

**UNIVERSIDAD IBEROAMERICANA**

**Programación en GTK+ 2.6**

Apuntes por:  
Ing. Andrés Tortolero Baena

<b>CAPÍTULO 1 .....</b>	<b>6</b>
PROGRAMACIÓN EN GTK+ 2.6 .....	6
INTRODUCCIÓN .....	6
1. Sistema X-Windows .....	6
2. Introducción a la programación orientada a objetos .....	7
2.1. Encapsulamiento .....	7
2.2. Clases .....	7
2.3. Clases y Objetos .....	7
2.4. Herencia .....	8
2.5. Polimorfismo .....	8
3. Introducción a GTK .....	8
3.1. GTK y la programación orientada a objetos .....	9
3.2. Compilación de un programa en GTK .....	13
3.3. Creación de un Makefile .....	13
4. Revisión de las aplicaciones de GTK+ .....	14
5. Los 7 pasos para la programación básica en GTK+ .....	18
5.1. Inicializar el ambiente .....	19
5.2. Crear los widgets y fijar sus atributos .....	20
5.3. Registrar las llamadas a las funciones .....	23
5.4. Definir la jerarquía de las instancias del programa .....	25
5.5. Mostrar los widgets .....	26
5.6. Procesar las señales y los eventos .....	27
5.7. Salir de la aplicación .....	27
6. Más sobre el control de un programa .....	28
7. Glib .....	30
7.1. Tipos de Datos de Glib .....	30
7.2. Macros Estándar .....	31
7.3. Macros Matemáticas .....	31
7.4. Detección de Errores .....	31
7.5. Otras funciones .....	32
7.6. Listas (una my sencilla introducción) .....	32
7.6.1. Creación y Destrucción de Listas .....	32
7.6.2. Inserción de Elementos en una lista .....	32
7.6.3. Preinsertar y anexar .....	32
7.6.4. Movimiento en la lista .....	33
7.7. Manejo de Memoria .....	33
7.8. Funciones útiles .....	33
8. Tipos de Datos Extendidos .....	33
9. Creación y Destrucción de Cadenas .....	33
9.1. Anexar y Preinsertar Cadenas .....	34
9.2. Inserción de una cadena dentro de otra .....	34
10. Timers (Cronómetros) .....	34
10.1. Creación y Destrucción de un Timer .....	35
10.2. Iniciar, Detener y Reiniciar el Cronómetro .....	35
10.3. Revisión del Cronómetro .....	35
11. Señales y Eventos .....	35
12. Elementos de la Interfase de Usuario .....	36
12.1. Elementos Adicionales en una Ventana .....	36
12.1.1. Título .....	36
12.1.2. Tamaño y Posición de una Ventana .....	36
12.1.3. Controles en las Ventanas .....	37
12.2. Botones .....	37

12.3.	Cajas de Empaquetamiento (Packing Boxes).....	37
12.3.1.	Definición.....	38
12.3.2.	Cajas para botones.....	39
12.4.	Establecimiento del tamaño de un widget.....	39
12.5.	GtkLabel -Widget para Etiquetas.....	39
12.6.	Flechas ( <i>Arrows</i> ).....	41
12.7.	GtkEntry - Obteniendo Texto del Usuario.....	41
12.8.	Implementación de una barra de menús.....	42
12.8.1.	Accesos directos en los menús ( <i>Aceleradores</i> ) .....	46
12.8.2.	Aceleradores subrayados.....	47
12.8.3.	Poniendo submenús dentro de los menús.....	48
12.8.4.	Poniendo íconos en un menú.....	51
12.9.	GtkTable widget.....	52
12.10.	File Selection Widget.....	54
12.11.	Font Selection Widget.....	58
12.12.	Barras de desplazamiento.....	60
12.13.	GtkScale.....	62
12.14.	Frames.....	63
12.15.	Calendario.....	64
12.16.	Simplificación de las tareas.....	64
12.17.	Compartiendo la información.....	65
12.18.	Diálogos.....	65
12.20.1.	Dialogos de Acerca de.....	67
12.19.	Diálogos y GtkWindow.....	68
12.20.	Modalidad.....	68
12.21.	Removiendo Ventanas.....	69
12.22.	gtk_main.....	69
12.23.	Tool tips.....	70
12.24.	Barras de Herramientas.....	71
12.25.	Barras de Estado.....	72
13.	Añadiendo gráficos a la aplicación.....	73
13.1.	Puntos.....	73
13.2.	Líneas.....	74
13.3.	Rectángulos.....	76
13.4.	Polígonos.....	77
13.5.	Arcos.....	77
13.6.	Dibujar texto.....	78
14.	Manejando GDK Events.....	79
14.1.	La estructura GdkEventButton.....	82
14.2.	Seleccionando Eventos.....	83
15.	Cambiar un texto en una etiqueta.....	83
16.	Hacer aparecer un widget.....	83
17.	Más sobre listas.....	84
18.	Más sobre Botones.....	84
18.1.	Toggle buttons.....	85
18.2.	Check buttons.....	85
18.3.	Radio buttons.....	86
19.	Combo Box.....	86
20.	Más opciones para un menú.....	89
20.1.	GtkTearOffMenuItem.....	89
20.2.	GtkRadioMenuItem.....	89
20.3.	GtkCheckMenuItem.....	90
21.	Más sobre contenedores.....	90
21.1.	GtkNoteBook widgets.....	90
21.2.	GtkFixed Widgets.....	92

21.3	GtkLayout Widgets .....	92
21.4	Paned Widgets.....	93
22.	Añadiendo imágenes a una aplicación .....	94
23.	Funciones especiales .....	94
<b>BIBLIOGRAFÍA .....</b>		<b>95</b>

**Índice de Ilustraciones**

FIGURA 1.- GRÁFICA DE HERENCIAS.....	10
FIGURA 2.- ÁRBOL DE CLASES DE GTK .....	11
FIGURA 3.- CLASES DE GTKWIDGET .....	12
FIGURA 4.- ÁRBOL DE INSTANCIAS .....	15
FIGURA 5.- ÁRBOL DE INSTANCIAS .....	15
FIGURA 6.- ÁRBOL DE CLASES DE GTKWINDOW .....	16
FIGURA 7.- ÁRBOL DE CLASES DE GTKBOX .....	16
FIGURA 8.- EJEMPLOS DE VBOX Y H BOX.....	17
FIGURA 9.- ÁRBOL DE CLASES DE GTKLABEL .....	17
FIGURA 10.- ÁRBOL DE CLASES DE GTKSEPARATOR.....	17
FIGURA 11.- ÁRBOL DE CLASES DE GTKBUTTON .....	18
FIGURA 12.- ÁRBOL DE INSTANCIAS COMPLETO .....	18
FIGURA 13.- FUNCIONES GTK_*_SET_* PARA UN WIDGET DE TIPO GTKWINDOW .....	22
FIGURA 14.- TIPOS DE DATOS EN GLIB .....	30
FIGURA 15.- EJEMPLO DE UNA INTERFASE GRÁFICA .....	39
FIGURA 16.- ÁRBOL DE CLASES DE LA CLASE GTKMENUITEM.....	43
FIGURA 17.- DISTRIBUCIÓN DE UN GTKTABLE.....	53
FIGURA 18.- ÁRBOL DE CLASES DE LA CLASE GTKSCROLLEDWINDOS .....	61

## Programación en GTK+ 2.6

### INTRODUCCIÓN

**Hasta hace poco la gente acusaba a Unix y particularmente a Linux de ser un sistema operativo poco amigable, aunque se reconocía su poder. Con la disponibilidad de poderosas herramientas que nos permitan crear interfaces gráficas de usuario, esto ha ido cambiando. Utilizando GTK+, GNOME y otras bibliotecas podremos escribir aplicaciones con interfaces gráficas.**

#### 1. Sistema X-Windows

X-Windows fue desarrollado en el MIT en 1984. El equipo de Ciencias de la Computación creó un ambiente gráfico de usuario para que se pudiera utilizar en una máquina o que permitiera que otras máquinas pudieran utilizar los servicios provistos por un equipo central.

XFree86 se distribuye sin costo para Linux pero es únicamente una interfase para que otras aplicaciones puedan escribir en el cliente. No es tan poderoso, para esto se necesita un Administrador de Ventanas, que provee una interfase al usuario para correr aplicaciones, navegar en el disco duro de la computadora o hacer cualquier actividad que un usuario pueda esperar. (Proveen un escritorio virtual)

A GNOME (GNU Network Object Model Environment) se le conoce como un administrador de escritorio. Mientras que un administrador de ventanas se contruye sobre X y nos permite usar un sistema, GNOME se construye sobre un administrador de ventanas y provee un mayor conjunto de servicios, gráficos y no gráficos, para hacer la interfase de usuario más poderosa.

GNOME utiliza las bibliotecas de GTK+ para dibujar las interfaces de usuario, es por esto que la visualización y métodos de acceso de las interfaces son comunes para todas las aplicaciones.

Existen diferentes administradores de escritorio como por ejemplo KDE y CDE. Desde el punto de vista de un programador una gran ventaja de usar GNOME es que es gratis.

Existen también una gran cantidad de bibliotecas y herramientas de desarrollo para interfases gráficas con GTK+/GNOME, algunas son las siguientes:

- GLib: es una biblioteca (glib.h) que contiene funciones prototipo y macros que las bibliotecas estandar de C no proveen. Por ejemplo tipos de datos cuya longitud puede ser garantizada no importando la plataforma de cómputo o sistema operativo.
- GDK: GNOME Drawing Kit. Provee una delgada capa sobre las bibliotecas X (Xlib) para simplificar su manejo (las bibliotecas Xlib son sencillas de usar).
- GTK+ (GIMP Toolkit): El resto del curso se centrará en el manejo de esta biblioteca, es la más conocida y no es parte de GNOME, puede administrar aspectos para el desarrollo de aplicaciones gráficas de usuario. Es una biblioteca totalmente orientada a objetos (aunque está escrita en C estándar) y provee un número de objetos para administrar cualquier aspecto de una interfase gráfica.
- GNOME: Conjunto de bibliotecas desarrolladas específicamente para GNOME. LibGnome, LibGnomeUI, LibGnomev2.

## 2. Introducción a la programación orientada a objetos

En términos filosóficos, podemos decir que un objeto es una entidad que puede reconocerse. En términos de programación, podemos decir que un objeto es una estructura de datos y sus funciones asociadas. Un objeto siempre va a tener algún comportamiento determinado. Va a existir con el fin de proporcionar alguna función al sistema. Cada uno de esos comportamientos (funciones) se llama operación. Por ejemplo, existe el objeto lápiz, cuya función es escribir.

Cada uno de los objetos de un sistema conoce su estado actual. Los objetos poseen todo el conocimiento dentro del sistema. Cada pieza de conocimiento se llama atributo. Cada atributo tiene un nombre y un valor. El estado de un objeto se define por los valores de los atributos. Por ejemplo, el objeto lápiz tiene el atributo de tener mucha o poca punta.

### 2.1. Encapsulamiento

El encapsulamiento esconde el modo en que las cosas trabajan y lo que saben, detrás de una interfase: *las operaciones del objeto*. También nos permite ignorar los detalles de bajo nivel sobre el funcionamiento interno de las cosas y nos libera para pensar en un nivel de abstracción más alto.

Los objetos pueden estar compuestos de otros objetos; también pueden ser parte de otros objetos. Esta característica, llamada agregación, nos facilita la reutilización de los objetos creados.

### 2.2. Clases

En la programación orientada a objetos, una clase es una plantilla para objetos. Una definición de clase especifica las operaciones y atributos para todas las instancias de dicha clase. Una clase describe un tipo. Podemos definir un grupo de objetos, los cuales tienen el mismo comportamiento y estructura. Podríamos decir entonces que una clase es una familia de objetos con características y comportamientos similares.

### 2.3. Clases y Objetos

Concluyendo, las clases son definiciones estáticas, que nos permiten entender a todos los objetos de una clase. Los objetos son las entidades (instancias) dinámicas que existen en el mundo real y en nuestra simulación del mismo.

Los objetos existen en tiempo de ejecución, las clases no.

## 2.4. Herencia

La herencia es una relación entre clases que se da a partir de un conjunto de características comunes entre ellas. Las características comunes se definen en la clase padre o superclase; las subclases utilizan la herencia para incluir propiedades.

## 2.5. Polimorfismo

Polimorfismo significa que la misma operación existe en diferentes clases, y que en cada una de ellas se realiza de diferente manera. Cada operación tiene el mismo significado, pero cada clase lleva a cabo dicha operación de una manera muy particular, es decir, tenemos muchos métodos (funciones) para una misma operación.

## 3. Introducción a GTK

GTK (*GIMP Toolkit*) es una biblioteca de funciones creada para desarrollar interfases gráficas. Se distribuye bajo la licencia de software libre (LGPL), de tal manera que se pueden desarrollar cualquier tipo de programas, tanto software libre como software comercial utilizando GTK sin la necesidad de invertir en algún tipo de licencias o regalías.

GTK es una colección gráfica de objetos, como botones, barras de desplazamiento, menús y las correspondientes llamadas a las funciones necesarias para implementarlos. Los objetos gráficos con los que podemos interactuar son conocidos como widgets (Widget Gadget), y la colección de objetos que componen a GTK+ es conocida como GTK+ widget set. Las correspondientes llamadas a las funciones crean un toolkit.

Se llama GIMP toolkit debido a que originalmente fue escrito para desarrollar el programa GIMP (*GNU Image Manipulation Program*), GIMP es un programa muy popular desarrollado por Spencer Kimball y Peter Mattis para ambientes Linux y Unix. Originalmente, GIMP fue desarrollado usando Motif, una herramienta basada en el sistema X Windows. Debido a que no tenía la flexibilidad deseada, los desarrolladores de GIMP crearon el GIMP ToolKit (GTK). GTK+ es una mejora del GIMP ToolKit original.

GTK se ha utilizado para el desarrollo de una gran variedad de proyectos, incluidos GNOME (*GNU Network Object Model Environment*). GTK está construido sobre la parte superior de GDK (*GIMP Drawing Kit*) el cual es básicamente una especie de envoltura (wrapper) de las funciones de bajo nivel para acceder a las funciones del manejo de ventanas (Xlib en el caso del sistema X Windows) y de gdk-pixbuf, una biblioteca de funciones del lado del cliente para la manipulación de imágenes.

Los creadores de GTK son:

- Peter Mattis
- Spences Kimball
- Josh MacDonald

La versión de GTK+ 1.26 contenía más de 60 widgets que se pueden usar para construir una interfase gráfica de usuario (GUI). Varían desde simples botones hasta componentes más complicados, como ventanas con barras de desplazamiento, menús, etc. Actualmente está disponible la versión 2.6 de GTK, la cual implementa algunos nuevos widgets, así como una manera diferente de crear los componentes.

Los Widgets tienen métodos asociados, que realizan acciones sobre ellos. Por ejemplo, si un widget es un botón, algunos métodos son llamados cuando se da clic en el botón. Algunos métodos forman parte del widget, como el



cambio de apariencia cuando el botón es seleccionado. Otros métodos son proporcionados por el programador, los cuales son conocidos como manejadores o *handlers*.

En GTK+, los tipos de widgets son conocidos como clases. Dichas clases definen la funcionalidad y los atributos de un widget. Por ejemplo, en GTK+ existen muchos tipos de botones, definidos por las widgets clases: `GtkButton`, `GtkToggleButton`, `GtkCheckButton`, `GtkRadioButton` y `GtkSpinButton`.

Una clase define los métodos por omisión que están asociados a los widgets que se están usando en una interface. Por ejemplo en un *radio button* o en un *toggle button*, podemos ver la apariencia del mismo al hacer clic en ellos.

Bjarne Stroustrup, inventor del lenguaje C++, dijo alguna vez que la programación orientada a objetos es una forma de escribir código. Podemos decir que un lenguaje orientado a objetos es aquel que de manera explícita nos provee de características en el lenguaje para para escribir código orientado a objetos. Aunque C no es un lenguaje de programación orientado a objetos, nada nos impide escribir una aplicación en el estilo orientado a objetos. GTK es esencialmente un lenguaje de programación que simula una programación orientada a objetos, implementando la idea de clases y llamadas a funciones (apuntadores a funciones).

Junto a GTK, existe un componente adicional que le ayuda a desarrollar sus tareas. GLib, la cual contiene una serie de reemplazos a algunas llamadas estándar, así como también funciones adicionales para manejar componentes adicionales, como listas encadenadas. Dichas funciones ayudan a GTK a incrementar su portabilidad y serán analizadas a detalle más adelante.

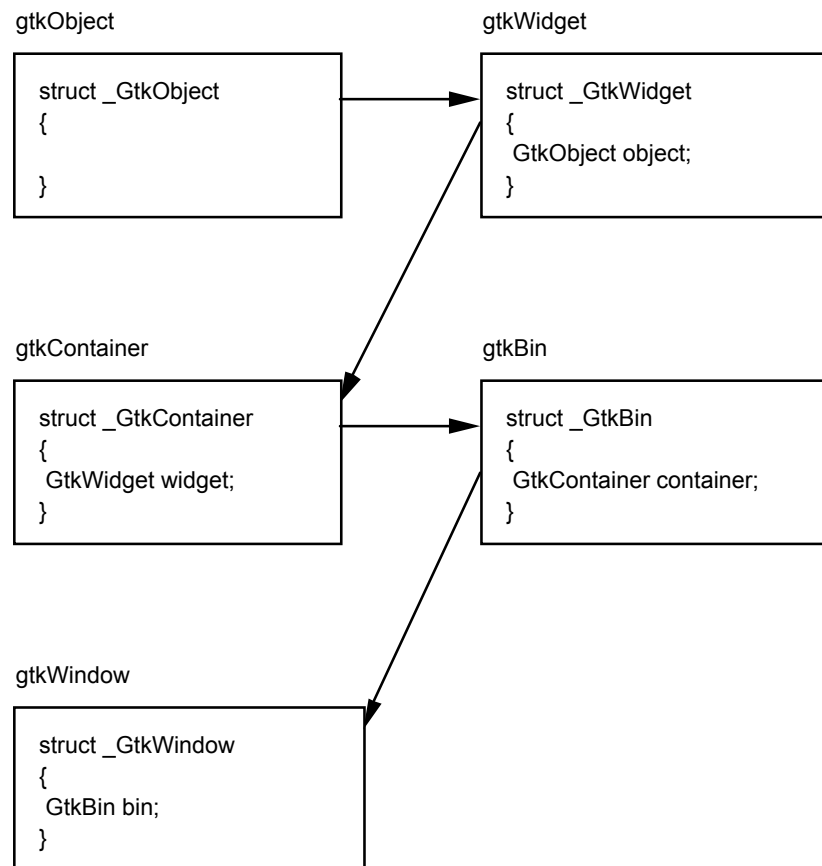
### 3.1. GTK y la programación orientada a objetos

Todos los tipos de datos propios de GTK+ no son más que una estructura. Por ejemplo, el tipo `GtkWidget` es una estructura que se utiliza como base para todos los otros widgets gráficos de una aplicación, incluyendo ventanas, botones, etiquetas, *check boxes*, elementos de menú, etc. La biblioteca de GTK+ provee funciones para trabajar con un tipo de estructura específica. Si una ventana (`GtkWindow`) es "derivada" de un `GtkWidget`, debe existir un macro que nos permita convertir un `GtkWindow` a un `GtkWidget` y viceversa. Esto se conoce como herencia; podemos construir objetos basándonos en otros objetos y en cualquier parte del código se puede tratar a un objeto como uno de su propio tipo o como de un tipo base de donde nuestro objeto fue heredado.

