo parámetro que especificamos al registrar la función de notificación, y que nos permite pasar datos propios de un sitio a otro del programa.

El identificador devuelto por `gconf_engine_notify_add` es especialmente útil cuando queremos dejar de recibir notificaciones de cambios. Para ello, usamos la función `gconf_engine_notify_remove`, que hace lo propio, y que recibe, como parámetro, dicho identificador. Una vez que hayamos llamado a esta función con éxito, no recibiremos más notificaciones de cambios en esa entrada.

```
guint cnx_id;
GError *error = NULL;

cnx_id = gconf_engine_notify_add(engine, "/apps/gnome/gnumeric", func, NULL, &error);
...
gconf_engine_notify_remove(engine, cnx_id);
```

## Gestión de errores

Hemos visto que casi todas las funciones mostradas hasta el momento incluyen un parámetro de tipo `GError`. Este es un tipo de datos que será incluido en GLib 2.0, pero que ya tenemos disponible en GConf, para así luego facilitar la transición. Como se puede adivinar por su nombre, es un tipo de datos que nos permite recibir información acerca de errores desde GConf, de forma que tengamos una información detallada del error (una descripción, un código de error, ...).

La forma correcta de usar estos parámetros de tipo `GError` es pasando un puntero a un puntero en el que GConf copiará la información del error en caso de que haya uno. Así, por ejemplo:

```
GError *error = NULL;

gconf_engine_set_int(engine, "/apps/gnome/mail", 0, &error);
if (error) {
  /* se produjo un error, mostrar información */
  printf("Error %d: '%s'\n", error->num, error->message);
}
```

La estructura `GError` contiene dos campos públicos, que son los dos que se muestran en el ejemplo anterior: *num*, que contiene un entero que indica el código del error (los códigos de error válidos podemos consultarlos en los ficheros de cabecera de GConf), y *message*, que contiene una cadena con la descripción del error.

# Capítulo 19. gnome-vfs

GNOME-VFS es una librería que ofrece un API (interfaz para el programador de aplicaciones) muy similar al sistema de E/S que normalmente se usa. Es decir, las funciones de E/S del estándar de C y similares.

La diferencia entre el sistema estándar de E/S y GNOME-VFS es que este último ofrece una abstracción del acceso a ficheros, sockets, protocolos de red, etc, en lo que se conoce como VFS o Virtual File System (Sistema virtual de ficheros). Gracias a esta abstracción, se podrá acceder, desde las aplicaciones, a recursos (ficheros, directorios, etc) disponibles en distintos sitios y a través de distintos protocolos de acceso.

Así, por ejemplo, si se usa GNOME-VFS, se podrá acceder a ficheros remotos, a través de FTP (o HTTP, o...) como si de ficheros locales se tratara, y todo ello totalmente transparente para la aplicación.

La arquitectura de GNOME-VFS está basada en módulos (o backends) que implementan el acceso a un sistema de ficheros real (ficheros en el disco, HTTP, FTP, WebDAV, etc), y que pueden ser instalados o desinstalados sin afectar para nada al resto del sistema. Este sistema de módulos permite añadir al sistema de GNOME-VFS acceso a ficheros de distintas formas. Por ejemplo, como ya se ha comentado, hay un módulo que permite el acceso a recursos a traves de WebDAV, o de HTTP, o FTP, o ficheros locales, pero, aparte de esto, se pueden añadir módulos que ofrezcan acceso a distintos dispositivos o recursos como si de ficheros se tratara. Por ejemplo, uno de los más interesantes es el módulo "camera", que permite acceder a las fotos almacenadas en una cámara de fotos digital como si de un simple directorio con ficheros se tratara (que, por cierto, es lo que es).

Se ha decidido recientemente, en el seno del proyecto GNOME, la obligatoriedad de usar GNOME-VFS para todas las operaciones de Entrada/Salida de las aplicaciones que componen el proyecto GNOME. Esto va a permitir, primero, la estandarización del acceso a cualquier tipo de recurso soportado por GNOME-VFS sin necesidad de que las aplicaciones sepan absolutamente nada de dicho recurso, y, más importante, va a permitir que todas las aplicaciones se integren perfectamente con Nautilus, el gestor de ficheros oficial del proyecto GNOME, que hace un uso intensivo de GNOME-VFS.

## URIs (Uniform Resource Identifier)

GNOME-VFS funciona por medio de URIs (Uniform Resource Identifiers), o, lo que es lo mismo, un modo de referenciación único de recursos. Cualquier persona que haya usado mínimamente Internet sabe lo que es un URI, que es una cadena de texto con la que se hace referencia a un recurso único en Internet. Así mismo, también se usa para acceder, a través de nuestro navegador, a los ficheros locales de nuestro disco (file:/home/rodrigo/index.html, por ejemplo).

Este modo de referenciación único es el que se usa cuando le indicamos a GNOME-VFS que realice determinada operación sobre un recurso, para especificar qué recurso es el que tiene que usar para dicha operación.

En GNOME-VFS se dispone de un conjunto bastante amplio de funciones para el manejo y validación de estos URIs. Así, por ejemplo, lo primero que se debería hacer es crear un objeto de la clase GnomeVFSURI, que se hace de la siguiente forma:

```
GnomeVFSURI *uri;

uri = gnome_vfs_uri_new ("file:/tmp/fichero");
```

Una vez se tiene el objeto *uri* creado, tenemos acceso a un número bastante importante de operaciones sobre él, que se listan a continuación de forma resumida:

- `gnome_vfs_append_file_name`, `gnome_vfs_append_string` y `gnome_vfs_append_path` permiten, dado un URI, añadirle un camino hacia un fichero o directorio. El usar estas funciones en vez de construir el URI (concatenando cadenas), se asegura el que GNOME-VFS realizará una validación del formato del URI, generando un error si se intentara crear un URI incorrecto.
- `gnome_vfs_uri_to_string`permite obtener una cadena de texto que represente un objeto de tipo GnomeVFSURI.
- `gnome_vfs_uri_is_local` comprueba si el URI especificado es local o no.
- `gnome_vfs_uri_get_host_name`, `gnome_vfs_uri_get_host_port`, `gnome_vfs_uri_get_username` y `gnome_vfs_uri_get_password` permiten obtener por separado cada una de las partes de un URI (nombre del host, puerto, nombre de usuario y contraseña) que pudieran estar incluidos en el URI. Estas funciones tienen su equivalente (`gnome_vfs_uri_set_password`, etc) que permiten modificar cada una de estas partes.

Todas estas funciones permiten acceder y alterar el contenido del URI, todo ello a través de sencillas funciones.

## Operaciones básicas

Las operaciones básicas que se pueden realizar con GNOME-VFS son las que cabe esperar de cualquier librería de E/S. Es decir, abrir ficheros, leerlos, modificarlos, todo ello a través de funciones que siguen la nomenclatura del estándar POSIX de E/S.

Así, por ejemplo, para abrir un fichero alojado en el servidor FTP de GNOME, se usaría el siguiente fragmento de código:

```
GnomeVFSHandle *handle = NULL;
gchar buffer[128];
GnomeVFSFileSize leidos;

gnome_vfs_open (&handle, "ftp://ftp.gnome.org/gnumeric.deb", GNOME_VFS_OPEN_READ)
gnome_vfs_read (handle, buffer, sizeof (buffer) - 1, &leidos);
g_print ("Se leyeron %d bytes:\n%s\n\n", leidos, buffer);
```

Como se puede observar en este ejemplo, el API de GNOME-VFS es muy parecido al del sistema POSIX de E/S de C (las funciones close, read, write, etc). Y, de hecho, ésto no es fortuíto, sino que ha sido intencionado el que el API fuese lo más parecida posible, para así facilitar los cambios necesarios en las aplicaciones para pasar del sistema POSIX a GNOME-VFS.

En este ejemplo, se puede observar que no es necesario usar ninguna variable de tipo GnomeVFSURI, como comentábamos en el punto anterior. Ésto es así porque se ha querido simplificar el ejemplo al máximo, y para ello se ha usado una de las múltiples funciones de GNOME-VFS, en este caso `gnome_vfs_open`, que permite directamente especificar el URI en una cadena de texto. Así que, usando `GnomeVFSURI`, el código anterior se traduciría de la siguiente manera:

```
GnomeVFSURI *uri;
GnomeVFSHandle *handle;

uri = gnome_vfs_uri_new ("ftp://ftp.gnome.org/gnumeric.deb");
gnome_vfs_open_uri (&handle, uri, GNOME_VFS_OPEN_READ);
...
```

Así, como puede verse, se dispone de un sistema de manejo de recursos muy sencillo de usar, al ser parecido al sistema al que ya se usaba antes. Por ello, siguiendo con las

similitudes con el estándar POSIX, éstas son las funciones de E/S básicas de las que se dispone en GNOME-VFS.

```
GnomeVFSResult gnome_vfs_open (GnomeVFSHandle **handle,
                                             const gchar *text_uri,
                                             GnomeVFSOpenMode open_mode);
GnomeVFSResult gnome_vfs_open_uri (GnomeVFSHandle **handle,
                                             GnomeVFSURI *uri,
                                             GnomeVFSOpenMode open_mode);
GnomeVFSResult gnome_vfs_create (GnomeVFSHandle **handle,
                                             const gchar *text_uri,
                                             GnomeVFSOpenMode open_mode,
                                             gboolean exclusive,
                                             guint perm);
GnomeVFSResult gnome_vfs_create_uri (GnomeVFSHandle **handle,
                                             GnomeVFSURI *uri,
                                             GnomeVFSOpenMode open_mode,
                                             gboolean exclusive,
                                             guint perm);
GnomeVFSResult gnome_vfs_close (GnomeVFSHandle *handle);
GnomeVFSResult gnome_vfs_read (GnomeVFSHandle *handle,
                                             gpointer buffer,
                                             GnomeVFSFileSize bytes,
                                             GnomeVFSFileSize *bytes_read);
GnomeVFSResult gnome_vfs_write (GnomeVFSHandle *handle,
                                             gconstpointer buffer,
                                             GnomeVFSFileSize bytes,
                                             GnomeVFSFileSize *bytes_written);
GnomeVFSResult gnome_vfs_seek (GnomeVFSHandle *handle,
                                             GnomeVFSSeekPosition whence,
                                             GnomeVFSFileOffset offset);
GnomeVFSResult gnome_vfs_tell (GnomeVFSHandle *handle,
                                             GnomeVFSFileSize *offset_return);
GnomeVFSResult gnome_vfs_get_file_info (const gchar *text_uri,
                                             GnomeVFSFileInfo *info,
                                             GnomeVFSFileInfoOptionsoptions);
GnomeVFSResult gnome_vfs_get_file_info_uri (GnomeVFSURI *uri,
                                             GnomeVFSFileInfo *info,
                                             GnomeVFSFileInfoOptionsoptions);
GnomeVFSResult gnome_vfs_get_file_info_from_handle
(GnomeVFSHandle *handle,
                                             GnomeVFSFileInfo *info,
                                             GnomeVFSFileInfoOptionsoptions);
GnomeVFSResult gnome_vfs_truncate (const gchar *text_uri,
                                             GnomeVFSFileSize length);
GnomeVFSResult gnome_vfs_truncate_uri (GnomeVFSURI *uri,
                                             GnomeVFSFileSize length);
GnomeVFSResult gnome_vfs_truncate_handle (GnomeVFSHandle *handle,
                                             GnomeVFSFileSize length);
GnomeVFSResult gnome_vfs_make_directory_for_uri
(GnomeVFSURI *uri,
                                             guint perm);
GnomeVFSResult gnome_vfs_make_directory (const gchar *text_uri,
                                             guint perm);
GnomeVFSResult gnome_vfs_remove_directory_from_uri
(GnomeVFSURI *uri);
GnomeVFSResult gnome_vfs_remove_directory (const gchar *text_uri);
GnomeVFSResult gnome_vfs_unlink_from_uri (GnomeVFSURI *uri);
GnomeVFSResult gnome_vfs_create_symbolic_link
(GnomeVFSURI *uri,
                                             const gchar *target_reference);
GnomeVFSResult gnome_vfs_unlink (const gchar *text_uri);
GnomeVFSResult gnome_vfs_move_uri (GnomeVFSURI *old_uri,
                                             GnomeVFSURI *new_uri,
                                             gboolean force_replace);
GnomeVFSResult gnome_vfs_move (const gchar *old_text_uri,
```

```
                                                     const gchar *new_text_uri,
                                                     gboolean force_replace);
GnomeVFSResult gnome_vfs_check_same_fs_uris (GnomeVFSU
```

Es una larga lista de operaciones que cubre todas las tareas básicas de E/S, como son la lectura/escritura (_read/_write), la búsqueda dentro del fichero (_seek), la obtención de información sobre el recurso (_get_file_info), etc, aparte de otra serie de funciones de más alto nivel que permiten hacer todo tipo de operaciones con URIs. Como también se puede observar, se ofrecen funciones para cada operación de forma que se puedan usar en esas operaciones tanto simples cadenas de texto representando un URI, como objetos GnomeVFSURI como GnomeVFSHandle (que es el equivalente a los descriptores de ficheros en el sistema POSIX).

Es importante destacar que gran parte de estas funciones devuelven un valor de tipo GnomeVFSResult, que es un tipo definido en los ficheros de cabecera de GNOME-VFS que puede contener distintos valores, cada uno de los cuales especifican una condición de error especial. En cualquiera de los casos, aunque en los ejemplos que aquí se muestra no aparezca, es conveniente comprobar, tras cada llamada a las funciones de GNOME-VFS el valor devuelto, y actuar en consecuencia.

## E/S Asíncrona

Se puede perfectamente decir que GNOME-VFS es todo lo que hace falta para las necesidades de un programador, en el ámbito de la E/S de datos, y para corroborar eso, GNOME-VFS ofrece, aparte de las funciones de E/S básicas comentadas anteriormente. Sus equivalentes asíncronos, que permiten realizar todas las operaciones anteriormente comentadas de forma asíncrona, de manera que dichas operaciones no bloquearán al resto de la aplicación, sino que se ejecutarán de forma concurrente a los demás procesos que realice nuestra aplicación.

La forma de funcionamiento de este conjunto de llamadas asíncronas es muy parecido al que se ha comentado anteriormente, con la única diferencia que se deberá añadir un nuevo parámetro a las llamadas a funciones de GNOME-VFS:

```
GnomeVFSAsyncHandle *handle = NULL;

gnome_vfs_async_open (&handle, "ftp://ftp.gnome.org/gnumeric.deb",
                      GNOME_VFS_OPEN_READ,
                      (GnomeVFSAsyncOpenCallback) funcion_callback,
              NULL);
```

`funcion_callback`, que es la función que será llamada cuando se complete la operación (consiguiendo así el asincronismo), tendría el siguiente aspecto:

```
void funcion_callback (GnomeVFSHandle *handle,
                       GnomeVFSResult result,
               gpointer user_data);
```

Donde *`handle`* es el identificador devuelto por `gnome_vfs_async_open`, *`result`* es el código del error producido en la operación (si se produjo alguno) y *`user_data`* es el puntero especificado como último parámetro en la llamada a la función `gnome_vfs_async_open`, y que permite pasar datos propios desde el lugar donde se inicia la operación asíncrona y la función de "callback".

## Módulos para GNOME-VFS

GNOME-VFS, aparte de ofrecer todas estas funciones que se ha comentado, ofrece un conjunto de funciones que permiten ampliar la funcionalidad de GNOME-VFS mediante el desarrollo de nuevos módulos. Ésta es una característica realmente interesante, pues permite modificar completamente el modo de almacenamiento de los datos con sólo escribir un nuevo módulo.

# Capítulo 20. Acceso a BBDD

Una de las tareas más importantes en cualquier aplicación ofimática es el acceso a bases de datos. Para ello, y como parte del proyecto GNOME Office[1], existe `GNOME-DB`, que es un metaproyecto compuesto de:

- `libgda`, que implementa el acceso a los distintos servidores de BBDD soportados (Oracle, MySQL, PostgreSQL, InterBase, etc).
- `libgnomedb`, que es una librería de "widgets" orientados a bases de datos, como por ejemplo, rejillas de datos, listas, cajas de texto, etc.
- `GNOME-DB`, que es una aplicación, del estilo de la ya conocida MS-Access, o de la menos conocida `TOAD`, que ofrece un bonito y potente interfaz gráfico a todas las funcionalidades de `libgda/libgnomedb`.

Este capítulo se centra única y exclusivamente en `libgda` y `libgnomedb`, que son las que interesan para el desarrollo de aplicaciones que accedan a bases de datos. `GNOME-DB` es una aplicación cuyo uso no entra en el alcance de este capítulo.

## libgda

Como se comentaba en la introducción de este capítulo, `libgda` es una librería que implementa acceso a distintas fuentes de datos. Aunque orientada al acceso a servidores de BBDD, por *fuentes de datos* se entiende cualquier almacén de datos, ya sea un servidor de BBDD (Oracle, PostgreSQL, MySQL, etc) como una BBDD embebida (SQLite, Berkeley DB, etc), así como ficheros XML, ASCII, servidores LDAP, de correo, etc.

La idea detrás de `libgda` es el servir de capa de acceso a datos, sin necesidad de ahondar en detalles de bajo nivel. Para ello, ofrece un API sencillo de usar a la vez que potente, que encapsula perfectamente el acceso directo a los distintos proveedores de datos soportados, ofreciendo un interfaz genérico para todos ellos, de forma que con un solo conjunto de funciones se puede acceder a multitud de fuentes de datos distintas.

`libgda` ofrece una arquitectura modular, basada en "plugins", en donde cada proveedor de datos soportado instala en el sistema uno de estos "plugins", que son detectados automáticamente por la librería, y puestos a disposición de los clientes. En un principio (año 1998), la arquitectura estaba basada en CORBA, pero con el tiempo, se fue cambiando la arquitectura, eliminando todas las dependencias con CORBA, resultando en una librería diminuta, con muy pocas dependencias pero una gran funcionalidad, apta para ser usada tanto en aplicaciones GNOME, como aplicaciones totalmente ajenas al proyecto GNOME, como pueden ser aplicaciones en sistemas embebidos o en otros sistemas operativos (Windows, Mac, etc). El sistema usado en la actualidad para los "plugins" es la sección de nombre *Como hacer plugins.* en Capítulo 4, el sistema de carga de módulos de `GLib`, lo que ha resultado en una arquitectura mucho más simple y ligera, sin perder absolutamente ninguna funcionalidad.

### Fuentes de datos

### Clientes GDA

El punto de entrada a toda la arquitectura de `libgda` está representado por la clase GdaClient, que permite el acceso a los distintos Esta funcionalidad incluye:

- Apertura y cierre de conexiones a distintas BBDD.