Cada estructura tiene en su estructura el nombre de la estructura que hereda, como primer elemento. Por ejemplo:

```
struct _GtkWindow{
    GtkBin bin;
    gchar *title;
    gchar *wmclass_name;
}
```

Lo que podemos ver aquí es que `GtkWindow` hereda directamente de `GtkBin`. A continuación se muestra un gráfica de las herencias de GTK+:



**Figura 1.- Gráfica de Herencias**

Generalmente es más sencillo crear un objeto del tipo simple, GtkWidget, y después podemos utilizar los macros de "casting" (o conversión de tipos) que se necesitan. Un ejemplo es con la función para mostrar en pantalla un widget, existe para GtkWidget pero no para GtkWindow.

Lo más sencillo es crear un widget básico y hacer un cast hacia widgets más complejos cuando lo necesitemos.

En la herencia es importante notar que existen funciones para algunos objetos, no para todos, esto hace que podamos heredar el comportamiento con las funciones.

GTK+ es un toolkit orientado a objetos. La herencia en un diseño como éste, tiene la idea de un árbol de clases. Las clases widget están definidas como hijos de otros clases widget, heredando las características y funcionalidad de sus padres. Esto hace el trabajo de un programador mucho más fácil. Cuando se crea un nuevo widget, sólo se tiene que desarrollar el código para las nuevas características de un widget, permitiéndolo heredar características de las clases de las que hereda. Un ejemplo del árbol de las clases de GTK+ es el siguiente:

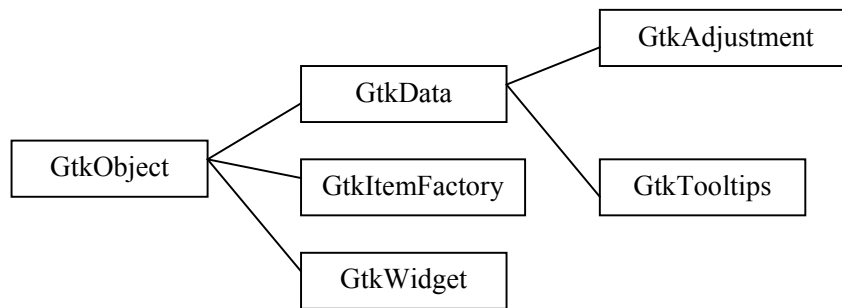


Figura 2.- Árbol de Clases de GTK

La clase `GtkObject` es la clase más alta en el árbol de GTK+. `GtkObject` define atributos y métodos disponibles en todas las clases en GTK+. No tiene el fin de ser un componente que tiene que existir en todas las aplicaciones, más bien es una **clase base**. Los dos hijos de la clase `GtkObject` son, `GtkData` y `GtkWidget` son también clases base. Dichas clases son clases que no representan componentes en una interface de usuario, únicamente son clases que definen funcionalidad, métodos y atributos que van a ser heredados por sus hijos.

`GtkData` es una clase base para los objetos cuyo fin es pasar información a nuestra aplicación, es decir, a nuestra interface de usuario. Por ejemplo, la clase `GtkAdjustment` representa el movimiento de barras de desplazamiento, escalas y otros widgets que permiten al programador poner todos los objetos de la interface de usuario juntos. La clase `GtkAdjustment` provee de todos los métodos que se necesitan para realizar dichas acciones.

Otro objeto de la clase `GtkData` es `GtkTooltips`, el cual nos provee de todos los métodos para desplegar mensajes cortos (tips) cuando el usuario se posiciona sobre algún widget en particular.

La última clase que se muestra en el árbol de clases de GTK+ es la clase `GtkWidget`. Ésta es la clase base para los widgets que hacen la interface de usuario. La clase `GtkWidget` tiene once hijos y es el padre de los más de 60 widgets previamente mencionados.

La siguiente figura muestra las 11 clases hijo de la clase `GtkWidget`:

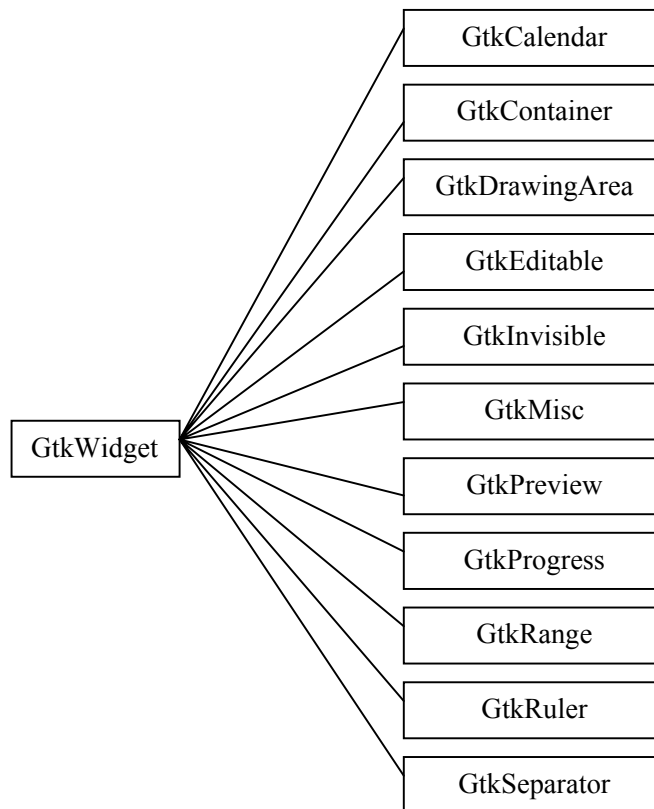


Figura 3.- Clases de GTKWidget

- La clase `GtkCalendar` crea un calendario mensual, y provee la capacidad de que el usuario recorra los diferentes meses.
- La clase `GtkContainer` es la clase base de los widgets que pueden estar a cargo de otros widgets en la interface de usuario. Por ejemplo, uno de los hijos de esta clase es una caja (box) que maneja y cambia el tamaño de todos los widgets que están dentro de ella. La clase `GtkContainer` tiene 12 hijos y un total de 48 clases que descenden de ella.
- La clase `GtkDrawingArea` crea un área específica para comandos de dibujo de GDK. `GtkDrawingArea` tiene sólo una clase hijo, `GtkCurve`.
- La clase `GtkEditable` es la clase base para los widgets que soportan entradas de texto que puede ser modificado por el usuario. Tiene 3 clases descendientes.
- La clase `GtkInvisible` crea un widget invisible creado para las operaciones de *drag-and-drop*, y normalmente no es implementado por nuestra aplicación.
- La clase `GtkMisc` es la clase base para los widgets que permiten rellenar y agregar nuevas características a nuestra interface, como etiquetas o mensajes. Tiene 5 clases descendientes.
- La clase `GtkPreview` es la clase usada para desplegar imágenes en escala de grises o color.

- `GtkProgressBar` es un widget usado para mostrar información del progreso, como un porcentaje de un archivo que ha sido transferido, y no tiene descendientes.
- La clase `GtkRange` es la clase base para los widgets que muestran un rango ajustable, como una escala o una barra de desplazamiento. Posee seis clases descendientes.
- La clase `GtkRuler` es la clase base para los widgets de reglas horizontal y vertical utilizados para mostrar medidas en los bordes de alguna área. Tiene dos clases descendientes.
- La clase `GtkSeparator` es la clase base para los widgets horizontal y vertical utilizados principalmente en los menús para separar grupos de objetos relacionados entre sí. Tiene dos clases descendientes.
- GTK+ posee una característica, llamada temas (*themes*), lo cual es una manera fácil de cambiar la apariencia de los programas desarrollados en GTK+. Básicamente, un tema define los colores y patrones de los widgets durante sus diferentes estados (como normal, activado, desactivado, etc).

### 3.2. Compilación de un programa en GTK

Para compilar un programa en GTK+ se tiene que teclear lo siguiente:

```
gcc -Wall -o archivo.out archivo.c `pkg-config --cflags --libs gtk+-2.0`
```

- `-Wall`: es para que `gcc` produzca en pantalla cualquier advertencia y mensaje de error, no importa que tan triviales sean.
- `-o` es para dar un nombre al programa de salida

La compilación de una aplicación como ésta requiere de más opciones en la línea de compilación, estas opciones nos las provee la utilidad `pkg-config`. Si no hiciéramos uso de la utilidad necesitaríamos incluir las siguientes opciones en el momento de compilar:

```
-I/usr/include/gtk-2.0 -I/usr/lib/gtk-2.0/include -I/usr/include/atk-1.0  
-I/usr/include/pango-1.0 -I/usr/openwin/include -I/usr/sfw/include  
-I/usr/sfw/include/freetype2 -I/usr/include/glib-2.0 -I/usr/lib/glib-2.0/include  
  
y  
  
-lgtk-x11-2.0 -lgdk-x11-2.0 -latk-1.0 -lgdk_pixbuf-2.0 -lm -lmlib -lpangoxft-1.0 -  
lpangox-1.0 -lpango-1.0 -lgobject-2.0 -lgmodule-2.0 -lglib-2.0
```

Las líneas anteriores indican todos los archivos de inclusión, bibliotecas y funciones especiales que necesita un programa desarrollado en GTK para poder compilar. La opción `-I` indica que se está utilizando un directorio para archivos de inclusión. La opción `-L` indica que se quiere incluir una biblioteca.

### 3.3. Creación de un Makefile

La creación de un Makefile para compilar programas desarrollados en GTK no difiere mucho de los Makefiles hasta ahora conocidos. Un ejemplo de un Makefile sería el siguiente:

```
# Ejemplo de un Makefile para gtk  
SOURCES = menu.c  
OBJS     = ${SOURCES:.c=.o}
```

```
CFLAGS = ` pkg-config --cflags gtk+-2.0`
LDADD  = ` pkg-config --libs gtk+-2.0`
CC      = gcc
PACKAGE = menu.out

all : ${OBJS}
      ${CC} -o ${PACKAGE} ${OBJS} ${LDADD}

.c.o:
      ${CC} ${CFLAGS} -c $<

# end of file
```

Un Makefile más sencillo sería el siguiente:

```
menu.out: menu.o
      gcc -o menu.out menu.o `pkg-config --libs gtk+-2.0`

menu.o: menu.c
      gcc -c menu.c `pkg-config --cflags gtk+-2.0`

clean:
      rm *.o
```

#### 4. Revisión de las aplicaciones de GTK+

Cada aplicación de GTK+ crea un widget como parte de la interface gráfica de usuario. En el mundo de la programación orientada a objetos, lo anterior es llamado *instanciar*, o crear la instancia de una clase. Una clase define las capacidades, características y apariencia de un widget. Por ejemplo, la clase `GtkButton` describe a un objeto (un botón) que el usuario selecciona para realizar alguna acción, como salir de alguna aplicación. Para poder hacer que un botón aparezca en nuestra aplicación, debemos crear una instancia de la clase `GtkButton`.

Cada aplicación desarrollada en GTK puede ser representada por un árbol, el cual describe todos los widgets que fueron instanciados y que forman parte de la interface gráfica de usuario. Se establece una relación padre/hijo entre los widgets instanciados para definir el árbol. El widget padre va a controlar el comportamiento de los hijos en cierto grado. El grado de control va a depender del tipo de la clase del widget padre. Algunos widgets padre (como `GtkBox` o `GtkTable`) pueden controlar muchos aspectos del comportamiento de sus hijos, como su tamaño, posición y lo que hacen cuando cambia de tamaño el contenedor. Otros padres tienen un control menor sobre sus hijos, más allá de su visibilidad en la pantalla.

El árbol de instancias de los widgets contiene información jerárquica en relación con los widgets que forman parte de la interface de usuario. Dicho árbol normalmente contiene información del tipo y de los ancestros de los widgets.

Supongamos que creamos un programa con la siguiente interface de usuario, cuyo nombre es `gtktest`:

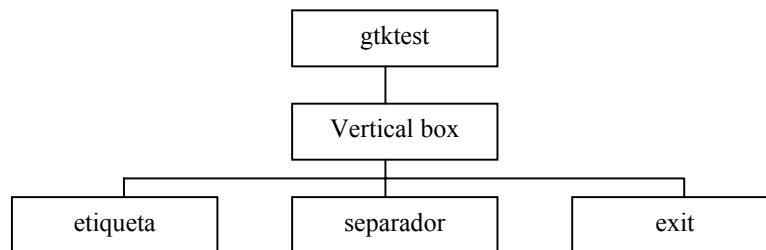


Figura 4.- Árbol de instancias

Como se puede ver, `gtktest` utiliza cinco widgets para crear la interfaz gráfica. Tradicionalmente, el widget que está en la posición más alta del árbol tiene el mismo nombre que el programa ejecutable. El texto “Programa de prueba” es almacenado en un `label` widget. El botón de salida es también un widget. Inclusive la línea entre el texto y el botón es un widget. Los tres widgets anteriores son almacenados dentro de una caja vertical (`vertical box widget`), la cual está contenida dentro del widget `gtktest`.

Un widget no puede ser visto (desplegado a la pantalla) a menos que él y cualquiera de sus widgets ancestros directos sean declarados como vistos. En el caso de nuestro ejemplo, significa que la etiqueta “Programa de prueba” no puede ser vista hasta que ambos, `gtktest` y el `vertical box` sean mostrados. De esta manera, si nosotros fijamos la visibilidad de `gtktest` como `true`, la aplicación completa se mostrará; de manera contraria, si fijamos la visibilidad como `false`, toda la aplicación “desaparecerá”. Esto previene que los widgets hijos estén flotando en la pantalla sin su padre (en una calculadora, los botones no aparecen sin que antes haya aparecido la calculadora en sí).

A continuación entremos más a detalle en el árbol de instancias que compone nuestro programa de prueba.

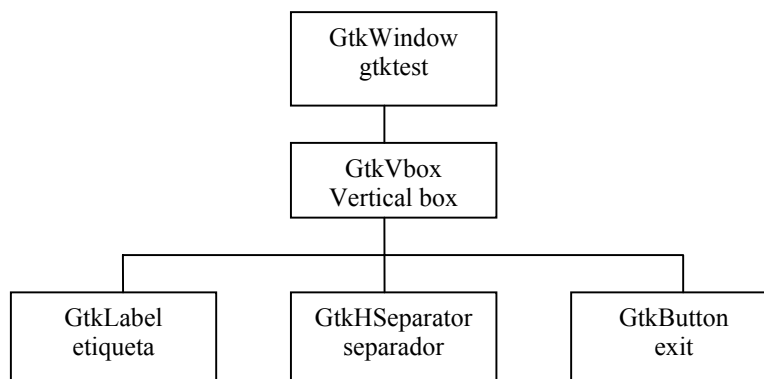
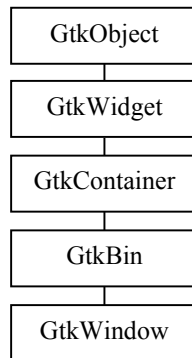


Figura 5.- Árbol de Instancias

El diagrama anterior es el mismo árbol que se vió anteriormente, pero con la adición de las clases widget de cada widget instanciado. El widget `gtktest` es un miembro de la clase `GtkWindow`. Un widget del tipo `GtkWindow` es usualmente el primer widget que se instancia en una aplicación y es normalmente conocido como top-level widget. La siguiente figura muestra la rama de `GtkWindow` del árbol de clases de GTK+:

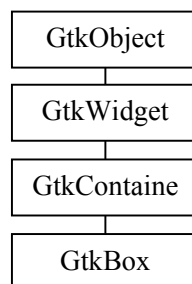


**Figura 6.- Árbol de clases de GtkWindow**

La función primaria de un widget del tipo `GtkWindow` es proveer una comunicación entre la aplicación y el administrador de ventanas. Cuando el usuario cambia el tamaño de la ventana principal con el window manager, dicha información es pasada a `GtkWindow`.

Como se puede ver en la figura anterior, `GtkWindow` es descendiente de la clase `GtkBin`. La clase `GtkBin` restringe el número de hijos de los widgets de este tipo a uno solo. Así, cualquier clase descendiente heredaré esta restricción. Por lo tanto, un widget del tipo `GtkWindow` sólo podrá tener un hijo.

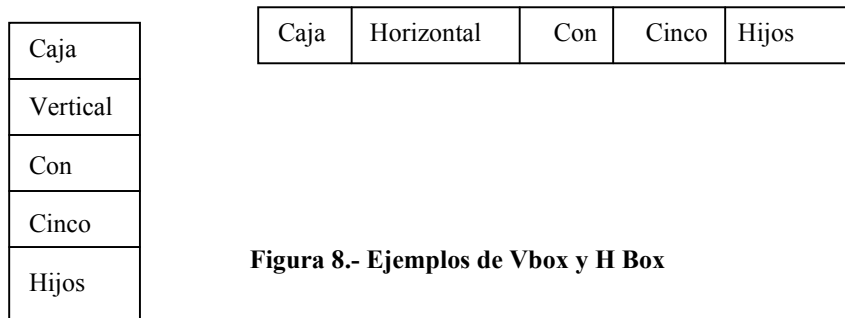
El `vertical box widget` es descendiente de la clase `GtkBox`. La rama del árbol de GTK+ al cual pertenece, se muestra a continuación:



**Figura 7.- Árbol de clases de GtkBox**

La clase `GtkBox` también es una clase base que define tamaño y atributos de espacio para las clases de cajas horizontales y verticales `GtkVBox` y `GtkHBox`. En los programa se necesitará instanciar ya sea un `GtkVBox` o un `GtkHBox`, dependiendo de la dirección en que se quieran extender a los widgets hijos.

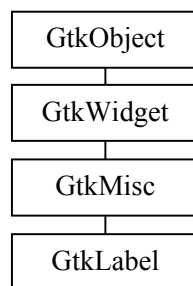




**Figura 8.- Ejemplos de Vbox y H Box**

Las cajas permiten al programador especificar si los hijos van a tener el mismo tamaño, si van a cambiar de tamaño al hacer más grande la ventana, etc.

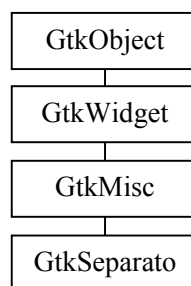
El `label` widget pertenece a la clase `GtkLabel`, a continuación se muestra la rama de la clase `GtkLabel` en el árbol de clases de GTK+:



**Figura 9.- Árbol de clases de GtkLabel**

Una etiqueta es una clase simple. Ninguna acción está asociada a las etiquetas. Se puede especificar solamente si el texto se va a desplegar centrado, alineado a la derecha o a la izquierda, además del tamaño y tipo de la letra.

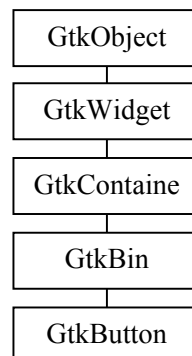
Un `separator` widget es parte de la clase `GtkSeparator`, su árbol es el siguiente:



**Figura 10.- Árbol de clases de GtkSeparator**

La clase `GtkSeparator` es también una clase base de las clases de separador horizontal y vertical `GtkHSeparator` y `GtkVSeparator`.