- Implementación de un 'pool' de conexiones, de forma que, en vez de abrir la misma conexión una y otra vez, simplemente se use una conexión compartida por varias partes de la aplicación. Esto es especialmente útil en el caso de BBDD propietarias, donde se pague por el uso de licencias concurrentes. El estar compartiendo la conexión entre 'n' clientes permite saltarse las restricciones de uso por número de licencias, siempre dentro de la legalidad.

Para crear un objeto de la clase GdaClient, se usa la siguiente función:

```
GdaClient *gda_client_new (void);
```

Esta función crea un objeto de la clase GdaClient, que servirá de punto de entrada al resto de operaciones que se pueden realizar con `libgda`. Y no sólo de punto de entrada, si no que también permite servir de controlador de las acciones que se vayan realizando con los distintos proveedores. Esto incluye la recepción de notificaciones de todos los eventos que se produzcan, tales como apertura de una nueva conexión, ejecución de un comando, etc, así como alguna que otra cosa más, que se detallan a continuación.

Una vez que se ha creado una instancia de GdaClient, el siguiente paso, normalmente, es la apertura de conexiones, que se realiza con la función `gda_client_open_connection`, que tiene la siguiente forma:

```
GdaConnection * gda_client_open_connection(GdaClient *client, const
gchar * dsn, const gchar * username, const gchar * password);
```

El primer parámetro, al igual que en el resto de librerías de GNOME, es un puntero al objeto sobre el que realizar la acción, en este caso un GdaClient. Los siguientes parámetros, especifican los datos a usar en la conexión. El primero de ellos, *dsn*, especifica el nombre de la fuente de datos a usar para establecer la conexión, tal y como se explicó en el apartado anterior. Los otros dos, *username* y *password*, permiten especificar el nombre de usuario y contraseña a usar en la conexión. En el caso de *username*, si no es NULL, sustituirá al nombre de usuario especificado al crear la fuente de datos; si es NULL, se usará el especificado en la fuente de datos, si es que se especificó al crearla.

`gda_client_open_connection` devuelve un nuevo objeto del tipo `GdaConnection`, que es otra clase disponible en `libgda` y que, como su nombre indica, permite la gestión de las conexiones abiertas por `GdaClient`.

Al igual que `GdaClient` es el punto de entrada para la realización de tareas con las distintas fuentes de datos soportadas en `libgda`, `GdaConnection` es el punto de entrada a todas las operaciones disponibles en las conexiones que sean establecidas a través de `libgda`. Las operaciones disponibles a través de GdaConnection son múltiples e incluyen:

- Ejecución de comandos, tanto SQL como de otros tipos (comentados más adelante), que permiten la obtención de resultados de las distintas fuentes de datos así como la abstracción de la fuente de datos a la que se esté accediendo.
- Ejecución de transacciones. Por supuesto, no todos los proveedores las soportan, para lo cual `libgda` ofrece distintas utilidades que permiten simularlas, lo que será explicado más adelante.
- Obtención de información sobre las características de la fuente de datos siendo accedida, lo que incluye no sólo la obtención de información sobre sus características, si no sobre todos los objetos (tablas, vistas, procedimientos, tipos, etc) almacenados en ella.

### Pools de conexiones

Como se comentaba, `GdaClient` implementa un 'pool' de conexiones, que se puede configurar al antojo del usuario, incluso deshabilitándolo completamente.

Este 'pool' de conexiones permite hacer una gestión eficiente de los recursos disponibles en la BBDD, así como, como se comentaba en la introducción de este apartado, la posibilidad de saltarse, legalmente, las restricciones por número de licencias presentes en algunas BBDD comerciales. -> Para seguir con esto ver aquí[2].

### Notificaciones desde los proveedores

## Gestión de errores

Una parte muy importante del acceso a datos en las aplicaciones es la gestión de los errores que puedan producirse como resultado de las distintas operaciones que se realicen sobre la fuente de datos en cuestión. Todas las funciones de `libgda` informan de cualquier error que se haya producido, pero la información que ofrecen es mínima (normalmente, simplemente un valor gboolean que especifica si la operación se completó con éxito o no). Por ello, `libgda` ofrece un sistema paralelo a este para informar con todo detalle de todos los errores que se produzcan. Para conseguir esto, se usa el sistema de señales del Capítulo 5sistema de objetos de GLib.

Todo esto se traduce a que tanto `GdaClient` como `GdaConnection` incluyen una señal "error" a la que los clientes pueden conectarse (mediante la función `g_signal_connect`) para ser informados detalladamente de cualquier error que se produzca. Realmente no es necesario que los clientes se conecten a la señal del `GdaClient` creado y de cualquier conexión que se establezca, pues la clase `GdaClient` ya captura todos los errores notificados en cada `GdaConnection` creado, de forma que sólo es necesario conectarse a la señal "error" de `GdaClient`.

El manejador de esta señal, para la clase `GdaConnection` tiene la siguiente forma:

```
void error_callback(GdaConnection * cnc, GList * errors, gpointer
user_data);
```

Para la clase `GdaClient`, en cambio, tiene la siguiente forma:

```
void error_callback(GdaClient * client, GdaConnection * cnc, GList *
errors, gpointer user_data);
```

Las dos funcionan de la misma forma, es decir, reciben una lista de errores (parámetro *errors*). La única diferencia radica en que el manejador de la señal para `GdaClient` recibe, además de la conexión en la que se ha producido el error, un puntero al cliente sobre el que se estableció la conexión con el manejador de señal.

Por tanto, en la mayor parte de los casos, la gestión de errores de los programas que hagan uso de `libgda` se reduce a las siguientes líneas de código:

```
        g_signal_connect (G_OBJECT (client), "error", error_handler, NULL);
```

Seguido del manejador de señal propiamente dicho, que sería la función `error_handler` en el código anterior, y que sería declarado de la siguiente forma:

```
   void error_handler (GdaClient *client,
                       GdaConnection *cnc,
                       GList *errors,
```

```
                                    gpointer user_data)
```

El parámetro más interesante de cuantos recibe el manejador de señal es, sin duda alguna, *errors*, que es una GList de GdaError, que es el tipo de datos usado para contener toda la información referente al error. La gestión de este nuevo tipo de datos se ve mucho mejor con un ejemplo:

```
void
error_handler (GdaClient *client, GdaConnection *cnc, GList *errors, gpointer user_data
{
}
```

Como se puede apreciar por lo comentado hasta ahora, la gestión de errores con `libgda` se realiza de forma asíncrona. Si bien este método es especialmente potente, pues permite separar la ejecución de operaciones de la gestión de los errores producidos en ella, hay ocasiones en las que es preferible el conocer, de manera síncrona, cuáles son los errores producidos. Para ello, se puede hacer uso del valor de retorno de las distintas funciones de `libgda`, que especifica si la operación se produjo con éxito o no. Esto, junto al uso de la función `gda_connection_get_errors` permiten la obtención de la información de errores inmediatamente después de la ejecución de cada función. Así, por ejemplo:

```
   GdaCommand *cmd;

   cmd = gda_command_new ("UPDATE table1 SET field1 = 0", GDA_COMMAND_TYPE_SQL, 0);
   if (gda_connection_execute_non_query (cnc, cmd, NULL) == -1) {
           const GList *errors = gda_connection_get_errors (cnc);
           ...
   }
```

Este código, que ejecuta un comando UPDATE y obtiene el número de filas afectadas por el comando (valor de retorno de `gda_connection_execute_non_query`) hace exactamente lo que se explicaba en el párrafo anterior. Es decir, obtiene el valor de retorno de la función de `libgda` (en este caso, `gda_connection_execute_non_query`), que es -1 en caso de producirse un error. Basado en este valor de retorno, el programa obtiene, si se produjo un error, información sobre cada uno de los errores producidos, esto último, mediante la función `gda_connection_get_errors`.

## Ejecución de comandos

Una vez obtenida una conexión a un proveedor de datos, las operaciones más corrientes a realizar con dicha conexión incluyen la ejecución de comandos, como se ha visto brevemente en el apartado anterior.

La base para ejecutar comandos en `libgda` es, aparte, por supuesto, de la clase `GdaConnection` (no se pueden ejecutar comandos sin una conexión abierta), es la clase `GdaCommand`, que ofrece una sencilla forma de crear comandos para su posterior ejecución. Esta clase es especialmente sencilla, y la mayor parte de las veces, simplemente será necesario el uso de una sola función:

```
GdaCommand *gda_command_new(const gchar *text, GdaCommandType type,
GdaCommandOptions options);
```

Esta función permite crear una instancia de la clase `GdaCommand` con una serie de valores específicos, para distintos parámetros:

- `text`: el texto del comando, que, normalmente, será uno o varios comandos SQL, aunque no siempre (ver explicación de `type`).
- `type`: el tipo de comando a crear. Puede ser *GDA_COMMAND_TYPE_SQL*, en cuyo caso el parámetro `text` contiene uno o varios comandos SQL (separados por punto y coma); o puede ser *GDA_COMMAND_TYPE_XML*, en cuyo caso `text` contiene un fragmento XML describiendo uno o varios comandos (ver sección sobre comandos XML); o *GDA_COMMAND_TYPE_PROCEDURE*, en cuyo caso `text` contiene el nombre y los parámetros de una llamada a un procedimiento almacenado/función definido/a en la BBDD en la que se va a ejecutar el comando; o *GDA_COMMAND_TYPE_TABLE*, en cuyo caso `text` contiene el nombre de una o varias tablas (separadas por punto y coma), y la ejecución del comando resultará en lo mismo que si se ejecutara un "SELECT * FROM ..." en cada una de las tablas especificadas; y, por último, este parámetro puede ser *GDA_COMMAND_TYPE_SCHEMA*, en cuyo caso el parámetro `text` contiene el nombre de uno de los esquemas definidos en `libgda` para la obtención de metadatos.
- `options`: las opciones de ejecución del comando, que no es más que una máscara de bits que permite especificar distintos parámetros de comportamiento. Los valores disponibles para este parámetro son los siguientes (que pueden combinarse mediante el uso del operador `|`):
  - *GDA_COMMAND_OPTION_IGNORE_ERRORS*
  - *GDA_COMMAND_OPTION_STOP_ON_ERRORS*

### Modelos de datos

### Metadatos

## libgnomedb

libgnomedb es la segunda librería del proyecto GNOME-DB, y que añade a la potencia de libgda, un conjunto de controles visuales para la visualización y manejo de los datos obtenidos a través de libgda.

## Notas

1. http://www.gnome.org/gnome-office
2. http://bugzilla.gnome.org/show_bug.cgi?id=73340

# Capítulo 21. Documentación con DocBook

## Por qué usar SGML?

Para la creación de articulos/libros/tutoriales/etc se hace necesario utilizar un medio de documentación estandar de manera que la modificación de los documentos fuentes sea mas sencilla al igual que la generación de estos en diferentes formatos sea un paso trivial.

### SGML

La documentación estructurada se construye sobre elementos estructurados: capítulos, secciones, párrafos, etc. donde los elementos se etiquetan claramente para que son: referencias, salida de programas, etc. No se da ningúna información explícita sobre como el documento debe ser escrito, solamente sobre su estructura y contenido.

Esto permite el proceso automático de los documentos, animando a los autores a que se concentren en el contenido de los documentos y no en el como generarlos.

Asi, el preguntarse "como colocar una palabra en negrilla con SGML?" tiene poca importancia, lo que uno se pregunta es como poner mas énfasis en un cierto texto.

El SGML (Standard Generalized Markup Languaje) es un lenguaje estandarizado previsto para facilitar la escritura de documentación estructurada. Mas especificamente es un metalenguaje. Realmente no se escribe el documento en SGML, El SGML se utiliza para describir un lenguaje estructurado especifico para un tipo de documento, llamado DTD (Document Type Definition), que especifica como los documentos pueden ser escritos.

Por lo tanto decir que un documento está escrito en formato SGML es técnicamente correcto, pero engañoso. Podra decir que un documento está escrito con el formato DocBook u otro formato definido en SGML.

SGML es un lenguaje de marcas. Todos los documentos incluyen texto mezclados con etiquetas que delimita los elementos. En el Ejemplo 21-1 se muestra el uso de las etiquetas.

**Ejemplo 21-1. Uso de Etiquetas**

```
<article>
<title>Proyecto GNOME Chile</title>
<para>Vamos que se puede!</para>
```

Se parace a HTML, pues HTML es teóricamente un DTD de SGML. Los elementos tienen un contenido, por ejemplo el contenido del elemento "para" tiene "Vamos que se puede!".

Los elementos pueden tener cualidades, para indicar mas información, por ejemplo:

```
<example id="ejemplo1">
.....
</example>
```

Un elemento es especial, el elemento de la "raiz" que es el elemento global, que contiene a todo el documento. En XML, la linea DOCTYPE indica que elemento es la raiz. En el Ejemplo 21-2 se muestra una linea de este tipo.

**Ejemplo 21-2. Elemento raiz de un documento**

```
<!DOCTYPE book PUBLIC "-//OASIS//DTD DocBook V4.1//EN">
```

### Archivos XML

Como no es el propsito de este documento el hablar explcitamente sobre XML, sólo diremos que los archivos XML comienzan primero con una "instrucción de proceso", que comienzan con <?, y en este caso indica que es un archivo XML, además de cierta meta-información. Esto lo vemos en el Ejemplo 21-3.

**Ejemplo 21-3. Encabezado de un archivo XML**

```
  <xml version="1.0" encoding="utf-8"?>
```

Los archivos XML deben ser "bien formateados", es decir, las etiquetas de comienzo deben siempre tener una etiqueta de término. También puede tener "elementos vacios" donde la etiqueta de inicio con la de término se convinan en una sola etiqueta escrita:

```
<etiqueta/>
```

### Qué es un DTD?

Un DTD (Document Type Definicion) es la descripción (en SGML) de un lenguaje específico. Puede escribir su propio DTD o puede usar un DTD ya existente, con la ventaja de poder intercambiar documentos con otras personas. Existe varios DTDs, tipicamente para propósitos de un grupo de gente dado (astrónomos, químicos,etc).

El DTD enumera los elementos y sus relaciones permitidas, por ejemplo, un "capitulo" debe tener por lo menos una "sección". Algunos DTDs que puede encontrar útiles son:

- DocBook: Utilizado para la documentación técnica especialmente sobre software.
- LinuxDoc: Utilizado por el proyecto de documentación de Linux, por ejemplo para el Linux-COMO.
- DebianDoc: Utilizado en parte por el proyecto de documentación de Debian.
- HTML: en teoría es un DTD pero muy poca páginas actuales respetan su definición.

## El DTD DocBook

Con docbook se pueden generar textos en diferentes formatos: html, latex, txt, pdf, postscript. Permite la creación de documentos tales como libros, articulos, manuales, etc. Se ha convertido junto a Open eBook en un estandar internacional para el procesamiento de este tipo de documentos.

### Instalación de programas

Para Debian/Sid instale los siguientes paquetes:

- sgml-base: Utilidades para mantención de catalogos SGML.
- sgml-data: SGML comn, XML DTDs y entidades.
- docbook: Sistema estandar SGML para la representación de documentos técnicos
- docbook-utils: Convertidor de archivos docbook a otros formatos.
- docbook-xsl: Estilos para el procesamiento de archivos DocBook XML.

Para instalar estos paquetes utilizamos el comando: apt-get install nombre_paquete. Si no encuentra estos nombres de paquetes busque con el comando: apt-get search sgml y apt-get search docbook los que correspondan.

## Donde Escribir el documento

Cualquier editor sirve para escribir nuestro documento, personalmente prefiero el "vim" con la opcin "syntax on" activada para el resalte de etiquetas con colores.

También puede usar Emacs con el módulo "SGML mode", el que permite escribir con mas facilidad estos documentos.

## Paso a otros formatos

Para pasar el documento fuente a otros formatos tenemos los siguientes comandos:

- db2dvi: genera un documento en formato dvi.
- db2html: genera un directorio con el nombre del archivo y el su interior archivos html.
- db2pdf: genera un documento en formato pdf.
- db2ps: genera un documento en formato postscript.
- db2rtf: genera un documento en formato rtf.

La instrucción para generar los distintos formatos es:

```
db2dvi fuente.sgml
db2html fuente.sgml
db2ps fuente.sgml
db2pdf fuente.sgml
db2rtf fuente.sgml
```

Si mientras el proceso de generación no arroja error entonces el nuevo documento ha sido creado correctamente.

## Tipos de Documentos

Docbook permite dos tipos de documentos; book (libro) y article (articulo). Los Articulos son mas sencillos de que los libros, estos no tienen índice de contenidos y ocupan menos hojas. Los libros permiten el uso de capítulos e índice de contenidos. Los libros además tienen la ventaja de ser que pueden contener más de un archivo, es decir, se pueden incluir otros archivos en un documento.

### Articulo

Como ya indicamos anteriormente, este tipo de documento es mucho mas sencillo que un libro. En el Ejemplo 21-4 vemos uno de estos documentos.

**Ejemplo 21-4. Ejemplo de un Articulo**

```
<!DOCTYPE article PUBLIC "-//OASIS//DTD DocBook V4.1//EN">
<article lang="es">
<articleinfo>
  <author>
    <firstname>Alejandro</firstname>
    <surname>Valdés Jiménez</surname>
  </author>
  <title>Ejemplo de un articulo</title>
</articleinfo>

<sect1 id="sect1">
  <title>Primera sección</title>
  <para>
  Contenido primera sección...
  </para>
</sect1>
</article>
```

## Ejemplo de un Libro

Docbook tiene otros elementos que se utilizan en este tipo de documentos y que permiten organizarlos de manera mas conveniente. En el Ejemplo 21-5 vemos como se crea un documento de este tipo.

**Ejemplo 21-5. Ejemplo de un Libro**

```
<!DOCTYPE book PUBLIC "-//OASIS//DTD DocBook V4.1//EN">
<book lang="es">
<bookinfo>
  <date>Diciembre 2003</date>
  <title>Ejemplo de un Libro</title>
  <author>
    <firstname>Alejandro</firstname>
    <surname>Valdés Jiménez</surname>
  </author>
</bookinfo>

<chapter id="chapter1">
  <title>Ambientes de escritorio</title>
  <para>
  Texto...
  </para>
  <sect1 id="sect1">
    <title>GNOME</title>
    <para>contenido de la sección 1</para>
  </sect1>
</chapter>
</book>
```

En el Ejemplo 21-5 notamos que el documento está todo escrito en un solo archivo, pero podemos formar nuestro documento final mediante la incorporación de otros archivos. En el Ejemplo 21-6 vemos como hacer esto.