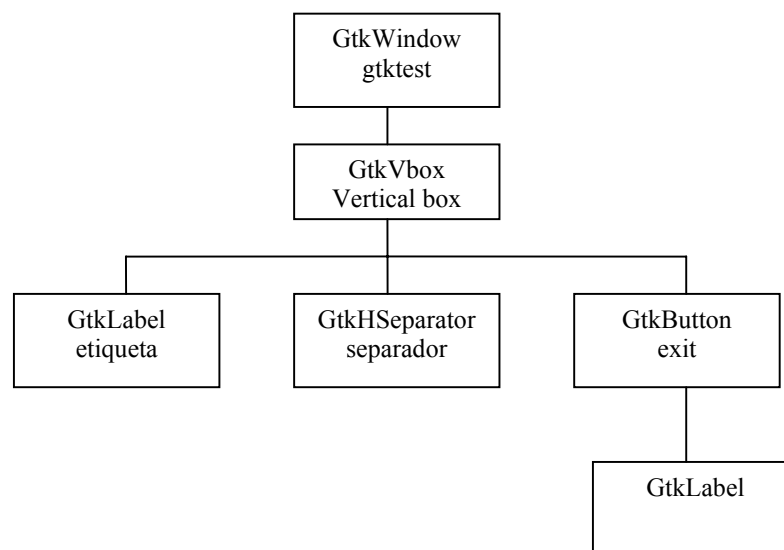
El widget `exit` es parte de la clase `GtkButton`:



**Figura 11.- Árbol de clases de GtkButton**

La clase `GtkButton` provee un widget que es capaz de realizar alguna acción definida por el usuario al hacer click en él.

Cabe aclarar que `GtkButton` es descendiente de `GtkBin`, eso implica que sólo puede manejar un hijo, el cual es una etiqueta (label), es decir, la etiqueta del botón.



**Figura 12.- Árbol de instancias completo**

## 5. Los 7 pasos para la programación básica en GTK+

Una interface gráfica de usuario puede ser muy compleja, además de requerir muchas líneas de código para crearla. La creación de aplicaciones con GTK+ usualmente puede dividirse en siete pasos básicos; aunque si bien es cierto, muchos de estos pasos incluyen muchas llamadas a funciones o pueden llegar a ser complejos, después de entender cada uno de los siete pasos, se podrá llegar a tener un buen entendimiento de las aplicaciones desarrolladas en GTK+.

Los siete pasos a seguir son:

1. Inicializar el ambiente
2. Crear los widgets y fijar sus atributos
3. Registrar las llamadas a las funciones
4. Definir la jerarquía de las instancias del programa
5. Mostrar los widgets
6. Procesar las señales y los eventos
7. Salir de la aplicación

### 5.1. Inicializar el ambiente

Este es el primer paso en todas las aplicaciones de GTK+. La función de GTK+ utilizada para inicializar el ambiente es

```
void gtk_init(int *argc_addr, char **argv_addr)
```

Los parámetros de la función `gtk_init` son los siguientes:

- *argc\_addr* La dirección de *argc*, la variable estándar de C pasada a la función *main* que contiene el número de argumentos pasados desde la línea de comandos.
- *argv\_addr* La dirección de *argv*, la variable estándar de C pasada a la función *main* que contiene los valores de los argumentos pasados desde la línea de comandos.

La función anterior deja listos los valores predeterminados, como los aspectos visuales tal y como los colores y posteriormente manda llamar a la función `gdk_init(gint *argc, gchar ***argv)` la cual inicializa las bibliotecas a utilizar, configura los manejadores predeterminados de las señales y verifica los argumentos que se pasaron al programa al momento de correr, buscando a alguno de los siguientes:

```
--display display_name para especificar otro X server diferente al propio. (--display "computer:0").
--sync, para correr la aplicación en modo síncrono (utilizado principalmente para debugging).
--gtk-module
--g-fatal-warnings
--gtk-debug
--gtk-no-debug
--gdk-debug
--gdk-no-debug
--name
--class
```

`gtk_init` buscará cualquier argumento que se pase, removerá cualquiera de las palabras conocidas (como `--display` o `--sync`), y regresará las variables *argv* y *argc* actualizadas para que sean utilizadas por la aplicación.

Todas las aplicaciones de GTK+ necesitan el archivo de inclusión `<gtk/gtk.h>`, el cual incluye todos los encabezados de los archivos necesarios para instanciar los widgets. El código siguiente es el fragmento de un programa en GTK+, que inicializa el ambiente:

```
/* Comienzo del programa */
#include<gtk/gtk.h>
int main(int argc, char *argv[])
{
    /* 1. Inicialización del ambiente */
    gtk_init(&argc, &argv);
    /* ..... */
}
```

Es muy importante resaltar que `gtk_init` debe ser siempre llamado antes de que la aplicación haga cualquier otra cosa.

## 5.2. Crear los widgets y fijar sus atributos

En GTK+, los widgets son creados (instanciados, siguiendo el término de la programación orientada a objetos) con `gtk_*_new()`, en donde `*` indica el tipo de widget que se quiere crear. Así, vemos que el ejemplo visto anteriormente incluye una etiqueta (`label`), un separador, una caja vertical (`vbox`), un botón y una ventana, por lo tanto, para crear cada uno de los componentes anteriores, tenemos lo siguiente:

```
gtk_label_new()  
gtk_hseparator_new()  
gtk_vbox_new()  
gtk_button_new()  
gtk_window_new()
```

Las funciones anteriores, regresan un apuntador a `GtkWidget`.

Los parámetros pasados a las funciones `gtk_*_new()` son específicos de cada una, a saber:

- `gtk_window_new()` tiene el siguiente prototipo:

```
GtkWidget *gtk_window_new(GtkWindowType type);
```

El parámetro para la función es el siguiente:

<u>Parámetro</u>	<u>Descripción</u>
<i>type</i>	El tipo de la ventana a ser creada. Las opciones válidas son: GTK_WINDOW_TOPLEVEL o GTK_WINDOW_POPUP.

Los dos tipos básicos de `GtkWindow` son `GTK_WINDOW_TOP_LEVEL` y `GTK_WINDOW_POPUP`. La ventana principal de una aplicación será `GTK_WINDOW_TOPLEVEL`. Una ventana de este tipo será manejada directamente por el window manager y no tendrá ningún padre. El tamaño predeterminado de un widget de tipo `GtkWindow` será de 200 x 200 píxeles.

- `gtk_vbox_new()`

Un widget del tipo `GtkVBox` es el siguiente elemento que utilizamos en nuestro ejemplo. Es un contenedor invisible que automáticamente ajusta el tamaño y la posición de sus hijos. El prototipo de la función para crear un widget de este tipo es el siguiente:

```
GtkWidget *gtk_vbox_new(gboolean homogeneous, gint spacing);
```

<u>Parámetro</u>	<u>Descripción</u>
<i>homogeneous</i>	TRUE indica que todos los widgets contenidos en la caja tienen el mismo tamaño, FALSE indica que deberán de conservar su tamaño original (especificado al crearlos).
<i>spacing</i>	La cantidad de espacio (en píxeles) entre los hijos.

- `gtk_label_new()` tiene el siguiente prototipo:

```
GtkWidget *gtk_label_new(const char *str);
```

<u>Parámetro</u>	<u>Descripción</u>
<i>str</i>	El texto a ser desplegado en la etiqueta.

- `gtk_hseparator_new()` tiene el siguiente prototipo:

```
GtkWidget *gtk_hseparator_new(void);
```

- `gtk_button_new()` tiene el siguiente prototipo:

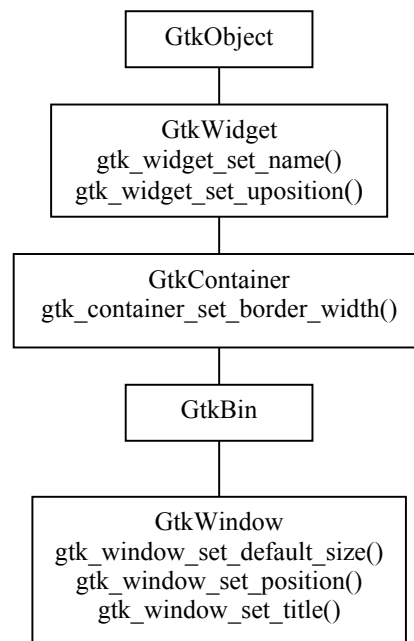
```
GtkWidget *gtk_button_new(void);  
GtkWidget *gtk_button_new_with_label(const gchar *text);
```

<u>Parámetro</u>	<u>Descripción</u>
<i>text</i>	El texto a ser desplegado en el botón.

El siguiente ejemplo muestra la creación de los widgets para el ejemplo visto anteriormente:

```
#include <gtk/gtk.h>  
  
int main( int argc, char *argv[] )  
{  
    GtkWidget *top_widget, *box, *label, *separator, *exit_btn;  
  
    /* 1. Initialize the environment */  
    gtk_init( &argc, &argv );  
  
    /* 2a. Create widgets */  
    top_widget = gtk_window_new( GTK_WINDOW_TOPLEVEL );  
    box = gtk_vbox_new( FALSE, 10 );  
    label = gtk_label_new( "GTK+ is fun!" );  
    separator = gtk_hseparator_new( );  
    exit_btn = gtk_button_new_with_label( "Exit" );  
    /* ..... */  
}
```

Para fijar los atributos de un widget, se deberán utilizar las funciones `gtk_*_set_*`. Debido a la característica de herencia de los widgets, también se pueden utilizar las funciones `gtk_*_set_*` de las clases padres. La siguiente figura muestra algunas de las funciones `gtk_*_set_*` para un widget del tipo `GtkWindow`:



**Figura 13.- Funciones gtk\_\*\_set\_\* para un widget de tipo GtkWindow**

La función `gtk_widget_set_name()` es utilizada para asignar nombres a los widgets instanciados. Su prototipo es el siguiente:

```
void gtk_widget_set_name(GtkWidget *widget, const gchar *name)
```

<u>Parámetro</u>	<u>Descripción</u>
<i>widget</i>	El widget al que se está poniendo el nombre
<i>name</i>	El string con el nombre del widget

La función `gtk_window_set_title()` es utilizada para fijar el título, que generalmente aparecerá en la parte superior de una ventana; su prototipo es el siguiente:

```
void gtk_window_set_title(GtkWindow *window, const gchar *title)
```

<u>Parámetro</u>	<u>Descripción</u>
<i>window</i>	La instancia de GtkWindow a la que se está poniendo el título
<i>title</i>	El string con el título de la ventana.

La función `gtk_container_set_border_width()` es utilizada para fijar el borde de un contenedor; su prototipo es el siguiente:

```
void gtk_container_set_border_width (GtkContainer *container,
                                     guint *border_width)
```

<u>Parámetro</u>	<u>Descripción</u>
<i>container</i>	El GtkContainer (o su descendiente) cuyo borde se está fijando
<i>border_width</i>	La cantidad de borde que se está fijando

Cabe aclarar que aunque los atributos se pueden heredar de acuerdo a la jerarquía del árbol de clases, los tipos de datos de los parámetros deben corresponder (después de todo, estamos programando en C). Los ID's de nuestros widgets son guardados en apuntadores de tipo `GtkWidget`. El primer parámetro de la función `gtk_window_set_title` tiene que ser un dato de tipo `GtkWindow`, y el primer parámetro de la función `gtk_container_set_border_width` tiene que ser de tipo `GtkContainer`, por lo tanto, si queremos aplicar dichas funciones a un widget de tipo `window`, tenemos que hacer un cast. Para ello, GTK+ nos provee de macros cuya tarea es realizar la conversión. Antes de realizar la conversión, dichos macros realizan una revisión para asegurarse de que estamos realizando una conversión válida. (No se puede hacer un cast de un botón a un separador, por ejemplo). Los macros para hacer las conversiones son los siguientes:

```
GTK_CONTAINER(contenedor_o_descendiente)
GTK_OBJECT(objeto_o_descendiente)
GTK_WIDGET(widget_o_descendiente)
GTK_WINDOW(window_o_descendiente)
```

**Nota:** Para que un cast pueda ser válido, deberá realizarse de acuerdo al árbol de clases de los widgets. El tipo viejo deberá ser descendiente del nuevo tipo, no se podrá realizar un cast entre familias diferentes, es decir, cuya descendencia no es en línea recta.

Por default, un `GtkWindow` puede crecer, pero no puede ser más chico que sus widgets hijos. Esta política puede ser fijada con la siguiente función:

```
void gtk_window_set_resizable (GtkWindow *window, gboolean resizable)
```

<u>Parámetro</u>	<u>Descripción</u>
<code>window</code>	La instancia de <code>GtkWindow</code> a la que se está fijando la política
<code>resizable</code>	TRUE si el usuario puede cambiar de tamaño la ventana.

A continuación el fragmento de código en donde se fijan los atributos de los componentes de nuestro ejemplo:

```
/* 2b. Set attributes */
gtk_window_set_title( GTK_WINDOW( top_widget ), "gtkfun" );
gtk_container_set_border_width( GTK_CONTAINER( top_widget ), 15 );
gtk_widget_set_name( top_widget, "gtkfun");
gtk_widget_set_name( box, "vertical box");
gtk_widget_set_name( label, "fun_label");
gtk_widget_set_name( separator, "separator");
gtk_widget_set_name( exit_btn, "exit");
/* ..... */
}
```

### 5.3. Registrar las llamadas a las funciones

Los widgets en GTK+ responden a eventos, como cuando un usuario presiona un botón para salir de una aplicación. Cuando ocurre un evento, el widget correspondiente emitirá una señal. El programador podrá enlazar dicha señal con una o varias funciones. Cada widget tiene asociado a él un conjunto de señales y de prototipos de funciones que pueden ser utilizados por el programador. Por ejemplo, un `GtkButton` provee señales cuando el usuario hace click en él, presiona, suelta, entra y sale del área del botón. Es deber del programador decidir qué señales son importantes para nuestro programa. Se debe, pues, escribir y registrar todas las funciones que se quieran que se ejecuten cuando una señal es detectada.

Para determinar todas las señales que están disponibles para ser usadas, es necesario revisar la información de las clases de los widgets. Los widgets heredan las señales de sus ancestros.

Algunas señales son específicas de cada widget (como `clicked` del `GtkButton`). Otras son más genéricas y son heredadas por otras clases.

Una vez que se ha escrito la función específica, se deberá registrar, de tal manera que se ejecute cuando la señal específica sea reconocida. En la versión 2.0 de GTK, todo el sistema encargado de manejar las señales se ha movido de GTK a GLib, de ahí que las funciones y tipos explicados en esta sección tengan un prefijo "g\_" en lugar de un prefijo "gtk\_".

Para poder hacer que un widget pueda ejecutar una acción, se debe primero configurar un *manejador de señales*, que será el encargado de "cuchar" las señales emitidas por los widgets y de mandar llamar la función que se encargará de realizar la acción deseada. Lo anterior es realizado con la ayuda de la función:

```
gulong g_signal_connect (gpointer *object, const gchar *name,  
                        GCallback func, gpointer func_data)
```

<u>Parámetro</u>	<u>Descripción</u>
<i>object</i>	El <code>GtkObject</code> (widget o su descendiente) que está emitiendo la señal que se quiere cchar
<i>name</i>	El nombre de la señal que se quiere detectar
<i>func</i>	La función que queremos que se ejecute al detectarse la señal.
<i>func_data</i>	La información que es pasada a <i>func</i> cuando es llamada.

La función cuyo nombre es especificado como tercer argumento de la función anterior depende mucho de la señal que se quiere detectar, comúnmente es llamada "*callback function*" y deberá tener generalmente el siguiente prototipo:

```
void callback_func (GtkWidget *widget, gpointer callback_data)
```

<u>Parámetro</u>	<u>Descripción</u>
<i>widget</i>	El <code>GtkButton</code> que fue presionado.
<i>callback_data</i>	La información que es pasada a <i>callback_func</i> cuando se llama (especificada cuando se hace el registro de la función).

Otra función que nos puede servir para registrar llamadas a funciones es la siguiente:

```
gulong g_signal_connect_swapped( gpointer *object, const gchar *name,  
                                GCallback func, gpointer *slot_object );
```

`g_signal_connect_swapped()` es parecida a la función `g_signal_connect()` excepto que la función que se llama en la primera solamente acepta como argumento un sólo parámetro, un apuntador a `GtkObject`. Así que el prototipo de la función que atenderá la señal que se conecta utilizando esta función deberá tener el siguiente prototipo:

```
void callback_func( GtkObject *object );
```

En donde el parámetro recibido será generalmente un widget. Generalmente se utiliza esta llamada para hacer uso de funciones propias de Gtk (como `gtk_widget_show` y `gtk_widget_hide` explicadas más adelante) que reciben un sólo parámetro (generalmente, un widget).

A continuación, un fragmento de código en donde se desarrolla una función que se quiere ejecutar cuando se presiona un botón:

```
void print_and_quit( GtkButton *was_clicked, gpointer user_data )  
{  
    /* Use glibs printf equivalent to print a message */
```



```
    g_print( "Thank you for using this program.\n" );
    gtk_main_quit();
}
```

Aquí está la declaración de la función `g_signal_connect` para agregar la llamada a la señal `clicked`:

```
/* 3. Register callbacks */
g_signal_connect( G_OBJECT( exit_btn ), "clicked",
                  G_CALLBACK( print_and_quit ), NULL );
```

El ejemplo anterior, no utiliza ninguna información para pasarla a la función. Aquí un ejemplo en donde sí se manda información a la función cuando es llamada:

```
void print_and_quit( GtkWidget *was_clicked, gpointer user_data )
{
    /* Use glib's printf equivalent to print a message */
    g_print( (gchar *)user_data );
    gtk_main_quit();
}

/* 3. Register callbacks */
g_signal_connect( G_OBJECT( exit_btn ), "clicked",
                  G_CALLBACK( print_and_quit ), "Thank you and good bye!" );
```

Revisando un poco más a detalle el prototipo de la función `g_signal_connect`, nótese que tiene un valor de retorno de tipo `guint`. Dicho valor es un identificador de cada una de las conexiones que se realizan a las diferentes señales que se quieren detectar y puede utilizarse, por ejemplo, para remover completamente o temporalmente la llamada a las funciones al detectarse la señal determinada por medio de las siguientes funciones:

```
void g_signal_handler_disconnect( gpointer object, gulong id );

void g_signal_handler_block( gpointer object, gulong id );

void g_signal_handlers_block_by_func( gpointer object, GCallback func, gpointer data );

void g_signal_handler_unblock( gpointer object, gulong id );

void g_signal_handlers_unblock_by_func( gpointer object, GCallback func, gpointer data );
```

## 5.4. Definir la jerarquía de las instancias del programa

Hasta ahora hemos creado desde un top-level widget hasta un botón que imprime un mensaje cuando es presionado. Sin embargo, no hemos establecido ninguna relación padre/hijo entre dichos widgets. En GTK+, dicha relación es establecida después de haber creado los widgets y antes de hacerlos aparecer en pantalla. Aquí no hay ningún valor predeterminado. Debemos establecer la relación jerárquica, o GTK+ no desplegará los widgets.

Para establecer la relación padre/hijo, se deberá agregar el hijo al control del widget padre. La función a ser llamada para dicho efecto, dependerá del tipo de padre. En el caso de nuestro ejemplo, el top-level widget es de la clase `GtkWindow`, que es un descendiente de un contenedor (*container*), por lo tanto, utilizaremos la función `gtk_container_add()`.

```
void gtk_container_add (GtkContainer *container, GtkWidget *widget)
```

<u>Parámetro</u>	<u>Descripción</u>
<code>container</code>	La instancia padre de un <code>GtkContainer</code> .

*widget* El widget que se quiere agregar al contenedor.

Como se mencionó anteriormente, un widget de la clase `GtkWindow` sólo puede tener un hijo, por lo tanto, `gtk_container_add()`, será utilizado para agregar la caja vertical al widget `GtkWindow`. El siguiente paso será agregar la etiqueta, el separador vertical y el botón a la caja vertical, la cual sí puede tener más de un hijo. Esto se hace por medio de la función `gtk_box_pack_start_defaults()`:

```
void gtk_box_pack_start_defaults (GtkBox *box, GtkWidget *widget)
```

<u>Parámetro</u>	<u>Descripción</u>
<i>box</i>	La instancia padre de un <code>GtkBox</code> .
<i>widget</i>	El widget que se quiere agregar al la caja.

La función anterior empieza a agregar los componentes a la caja desde el inicio de ésta (desde arriba en el caso de una caja vertical).

A continuación se enuncia el código correspondiente para establecer la jerarquía de los componentes de nuestro ejemplo:

```
/* 4. Define instance hierarchy (pack the widgets) */
gtk_container_add( GTK_CONTAINER( top_widget ), box );
gtk_box_pack_start_defaults( GTK_BOX( box ), label );
gtk_box_pack_start_defaults( GTK_BOX( box ), separator );
gtk_box_pack_start_defaults( GTK_BOX( box ), exit_btn );
```

## 5.5. Mostrar los widgets

Una vez que se ha determinado la jerarquía de los componentes, éstos pueden ser mostrados en pantalla. Lo anterior puede hacerse mostrando uno a uno los componentes de nuestra aplicación con la función `gtk_widget_show()`, o mostrando todos a la vez con la función `gtk_widget_show_all()`

```
void gtk_widget_show (GtkWidget *widget)
```

<u>Parámetro</u>	<u>Descripción</u>
<i>widget</i>	El widget que se quiere agregar al la caja.

```
void gtk_widget_show_all (GtkWidget *pwidget)
```

<u>Parámetro</u>	<u>Descripción</u>
<i>pwidget</i>	El widget padre de los widgets que se quieren mostrar. Esto causa que <i>pwidget</i> y todos sus descendientes en el árbol de jerarquías de nuestra aplicación se muestren.

El siguiente código es el código necesario para mostrar los widgets de nuestro ejemplo:

```
/* 5. Show the widgets */
gtk_widget_show_all( top_widget );
```

también se podrían mostrar uno por uno:

```
/* 5. Show the widgets */
gtk_widget_show( exit_btn );
gtk_widget_show( separator );
gtk_widget_show( label );
gtk_widget_show( box );
gtk_widget_show( top_widget );
```

## 5.6. Procesar las señales y los eventos

Una vez que hemos escrito y registrado todas las funciones que queremos asociar con los widgets de nuestra aplicación, es tiempo de correrla y de pasar el control del mismo a GTK+ y al usuario de la aplicación. El resto de nuestro programa es simplemente un ciclo, el cual acepta señales y llama al método apropiado para manejarla. Cuando la función termina, el control del programa es regresado al ciclo, el cual continúa aceptando y procesando señales.

Todo lo anterior, es controlado por la función de GTK+ `gtk_main()`, la cual no recibe ni regresa nada:

```
void gtk_main()
```

La aplicación se mantendrá en un ciclo hasta que se reciba alguna instrucción de salida, o se dé algún “*fatal error*”.

El siguiente fragmento de código muestra el ciclo principal de las aplicaciones en GTK+:

```
/* 6. Processing Loop */
gtk_main();
g_print( "Bye!\n" );
```

## 5.7. Salir de la aplicación

La función de GTK+ `gtk_main_quit()` es llamada para romper el ciclo principal después de que se ha hecho la llamada a alguna función.

```
void gtk_main_quit()
```

Una vez que la llamada a la función se terminó, el control del programa pasa a la siguiente línea de código en el programa principal después de `gtk_main()`.

La función `gtk_exit()` es utilizada para terminar inmediatamente la ejecución del programa y salir de él, dando un código de error. Es el equivalente a la función `exit()` de C.

A continuación se enlista el código completo del programa de ejemplo, el cual enlista los siete pasos necesarios para la creación de un programa en GTK+:

```
#include <gtk/gtk.h>

/* Callbacks */
void print_and_quit( GtkWidget *was_clicked, gpointer user_data )
{
    /* Use glibs printf equivalent to print a message */
    g_print( "Thank you for using this program.\n" );
    gtk_main_quit();
}

int main( int argc, char *argv[] )
{
    GtkWidget *top_widget, *box, *label, *separator, *exit_btn;

    /* 1. Initialize the environment */
    gtk_init( &argc, &argv );

    /* 2a. Create widgets */
    top_widget = gtk_window_new( GTK_WINDOW_TOPLEVEL );
```

```
box = gtk_vbox_new( FALSE, 10 );
label = gtk_label_new( "GTK+ is fun!");
separator = gtk_hseparator_new( );
exit_btn = gtk_button_new_with_label( "Exit" );
/* 2b. Set attributes */
gtk_window_set_title( GTK_WINDOW( top_widget ), "gtkfun" );
gtk_container_set_border_width( GTK_CONTAINER( top_widget ), 15 );
gtk_widget_set_name( top_widget, "gtkfun");
gtk_widget_set_name( box, "vertical box");
gtk_widget_set_name( label, "fun_label");
gtk_widget_set_name( separator, "separator");
gtk_widget_set_name( exit_btn, "exit");

/* 3. Register callbacks */
g_signal_connect( G_OBJECT( exit_btn ), "clicked",
                  G_CALLBACK( print_and_quit ), NULL );

/* 4. Define instance hierarchy (pack the widgets) */
gtk_container_add( GTK_CONTAINER( top_widget ), box );
gtk_box_pack_start_defaults( GTK_BOX( box ), label );
gtk_box_pack_start_defaults( GTK_BOX( box ), separator );
gtk_box_pack_start_defaults( GTK_BOX( box ), exit_btn );

/* 5. Show the widgets */
gtk_widget_show_all( top_widget );

/* 6. Processing Loop */
gtk_main();
g_print( "Bye!\n");
return 0;
}
```

## 6. Más sobre el control de un programa

Supongamos que nosotros corremos el ejemplo anterior, el programa se sale cuando el usuario presiona el botón de “Exit”. ¿Qué pasa si el usuario quiere cerrar la aplicación por medio del botón “cerrar” de la ventana principal de la aplicación? Resulta que cuando se quiere cerrar un programa por medio del botón cerrar de la ventana, una señal `delete-event` es generada por la ventana. Si el programador quiere cerrar la aplicación por este medio, deberá tomar en cuenta la señal `destroy`, que es generada por el `top-level window` de la aplicación.

Primero que nada, debemos de proveer al programa con una función que pueda manejar la señal `delete_event`, de tal manera que podamos saber cuando el usuario quiere cerrar la ventana. El prototipo de la función es el siguiente:

```
gboolean user_function (GtkWidget *widget, GdkEvent *event,
                        gpointer user_data)
```

El parámetro `widget` es dado por GTK+ cuando la función es llamada y corresponde al ID del widget que causó el evento `delete_event`. El parámetro `user_data` es dado por el programador, cuando la función es registrada. El parámetro `event` es la dirección de la estructura que contiene todo lo que queremos saber acerca del evento y será tratada a detalle más adelante. El valor de retorno de nuestra función indica si nosotros estamos previniendo a la aplicación de ser cerrada. Un valor `TRUE` indica que la aplicación será prevenida de ser cerrada, un valor `FALSE` indica lo contrario (y una señal `destroy` será emitida). A continuación se enuncia un ejemplo de la llamada a la función `delete_event`:

```
gboolean delete_event_handler (GtkWidget *widget, GdkEvent *event,
                              gpointer user_data)
{
```

```
/* Received closure from the window manager */
g_print("The window manager is asking to close this application\n");
return(FALSE); /* FALSE - do no prevent closure */
}
```

La función para manejar la señal `delete_event` es agregada al top-level window debido a que es la que se comunica con el window manager. El siguiente es un ejemplo de un `g_signal_connect()` para un manejador de la señal `delete_event`:

```
g_signal_connect (G_OBJECT( top_widget ), "delete_event",
                  G_CALLBACK(delete_event_handler), NULL);
```

Con lo anterior, la aplicación todavía no se va a cerrar. Es necesario agregar un manejador para la señal `destroy`. El prototipo es el siguiente:

```
void user_function (GtkWidget *widget, gpointer user_data)
```

<i>Parámetro</i>	<i>Descripción</i>
<i>widget</i>	El <code>GtkObject</code> que recibe la señal <code>destroy</code> (pasado por GTK+).
<i>user_data</i>	La información pasada por el usuario cuando la función es llamada.

El programa completo sería el siguiente:

```
#include <gtk/gtk.h>

/* Callbacks */
void print_and_quit( GtkWidget *was_clicked, gpointer user_data )
{
    /* Use glibs printf equivalent to print a message */
    g_print( "Thank you for using this program.\n" );
    gtk_main_quit();
}

gboolean delete_event_handler (GtkWidget *widget, GdkEvent *event,
                              gpointer user_data)
{
    /* Received closure from the window manager */
    g_print("The window manager is asking to close this application\n");
    return(FALSE); /* FALSE - do no prevent closure */
}

int main( int argc, char *argv[] )
{
    GtkWidget *top_widget, *box, *label, *separator, *exit_btn;

    /* 1. Initialize the environment */
    gtk_init( &argc, &argv );

    /* 2a. Create widgets */
    top_widget = gtk_window_new( GTK_WINDOW_TOPLEVEL );
    box = gtk_vbox_new( FALSE, 10 );
    label = gtk_label_new( "GTK+ is fun!" );
    separator = gtk_hseparator_new( );
    exit_btn = gtk_button_new_with_label( "Exit" );

    /* 2b. Set attributes */
    gtk_window_set_title( GTK_WINDOW( top_widget ), "gtkfun" );
    gtk_container_set_border_width( GTK_CONTAINER( top_widget ), 15 );
    gtk_widget_set_name( top_widget, "gtkfun" );
```

```
gtk_widget_set_name( box, "vertical box");
gtk_widget_set_name( label, "fun_label");
gtk_widget_set_name( separator, "separator");
gtk_widget_set_name( exit_btn, "exit");

/* 3. Register callbacks */
g_signal_connect( G_OBJECT( exit_btn ), "clicked",
                  G_CALLBACK( print_and_quit ), NULL );
g_signal_connect( G_OBJECT( top_widget ), "delete_event",
                  G_CALLBACK( delete_event_handler), NULL);
g_signal_connect( G_OBJECT( top_widget ), "destroy",
                  G_CALLBACK( print_and_quit), NULL);

/* 4. Define instance hierarchy (pack the widgets) */
gtk_container_add( GTK_CONTAINER( top_widget ), box );
gtk_box_pack_start_defaults( GTK_BOX( box ), label );
gtk_box_pack_start_defaults( GTK_BOX( box ), separator );
gtk_box_pack_start_defaults( GTK_BOX( box ), exit_btn );

/* 5. Show the widgets */
gtk_widget_show_all( top_widget );

/* 6. Processing Loop */
gtk_main();
g_print( "Bye!\n");
return 0;
}
```

## 7. Glib

Glib es una biblioteca de bajo nivel que provee de muchas definiciones y funciones útiles que están disponibles cuando se está desarrollando un programa en GDK y GTK+. Lo anterior incluye un conjunto de funciones y macros para: administración de memoria, administración de listas y depuración de código. También proveen un reemplazo de tipos que nos da independencia de plataforma o tipos más convenientes (gboolean, gpointer).

### 7.1. Tipos de Datos de Glib

Estos tipos nos permiten portar el código de manera más sencilla

gboolean	True/False, on/off TRUE y FALSE están en Glib
gpointer	Apuntador genérico (void *) Es más fácil de leer
gchar y gchar*	Caracter y caracter sin signo
gint, guint, gshort, glong, gulong	
gint8, guint8, gint16, guint16, gint32, guint32 gint64, guint64	Son de tamaño conocido  Son de 64 bits, solo se soporta en Algunas máquinas. Se debe usar el macro G_HAVE_GINT64 para saber si nuestro equipo lo soporta
gfloat, gdouble	reemplazo float y double
gsize	Unsigned data type (lo veremos después), manejo de tamaño de estructuras
gssize	Signed data type manejo de tamaño de estructuras

Figura 14.- Tipos de datos en Glib

## 7.2. Macros Estándar

Constantes útiles TRUE, FALSE, NULL

G\_DIR\_SEPARATOR , caracter de separación, depende del SO

G\_DIR\_SEPARATOR\_S, caracter de deseparación incluido el caracter 0

## 7.3. Macros Matemáticas

Min(x,y), Max(x,y), ABS

Nota: Debemos procurar usarlos con cuidado, ya que pueden insertar efectos laterales en nuestro código

## 7.4. Detección de Errores

Assertion Functions (suposiciones)

Si tengo una función que toma parámetros y escribo código asumiendo que el valor debe ser siempre mayor a 0, entonces puedo asumir o suponer esto en el código. Se hace a través de la función `g_assert`

```
void funcion(gint value)
{
    g_assert(value > 0)
    :
    :
}
```

Si lo que asumimos es incorrecto, la función detendrá el programa y enviará un mensaje de error. Indicando el lugar en el código donde sucedió eso y qué condición falló.

`g_assert_not_reached`

La usamos para definir algo que suponemos no será alcanzado nunca en el código(no vamos a pasar por ahí)

```
switch (value){
    case 1:
        break;
    case 2:
        break;
    default :
        g_assert_not_reached(); // Esto no deberia suceder
        break;
}
```

Sucede lo mismo que con `g_assert()`

Estas funciones son importantes para revisar el funcionamiento de nuestro código, debemos recordar que detienen el programa.

Hay casos en los que no es necesario detener la ejecución, se puede corregir y continuar.

`g_return_if_fail`, `g_return_val_if_fail`

Se les provee una condición y en caso de que la condición (suposición) no se cumpla, se despliega un error. Después de esto se continúa la ejecución del programa.

En caso de que ocurra en una función se termina la ejecución de ésta.

`g_return_val_if_fail(nvalue > 0 , -1)`

Si `nvalue` es menor o igual a 0 se regresa un `-1` en la función donde se utilizó `g_return_val`. De esta manera el código que llama a la función puede lidiar con el error.

## 7.5. Otras funciones

`g_error()`, `g_warning("Warning")`

Estas regresan un mensaje al usuario, terminan la ejecución del código.

Existen funciones para trabajar con segmentos de memoria, de momento no las vamos a ver.

## 7.6. Listas (una my sencilla introducción)

Existen muchas aplicaciones donde se pueden utilizar diferentes tipos de listas. Este tema es muy amplio pero se cubrirá de manera superficial

Qué es una lista? es una colección ordenada de datos.

Se tienen funciones para listas encadenadas sencillas (`GSList`), listas doblemente encadenadas (`GList`) y árboles.

### 7.6.1. Creación y Destrucción de Listas

En GLib una lista se inicia pasando un apuntador de tipo `GSList` o `GList`, por ejemplo, inicializado en `NULL`. Posteriormente las funciones de GLib se encargan de adicionar nodos a la lista, asignando memoria.

Debemos tener cuidado de liberar memoria una vez que no necesitemos la lista, esto se hace a través de las funciones `g_list_free` y `g_slist_free`.

### 7.6.2. Inserción de Elementos en una lista

Anexar, preinsertar, insertar en la parte media, insertar en orden.

### 7.6.3. Preinsertar y anexar

`g_list_append`, `g_slist_append`  
`g_list_prepend`, `g_slist_prepend`

Se les pasa un apuntador al primer elemento de la lista, así como un apuntador a los datos que queremos adicionar. Se regresa un apuntador a la cabeza de la lista.

```
GList *NameList = NULL;
NameList = g_list_append(NameList, "Pete");
NameList = g_list_prepend(NameList, "Richard");
NameList = g_list_append(NameList, "Dave");
```

Richard, Pete, Dave

La inserción de elementos en la lista requiere de un argumento adicional; un entero que indique donde en la lista se va a insertar el elemento. Si el número especificado es menor a 0 o más grande que el número de elementos de la lista, `GList` insertará el elemnto al final de la lista.

```
NameList = g_list_insert(NameList, "Wright", 1);
```

Nota: La numeración comienza en 0.



#### 7.6.4. Movimiento en la lista

En una lista doblemente encadenada se pueden utilizar las funciones `g_list_next` y `g_list_last`. Así como `g_list_first` y `g_list_previous`.

Para una lista sencilla existen las funciones: `g_slist_next` y `g_slist_last`.

Podemos movernos a un elemento en particular con las funciones `g_list_nth` y `g_slist_nth`. Se pasa un apuntador al inicio de la lista y un número que indique en que posición queremos situarnos.

#### 7.7. Manejo de Memoria

GLib nos provee de las siguientes funciones, que son un reemplazo de las funciones conocidas, para el manejo eficiente de la memoria:

```
gpointer g_malloc(gulong size);
```

La función anterior es un reemplazo de la función `malloc()`, con la ventaja que no tenemos que preocuparnos por el valor de retorno, lo cual es realizado dentro de la misma función. Si por alguna razón falla la ejecución de la función, el programa terminará.

```
void g_free( gpointer mem );
```

La función anterior libera la memoria.

#### 7.8. Funciones útiles

`void g_print(gchar *format...)` Función que reemplaza a la función `printf()`.

`void g_error(gchar *format...)` Función que imprime un mensaje de error, agregándole a dicho mensaje el texto "\*\*\*ERROR\*\*:" y se sale del programa. Se debe usar para errores graves.

`void g_warning(gchar *format...)` Función que imprime un mensaje de advertencia, agregándole a dicho mensaje el texto "\*\*\*WARNING\*\*:" pero sin salirse del programa.

`void g_message(gchar *format...)` Función que imprime un mensaje en la pantalla, agregándole a dicho mensaje el texto "message:" pero sin salirse del programa.

### 8. Tipos de Datos Extendidos

Además de los tipos de datos que son introducidos en GLib para ayudar con la portabilidad, también provee un conjunto de funciones para la manipulación de cadenas, el tipo `GString`.

En realidad `GString` es una estructura que consiste de un apuntador a una cadena de tipo `gchar`, se llama `str`, y un entero `len` de tipo `gint`, que contiene la longitud de la cadena. El verdadero poder lo podemos ver en el conjunto de funciones diseñada para trabajar con esta estructura.

GLib asigna dinámicamente la memoria para mantener las cadenas que creamos y nos permite extender o recortar texto, permite concatenar cadenas, insertar uno después de otro y demás. Todo el tiempo GLib se asegura del uso adecuado de memoria, por lo que `GString` es muy eficiente y más simple de usar.

### 9. Creación y Destrucción de Cadenas

La forma más sencilla de crear un `GString`, es con la función `g_string_new`. Toma como argumento un apuntador a una cadena y regresa un apuntador a la cadena creada.

```
GString *MyName;  
MyName = g_string_new("Peter Wright");
```

Podemos asignar un tamaño para la cadena de caracteres con la función `g_string_sized_new`

```
MyName = g_string_sized_new(10);
```

Una vez asignado el tamaño podemos asignar texto con la función `g_string_assign`. Esta función necesita dos argumentos, el primero es un apuntador a la cadena donde queremos que la cadena sea asignada, el segundo es un apuntador a la cadena de caracteres que será asignada. La función regresa un apuntador a la cadena recién asignada. Esto es importante pues en la asignación de texto puede haber una reasignación de memoria y esto hará que la dirección de la cadena cambie.

```
MyName = g_string_assign(MyName, "Pete 'Show Me Da Money' Wright");
```

Este ejemplo demuestra lo anterior. `MyName` tenía asignado 10 espacios y la nueva cadena ocupa más.

Para destruir una cadena se utiliza la función `g_string_free`. Usa dos argumentos, el primer argumento es el apuntador a la cadena que queremos eliminar y el segundo es un tipo `gboolean`, de momento lo pondremos en `TRUE`.

## 9.1. Anexar y Preinsertar Cadenas

Esto se realiza con dos funciones `g_string_append` y `g_string_prepend`. Trabajan de la misma manera. Las dos funciones esperan un argumento a `GString` y un apuntador a un arreglo de caracteres terminado en `NULL`. GLib reasigna memoria y construye el nuevo tipo `GString`. Finalmente se regresa un apuntador al nuevo `GString`.

```
GString *MyName;  
MyName = g_string_new("Pete ");  
MyName = g_string_append(MyName, "Wright");  
MyName = g_string_prepend(MyName, "Mr. ");
```

Ahora la cadena queda como "Mr. Pete Wright".