**Ejemplo 21-6. Un libro creado con varios archivos**

```
<!DOCTYPE book PUBLIC "-//OASIS//DTD DocBook V4.1//EN"[
<!ENTITY capitulo2.sgml SYSTEM "capitulo2.sgml">
]>
<book lang="es">
<bookinfo>
  <date>Diciembre 2003</date>
  <title>Ejemplo de un Libro con varios archivos</title>
  <author>
    <firstname>Alejandro</firstname>
    <surname>Valdés Jiménez</surname>
  </author>
</bookinfo>

<chapter id="chapter1">
  <title>Ambientes de escritorio</title>
  <para>
  Texto...
  </para>
  <sect1 id="sect1">
    <title>GNOME</title>
    <para>contenido de la sección 1</para>
  </sect1>
</chapter>
&capitulo2.sgml;
</book>

-- capitulo2.sgml --
<chapter id="chapter1">
  <title>Sistemas Operativos</title>
  <sect1 id="Linux">
    <title>GNU/Linux</title>
    <para>Linux....</para>
  </sect1>
</chapter>
```

# Elementos

En esta sección veremos como se utilizan algunos elementos definidos para docbooks.

## Lista con items

- Item 1
- Item 2

Con docbook:

```
<itemizedlist>
  <listitem><para>Item 1</para></listitem>
  <listitem><para>Item 2</para></listitem>
</itemizedlist>
```

## Listas ordenadas

1. Item 1
2. Item 2

Con docbook:

```
<orderedlist numeration="arabic">
  <listitem><para>Item 1</para></listitem>
  <listitem><para>Item 2</para></listitem>
</orderedlist>
```

El atributo "numeration" define el tipo de numeración a utilizar, existen los siguientes valores:

- arabic: numeración arábica.
- loweralpha: alfanumérico en minúsculas.
- lowerroman: romano en minúsculas.
- upperalpha: alfanumérico en mayúsculas.
- upperroman: romano en mayúsculas.

## Tablas

**Tabla 21-1. Tabla de ejemplo**

| Titulo columna |
|---|
| fila 1 |
| fila 2 |

Con docbook:

```
<table id="tabla-ejemplo">
  <title>Tabla de ejemplo</title>
  <tgroup cols="1">
    <thead>
      <row>
      <entry>Titulo columna</entry>
      </row>
    </thead>
    <tbody>
      <row>
        <entry>fila 1</entry>
      </row>
      <row>
        <entry>fila 2</entry>
      </row>
    </tbody>
  </tgroup>
</table>
```

## Imgenes

**Figura 21-1. logo**

Con docbook:

```
<figure id="figura-ejemplo">
  <title>logo</title>
  <graphic fileref="xd2.png" format="PNG">
</figure>
```

## Referencias dentro de un documento

Todas las etiquetas tienen un atributo "id", el que permite asignarles un identificador, éste identificador es un texto que debe ser único. La idea de otorgarles este identificador es que eventualmente se les pueda hacer referencia desde cualquier parte del documento.

La línea siguiente muestra como hacer referencia a algún elemento del documento:

```
<para>
  en la <xref linkend="tabla-ejemplo"> se muestra una tabla.
</para>
```

## Enlaces a internet

Para hacer referencias a algna página o dirección de correo. Por ejemplo, la dirección web de GNOME Chile[1] y una dirección de correo[2]

```
<para>Una dirección web:
  <ulink url="http://www.gnome.cl/"> GNOME Chile </ulink>
</para>

<para>Una direccin de correo:
  <ulink url="mailto:avaldes@utalca.cl"> avaldes@utalca.cl </ulink>
</para>
```

## Notas al pie de página

Las notas al pie de página aparecen la final de la hoja en los documentos pdf y al final del documento html. El siguiente ejemplo es una nota al pie de página [3]

Con docbook:

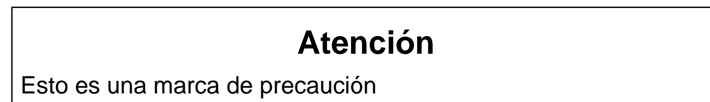
```
<para>
El siguiente ejemplo es una nota al pie de página
```

```
<footnote>
  <para>Para mas información visite: <ulink url="http://libros.es.gnome.org/">Libro de
</footnote>
</para>
```

### Marcas

Las marcas sirven para resaltar o señalar cierta información que se encuentra en el documento, tenemos:

- caution




- important

  **Importante:** Esta es una marca importante

- tip

  **Sugerencia:** Esta es una marca de sugerencia

Con docbook

```
<caution><para>Esto es una marca de precaucin</para>/caution>

<important><para>Esta es una marca importante</para></important>

<tip><para>Esta es una marca de sugerencia</para></tip>
```

## Notas

1. http://www.gnome.cl/
2. mailto:avaldes@utalca.cl
3. Para mas información visite:
    4. http://libros.es.gnome.org/
4. http://libros.es.gnome.org/

# Apéndice A. Cómo migrar aplicaciones a la plataforma Gnome 2.0

## Introducción

En los próximos meses, la plataforma GNOME irá moviéndose hacia su versión 2.0. Esto representa un hito importante en el desarrollo de GNOME e incorpora un gran número de cambios sobre las versiones actuales 1.x.

### Â¿A quién está dirigido esto?

Este documento pretende ofrecer una guía para los desarrolladores de aplicaciones que deseen portar productos desde la actual versión GNOME 1.4 a la versión 2.0. He basado estas notas en mi propia experiencia al portar un par de aplicaciones, mezclados con los comentarios de otros desarrolladores recibidos a través de email y discusiones en el IRC.

No puedo ofrecer una guía definitiva para portar cualquier aplicación en este documento. En particular, las tecnologías basadas en componentes (`ORBit` y `bonobo`) han sido omitidas, dada mi falta de experiencia con ellas y a que desde mi punto de vista su funcionamiento es pura magia. Posiblemente, una versión posterior de este documento incorpore información sobre estas cosas una vez que alguien escriba las secciones apropiadas.

Se asume un razonable nivel de familiaridad con las bibliotecas GNOME 1.4 actualmente existentes, dado que este es un documento sobre cómo migrar aplicaciones y no un tutorial sobre cómo escribir aplicaciones GNOME desde 0.

### Además de esta guía ...

En las siguientes secciones, sólo se cubren los aspectos técnicos de la migración a GNOME 2. Sin embargo, ésto no debería ser todo. Existe un esfuerzo concertado para hacer GNOME tanto accesible para personas con discapacidades como para hacerlo más amigable y sencillo de utilizar para todos los usuarios

El tema de accessibilidad se cubre brevemente em este documento en la sección titulada, sorprendentemente, Capítulo 11 . La facilidad de uso para el usuario se cubre en otro documento, escrito por el equipo GNOME Usability Project[1]. Puedes leer un borrador de su documento en GNOME Human Interface Guidelines[2], te recomendamos hacerlo. Para las aplicaciones significativas del proyecto GNOME 2, los problemas de usabilidad serán considerados bugs que podrían excluir a la aplicación de formar parte de la distribución GNOME o símplemente de ser ampliamente aceptada.

## Â¿Qué significa *migración*?

Este documento trata sobre la migración de aplicaciones a GNOME 2; ya lo sabes. Sin embargo, eso puede significar muchas cosas diferentes según la persona, por lo que en esta sección he intentado dar una idea de qué puede acarrear una migración en general y cuáles son los diferentes niveles.

No es descabellado pensar en la migración como una serie de pasos a seguir de entre los de la siguiente lista (no necesariamente en orden):

1. Realizar las conversiones mínimas necesarias para conseguir compilar y ejecutar la aplicación con las bibliotecas GNOME 2 (y cualquier otra biblioteca auxiliar).

2. Compilar sin necesidad de definir `GTK_ENABLE_BROKEN` .
3. Compilar con las constantes `GTK_DISABLE_DEPRECATED` y `GNOME_DISABLE_DEPRECATED` definidas
4. Modernizar la documentación. Esto no significa sólo el convertir la documentación a los estándares de GNOME 2. Hay más trabajo relacionado si quieres hacerlo realmente bien.
   - Si no tienes documentación para el usuario y estás escribiendo una aplicación que no es símplemente una biblioteca de funciones, entonces necesitas escribirla (o encontrar un voluntario que quiera hacerlo)
   - Si tu aplicación va a ser usada por otras aplicaciones (esto es, es una biblioteca o componente), entonces necesitas tener documentación sobre el API. Esto puede ser tan simple como poner los comentarios necesarios en el código y ejecutar `gtk-doc` como parte del proceso de compilación. Puedes ir mucho más allá, pero al menos *algo* de ayuda a los desarrolladores que quieran usar tu producto.
   - Limpia y ordena cualquier documentación que tengas por ahí. Existe un gran número de ficheros "flotando" por el repositorio CVS de GNOME que fueron en su momento básicamente "volcados" del contenido del cerebro de los desarrolladores hace dos años y ahora no están relacionados con nada. O tal vez tengas un documento que describe cómo pasar de una versión 2 a una versión 3 de tu aplicación, pero fue hace 8 meses que se editó por última vez y seguramente ahora esÂ´te obsoleto.

     Si no quieres borrar alguno de estos documentos históricos (lo cual es perfectamente entendible), al menos deberías moverlos a un nuevo directorio que se llame, por ejemplo,`old-docs` para aclarar su importancia en el proyecto.

   - Échale un vistazo a los ficheros `README`, `TODO`, `NEWS`, `AUTHORS` y `HACKING` (o `README.cvs-commits`) de tu proyecto. Â¿Están actualizados? Â¿Las direcciónes email de contacto siguen siendo válidasi y has indicado cláramente cuáles son las vías para contactar (algunos autores podrían haberse mudado sin posibilidad de contactar con ellos, por ejemplo)? Todos estos ficheros tienden a atrofiarse dado que las personas encargadas de actualizarlos se olvidan de ello y por tanto la gente deja de creer en ellos. Â¡Tu versión GNOME 2 es una ocasión perfecta para remediar esta situación (en caso de existir)!
5. Haz que tu aplicación se accesible. Â´Echale un vistazo a la sección Capítulo 11 de este documento y revisa el sitio web del Proyecto de Accesibilidad [3].
6. Usa `gconf` en lugar de `gnome-config.h` para todo lo relacionado con tu configuración tanto como puedas. Así mismo, utiliza la configuración de la propia plataforma `gconf` para la configuración de cosas como los proxys HTTP. Esto significará que tus usuarios pueden configurar todo de una forma consistente y no necesitarán hacerlo para todas y cada una de las aplicaciones.
7. Diseña tu aplicación para que funcione en un entorno donde el usuario puede haber hecho login muchas veces simultáneamente. En otras palabras, debes estar preparado para gestionar parte del estado de la aplicación basándote en la sesión en particular que la está usando, si es necesario.
8. Implementa las recomendaciones de las *Human Interface Guidelines*[4], como se menciona en la introducción.

Saber cuántos de estos requerimientos deberías de implementar en tu propia aplicación es una decisión personal tuya y de los demás desarrolladores del proyecto. Sin embargo, no es algo descabellado esperar que las principales aplicaciones en GNOME 2 puedan cumplir con todos los criterios y otras aplicaciones populares intenten completar todas aquellas que les sea posible.

Como regla general, el funcionar correctamente con `gconf` y la gestión de sesiones son dos nuevas adiciones para GNOME 2, pero deberían considerarse muy importantes como parte del "juego limpio" con el resto de la plataforma. De forma similar, la accesibilidad y la usabilidad son dos facetas a las que se les está prestando mucha atención en GNOME 2 y las aplicaciones que no hagan un esfuerzo en éstas áreas son candidatas a destacar del resto (y no de forma positiva)

### Consideraciones generales

Los siguientes elementos no están realmente relacionados con los aspectos técnicos de la migración, pero son a menudo infravalorados y se podría hacer algo más que considerarlos durante la fase "de limpieza" de tu proceso de migración.

- Asegúrate de que el comando **make distcheck** funciona con tu aplicación. Preferiblemente, intenta que funcione siempre. Esto asegurará que tu aplicación tiene todos los ficheros necesarios cuando crees el paquete tar y que, entre otras cosas, asegurará la compilación de la aplicación cuando el directorio de compilación no sea el mismo que el directorio original (un problema al que no se le dedica la atención necesaria)
- Antes de liberar la versión compatible con GNOME2 de tu aplicación, intenta compilarla desde cero en un sistema "limpio". Demasiado a menudo, el sistema principal del desarrollador tendrá pequeñas piezas de versiones o instalaciones de bibliotecas anteriores que estén siendo accidentalmente usadas por el nuevo paquete. O descubrirás que algo que asumías que estaría disponible en una instalación estándar realmente necesita ser descargado de forma separada. Poner una dosis de atención extra a este tipo de detalles antes de lanzar una nueva versión te evitará perder un montón de tiempo e incomodidades a tí y a tus usuarios en el futuro.

## Las bibliotecas de la plataforma

GNOME 2 consistirá en un grupo de bibliotecas de plataforma, algunas bibliotecas que no serán de plataforma y otro montón (esperemos que así sea) de aplicaciones que dependerán de los anteriores dos grupos.

Probablemente sea necesario clarificar las diferencias que existen entre una biblioteca *de plataforma* frente a una biblioteca que no lo sea, dado que realmente estaremos discutiendo sobre las bibliotecas de plataforma en las siguientes secciones. Algún email[5] de Maciej Stachowiaw a la lista de correo gnome-2-0 probablemente lo explique lo más sucíntamente posible:

> Una "biblioteca de plataforma" es aquella en la que nos comprometemos a apoyar y dar soporte como parte integrante de la plataforma y que recomendamos a los desarrolladores internos. Algunos aspectos de esto, incluyen la adhesión a la compatibilidad total tanto a nivel fuente como a nivel binario dentro de una versión principal (major version) de la plataforma GNOME, y con una licencia apropiada tanto para su uso por el software libre como por el software propietario [6] .
>
> Pero incluso en el caso de aquellas bibliotecas cuyos responsables de mantenerlas estén deseando seguir estos criterios, podríamos no querer incluírlas si creemos que no están lo suficientemente maduras, si duplican funcionalidades ya presentes en la plataforma, si no interactúan bien con otras partes de la plataforma, etc.

Actualmente, las siguientes bibliotecas forman parte de la plataforma GNOME 2. El orden de esta lista es un posible orden de compilación que funciona (ver la sección de nombre *Descargando los paquetes* para más información).

**Tabla A-1. Módulos de bibliotecas de plataforma GNOME 2**

| | | |
|---|---|---|
| intltool | gnome-common | esound [a] |
| glib | pango | atk |
| gtk+ | libxml2 [b] | linc |
| libIDL | ORBit2 | bonobo-activation |
| gconf | libbonobo | gnome-vfs |
| libart_lgpl | bonobo-config | libgnome |
| libgnomecanvas | libbonoboui | libgnomeui |
| libglade | gail | libgnomeprint |
| libgnomeprintui | libxslt | |

Notas de Tabla:
a. Esta biblioteca simplemente se ha portado desde la plataforma GNOME 1 sin cambios.
b. En el repositorio GNOME CVS, se llama `gnome-xml`.

## Preparación del ambiente

Naturalmente, antes de que comience cualquier trabajo de migración, necesitas disponer de todas las herramientas y bibliotecas necesarias con las que compilar. En esta sección se explicará cómo recopilar el software y cómo configurar tu sistema para compilar y ejecutar aplicaciones con el entorno GNOME 2.

### Requerimientos para compilar los módulos

Aquí tenemos una lista de las herramientas que necesitarás para conseguir compilar las bibliotecas de plataforma GNOME 2 y para compilar tus aplicaciones contra estas bibliotecas.

**Tabla A-2. Paquetes requeridos para compilar la plataforma GNOME 2**

| Paquete | Mínima Versión Requerida |
|---|---|
| autoconf | 2.52 |
| automake | 1.4-p4 |
| libtool | 1.4 |
| pkgconfig | 0.8.0 |
| gettext | 0.10.40 |

Algunas notas generales sobre estos paquetes:

- Los paquetes `autoconf`, `automake` y `libtool` sólo son necesarios si estás compilando las bibliotecas desde el CVS.
- La versión requerida de `automake` no es la última versión disponible (o sea, la versión 1.5 1.5). Si estás usando la versión 1.5 de `automake`, te serán de utilidad los siguientes consejos (enviados originalmente[7] a la lista gnome-2-0 por James Henstridge):
  - Si la compilación falla cuando se intenta construir algunos ficheros que contengan código ensamblador, necesitarás añadir la línea

    ```
    AM_PROG_AS
    ```

a tu fichero `configure.in`. Esto será necesario actualmente cuando compiles `GTK+` con la versión 1.5, por ejemplo.

- Algunos paquetes fallarán cuando ejecutes **make distcheck**. Existen fundamentalmente dos razones para este comportamiento: primero, el directorio fuente para la compilación de test ha pasado a modo sólo-lectura durante la compilación. Puedes evitar este problema si no pones ningún fichero generado durante la compilación en el directorio fuente (esto es un problema para la documentación) ó poniendo explícitamente:

```
chmod u+w $(sourcedir)
```

  en las reglas de tu `Makefile`.

  La otra razón por la que un distcheck puede fallar es si un **make uninstall** realmente no consigue desinstalar todos los ficheros instalados por un **make install**. De forma similar, si **make distclean** deja ficheros generados en el directorio de compilación, `automake 1.5` fallará.

- Es posible usar una versión de `gettext` anterior a la 0.10.40. Las versiones 0.10.38 y superiores son correctas. Sin embargo, la 0.10.38 acarreará problemas ocasionales cuando funcione en combinación con `autoconf 2.52`, lo que se ha arreglado ya en la versión 0.10.39 y 0.10.40.

La aplicación `pkg-config` es un intento de evitar la necesidad de que cada paquete y biblioteca tengan su propio script de configuraciñon para pasar las opciones necesarias de compilador y enlazador de C al script **configure**. Ahora cada paquete simplemente instala un fichero `package.pc` que contiene información sobre el destino de instalación del paquete, de qué otros paquetes depende, y qué opciones de compilador y enlazador necesita. El tema de migrar tu aplicación para que use `pkg-config` se cubre en la sección titulada la sección de nombre *Cambios al entorno de compilación*.

## Descargando los paquetes

Existen pocas alternativas para descargar todas las bibliotecas necesarias — desde el CVS, de tarballs pre-empaquetados o en un formato de empaquetamiento adecuado a tu distribución.