Podemos tener acceso a la cadena por medio del elemento de la estructura `GString`:

```
g_print(MyName->str);
```

## 9.2. Inserción de una cadena dentro de otra

Podemos insertar texto en cualquier parte de una cadena `GString`, a través de la función `g_string_insert`. Se debe pasar el apuntador a la cadena donde se insertará la cadena, la posición donde queremos que se realice la inserción y por último el texto a insertar.

```
GString = *MyName;  
MyName = g_string_new("Beginning GNOME");  
MyName = g_string_insert(MyName, 10, "GTK+");
```

Existen más funciones para la manipulación de el tipo `GString` pero de momento no las veremos.

## 10. Timers (Cronómetros)

Un tipo de timer, que es el que se cubre a continuación se puede utilizar como un cronómetro. Este cronómetro se crea y se revisa periódicamente para comprobar el tiempo que ha pasado. Una vez que no se necesita el cronómetro, se detiene y nos liberamos de él.

Para manipular los cronómetros se necesitan seis funciones:

- Crear un cronómetro `g_timer_new`. Regresa un apuntador a un objeto `GTimer`
- Inicializar el cronómetro se usa la función `g_timer_start`
- Encontrar cuanto tiempo ha pasado, `g_timer_elapsed`
- Detener el cronómetro `g_timer_stop`
- Reiniciar y empezar `g_timer_reset`
- Si queremos deshacernos del cronómetro `g_timer_destroy`

### 10.1. Creación y Destrucción de un Timer

```
GTimer *MyTimer;  
MyTimer = g_timer_new();  
  
// Código  
  
g_timer_destroy(MyTimer);
```

### 10.2. Iniciar, Detener y Reiniciar el Cronómetro

```
GTimer *MyTimer;  
MyTimer = g_timer_new();  
  
while (bStillNeeToTimeStuff)  
{  
    g_timer_start(MyTimer);  
    // Código  
    g_timer_stop(MyTimer);  
    g_timer_reset(MyTimer);  
}  
g_timer_destroy(MyTimer);
```

### 10.3. Revisión del Cronómetro

Para revisar el valor del cronómetro se utiliza la función `g_timer_elapsed`. Esta función nos permite medir el tiempo en milésimas de segundo, si así lo requiriéramos.

La función regresa un `gdouble`, que es el tiempo transcurrido en segundos. Se necesitan dos argumentos, el primero es el apuntador a `GTimer` que estamos utilizando y el segundo es un apuntador a `gulong` (GLib unsigned long integer). Si pasamos un apuntador válido, en lugar de `NULL`, el valor que regrese la función será el equivalente en milisegundos.

Ver el código `spooky.c`

## 11. Señales y Eventos

Este es un buen momento para diferenciar una señal de un evento. El manejador de una señal es llamado después del hecho, así, una ventana es destruida y después de esto el manejador es llamado (ver código)

Los eventos suceden antes del hecho. Así, con una ventana, existe un evento llamado `delete_event`, que sucede antes de que la ventana se borra. Existen métodos predefinidos para manejar los eventos. Los manejadores que conectamos a los eventos se espera que regresen un valor booleano. Éste indica si nuestro

manejador realizó todo lo necesario del evento o si es necesario que GTK+ lo maneje a su manera. Si regresamos un `FALSE` le indicamos a GTK que él debe manejar el evento. Si regresamos un `TRUE` le decimos GTK que ya no debe manejar el evento. Si hacemos esto con un `delete_event` podremos prevenir que GTK destruya la ventana.

Ahora se captura el evento `delete_event`, en lugar de la señal `destroy`. El prototipo de la función es

```
gboolean functionname (GtkWidget *, GdkEvent *, gpointer);
```

podemos cambiar la función manejadora de evento a la siguiente forma:

```
gboolean StopTheApp (GtkWidget *The Window, GdkEvent *event, gpointer data)
{
    gtk_main_quit();
    return FALSE;
}
```

Nota: Si regresáramos `TRUE` la ventana se quedaría abierta pues GTK ya no procesaría el evento.

`GdkEvent` es una estructura que contiene la información sobre el evento, nosotros podemos ver a través de sus elementos para saber qué ha pasado. Esta es otra diferencia entre eventos y señales. Los manejadores de eventos usan como argumento una estructura `GdkEvent`.

Sólo debemos tener cuidado con cambiar el código de conexión. Se debe modificar el texto de `destroy` por `delete_event`.

La forma exacta de la estructura de un `GdkEvent` varía dependiendo del widget que lo emitio. Sería muy extenso describir en detalle esto. Pero existe un tipo común para todos los `GdkEvent`: el tipo.

El elemento de tipo puede ser examinado para saber exactamente qué tipo de evento fue disparado. Se tiene un enumerado `GdkEventType` y todos los valores posibles están listados en `gdk/gdktypes.h`

Ver programa `events.c`

## **12. Elementos de la Interfase de Usuario**

En esta sección estudiaremos de manera sencilla cómo se pueden posicionar los widgets dentro de una ventana. Es importante tener un control en la presentación de dichos elementos.

### **12.1. Elementos Adicionales en una Ventana**

#### **12.1.1. Título**

Generalmente se querrá tener una ventana con un título descriptivo, esto se hace con la función

```
gtk_window_set_title(GTK_WINDOW(Window), "Titulo de la Ventana");
```

#### **12.1.2. Tamaño y Posición de una Ventana**

Se tiene la libertad de que GTK escoga el tamaño y posición de una ventana, aún así, existen funciones que nos permiten manipular estas propiedades.

```
gtk_window_set_default_size(GTK_WINDOW(window), 640, 480);
```

El segundo argumento es la altura y el tercero el ancho de la ventana. Con esta función podemos modificar el tamaño de la ventana.

```
void gtk_window_set_position(GtkWindow *window, WindowPosition position);
```

El segundo argumento es un tipo enumerado para marcar la posición de la ventana, el valor que puede tomar es uno de los siguientes 3.

GTK\_WIN\_POS\_NONE -> GTK decide donde poner la ventana

GTK\_WIN\_POS\_CENTER -> GTK coloca la ventana al centro de la pantalla

GTK\_WIN\_POS\_MOUSE -> GTK coloca la ventan donde se encuentre el ratón

Nota: ver el programa `windowposition.c`

### 12.1.3. Controles en las Ventanas

Debemos recordar que un objeto de tipo `GtkWindow` es derivado de `GtkWidget`, lo que podemos hacer con un `GtkWidget` lo podemos hacer con un `GtkWindow` (por herencia), pero no al revés.

De la misma manera `GtkWidget` deriva de `GtkObject`

`GtkObject` -> `GtkWidget` -> `GtkContainer` -> `GtkBin` -> `GtkWindow`

Un `GtkContainer` es importante pues es un objeto que nos permite contener otros objetos.

En GTK un botón es un contenedor, como ejemplo nos pemite que contenga una etiqueta.

## 12.2. Botones

Los pasos para crear un botón son:

- 1) Crear un widget (botón)
- 2) ponerlo en el contenedor
- 3) establecer sus llamadas de señales y objetos
- 4) mostrarlo

Ver el archivo `buttonclick.c`

En este programa es importante resaltar el apoyo en funciones que realicen determinadas tareas.

Una función que debemos mencionar es:

```
gtk_container_border_width(GTK_CONTAINER(window), 5);
```

El segundo argumento estará a 5 pixeles del margen de la ventana.

Si queremos insertar más botones no tendremos éxito en este caso, para esto se requiere de un mejor contenedor.

Nota: Podemos imaginarnos a una ventana como un papel y a un contenedor como una tarjeta que contiene elementos. Se posicionan los elementos en las tarjetas y éstas a su vez en la hoja de papel.

### 12.3. Cajas de Empaquetamiento (Packing Boxes)

Este es el tipo más simple de contenedor. En realidad no se utiliza el tipo `GtkBox`, se utiliza uno de sus tipos derivados:

`GtkHBox` -> ordena horizontalmente

GtkVBox -> ordena verticalmente

### 12.3.1. Definición

```
GtkWidget *gtk_hbox_new(gboolean homogeneous, gint spacing);
```

El primer argumento nos indica como se colocarán los widgets en el contenedor, TRUE de manera homogénea, FALSE de manera no homogénea. El segundo argumento es para indicar el espaciamiento entre widgets.

Para que podamos controlar el tamaño de los widgets si aumentamos o disminuimos el tamaño del contenedor dentro del cual están podemos utilizar cualquiera de las siguientes funciones:

```
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)
```

<u>Parámetro</u>	<u>Descripción</u>
<i>box</i>	La caja en la que estamos poniendo los widgets
<i>child</i>	El widget que estamos agregando
<i>expand</i>	TRUE expande a <i>child</i> para que llene cualquier espacio extra en la caja; FALSE causa que <i>box</i> encoja alrededor de <i>child</i> .
<i>fill</i>	TRUE causa que cualquier espacio extra sea rellenado por <i>child</i> , FALSE causa que el espacio extra no se rellene y se considere como otro padding. Sólo es válido si <i>expand</i> es TRUE
<i>padding</i>	La cantidad específica para rellenar el espacio alrededor de <i>child</i> .

La diferencia entre `gtk_box_pack_start()` y `gtk_box_pack_start_defaults()` es que la segunda automáticamente pone como TRUE los valores correspondientes a *fill* y *expand* y 0 a *padding*.

El parámetro "*expand*" del widget hijo es forzado automáticamente a TRUE si la caja dentro de la que está es declarada como homogénea aunque esté declarado como FALSE. Así que una caja vertical u horizontal que ha sido creada como homogénea hará que sus widgets hijos se expandan hasta que llenen cualquier espacio que quede vacío. Este parámetro fuerza a que todos los widgets en una caja vertical tengan el mismo ancho y a que los widgets que estén en una caja horizontal tengan el mismo alto.

Nota: En el primer argumento se tiene un apuntador a `GtkBox`. Será necesario hacer un cast a nuestro apuntador de caja (`box`) con el macro `GTK_BOX`. Pues definiremos al apuntador al tipo `Box` como `Widget`

También se pueden modificar los atributos actuales de packing con la función `gtk_box_set_child_packing()`, obtener los valores actuales con `gtk_box_query_child_packing()`, cambiar el orden de los hijos con `gtk_box_reorder_child()`, y fijar las características de homogeneidad y espacio con `gtk_box_set_homogeneous()` y `gtk_box_set_spacing()`.

Ver ejercicio `packingbox.c`

Nuestra primera visión de las cajas (packing boxes) es que pueden ser limitadas. Sólo pueden colocar los widgets en renglones y columnas. Pero de hecho son muy poderosas, y adecuadas para la mayor parte de las aplicaciones.

Consideremos la siguiente interfase:

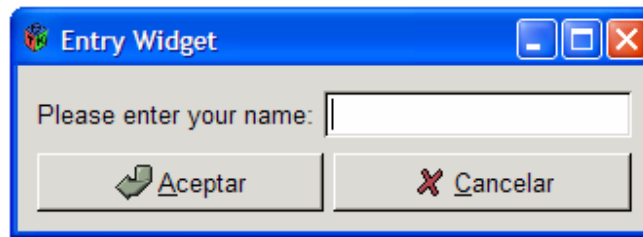


Figura 15.- Ejemplo de una interfase gráfica

Esta interfase se realiza utilizando tres packing boxes. Una horizontal para la etiqueta y el área de texto. Otra más, también horizontal para los botones. Estas dos cajas se empaquetan dentro de una caja vertical, y esta caja vertical se adiciona a su vez a la ventana. Este es el método para producir interfases complejas en GTK+

Ver el código `4interface.c`

Una función nueva que se puede utilizar, en lugar de `gtk_widget_show` es `gtk_widget_show_all`, pasando un apuntador al contenedor donde se encuentran los widgets que queremos ver, nos los mostrará todos de una vez.

### 12.3.2. Cajas para botones

Las cajas para botones son una manera conveniente para ordenar rápidamente un grupo de botones. Existen tanto horizontales como verticales. Para crearlas, se puede utilizar alguna de las siguientes funciones:

```
GtkWidget *gtk_hbutton_box_new( void );
```

```
GtkWidget *gtk_vbutton_box_new( void );
```

Para agregarles elementos, se debe hacer lo siguiente:

```
gtk_container_add (GTK_CONTAINER (button_box), child_widget);
```

### 12.4. Establecimiento del tamaño de un widget

Antes de pasar a ver cómo se crean más widgets, vamos a presentar una función que nos permite fijar el tamaño de un widget, diferente a un top-level window:

```
void gtk_widget_set_size_request(GtkWidget *widget, gint width, gint height)
```

<i>Parámetro</i>	<i>Descripción</i>
<i>widget</i>	El widget al que le estamos cambiando el tamaño
<i>width</i>	El ancho del widget (en pixeles).
<i>height</i>	El alto del widget (en pixeles).

Ejemplo:

```
gtk_widget_set_size_request(label, 200, 100)
```

### 12.5. GtkLabel -Widget para Etiquetas

En la función `gtk_button_new_with_label` lo que sucede es que GTK crea un botón, luego una etiqueta y utilizando la función `gtk_container_add` incluye la etiqueta en el botón.

Un uso de una etiqueta es para describir algo, por ejemplo se pone a un lado de una entrada de texto para dar alguna descripción al usuario de dicha entrada.

Un objeto `GtkLabel` es únicamente un texto fijo en la interfase de usuario.

En general un `GtkLabel` no se conecta a manejadores de señales y es muy raro que el usuario interactúe con éstos. Aún así podemos decir que son muy útiles.

Se crea una etiqueta con alguna de las siguientes funciones `gtk_label_new`

```
GtkWidget *gtk_label_new(const gchar *str);  
GtkWidget *gtk_label_new_with_mnemonic( const char *str );
```

El único argumento es un apuntador a la cadena de texto que queremos que aparezca. Una vez que creamos una etiqueta podemos hacer varias cosas.

Podemos modificar el texto de una etiquet con la función

```
void gtk_label_set_text(GtkLabel *label, const gchar *str);
```

Podemos obtener el texto de una etiqueta con la siguiente función:

```
const gchar* gtk_label_get_text( GtkLabel *label );
```

Se puede justificar un texto de una etiqueta con la siguiente función:

```
void gtk_label_set_justify( GtkLabel *label, GtkJustification jtype );
```

<u>Parámetro</u>	<u>Descripción</u>
<i>label</i>	La etiqueta a justificar
<i>jtype</i>	Los valores válidos son: <code>GTK_JUSTIFY_LEFT</code> , <code>GTK_JUSTIFY_RIGHT</code> , <code>GTK_JUSTIFY_CENTER</code> (el default) o <code>GTK_JUSTIFY_FILL</code>

Nota: Podemos borrar el texto de una etiqueta pasando un apuntador a una cadena vacía.

Una etiqueta también puede ajustar automáticamente el cambio de línea por medio de la función:

```
void gtk_label_set_line_wrap (GtkLabel *label, gboolean wrap);
```

<u>Parámetro</u>	<u>Descripción</u>
<i>label</i>	La etiqueta a modificar
<i>wrap</i>	Toma un argumento como <code>TRUE</code> o <code>FALSE</code>

Si se quiere desplegar una etiqueta subrayada, se podrá usar la siguiente función:

```
void gtk_label_set_pattern (GtkLabel *label, const gchar *pattern);
```

<u>Parámetro</u>	<u>Descripción</u>
<i>label</i>	La etiqueta a subrayar
<i>pattern</i>	Deberá ser una cadena del mismo tamaño que <i>label</i> y que especifica para cada caracter el caracter espacio o el caracter guión bajo para determinar si dicho caracter dentro de <i>label</i> deberá aparecer subrayado o no.

ver `labelsample.c`



## 12.6. Flechas (Arrows)

Este widget permite desplegar una flecha en diferentes posiciones y con diferentes formatos, que resulta muy útil cuando se pone dentro de un botón. Al igual que la etiqueta, no emite ningún tipo de señal.

```
GtkWidget *gtk_arrow_new( GtkArrowType arrow_type, GtkShadowType shadow_type );
```

<i>Parámetro</i>	<i>Descripción</i>
<i>arrow_type</i>	La dirección hacia donde apunta la flecha. Podrá ser alguno de los siguientes: GTK_ARROW_UP, GTK_ARROW_DOWN, GTK_ARROW_LEFT o GTK_ARROW_RIGHT.
<i>shadow_type</i>	El tipo de la flecha. Podrá ser alguno de los siguientes valores: GTK_SHADOW_IN, GTK_SHADOW_OUT (default), GTK_SHADOW_ETCHED_IN o GTK_SHADOW_ETCHED_OUT.

Si se quiere cambiar el estilo y tipo de una flecha que previamente se ha creado, se puede utilizar la siguiente función:

```
void gtk_arrow_set( GtkArrow *arrow, GtkArrowType arrow_type,
                  GtkShadowType shadow_type );
```

<i>Parámetro</i>	<i>Descripción</i>
<i>arrow</i>	La flecha a la que se quiere cambiar el estilo y tipo.
<i>arrow_type</i>	La dirección hacia donde apunta la flecha. Podrá ser alguno de los siguientes: GTK_ARROW_UP, GTK_ARROW_DOWN, GTK_ARROW_LEFT o GTK_ARROW_RIGHT.
<i>shadow_type</i>	El tipo de la flecha. Podrá ser alguno de los siguientes valores: GTK_SHADOW_IN, GTK_SHADOW_OUT (default), GTK_SHADOW_ETCHED_IN o GTK_SHADOW_ETCHED_OUT.

## 12.7. GtkEntry - Obteniendo Texto del Usuario

Otro objeto muy utilizado es GtkEntry. Este widget nos permite que el usuario teclee información en una caja de entrada, en una ventana.

```
GtkWidget *gtk_entry_new(void);
GtkWidget *gtk_entry_new_with_max_length(guint16 max);
```

La primer función crea una caja de entrada vacía y regresa un apuntador a su widget. La segunda crea una caja que limita el número máximo de caracteres que se pueden teclear en la caja.

Algo muy importante de las cajas de texto es que nos permiten mostrar y capturar texto. Las funciones para permitir al usuario teclear y capturar texto son:

```
gchar *gtk_entry_get_text(GtkEntry *entry);
void gtk_entry_set_text(GtkEntry *entry, const gchar *text);
```

También habrá ocasiones en que no queremos que el texto que el usuario teclee se vea. Un ejemplo es un password.

```
void gtk_entry_set_visibility(GtkEntry *entry, gboolean visible);
```

Cuando el último argumento es TRUE, la caja de texto funciona como debería ser, desplegando el contenido, al ponerlo en FALSE el texto se oculta.

En ocasiones se necesita crear un entry que no permita que el usuario escriba en él, para esto se podrá utilizar la siguiente función:

```
void gtk_editable_set_editable( GtkEditable *entry, gboolean editable );
```

Un valor de TRUE (defalut) permite escribir en el entry, un valor FALSE no lo permite.