Los tarballs pueden descargarse de ftp://ftp.gnome.org/pub/gnome/pre-gnome2/latest/sources/. Sin embargo, debido al pobre rendimiento general de ftp.gnome.org, se recomienda que elijas un mirror de http://www.gnome.org/mirrors/ftpmirrors.php3 y desde ahí elijas el directorio equivalente.

Las rutas de instalación de estos paquetes han sido convenientemente configuradas para hacer posible la instalación de las bibliotecas GNOME 2 al mismo tiempo que dispones de los paquetes GNOME 1. Por ejemplo, si los ficheros de cabeceras de GNOME 1 para `libgnome` se instalan (típicamente) en `/usr/include/libgnome`, las versiones GNOME 2 se instalarán en `/usr/include/libgnome-2.0/libgnome`.

Si utilizas consistentemente `pkg-config` para averiguar las rutas de los includes necesarios y de las opciones de biblioteca, todos estos detalles serán tenidos en cuenta por el sistema. Es suficiente con saber que las dos plataformas pueden ser instaladas simultáneamente.

> **Nota:** Algunos de los paquetes separados listados en la sección la sección de nombre *Las bibliotecas de la plataforma* pueden ser mezclados en un sólo tarball, por lo que no te preocupes si parece que no hay exactamente el número correcto de tarballs.

Los paquetes específicos de cada distribución pueden venir de distintas fuentes.

- Algunos paquetes Debian para Sid están disponibles en los mirrors habituales. Por el momento, puedes ejecutar **apt-cache search gtk+ 1.3** para descargar e instalar `gtk+`, `glib`, `atk` y `pango`. A medida que se disponga de más paquetes, podrás descargarlos de un forma similar (y esta nota será actualizada para reflejar el status más reciente).
- Los paquetes para Red Hat 7.2, llamados en su conjunto *gnomehide*, han sido compilados por Havoc Pennington y están disponibles en ftp://people.redhat.com/hp/gnomehide/. También existe una lista de distribución para estos paquetes (sólo para problemas de compilación e instalación) en http://mail.gnome.org/mailman/listinfo/gnome-redhat-list.
- Jacob Berkman de Ximian[12] está realizando compilaciones diarias de todos los paquetes y dejándolas disponibles a través de la aplicación de gestión de paquetes Red Carpet[13] de Ximian, dentro del canal de desarrolladores (developers). Antes de utilizar estos paquetes, es recomendable que leas este documento http://primates.ximian.com/~jacob/gnome-2-snapshots/.
- Había un rumor flotando por ahí que decía que Sun estaba realizando compilaciones nocturnas y que iba a dejar disponibles para su plataforma. Se añadirán más detalles sobre esto cuando se confirme o se deniegue este rumor.

Para instalar y compilarlo todo desde el CVS, necesitarás hacer un checkout de todos los módulos mencionados en la sección de nombre *Las bibliotecas de la plataforma* y compilarlos en el orden especificado ahí.

Una de las formas más sencillas para conseguir todos los módulos adecuados desde el CVS y compilarlos es bajándose el módulo llamado `vicious-build-scripts` del repositorio GNOME CVS. Lee el fichero `README` de este módulo detenidamente y sigue las instrucciones de instalación. Después podrás (opcionalmente) actualizar automáticamente tu árbol de directorios y compilar la mayoría de los módulos necesarios en el orden correcto.

Por el momento, `vicious-build-scripts` no incluye `gail`, `libgnomeprint`, ó `libgnomeprintui`. Sin embargo, tras compilar todos los módulos anteriores, compilar estos últimos a mano si realmente son necesarios es un proceso bastante directo.

## Otros paquetes necesarios para ejecutar GNOME 2

La plataforma GNOME 2 dependerá también de unos pocos paquetes extra que no están mantenidos por los desarrolladores de GNOME, por lo que se espera que los usuarios los traten de forma independiente y que tu aplicación compruebe su presencia si es necesario.

La mayoría de estos paquetes extra formarán parte de cualquier distribución estándar de Linux o Unix. Sin embargo, algunos no son tan comunes, por lo que aquí presentamos la lista de los que necesitarás instalar de forma especial.

**Tabla A-3. Paquetes extra requeridos para ejecutar GNOME 2**

| Paquete | Versión Mínima | Descargar desde ... |
| --- | --- | --- |
| scrollkeeper | 0.2 | http://scrollkeeper.sourceforge.net/ |

| Paquete | Versión Mínima | Descargar desde ... |
|---|---|---|
| Hojas de estilo DocBook de Norm Walsh [a] | *(desconocido)* | http://sourceforge.net/projects/docbook/ |
| Notas de Tabla:<br>a. Se usan como plantillas para algunas hojas de estilo personalizadas de GNOME a la hora de convertir documentos de ayuda. | | |

**Nota:** Los enlaces de descarga superiores son sólo punteros a la página inicial de cada proyecto. También podrás conseguir paquetes para tu distribución particular desde las fuentes habituales.

## Miscelánea de consejos de configuración

*(Con agradecimientos a Alan Cox.)*

### Pango

Para configurar Pango para probarlo, crea un fichero en tu directorio home y dale el nombre `.pangorc` y escribe estas líneas como su contenido:

```
[PangoFT2]
 FontPath = /usr/share/fonts/default/Type1:/usr/share/fonts/default/TrueTyp
```

Además, copia el fichero `examples/pangoft2.aliases` del directorio fuente de `pango` al fichero `.pangoft2_aliases` en tu directorio home.

### Compilando Red Hat 7.1

Compilar toda la plataforma desde cero requiere al menos la versión 2.0 de `Python`. Esto es debido a un script de conversión requerido por `libglade` lee la sección sobre Capítulo 12 para más información).

Si dispones de una instalación estándar de Red Hat 7.1, necesitarás conseguir el rpm `python2` desde los paquetes Rawhide, dado que la versión que viene con las Power Tools de la 7.1 no incluye el soporte XML necesario. Un lugar desde donde descargar el rpm necesario es desde www.rpmfind.net[15].

## Requerimientos de espacio en disco

GNOME no consume cero recursos de espacio en disco. De hecho, requiere bastante espacio, aunque en estos días de discos de multi-gigabytes, no es algo serio. Sin embargo, si el espacio en disco necesario es algo que te preocupe, aquí hay algo de información al respecto. Son números aproximativos — existen algunas variaciones debido a las opciones de empaquetamiento, por ejemplo — y están basados únicamente en mis observaciones en un par de máquinas.

- Las fuentes para las bibliotecas de plataforma requieren unos 170MB antes de ser compiladas (este es el tamañano del código fuente -- es algo menos en el CVS porque algunos de los ficheros son generados como parte del script `autogen.sh` y no son limpiados tras el **make distclean**).

- La plataforma base una vez instalada (sin las fuentes y sin limpiar (stripping) ninguna biblioteca) ronda los 200MB.
- Si estás compilando las bibliotecas usando `vicious-build-scripts` y la opción `LEAN=yes` (que ejecuta **make clean distclean** tras la instalación de cada módulo), necesitarás aproximadamente unos 140MB de espacio en disco además de los ya requerido por las fuentes. El módulo especialmente voluminoso es `gtk` — si tienes suficiente espacio para compilarlo, no tendrás problemas para compilar el resto de las fuentes.
- Compilar todas las bibliotecas de plataforma requiere aproximadamente unos 800MB, incluyendo el código fuente y excluyendo el espacio requerido para instalar las bibliotecas en su destino final.

## Cambios al entorno de compilación

En general, la configuración de la compilación para una aplicación GNOME 2 será muy parecida a la configuración para GNOME 1. Los únicos cambios requeridos son debidos a versiones más modernas de las herramientas como `autoconf` y para gestionar la inclusión de `pkg-config`, en lugar de scripts específicos de biblioteca como `gnome-config`.

### Cambios a `autogen.sh`

El único cambio que deberías tener en cuenta aquí es el hacer que el script `autogen.sh` de tu paquete llame al script `gnome-autogen.sh` con la variable de entorno USE_GNOME2_MACROS inicializada. Por ejemplo, observa el script sample autogen.sh que aparece más abajo. En general es bastante seguro simplemente copiar el `autogen.sh` de otro sitio, como por ejemplo de los módulos `gnome-hello` y luego símplemente modificar el nombre del paquete.

**Ejemplo A-1. Fichero `autogen.sh` de ejemplo**

```
#!/bin/sh
# Run this to generate all the initial makefiles, etc.

srcdir=`dirname $0`
test -z "$srcdir" && srcdir=.

PKG_NAME="Gnome Hello Demonstration Application"

(test -f $srcdir/configure.in \
  && test -f $srcdir/ChangeLog \
  && test -d $srcdir/src) || {
    echo -n "**Error**: Directory "\`$srcdir\'" does not look like the"
    echo " top-level $PKG_NAME directory"
    exit 1
}

which gnome-autogen.sh || {
    echo "You need to install gnome-common from the GNOME CVS"
    exit 1
}
USE_GNOME2_MACROS=1 . gnome-autogen.sh
```

### Pkg-config

Como se mencionó en la corta descripción de pkg-config, ésta es la nueva forma de hacer las cosas que se hacían anteriormente con llamadas a scripts específicos de cada aplicación como `gnome-config`.

Puedes listar todas las bibliotecas que están bajo el control de `pkg-config` ejecutando el comando **pkg-config --list-all**. Si lo haces, verás que todas las bibliotecas que son parte de la plataforma GNOME 2 son conocidas pra `pkg-config`.