Se puede también hacer que el texto del entry aparezca seleccionado con la ayuda de la función:

```
void gtk_editable_select_region( GtkEditable *entry, gint start, gint end );
```

ver `buttonsendry.c` y `buttonsendry2.c`, `entrysample.c`

Este tipo de widgets suele generar muchas señales, las más importantes son `activated` y `changed`, que se genera cuando ocurre un cambio en el texto. El prototipo de la función que se tiene que llamar cuando se genera esta señal es el siguiente:

```
void user_function (GtkEditable *editable, gpointer user_data)
```

Cuando queremos trabajar con más de una línea de texto, podremos utilizar un `GtkTextView` widget

Un `GtkTextView` se crea con la siguiente función:

```
GtkWidget *gtk_text_view_new(void);
```

```
void gtk_text_view_set_editable(GtkText *text, gboolean editable)
```

La función que nos permite obtener el texto es la siguiente:

```
gchar *gtk_editable_get_chars(GtkEditable *editable, gint inicio, gint fin).
```

<i>Parámetro</i>	<i>Descripción</i>
<i>editable</i>	El widget de la familia <code>GtkEditable</code> del que se quiere obtener el texto. Hay que recordar que el <code>GtkText</code> descende de la familia <code>GtkEditable</code> , es por eso que se puede utilizar esta función sin ningún problema, teniendo cuidado únicamente de hacer el correspondiente cast.
<i>inicio</i>	El caracter dentro del <code>GtkEditable</code> desde donde se quiere obtener el texto.
<i>Fin</i>	El caracter dentro del <code>GtkEditable</code> hasta donde se quiere obtener el texto.

## 12.8. Implementación de una barra de menús

En las aplicaciones gráficas más comunes, se puede observar una barra de menú en la parte superior de la aplicación, la cual generalmente contiene funciones estándar del programa (como Abrir, Salvar, Salir, etc). Para poder agregar algo similar a nuestra aplicación debemos de crear un widget capaz de hacer lo anterior.

La barra de menú será manejada por la caja vertical, y debido a que la rutina de empaquetamiento que utilizaremos para agregar la barra al contenedor es `gtk_box_pack_start()`, la barra de menú debe de ser el primer elemento que se agrega al contenedor (la caja vertical) de tal manera que aparezca antes que cualquier widget. Una barra de menú vacía es creada con la función:

```
GtkWidget *gtk_menu_bar_new(void)
```

Ejemplo:

```
GtkWidget *vbox, *menu_bar;  
/* ..... */  
menu_bar = gkt_menu_bar_new();
```

Una vez que hemos creado nuestra barra de menú, podemos agregarle todos los menús que queramos a la barra.

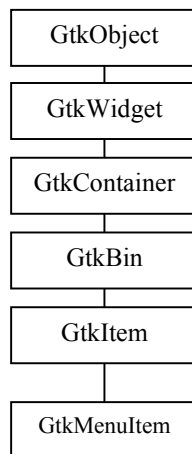


Figura 16.- Árbol de clases de la clase GtkWidget

Nótese que GtkWidget es descendiente de la clase GtkWidget, lo que significa que sólo podrá tener un hijo, el cual será, igual que los botones, la etiqueta que se le asigna (o sea su nombre). Para crear cada uno de los menús que irán dentro de la barra de menú, se podrá utilizar alguna de las siguientes funciones:

```

GtkWidget *gtk_menu_item_new(void);

GtkWidget *gtk_menu_item_new_with_label(const gchar *text)

GtkWidget *gtk_menu_item_new_with_mnemonic(const char *label);

```

Parámetro	Descripción
<i>text</i>	El nombre del menú que se acaba de crear.

Una vez que creamos el menú, debemos de agregarlo a la barra de menú que creamos anteriormente, esto se hace con cualquiera de las siguientes funciones:

```

void gtk_menu_bar_append(GtkMenuBar *menu_bar, GtkWidget *child)
void gtk_menu_bar_prepend(GtkMenuBar *menu_bar, GtkWidget *child)

```

Parámetro	Descripción
<i>menu_bar</i>	La barra de menú a la que estamos agregando el menú
<i>child</i>	El menú que estamos agregando.

Ejemplo:

```

GtkWidget *file_item;
/* ..... */
file_item = gtk_menu_item_new_with_label("Archivo");
gtk_menu_bar_append(GTK_MENU_BAR(menu_bar), file_item);
gtk_widget_show(file_item);

```

El siguiente paso es agregar un menú al elemento Archivo en nuestra barra de menú. Dentro de dicho menú, habrán elementos, tal y como Abrir, Salvar, Salvar como..., Imprimir, Salir. En GTK+, los menús vacíos son creados con la función:

```

GtkWidget *gtk_menu_new (void)

```

Cada uno de los elementos de nuestro menú (como Salir), son GtkMenuItems, tal y como es Archivo en la barra de menú, así que cada uno de los elementos que irán dentro de nuestro menú se crearán de la misma manera en que se crearon los componentes de nuestra barra de menús, con `gtk_menu_item_new_with_label()`. Generalmente, la opción Salir de un menú está separada del resto de las demás por medio de una línea horizontal (un separador), el cual se puede crear con la función:

```
GtkWidget *gtk_menu_item_new(void)
```

Ejemplo:

```
GtkWidget *file_menu, *separator_item, *open_item, *exit_item;
/* ..... */
file_menu = gtk_menu_new();
separator_item = gtk_menu_item_new(void);
exit_item = gtk_menu_item_new_with_label("Salir");
open_item = gtk_menu_item_new_with_label("Abrir");
```

Una vez que hemos creado los elementos de nuestro menú, tenemos que agregarlos a él, esto se hace con cualquiera de las siguientes funciones (la manera más común de hacerlo es con `append`):

```
void gtk_menu_shell_append(GtkMenuShell *menu_shell, GtkWidget *child)
void gtk_menu_shell_prepend(GtkMenuShell *menu_shell, GtkWidget *child)
void gtk_menu_shell_insert(GtkMenuShell *menu_shell, GtkWidget *child,
                           gint position)
```

<u>Parámetro</u>	<u>Descripción</u>
<i>menu_shell</i>	El menú al que estamos agregando el elemento.
<i>child</i>	El elemento que estamos agregando.
<i>position</i>	La posición dentro del menú en la cual estamos agregando el elemento.

El siguiente ejemplo muestra la manera en que se agregan elementos a un menú, así la manera en que se asocia la llamada a una función en el momento de seleccionar “Salir” del menú:

```
gtk_menu_shell_append(GTK_MENU_SHELL(file_menu), open_item);
gtk_menu_shell_append(GTK_MENU_SHELL(file_menu), separator_item);
gtk_menu_shell_append(GTK_MENU_SHELL(file_menu), exit_item);
g_signal_connect(G_OBJECT(exit_item), "activate",
                 G_CALLBACK( print_and_quit ), NULL);
```

Una vez que hemos creado el menú y que le hemos agregado los componentes necesarios, lo que falta hacer es agregar el menú al `menu_item` que previamente agregamos a la barra de menú. Esto se hace con la función:

```
void gtk_menu_item_set_submenu(GtkMenu *menu, GtkWidget *sub_menu)
```

<u>Parámetro</u>	<u>Descripción</u>
<i>menu</i>	El <code>menu_item</code> al que estamos agregando el menú
<i>sub_menu</i>	El menú al que estamos agregando al <code>menu_item</code>

Adicionalmente los elementos del menú tienen que mostrarse, a continuación se pone el código necesario:

```
gtk_widget_show(open_item);
gtk_widget_show(exit_item);
gtk_widget_show(separator_item);
gtk_menu_item_set_submenu(GTK_MENU_ITEM(file_item), file_menu);
```

A continuación se enseña el código completo de una aplicación que crea un menú:

```
#include <gtk/gtk.h>

/* Callbacks */
void print_and_quit( GtkWidget *was_clicked, gpointer user_data )
{
    /* Use glibs printf equivalent to print a message */
    g_print( "Thank you for using this program.\n" );
    gtk_main_quit();
}

gboolean delete_event_handler( GtkWidget *widget, GdkEvent *event,
                               gpointer user_data )
{
    /* Received closure from the window manager */
    g_print("The window manager is asking to close this application\n");
    return( FALSE ); /* FALSE - do not prevent closure */
}

int main( int argc, char *argv[] )
{
    GtkWidget *top_widget, *box, *label, *separator_item;
    GtkWidget *menu_bar, *file_item, *file_menu, *exit_item;

    /* 1. Initialize the environment */
    gtk_init( &argc, &argv );

    /* 2a. Create widgets */
    top_widget = gtk_window_new( GTK_WINDOW_TOPLEVEL );
    box = gtk_vbox_new( FALSE, 0 );
    menu_bar = gtk_menu_bar_new( );
    file_item = gtk_menu_item_new_with_label( "File" );
    file_menu = gtk_menu_new();
    separator_item = gtk_menu_item_new( );
    exit_item = gtk_menu_item_new_with_label( "Exit" );
    label = gtk_label_new( "GTK+ is fun!" );

    /* 2b. Set attributes */
    gtk_window_set_title( GTK_WINDOW( top_widget ), "new_gtkfun" );
    gtk_container_set_border_width( GTK_CONTAINER( top_widget ), 0 );
    gtk_widget_set_name( top_widget, "new_gtkfun" );

    /* 3. Register callbacks */
    g_signal_connect( G_OBJECT( exit_item ), "activate",
                     G_CALLBACK( print_and_quit ), NULL );
    g_signal_connect( G_OBJECT( top_widget ), "delete_event",
                     G_CALLBACK( delete_event_handler ), NULL );
    g_signal_connect( G_OBJECT( top_widget ), "destroy",
                     G_CALLBACK( print_and_quit ), NULL );

    /* 4. Define instance hierarchy (pack the widgets) */
    gtk_container_add( GTK_CONTAINER( top_widget ), box );
    gtk_box_pack_start_defaults( GTK_BOX( box ), menu_bar );
    gtk_box_pack_start_defaults( GTK_BOX( box ), label );

    /* Pack the menu bar */
    gtk_menu_bar_append( GTK_MENU_BAR( menu_bar ), file_item );
    gtk_menu_item_set_submenu( GTK_MENU_ITEM( file_item ), file_menu );
    gtk_menu_shell_append( GTK_MENU_SHELL( file_menu ), separator_item );
    gtk_menu_shell_append( GTK_MENU_SHELL( file_menu ), exit_item );

    /* 5. Show the widgets */
}
```

```
gtk_widget_show_all( top_widget );

/* 6. Processing Loop */
gtk_main();
g_print( "Bye!\n");
return 0;
}
```

Podemos utilizar la siguiente función para justificar hasta la derecha un menú en nuestra barra de menús:

```
void gtk_menu_item_right_justify(GtkMenuItem *menu_item)
```

<u>Parámetro</u>	<u>Descripción</u>
<i>menu_item</i>	El menú que se quiere justificar a la derecha

Ejemplo:

```
/* ..... */
gtk_menu_bar_append(GTK_MENU_BAR(menu_bar), help_item);
gtk_menu_item_right_justify(GTK_MENU_ITEM(help_item));
```

Es común que los menús de ayuda tengan la información necesaria acerca del programa, la cual normalmente incluye el autor, la versión y la información de derechos de autor del mismo. Típicamente, esta información es la última opción dentro del menú de ayuda. El siguiente fragmento de código crea un menú de ayuda:

```
/* Asumimos que ya han sido declaradas variables de tipo GtkWidget *help_menu,
 *query_item, *separator, *about_help */
help_menu = gtk_menu_new();
gtk_menu_item_set_submenu(GTK_MENU_ITEM(help_item), help_menu);

/* Creando los elementos en el menu help */

query_item = gtk_menu_item_new_with_label ("What's this");
gtk_menu_shell_append(GTK_MENU_SHELL(help_menu), query_item);
gtk_widget_show(query_item);

separator = gtk_menu_item_new();
gtk_menu_shell_append(GTK_MENU_SHELL(help_menu), separator);
gtk_widget_show(separator);

about_help = gtk_menu_item_new_with_label("About my program...");
gtk_menu_shell_append(GTK_MENU_SHELL(help_menu), about_help);
gtk_widget_show(about_help);
```

### 12.8.1. Accesos directos en los menús (*Aceleradores*)

En GTK+, los accesos directos son conocidos como *aceleradores*. Para crear un grupo de *aceleradores* se utilizará la función:

```
GtkAccelGroup *gtk_accel_group_new( void )
```

Para agregar el nuevo grupo de *accelerators* al top-level widgets se utilizará la siguiente función:

```
void gtk_window_add_accel_group(GtkWindow *window, GtkAccelGroup *accel_group)
```

<u>Parámetro</u>	<u>Descripción</u>
<i>window</i>	La ventana a la cual estamos agregando el <i>accel_group</i>
<i>accel_group</i>	El grupo de aceleradores que queremos agregar

Ahora necesitamos asignar los aceleradores a cada uno de los elementos de nuestros menús. Esto lo hacemos con la siguiente función:

```
void gtk_widget_add_accelerator(GtkWidget *widget,
                              const gchar *accel_signal,
                              GtkAccelGroup *accel_group, guint accel_key,
                              guint accel_mods, GtkAccelFlags accel_flags)
```

<u>Parámetro</u>	<u>Descripción</u>
<i>widget</i>	El widget al que se quiere agregar el acelerador.
<i>accel_signal</i>	La señal que quiere emitirse (debe de ser válida para el widget).
<i>accel_group</i>	El grupo de aceleradores al que se quiere agregar el acelerador.
<i>accel_key</i>	La tecla del teclado que quiere utilizarse como acceso directo
<i>accel_mods</i>	Cualquier modificador para seleccionar el acceso directo (como Control, Shift, etc.). Las opciones válidas son: GDK_BUTTONn_MASK (en donde <i>n</i> va de 1 a 5), GDK_CONTROL_MASK, GDK_LOCK_MASK, GDK_MODn_MASK (en donde <i>n</i> va de 1 a 5), GDK_RELEASE_MASK y GDK_SHIFT_MASK.
<i>accel_flags</i>	Información acerca del despliegue del acelerador. Las opciones válidas son: GTK_ACCEL_VISIBLE, GTK_ACCEL_SIGNAL_VISIBLE y GTK_ACCEL_LOCKED.

Si en el campo *accel\_flags* queremos poner más de una opción, podemos utilizar el operador | (OR), por ejemplo `GTK_ACCEL_VISIBLE | GTK_ACCEL_SIGNAL_VISIBLE`. La bandera `GTK_ACCEL_VISIBLE` normalmente está prendida, de tal manera que la tecla que llama al acelerador es desplegada en el widget. `GTK_ACCEL_SIGNAL_VISIBLE` es encendida cuando queremos indicar qué señal será enviada como respuesta a la llamada del acelerador. `GTK_ACCEL_LOCKED` se enciende si no queremos cambiar el despliegue del acelerador.

El siguiente ejemplo es un fragmento de código que nos permite agregar el acelerador “Ctrl+Q” a la opción salir del menú “Archivo”.

```
/* Asume que previamente hemos declarado GtkAccelGroup *accel_group */
accel_group = gtk_accel_group_new();
gtk_accel_group_attach(accel_group, GTK_OBJECT(top_widget));
gtk_widget_add_accelerator(exit_item, "activate", accel_group, 'Q',
                          GDK_CONTROL_MASK, GTK_ACCEL_VISIBLE);
```

### 12.8.2. Aceleradores subrayados

Este tipo de aceleradores se utiliza cuando un menú está activo, para poder tener acceso a la opción por medio de la letra subrayada, en lugar de hacer click con el mouse. Para crear un acelerador de este tipo utilizaremos la siguiente función:

```
GtkAccelGroup *gtk_menu_ensure_uline_group(GtkMenu *menu)
```

<u>Parámetro</u>	<u>Descripción</u>
<i>menu</i>	El menú al cual se está agregando el acelerador.

El siguiente paso es agregar el acelerador a la opción del menú, lo cual se logra por medio de dos pasos:

1. Se debe crear una etiqueta, la cual debe contener el caracter subrayado, utilizando la función `gtk_label_parse_uline()`.
2. Después se deberá fijar el caracter correspondiente como un acelerador la opción del menú, utilizando `gtk_widget_add_accelerator()`.

La función que crea la etiqueta que contiene el caracter subrayado creará toda la etiqueta, no únicamente el caracter subrayado. Por lo tanto tendremos que redefinir el proceso para crear la opción en el menú. Esto lo haremos creando un `menu_item` sin etiqueta, y después, llamando a la función `gtk_label_parse_uline()`, crearemos la etiqueta con el caracter subrayado.

```
guint gtk_label_parse_uline(GtkLabel *label, const gchar *string)
```

<u>Parámetro</u>	<u>Descripción</u>
<i>label</i>	El label widget que almacenará la etiqueta.
<i>string</i>	La cadena que se desplegará en la etiqueta, conteniendo el caracter subrayado.

Ejemplo:

```
/* Asume que se han declarado las siguientes variables:
 * GtkWidget *file_item, *file_menu;
 * GtkWidget *exit_item;
 * GtkAccelGroup *file_accel_group;
 * guint key;
 */

/*Crea el File item y su menu */
file_item = gtk_menu_item_new_with_label("Archivo");
file_menu = gtk_menu_new();
exit_item = gtk_menu_item_new_with_label("");
file_accel_group = gtk_ensure_uline_accel_group(GTK_MENU(file_menu));
key = gtk_label_parse_uline(GTK_LABEL(GTK_BIN(exit_item)->child),
    "_Exit");
gtk_widget_add_accelerator(exit_item, "activate", file_accel_group, key,
    GTK_NONE, GDK_NONE);
```

Un miembro de la clase `GtkMenuItems` puede ser padre, ya que desciende de la clase `GtkBin`. Para entender el código anterior, es necesario conocer la estructura de la clase `GtkBin`:

```
typedef struct _GtkBin GtkBin;
struct _GtkBin{
    GtkContainer container;
    GtkWidget *child;
};
```

Es por esto que para tener acceso al hijo de `exit_item` (la etiqueta), tenemos que hacer un cast a `GtkBin`, y después podemos tener acceso a la etiqueta:

```
GTK_BIN(exit_item)->child.
```

### 12.8.3. Poniendo submenús dentro de los menús

Para poder agregar un menú dentro de otro menú, lo que tenemos que hacer es algo similar a lo que se vió anteriormente. Es decir, antes, para crear un menú, agregábamos un `menu_item` a un `menu_bar`, y después, agregábamos un menú al `menu_item`. Dentro del menú agregábamos las opciones del mismo. Siguiendo el mismo esquema, si en lugar de agregar una opción como tal dentro de un menú, queremos agregar un submenú, lo que tenemos que hacer es agregar el submenú a la opción del menú. El siguiente código ejemplifica lo anterior:

```
#include <gtk/gtk.h>
#define NO_PADDING 0
#define NO_SPACING 0

/* Callbacks */
void line_plot( GtkWidget *was_clicked, GtkWidget *where_to_draw );
void print_and_quit( GtkWidget *was_clicked, gpointer user_data );
```



```
gint delete_event_handler( GtkWidget *widget, GdkEvent *event,
                           gpointer user_data );
gint dialog_delete_handler( GtkWidget *widget, GdkEvent *event,
                           gpointer user_data );

int main( int argc, char *argv[] )
{
    GtkWidget *top_widget;
    GtkWidget *vbox, *menu_bar, *file_item, *file_menu, *exit_item;
    GtkWidget *plot_item, *plot_area;
    GtkWidget *help_item;
    GtkWidget *help_menu, *query_item, *separator, *about_help;
    GtkWidget *copyright_dlg, *copyright_label, *copyright_ok;
    GtkWidget *overview_dialog, *overview_label, *overview_ok;
    GtkWidget *help_submenu, *copyright, *overview;

    GtkAccelGroup *accel_group, *file_accel_group;
    guint key;

    /* 1. Initialize the environment */
    gtk_init( &argc, &argv );

    /* 2a. Create widgets */
    top_widget = gtk_window_new( GTK_WINDOW_TOPLEVEL );
    vbox = gtk_vbox_new( FALSE, NO_SPACING );
    menu_bar = gtk_menu_bar_new( );

    /* Create the File item and its menu */
    file_item = gtk_menu_item_new_with_label( "File" );
    file_menu = gtk_menu_new();
    exit_item = gtk_menu_item_new_with_label( "" );
    file_accel_group = gtk_menu_ensure_uiline_accel_group(
        GTK_MENU( file_menu ));
    key = gtk_label_parse_uiline( GTK_LABEL( GTK_BIN( exit_item )->child),
        "_Exit" );
    gtk_widget_add_accelerator( exit_item, "activate", file_accel_group,
        key, GDK_NONE, GDK_NONE );

    /* Create the Plot item */
    plot_item = gtk_menu_item_new_with_label( "Plot" );

    /* Create the Help item and its menus */
    help_item = gtk_menu_item_new_with_label( "Help" );
    help_menu = gtk_menu_new();
    query_item = gtk_menu_item_new_with_label( "What's This?" );
    separator = gtk_menu_item_new();
    about_help = gtk_menu_item_new_with_label( "About my program" );
    help_submenu = gtk_menu_new( );
    copyright = gtk_menu_item_new_with_label( "Coyright information" );
    overview = gtk_menu_item_new_with_label( "Overview" );

    /* Create the Work Area */
    plot_area = gtk_drawing_area_new();

    /* Create pop-up dialogs */
    copyright_dlg = gtk_dialog_new();
    copyright_label = gtk_label_new( "Author: You\n Version 0.0.1\n"
        "Freely distributable under LGPL" );
    copyright_ok = gtk_button_new_with_label( "OK" );

    /* Put the label in vbox, and the button in action_area */
```

```
gtk_box_pack_start( GTK_BOX( GTK_DIALOG(copyright_dlg)->vbox),
    copyright_label, FALSE, FALSE, NO_PADDING );
gtk_box_pack_start( GTK_BOX(GTK_DIALOG(copyright_dlg)->action_area),
    copyright_ok, FALSE, FALSE, NO_PADDING );
gtk_widget_show( copyright_label );
gtk_widget_show( copyright_ok );

overview_dialog = gtk_dialog_new();
overview_label = gtk_label_new( "This is a simple plotting program\n"
    "It currently accepts no outside data, \n"
    "though this feature will be available in a future\n"
    "release.\n");
overview_ok = gtk_button_new_with_label( "OK" );

/* Put the label in vbox, and the button in action_area */
gtk_box_pack_start( GTK_BOX( GTK_DIALOG(overview_dialog)->vbox),
    overview_label, FALSE, FALSE, NO_PADDING );
gtk_box_pack_start( GTK_BOX(GTK_DIALOG(overview_dialog)->action_area),
    overview_ok, FALSE, FALSE, NO_PADDING );
gtk_widget_show( overview_label );
gtk_widget_show( overview_ok );

/* Set up the accelerator group */
accel_group = gtk_accel_group_new();
gtk_accel_group_attach( accel_group, GTK_OBJECT( top_widget ) );
gtk_widget_add_accelerator( exit_item, "activate", accel_group,
    'Q', GDK_CONTROL_MASK, GTK_ACCEL_VISIBLE );

/* 2b. Set Attributes */
gtk_window_set_title( GTK_WINDOW( top_widget ), "Plotter" );
gtk_drawing_area_size( GTK_DRAWING_AREA(plot_area), 300, 200 );

/* 3. Registering Callback routines */
g_signal_connect( G_OBJECT( overview_dialog ), "delete_event",
    G_CALLBACK( dialog_delete_handler), NULL );
g_signal_connect( G_OBJECT( copyright_dlg ), "delete_event",
    G_CALLBACK( dialog_delete_handler), NULL );
g_signal_connect( G_OBJECT( top_widget ), "delete_event",
    G_CALLBACK( delete_event_handler), NULL );
g_signal_connect( G_OBJECT( top_widget ), "destroy",
    G_CALLBACK( print_and_quit), NULL );
g_signal_connect( G_OBJECT( exit_item ), "activate",
    G_CALLBACK( print_and_quit), NULL );
g_signal_connect( G_OBJECT( plot_item ), "activate",
    line_plot, plot_area );
g_signal_connect_object( GTK_OBJECT( copyright ), "activate",
    G_CALLBACK( gtk_widget_show ), GTK_OBJECT( copyright_dlg ) );
g_signal_connect_object( GTK_OBJECT( copyright_ok ), "clicked",
    G_CALLBACK( gtk_widget_hide ), GTK_OBJECT( copyright_dlg ) );
g_signal_connect_object( GTK_OBJECT( overview ), "activate",
    G_CALLBACK( gtk_widget_show ), GTK_OBJECT( overview_dialog ) );
g_signal_connect_object( GTK_OBJECT( overview_ok ), "clicked",
    G_CALLBACK( gtk_widget_hide ), GTK_OBJECT( overview_dialog ) );

/* 4. Defining the Instance Hierarchy */
gtk_container_add( GTK_CONTAINER( top_widget ), vbox );
gtk_box_pack_start( GTK_BOX(vbox), menu_bar, FALSE, FALSE, NO_PADDING);
gtk_box_pack_start( GTK_BOX(vbox), plot_area, FALSE, FALSE, NO_PADDING );

/* Pack the Menu Bar */
gtk_menu_bar_append( GTK_MENU_BAR( menu_bar ), file_item );
```

```

gtk_menu_bar_append( GTK_MENU_BAR( menu_bar ), plot_item );
gtk_menu_bar_append( GTK_MENU_BAR( menu_bar ), help_item );

/* Attach and pack the menus */
gtk_menu_item_set_submenu( GTK_MENU_ITEM( file_item ), file_menu );
gtk_menu_shell_append(GTK_MENU_SHELL(file_menu), exit_item );

gtk_menu_item_set_submenu( GTK_MENU_ITEM( help_item ), help_menu );
gtk_menu_shell_append(GTK_MENU_SHELL(help_menu), query_item );
gtk_menu_shell_append(GTK_MENU_SHELL(help_menu), separator );
gtk_menu_shell_append(GTK_MENU_SHELL(help_menu), about_help );
gtk_menu_item_set_submenu( GTK_MENU_ITEM( about_help ), help_submenu );
gtk_menu_shell_append(GTK_MENU_SHELL(help_submenu), copyright );
gtk_menu_shell_append(GTK_MENU_SHELL(help_submenu), overview );

/* 5. Showing the Widgets */
gtk_widget_show_all( top_widget );

/* 6. Processing Loop */
gtk_main();
g_print( "Bye!\n" );
return 0;
}

```

#### 12.8.4. Poniendo íconos en un menú

Hasta ahora se han estado creando *menu items* con una etiqueta para los elementos de un menú y *menu items* sin etiqueta para los separadores dentro de un menú. Ahora crearemos un *menu item* con una etiqueta y un ícono. Un *menu item* solamente puede manejar un solo hijo, porque descende de la clase GtkBin. Para poder manejar dos widgets en un *menu item*, se necesitará crear un GtkHBox como hijo del GtkMenuItem; una vez hecho lo anterior, el ícono y la etiqueta serán puestas dentro del GtkHBox. El siguiente código ejemplifica lo anterior.

```

#include <gtk/gtk.h>

/* ***** CALLBACKS ***** */
gboolean delete_event_handler(GtkWidget *widget, GdkEvent *event,
                             gpointer user_data){
    //g_print("The window manager has asked to close the window\n");
    return (FALSE);
}

void quit(GtkWidget *clicked, gpointer user_data){
    //g_print("Saliendo de la aplicacion...\n");
    gtk_main_quit();
}

int main(int argc, char *argv[])
{
    GtkWidget *top_widget, *box, *label, *drawing;
    GtkWidget *menu_bar, *file_item, *file_menu;
    GtkWidget *smile_item, *smile_hbox, *smiley_icon, *smiley_label;
    GdkPixmap *small_smiley;
    GdkBitmap *transparent;

    /* Inicializa el ambiente */
    gtk_init(&argc, &argv);

    /* Crea la ventana */
    top_widget = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(top_widget), "Smile");

```

```
gtk_widget_set_name(top_widget, "smile");

g_signal_connect(G_OBJECT(top_widget), "delete_event",
                 G_CALLBACK( delete_event_handler),NULL);

g_signal_connect(G_OBJECT(top_widget), "destroy",
                 G_CALLBACK( quit), NULL);

/*Crea la caja vertical */
box = gtk_vbox_new (FALSE, 0);
gtk_container_add(GTK_CONTAINER(top_widget),box);

/* Crea la barra de menus y sus componentes */
menu_bar = gtk_menu_bar_new();
gtk_box_pack_start_defaults(GTK_BOX(box), menu_bar);

/* Crea el file item y su menu */
file_item = gtk_menu_item_new_with_label ("File");
gtk_menu_bar_append(GTK_MENU_BAR(menu_bar),file_item);
file_menu = gtk_menu_new();
gtk_menu_item_set_submenu(GTK_MENU_ITEM(file_item),file_menu);

/*Crea los elementos del file menu */
smile_item = gtk_menu_item_new();
gtk_menu_shell_append(GTK_MENU_SHELL(file_menu), smile_item);

smile_hbox = gtk_hbox_new(FALSE,0);
gtk_container_add(GTK_CONTAINER(smile_item),smile_hbox);

/* Creamos el icono */
small_smiley = gdk_pixmap_colormap_create_from_xpm(NULL,
                                                    gdk_colormap_get_system(),&transparent,
                                                    NULL,"smile.xpm");
smiley_icon = gtk_pixmap_new(small_smiley, transparent);
gdk_pixmap_unref(small_smiley);
gtk_box_pack_start_defaults(GTK_BOX(smile_hbox), smiley_icon);

/* Creamos la etiqueta */
smiley_label = gtk_label_new("Smile");
gtk_box_pack_start_defaults(GTK_BOX(smile_hbox),smiley_label);

/* Creamos los demas componentes de la interfase */
label = gtk_label_new ("Ejemplo ");
gtk_box_pack_start_defaults(GTK_BOX(box), label);

drawing = gtk_drawing_area_new();
gtk_drawing_area_size(GTK_DRAWING_AREA(drawing), 200, 200);
gtk_box_pack_start_defaults(GTK_BOX(box),drawing);

gtk_widget_show_all(top_widget);

gtk_main();

return 0;
}
```

## 12.9. GtkTable widget

Hasta ahora, el único tipo de widgets que hemos visto que pueden hacerse cargo de muchos widget es una caja (Vbox o Hbox). Sin embargo, habrán ocasiones en las que necesitaremos manejar muchas columnas y/o

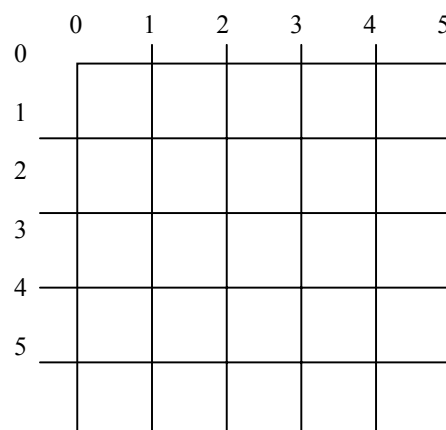
renglones. Una solución sería poner dentro de una caja vertical muchas cajas horizontales. Esto podría ser una buena opción, sin embargo, existe una mejor opción: utilizar un `GtkTable`. Este widget consiste de un número de renglones y columnas (celdas) en las cuales se pueden poner más widgets. Los widgets hijos pueden ocupar más de una celda, a diferencia de las cajas.

Para crear una instancia de un `GtkTable`, se utilizará la siguiente función:

```
GtkWidget *gtk_table_new(guint rows, guint columns, gboolean homogeneous)
```

<u>Parámetro</u>	<u>Descripción</u>
<i>rows, columns</i>	El número inicial de renglones y columnas en la tabla. La tabla automáticamente crecerá si se agrega un hijo fuera del tamaño original.
<i>homogeneous</i>	TRUE indica que todas las celdas serán del mismo tamaño, FALSE indica lo contrario.

Todas las celdas de una columna tendrán el mismo ancho y alto, inclusive cuando el parámetro *homogeneous* es FALSE. El ancho de una columna será igual a la celda más ancha en dicha columna, lo mismo pasará con el alto de un renglón. Los widgets dentro de una tabla podrán abarcar más de dos celdas. Cuando un widget es agregado a una tabla, se deberán de pasar como parámetros los límites de la tabla dentro de los cuales estará dicho widget. Supongamos que tenemos una tabla de 4 columnas y 5 renglones:



**Figura 17.- Distribución de un GtkTable**

Para poder agregar los widgets a la tabla, utilizaremos la siguiente función:

```
void gtk_table_attach_defaults(GtkTable *table, GtkWidget *widget,
                              guint left_attach, guint right_attach,
                              guint top_attach, guint bottom_attach)
```

<u>Parámetro</u>	<u>Descripción</u>
<i>table</i>	La tabla a la cual se están agregando los widgets.
<i>widget</i>	El widget que se está agregando a la tabla.
<i>left_attach, right_attach</i>	Los límites izquierdo y derecho de la tabla entre los cuales se agregará el widget.
<i>top_attach, bottom_attach</i>	Los límites superior e inferior entre los cuales se agregará el widget.

De igual manera que las cajas vertical y horizontal, una tabla tiene atributos de “empaquetamiento” que se fijan el momento en que se agregan los widgets. Para fijar dichos atributos, en lugar de la función anterior, se puede utilizar la siguiente para agregar los widgets a la tabla:

```
void gtk_table_attach(GtkTable *table, GtkWidget *widget,
```

```
guint left_attach, guint right_attach,
guint top_attach, guint bottom_attach,
GtkAttachOptions xoptions, GtkAttachOptions yoptions,
guint xpadding, guint ypadding)
```

<u>Parámetro</u>	<u>Descripción</u>
<i>table</i>	La tabla a la cual se están agregando los widgets.
<i>widget</i>	El widget que se está agregando a la tabla.
<i>left_attach, right_attach</i>	Los límites izquierdo y derecho de la tabla entre los cuales se agregará el widget.
<i>top_attach, bottom_attach</i>	Los límites superior e inferior entre los cuales se agregará el widget.
<i>xoptions, yoptions</i>	Los atributos de empaquetamiento en las direcciones x y y del widget hijo.
<i>xpadding, ypadding</i>	La cantidad de espacio alrededor de los hijos, en dirección x y y.

Las opciones válidas para los atributos de empaquetamiento en "x" y "y" son las siguientes:

- **GTK\_EXPAND:** Causará que la tabla se expanda haciendo uso de cualquier espacio que quede libre en la ventana.
- **GTK\_FILL:** Si la tabla es más grande que los widgets que contiene y este argumento es especificado, los widgets se expandirán para usar todo el espacio disponible.
- **GTK\_SHRINK:** Si la tabla cambia de tamaño, haciéndose más pequeña de tal manera que los widgets que contiene no caben en ella (usualmente ocurre esto al cambiar de tamaño la ventana), entonces los widgets normalmente desaparecerán de la ventana. Si este argumento es especificado, los widgets se encogerán junto con la tabla.

Existen también funciones relacionadas con las tablas que permiten fijar un espacio determinado entre los renglones o entre un renglón o columna específica:

```
void gtk_table_set_row_spacing( GtkTable *table, guint row, guint spacing );
void gtk_table_set_col_spacing ( GtkTable *table, guint column, guint spacing );
```

Para las columnas, el espacio se creará a la derecha de la columna indicada y para los renglones se creará debajo de ésta.

Se puede fijar también un espacio consistente para todas las columnas o renglones por medio de las siguientes funciones:

```
void gtk_table_set_row_spacings( GtkTable *table, guint spacing );
void gtk_table_set_col_spacings( GtkTable *table, guint spacing );
```

Cabe hacer notar que al usar cualquiera de las funciones anteriores, el último renglón o columna no tendrá ningún espacio.

## 12.10. File Selection Widget

```
GtkWidget *gtk_file_selection_new(const gchar *title)
```

<u>Parámetro</u>	<u>Descripción</u>
<i>title</i>	El título a poner en la ventana. ("Save as...", "Load data...", "Select Logging File...", etc).

Si queremos modificar la apariencia de nuestro widget, como ocultar los file operation buttons, podemos utilizar la siguiente función:

```
void gtk_file_selection_hide_fileop_buttons(GtkFileSelection *filesel)
```

Parámetro	Descripción
<i>filesel</i>	El file selection widget al que se quieren ocultar los botones.

Si queremos mostrar los botones de nuevo podemos utilizar la siguiente función:

```
void gtk_file_selection_show_fileop_buttons(GtkFileSelection *filesel)
```

<u>Parámetro</u>	<u>Descripción</u>
<i>filesel</i>	El file selection widget al que se quieren mostrar los botones.

Ejemplo:

```
/* Asume que se ha declarado un GtkWidget *message_fsb */
/* Crea un file selection */
message_fsb = gtk_file_selection_new("Select new message file");
gtk_file_selection_hide_fileop_buttons(GTK_FILE_SELECTION(message_fsb));
```

La ventana de selección de archivos se posiciona en el directorio en el que esté el usuario en el momento de abrir la ventana. Sin embargo, se puede predeterminar el directorio en el cual se abra la ventana:

```
void gtk_file_selection_set_filename(GtkFileSelection *filesel
                                   const gchar *filename)
```

<u>Parámetro</u>	<u>Descripción</u>
<i>filesel</i>	El file selection widget al cual se quiere fijar el directorio
<i>filename</i>	El path del directorio que se quiere mostrar

Las funciones anteriores nos sirven para crear e inicializar nuestro file selection widget. Una vez que hemos hecho lo anterior, necesitamos agregar *signal handlers* a los botones de Ok y Cancel, de acuerdo a nuestras necesidades específicas. Normalmente un click en el botón de Ok nos regresa el archivo seleccionado por el usuario y esconde o destruye la ventana de selección de archivos, mientras que un click en el botón de Cancel únicamente esconde o destruye la ventana.

Para obtener el archivo seleccionado por el usuario utilizaremos la siguiente función:

```
gchar *gtk_file_selection_get_filename(GtkFileSelection *filesel)
```

<u>Parámetro</u>	<u>Descripción</u>
<i>filesel</i>	El file selection widget al cual se quiere obtener el archivo

Los botones de Ok y Cancel son del tipo GtkWidget, así que necesitaremos agregarles llamadas a funciones cuando el evento clicked se dé. El prototipo para esas funciones será:

```
void clicket_handler(GtkButton *button, gpointer user_data)
```

<u>Parámetro</u>	<u>Descripción</u>
<i>button</i>	El GtkWidget que fue presionado (pasado por GTK+)
<i>user_data</i>	La información pasada cuando el manejador de la señal es llamada (especificada por el usuario).