En la mayoría de los casos, es suficiente con buscar las bibliotecas de plataforma que aparecen en la *parte superior* del árbol de dependencias. Esto es, la biblioteca cuyas dependencias incluye a todas las otras bibliotecas que necesitarás. Normalmente, `libgnomeui-2.0` suele ser suficiente, dado que sus dependencias incluyen a `gtk+`, `gconf` y `libbonoboui` entre otras. .

Si crees que necesitas algo más específico o más complejo que ésto, échale un vistazo a la página del manual de `pkg-config` para más información.

Para usar paquetes gestionados por `pkg-config`, todos los cambios están en `configure.in`,que se cubre en la sección sobre required changes to configure.in. Si lo que buscas es hacer usable tu propio paquete via `pkg-config`, entonces necesitas crear un fichero `foo.pc` apropiado (en caso de que tu paquete se llamara `foo`).

Normalmente, no crearías el fichero `foo.pc` directamente, dado que contiene información sobre las localizaciones de los componentes instalados de `foo`. En lugar de eso, crearías el fichero `foo.pc.in` con las variables apropiadas que serán luego rellenadas desde `configure.in`. A modo de ejemplo, aquí mostramos una pequeña variación del fichero `gobject.pc.in` (la línea `Conflicts` no es normal y hay algunos comentarios extra). Este fichero será luego convertido en `gobject.pc` gracias a una línea apropiadamente puesta en la sección AC_OUTPUT del fichero `configure.in`.

**Ejemplo A-2. fichero de ejemplo `gobject.pc.in`**

```
# This is a comment
prefix=@prefix@
exec_prefix=@exec_prefix@
libdir=@libdir@
includedir=@includedir@

Name: GObject                            # human-readable name
Description: Object/type system for GLib # human-readable description
Version: @VERSION@
Requires: glib-2.0 = 1.3.1
Conflicts: foobar <= 4.5
Libs: -L${libdir} -lgobject-1.3
Cflags: -I${includedir}/glib-2.0 -I${libdir}/glib/include
```

La mayor parte de lo aquí expuesto no debería sorprender demasiado a aquellas personas acostumbradas a compilar programas: especificar las bibliotecas requeridas, el enlazado y las opciones de C para este paquete y para cualquiera que entre en conflicto con éste. Para más ejemplos, observa la localización estándar donde `pkg-config` almacena los ficheros `*.pc` (normalmente `$(prefix)/lib/pkgconfig/`.

## Cambios al `configure.in`

El fichero `configure.in` es probablemente el fichero que más cambios requiere de todos los que forman parte de la infraestructura de compilación de la aplicación. Hay varias razones para esto.

En GNOME 2, la necesidad de tener un subdirectorio `macros/` para cada módulo CVS, conteniendo un grupo completo de macros `.m4` comunes, ha sido

eliminada. En su lugar, las macros comunes son parte del módulo `gnome-common` y normalmente estarán disponibles (cuando estén instaladas) bajo el directorio `$(prefix)/share/aclocal/gnome2-macros`.

De forma similar, en general hay poca necesidad de usar `gettext` del directorio `intl/` que aparece en la mayoría de los módulos, dado que `gettext` está por lo general ya instalado en la máquina del desarrollador.

Finalmente, se requieren algunos cambios dada la actualización de varias herramientas. Por ejemplo, algunos están recomendados como parte de la actualización a `autoconf 2.52`. Estos cambios permiten mejores mensajes de diagnóstico, la mayor parte de las veces. La inclusión de `pkg-config` también requerirá añadir unas cuantas llamadas.

Por supuesto, como en todas las sugerencias de este documento, si necesitas algo más complejo que lo anterior, seguro que ya sabes por qué necesitas algo diferente y tienes el suficiente conocimiento para cambiarlo. Este documento símplemente ofrece una lista de comprobación para la mayoría de las aplicaciones; no se asegura que funcione para aplicaciones con necesidades extrañas o para casos raros y especiales.

Ahora que las justificaciones ya han sido dadas, aquí se muestran los pasos que funcionarán en la mayoría de los casos:

1. Borrar cualquier llamada a la macro GNOME_COMMON_INIT. Está obsoleta dada la forma en que `gnome-autogen.sh` configura actualmente el entorno.
2. En `autoconf 2.52`, la macro AC_INIT toma ahora dos parámetros, mas un tercero opcional. Los parámetro son *`package-name`*, *`version-number`* y *`bug-address`*, siendo éste último opcional.

   En GNOME 1, tu fichero `configure.in` habría tenído una línea que dijera algo como

   ```
   AC_INIT(foo.c)
   AM_INIT_AUTOMAKE(foo, 1.23)
   ```

   Para GNOME 2, esto debería cambiarse por

   ```
   AC_INIT(foo, 1.23, http://bugzilla.gnome.org/enter_bug.cgi?product=foo)
   AC_CONFIG_SRCDIR(foo.c)
   AM_INIT_AUTOMAKE(AC_PACKAGE_NAME, AC_PACKAGE_VERSION)
   ```

   Observa que AM_INIT_AUTOMAKE se beneficia del hecho de que AC_INIT() ahora define las variables `AC_PACKAGE_NAME` y `AC_PACKAGE_VERSION`.

3. Incluir las macros adecuadas para `pkg-config` es un proceso relativamente simple y directo una vez que determinado el tope superior de tu árbol de dependencias de entre las bibliotecas de la plataforma . En lo que sigue, asumimos que el paquete que quieres compilar se llama `foo`.

   Para conseguir las opciones de C y del enlazado de bibliotecas, usa la siguiente secuencia de macros.

   ```
   PKG_CHECK_MODULES(FOO, libgnomeui-2.0)
   AC_SUBST(FOO_CFLAGS)
   AC_SUBST(FOO_LIBS)
   ```

   Ahora tu fichero `Makefile.am` puede hacer referencias a las variables `$(FOO_CFLAGS)` y `$(FOO_LIBS)` en las líneas de asignación `INCLUDES` y `LDADD` respectivamente.

   En el inusual caso de que requirieras ciertas versiones mínimas de algunas bibliotecas (por ejemplo, para evitar un bug presente en una versión anterior), podrías hacer algo como lo que se muestra en el siguiente extracto.

```
dnl En general es una 'buena práctica' poner los requerimientos de  pkg-config
dnl al comienzo, para que los cambios en sucesivas versiones sean más fáciles de r
GTK_REQUIRED=1.3.1
LIBGNOMEUI_REQUIRED=1.96.0

PKG_CHECK_MODULES(FOO, gtk+-2.0 >= $GTK_REQUIRED
                       libgnomeui-2.0 >= $LIBGNOMEUI_REQUIRED)
AC_SUBST(FOO_CFLAGS)
AC_SUBST(FOO_LIBS)
```

4. Finalmente, si deseas usar una versión instalada en local de `gettext`, se recomienda usar AM_GLIB_GNU_GETTEXT. Esto reemplaza cualquier llamada a AM_GNOME_GETTEXT o incluso AM_GNOME2_GETTEXT.

   También necesitarás definir la constante `GETTEXT_PACKAGE` usando algo como lo siguiente.

   ```
   GETTEXT_PACKAGE=foo
   AC_SUBST(GETTEXT_PACKAGE)
   AC_DEFINE_UNQUOTED(GETTEXT_PACKAGE, "$GETTEXT_PACKAGE")
   ```

   Observa que si tu aplicación está pensada para ser instalada en paralelo junto a su versión para GNOME 1, deberías entonces escribir `GETTEXT_PACKAGE=foo-2.0`. En cualquier caso, cada vez que llames a funciones `gettext` como `bindtextdomain()` ó `bind_textdomain_codeset()` en tu código, deberías utilizar `GETTEXT_PACKAGE` en lugar del nombre literal del paquete como primer argumento para estas funciones.

5. Elimina todas referencias a `intl/Makefile`, `macros/Makefile`. Éstas ya no se necesitan, dados los cambios anteriores.
6. Si estás generando funciones para hacer marshalling de señales (signal marshallers) en `gtk`, necesitarás determinar la localización de **glib-genmarshal**. El cambio requerido en el fichero `configure.in` .

### Cambios en `Makefile.am`

Normalmente, no necesitarás realizar cambios significativos en tu `Makefile.am`, dado que símplemente contiene la cantidad mínima de información necesaria para poder compilar tu aplicación. Lo único necesario sería eliminar cualquier referencia a los directorios `intl` y `macros` en variables como `DIST_SUBDIRS` y `SUBDIRS`.

## Notas

1. http://developer.gnome.org/projects/gup/
2. http://developer.gnome.org/projects/gup/hig/
3. http://developer.gnome.org/projects/gap/
4. http://developer.gnome.org/projects/gup/hig/
5. http://mail.gnome.org/archives/gnome-2-0-list/2001-September/msg00141.html
6. Todas las bibliotecas de plataforma actuales están licenciadas bajo la LGPL.
7. http://mail.gnome.org/archives/gnome-2-0-list/2001-August/msg00212.html
8. ftp://ftp.gnome.org/pub/gnome/pre-gnome2/latest/sources/

9. http://www.gnome.org/mirrors/ftpmirrors.php3
10. ftp://people.redhat.com/hp/gnomehide/
11. http://mail.gnome.org/mailman/listinfo/gnome-redhat-list
12. http://www.ximian.com
13. http://www.ximian.com/products/ximian_red_carpet/
14. http://primates.ximian.com/~jacob/gnome-2-snapshots/
15. http://www.rpmfind.net/linux/rpm2html/search.php?query=python