El siguiente ejemplo muestra la llamada a la función cuando el botón Ok fue presionado:

```
#include<gtk/gtk.h>
static gchar *filename;

void simple_ok_callback(GtkWidget *was_clicked, GtkWidget *file_sel)
{
    gtk_widget_hide(file_sel);
    filename=gtk_file_selection_get_filename(GTK_FILE_SELECTION(file_sel));
}
```

Para poder conectar las llamadas a las funciones cuando se hace click en alguno de los botones del widget GtkFileSelection, necesitamos conocer la estructura de este tipo de datos:

```
typedef struct _GtkFileSelection GtkFileSelection;
```

```
struct _GtkFileSelection{
    GtkWidget window;

    GtkWidget *dir_list;
    GtkWidget *file_list;
    GtkWidget *selection_entry;
    GtkWidget *selection_text;
    GtkWidget *main_vbox;
    GtkWidget *ok_button;
    GtkWidget *cancel_button;
    GtkWidget *help_button;
    GtkWidget *history_pulldown;
    GtkWidget *history_menu;
    GtkWidget *history_list;
    GtkWidget *file_dialog;
    GtkWidget *fileop_entry;
    gchar *fileop_file;
    gpointer *cmpl_state;

    GtkWidget *fileop_c_dir;
    GtkWidget *fileop_del_file;
    GtkWidget *fileop_ren_file;

    GtkWidget *button_area;
    GtkWidget *action_area;
}
```

El siguiente código ejemplifica lo anterior:

```
#include <gtk/gtk.h>
#include <stdio.h>
#define SPACING 10
#define PADDING 0

/* Callbacks */
void print_and_quit( GtkWidget *was_clicked, gpointer user_data )
{
}

gboolean delete_event_handler( GtkWidget *widget, GdkEvent *event, gpointer
user_data )
{
}

gboolean dialog_delete_handler( GtkWidget *widget, GdkEvent *event, gpointer
user_data )
```



```
{
}

void ok_callback( GtkWidget *was_clicked, GtkWidget *label )
{
    FILE *msg_fd;           /* File descriptor */
    gchar *msg_file;        /* File name */
    gchar c;                /* Character retrieved from file */
    GString *text;          /* Message text */
    GtkWidget *file_sel;    /* Parenting file selection widget */

    file_sel = gtk_widget_get_ancestor( was_clicked,
                                       GTK_TYPE_FILE_SELECTION );

    gtk_widget_hide( file_sel );
    /* Open file */
    msg_file=gtk_file_selection_get_filename(GTK_FILE_SELECTION(file_sel));
    msg_fd = fopen( msg_file, "r" );

    /* Create an empty GString for the message text */
    text = g_string_new("");

    /* Retrieve message */
    while ((c = fgetc(msg_fd)) != EOF )
        g_string_append_c( text, c );

    /* Update the label's text */
    gtk_label_set_text( GTK_LABEL(label), text->str );
}

int main( int argc, char *argv[] )
{
    GtkWidget *top_widget;
    GtkWidget *vbox, *menu_bar, *file_item;
    GtkWidget *file_menu, *change, *separator, *exit_item;
    GtkWidget *message, *message_fsb;

    /* 1. Initialize the environment */
    gtk_init( &argc, &argv );

    /* 2a. Create widgets */
    top_widget = gtk_window_new( GTK_WINDOW_TOPLEVEL );
    vbox = gtk_vbox_new( FALSE, SPACING );
    menu_bar = gtk_menu_bar_new( );

    /* Create the File item and its menu */
    file_item = gtk_menu_item_new_with_label( "File" );
    file_menu = gtk_menu_new();
    change = gtk_menu_item_new_with_label( "Change Message" );
    separator = gtk_menu_item_new( );
    exit_item = gtk_menu_item_new_with_label( "Exit" );

    /* Create the Label item */
    message = gtk_label_new( "Your message is:\n There is a Software Design "
                           "Meeting\n at 10:00 this morning.\n");

    /* Create the File Selection */
    message_fsb = gtk_file_selection_new( "Select new message file");
    gtk_file_selection_hide_fileop_buttons(GTK_FILE_SELECTION(message_fsb));
    gtk_file_selection_set_filename( GTK_FILE_SELECTION(message_fsb),
                                    "/home/messages/*" );
}
```

```
/* 2b. Set Attributes */
gtk_window_set_title( GTK_WINDOW( top_widget ), "Message" );

/* 3. Registering Callback routines */
g_signal_connect_swapped( G_OBJECT( change ), "activate",
                          G_CALLBACK( gtk_widget_show ),
                          GTK_OBJECT( message_fsb ) );
g_signal_connect( G_OBJECT( message_fsb ), "delete_event",
                  G_CALLBACK( dialog_delete_handler ), NULL );
g_signal_connect( G_OBJECT( GTK_FILE_SELECTION( message_fsb )->ok_button ),
                  "clicked", G_CALLBACK( ok_callback ), message );

g_signal_connect_swapped( G_OBJECT(
                          GTK_FILE_SELECTION( message_fsb )->cancel_button ),
                          "clicked",
                          G_CALLBACK( gtk_widget_hide ), GTK_OBJECT( message_fsb ) );

g_signal_connect( G_OBJECT( top_widget ), "delete_event",
                  G_CALLBACK( delete_event_handler ), NULL );

g_signal_connect( G_OBJECT( top_widget ), "destroy",
                  G_CALLBACK( print_and_quit ), NULL );

g_signal_connect( G_OBJECT( exit_item ), "activate",
                  G_CALLBACK( print_and_quit ), NULL );

/* 4. Defining the Instance Hierarchy */
gtk_container_add( GTK_CONTAINER( top_widget ), vbox );
gtk_box_pack_start( GTK_BOX( vbox ), menu_bar, FALSE, FALSE, PADDING );
gtk_box_pack_start( GTK_BOX( vbox ), message, FALSE, FALSE, PADDING );

/* Pack the Menu Bar */
gtk_menu_bar_append( GTK_MENU_BAR( menu_bar ), file_item );

/* Attach and pack the menus */
gtk_menu_item_set_submenu( GTK_MENU_ITEM( file_item ), file_menu );
gtk_menu_shell_append( GTK_MENU_SHELL( file_menu ), change );
gtk_menu_shell_append( GTK_MENU_SHELL( file_menu ), separator );
gtk_menu_shell_append( GTK_MENU_SHELL( file_menu ), exit_item );

/* 5. Showing the Widgets */
gtk_widget_show_all( top_widget );

/* 6. Processing Loop */
gtk_main();
return 0;
}
```

### 12.11. Font Selection Widget

Si queremos crear una ventana en la que podamos seleccionar el tipo de letra, usaremos lo siguiente:

```
GtkWidget *gtk_font_selection_dialog_new(const gchar *title)
```

<u>Parámetro</u>	<u>Descripción</u>
<i>title</i>	El título que queremos que aparezca en la ventana

Ejemplo:

```
#include <gtk/gtk.h>
#define HOMOGENEOUS_FALSE FALSE
#define PADDING 0

GtkWidget *append_menu_item( GtkWidget *menu, const gchar *text,
                             GtkSignalFunc callback, gpointer user_data);
gboolean dialog_delete_handler( GtkWidget *widget, GdkEvent *event,
                               gpointer user_data );
gboolean delete_event_handler( GtkWidget *widget, GdkEvent *event,
                               gpointer user_data );
void print_and_quit( GtkWidget *was_clicked, gpointer user_data );
void plot_values( GtkWidget *was_clicked, gpointer user_data );
GtkWidget *create_file_and_menu( GtkWidget *menu_bar );
GtkWidget *create_top_window( const gchar *title );
void ok_callback( GtkWidget *was_clicked, GtkWidget *fontsel );

int main( int argc, char *argv[] )
{
    GtkWidget *top_widget, *box, *pie_area;
    GtkWidget *menu_bar, *pie_item, *pie_menu ;
    GtkWidget *font_item, *fontsel_dlg;

    /* Initialize the environment */
    gtk_init( &argc, &argv );

    /* Create toplevel widget */
    top_widget = create_top_window( "Pie Chart" );

    /* Create managing vertical box widget */
    box = gtk_vbox_new( HOMOGENEOUS_FALSE, PADDING );
    gtk_container_add( GTK_CONTAINER( top_widget ), box );

    /* Create the menu bar widget */
    menu_bar = gtk_menu_bar_new( );
    gtk_box_pack_start_defaults( GTK_BOX( box ), menu_bar );

    /* Create the drawing area widget */
    pie_area = gtk_drawing_area_new( );
    gtk_drawing_area_size( GTK_DRAWING_AREA( pie_area ), 200, 250 );
    gtk_box_pack_start_defaults( GTK_BOX( box ), pie_area );

    /* Create the File item and its menu widget */
    create_file_and_menu( GTK_MENU_BAR( menu_bar ) );

    /* Create the Pie item and its menu widget */
    pie_item = gtk_menu_item_new_with_label( "Pie" );
    gtk_menu_bar_append( GTK_MENU_BAR( menu_bar ), pie_item );
    pie_menu = gtk_menu_new();
    gtk_menu_item_set_submenu( GTK_MENU_ITEM( pie_item ), pie_menu );

    /* Create Pie menu items */
    font_item = append_menu_item( GTK_MENU( pie_menu ), "Change Font",
                                NULL, NULL );
    append_menu_item( GTK_MENU( pie_menu ), NULL, NULL, NULL );
    append_menu_item( GTK_MENU( pie_menu ), "Draw pie chart", plot_values,
                    pie_area );

    /* Create the Font Selection Dialog, and attach the signal handler
       to bring it up */
    fontsel_dlg = gtk_font_selection_dialog_new( "Select Pie label's new font" );
    g_signal_connect_swapped( G_OBJECT( font_item ), "activate",
```

```
        G_CALLBACK( gtk_widget_show),
        GTK_OBJECT(fontsel_dlg) );
g_signal_connect( G_OBJECT( fontsel_dlg ), "delete_event",
        G_CALLBACK( dialog_delete_handler), NULL );

g_signal_connect(G_OBJECT(
        GTK_FONT_SELECTION_DIALOG(fontsel_dlg)->ok_button ),
        "clicked", G_CALLBACK( ok_callback ), fontsel_dlg );
g_signal_connect_swapped(G_OBJECT(
        GTK_FONT_SELECTION_DIALOG(fontsel_dlg)->cancel_button),
        "clicked", G_CALLBACK( gtk_widget_hide),
        GTK_OBJECT(fontsel_dlg) );

/* Show the widgets */
gtk_widget_show_all( top_widget );

/* Processing Loop */
gtk_main();
g_print( "Bye!\n");
return 0;
}
```

Una vez que aparece la ventana con los tipos de fuentes, el usuario debe seleccionar un tipo y hacer click en el botón de OK; a continuación se presente el ejemplo para conectar la señal:

```
g_signal_connect(G_OBJECT(
        GTK_OBJECT_SELECTION_DIALOG(fontsel_dlg)->ok_button),
        "clicked", G_CALLBACK( ok_callback ), fontsel_dlg);
```

para obtener el tipo de fuente seleccionada por el usuario se usa la siguiente función:

```
GdkFont *gtk_font_selection_dialog_get_font(GtkFontSelectionDialog *fsd)
```

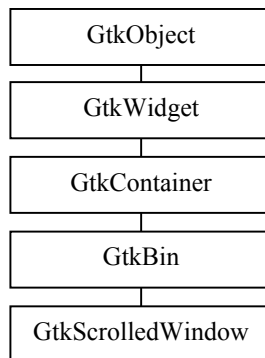
<u>Parámetro</u>	<u>Descripción</u>
<i>fsd</i>	La ventana con los tipos de fuentes que se quiere seleccionar

El *handler* quedaría de la siguiente manera:

```
GdkFont *label_font = NULL;
void ok_callback(GtkWidget *was_clicked, GtkWidget *font_dlg)
{
    label_font = gtk_font_selection_dialog_get_font(
        GTK_FONT_SELECTION_DIALOG(font_dlg) );
    gtk_widget_hide(font_dlg);
}
```

## 12.12. Barras de desplazamiento

Para que podamos utilizar ventanas con barras de desplazamiento, es necesario utilizar un widget del tipo `GtkScrolledWindow`



**Figura 18.- Árbol de clases de la clase GtkScrolledWindos**

La siguiente función nos permite crear una interface gráfica con una ventana desdesplazamiento:

```
GtkWidget *gtk_scrolled_window_new(GtkAdjustment *hadjustent,
                                   GtkAdjustment *vadjustment)
```

<i>Parámetro</i>	<i>Descripción</i>
<i>hadjustent</i>	El GtkAdjustment utilizado cuando se mueve el hijo en dirección horizontal. Especificar NULL para que la ventana cree su propio ajuste.
<i>vadjustent</i>	El GtkAdjustment utilizado cuando se mueve el hijo en dirección vertical. Especificar NULL para que la ventana cree su propio ajuste.

Debido a que un label widget no hereda las capacidades de scroll, necesitaremos utilizar la siguiente función:

```
void gtk_scrolled_window_add_with_viewport(
    GtkScrolledWindow *scrolled_window, GtkWidget *child)
```

<i>Parámetro</i>	<i>Descripción</i>
<i>scrolled_window</i>	El GtkScrolledWindow widget padre.
<i>child</i>	El widget hijo que necesita los scrollbars.

Ejemplo:

```
/* Asume que se ha declarado un GtkWidget *scroll */
scroll = gtk_scolled_window_new(NULL, NULL);
gtk_widget_set_usize(scroll, 200, 100);
gtk_scrolled_window_add_with_viewport(GTK_SCROLLLED_WINDOW(scroll),
                                     message);
```

Para especificar si queremos que las barras de desplazamiento estén presentes siempre o sólo cuando se necesitan podemos utilizar las siguientes funciones:

```
void gtk_scrolled_window_set_policy(GtkScrolledWindow *scrolled_window,
                                   GtkPolicy *hscrollbar_policy,
                                   GtkPolicy *vscrollbar_policy)
```

<i>Parámetro</i>	<i>Descripción</i>
<i>scrolled_window</i>	El GtkScrolledWindow al que se quiere cambiar la política.
<i>hscrollbar_policy</i>	La política para la barra horizontal. Las opciones válidas son: GTK_POLICY_ALWAYS (por default), GTK_POLICY_AUTOMATIC, (sólo aparecen cuando se necesitan) y GTK_POLICY_NEVER.

*vscrollbar\_policy* La política para la barra vertical. Las opciones válidas son: GTK\_POLICY\_ALWAYS (por default), GTK\_POLICY\_AUTOMATIC, (sólo aparecen cuando se necesitan) y GTK\_POLICY\_NEVER.

Para cambiar el lugar en donde queremos que aparezcan las barras de desplazamiento utilizaremos la siguiente función:

```
void gtk_scrolled_window_set_placement(GtkScrolledWindow *scrolled_window,  
                                     GtkCornerType *window_placement)
```

<u>Parámetro</u>	<u>Descripción</u>
<i>scrolled_window</i>	El GtkScrolledWindow al que se quiere cambiar la ubicación de las barras de desplazamiento.
<i>window_placement</i>	La ubicación de las barras de desplazamiento. El valor por default es GTK_CORNER_TOP_LEFT. Las otras opciones son GTK_CORNER_TOP_RIGHT, GTK_CORNER_BOTTOM_LEFT, GTK_CORNER_BOTTOM_RIGHT.

Si se quieren utilizar las barras de desplazamiento con un GtkText, se deberá compartir el objeto del ajuste del texto, de la misma manera en que la barra de desplazamiento comparte el valor del ajuste de la escala, por lo tanto, se tendrá que hacer algo como lo que se muestra a continuación:

```
GtkWidget *text, *vscroll, *hscroll;  
  
text = gtk_text_new(NULL, NULL);  
  
vscroll = gtk_vscrollbar_new(GTK_TEXT(text)->vadj);  
hscroll = gtk_hscrollbar_new(GTK_TEXT(text)->hadj);
```

y posteriormente agregar tanto las variables *text*, *vscroll* y *hscroll* a una caja (vertical u horizontal).

Otra manera de agregar barras de desplazamiento a un GtkText, crear un GtkScrolledWindow y, por medio de un *gtk\_container\_add*, agregarle el GtkText.

### 12.13. GtkScale

El widget GtkScale permite crear un widget para desplegar una escala, ya sea vertical u horizontal. Para crearlo, se debe utilizar cualquiera de las siguientes funciones:

```
GtkWidget *gtk_vscale_new(GtkAdjustment *adjustment);  
  
GtkWidget *gtk_hscale_new(GtkAdjustment *adjustment);
```

<u>Parámetro</u>	<u>Descripción</u>
<i>adjustment</i>	El objeto de GTK+ que controla el rango de la escala. (0 a 100%, por ejemplo). Puede fijarse como NULL.

Los ajustes permiten fijar los límites superiores e inferiores representados por un scrollbar. Un objeto de tipo *GtkAdjustment* puede crearse con la función:

```
GtkObject *gtk_adjustment_new(gfloat value, gfloat lower, gfloat upper,  
                              gfloat step_increment, gfloat page_increment,  
                              gfloat page_size);
```

<u>Parámetro</u>	<u>Descripción</u>
<i>value</i>	El valor inicial del ajuste
<i>lower</i>	El valor mínimo

<i>upper</i>	El valor máximo
<i>step_increment</i>	El tamaño del incremento al que puede moverse el ajuste
<i>page_increment</i>	La cantidad de incremento por página
<i>page_size</i>	El tamaño de la página

También se puede controlar el despliegue del valor representado en la escala, con la función:

```
void gtk_scale_set_digits(GtkScale *scale, gint digits);
```

<u>Parámetro</u>	<u>Descripción</u>
<i>scale</i>	El GtkVScale o GtkHScale del que se quiere cambiar los valores.
<i>digits</i>	El número de dígitos a la derecha del punto decimal que se van a mostrar.

Si no se quiere que aparezca una cadena mostrando el valor actual de la escala, se puede utilizar la función:

```
void gtk_scale_set_draw_value(GtkScale *scale, gboolean draw_value);
```

<u>Parámetro</u>	<u>Descripción</u>
<i>scale</i>	El GtkVScale o GtkHScale del que se quiere cambiar los valores.
<i>draw_value</i>	TRUE si se quiere que aparezca el valor, FALSE en caso contrario.

## 12.14. Frames

Un GtkWidget es un widget de la familia bin que permite desplegar los widgets de una manera muy amigable, ya que tiene la opción de desplegar un borde decorativo, así como una etiqueta opcional.

Un GtkWidget es creado con la función:

```
GtkWidget *gtk_frame_new(const gchar *label);
```

<u>Parámetro</u>	<u>Descripción</u>
<i>label</i>	La etiqueta que se agregará al Frame

La posición de la etiqueta del frame puede determinarse con la función:

```
void gtk_frame_set_label_align(GtkFrame *frame, gfloat xalign, gfloat yalign);
```

<u>Parámetro</u>	<u>Descripción</u>
<i>frame</i>	El frame cuya etiqueta se quiere alinear
<i>xalign</i>	La alineación de la etiqueta (valor entre 0.0 y 1.0)
<i>yalign</i>	No implementado todavía. Se deberá utilizar 0.0.

También se puede determinar el tipo de sombra que puede acompañar al frame. Lo anterior se podrá realizar mandando llamar la función:

```
void gtk_frame_set_shadow_type(GtkFrame *frame, GtkShadowType type);
```

<u>Parámetro</u>	<u>Descripción</u>
<i>frame</i>	El frame cuya sombra se quiere fijar
<i>type</i>	El tipo de sombra. Las opciones válidas son: GTK_SHADOW_NONE, GTK_SHADOW_IN, GTK_SHADOW_OUT, GTK_SHADOW_ETCHED_IN y GTK_SHADOW_ETCHED_OUT.

Para agregar widgets a un frame, se podrá utilizar la función `gtk_container_add()`.

### 12.15. Calendario

El widget GtkCalendar permite desplegar en pantalla un calendario mensual que el usuario puede usar para seleccionar alguna fecha en particular.

Para crearlo, se deberá utilizar la función:

```
GtkWidget *gtk_calendar_new(void);
```

La fecha seleccionada en el calendario puede obtenerse con la función:

```
void gtk_calendar_get_date(GtkCalendar *calendar, guint *year, guint *month,
                           guint *day);
```

<i>Parámetro</i>	<i>Descripción</i>
<i>calendar</i>	El widget calendario
<i>year, month, day</i>	Las variables en donde se almacenará la fecha seleccionada

Las señales emitidas por este widget, así como los correspondientes prototipos de las funciones que se deben implementar son las siguientes:

```
"month-changed"    void user_function (GtkCalendar *calendar, gpointer user_data);
"day-selected"     void user_function (GtkCalendar *calendar, gpointer user_data);
"day-selected-double-click" void user_function (GtkCalendar *calendar,
                                                gpointer user_data);
"prev-month"       void user_function (GtkCalendar *calendar, gpointer user_data);
"next-month"       void user_function (GtkCalendar *calendar, gpointer user_data);
"prev-year"        void user_function (GtkCalendar *calendar, gpointer user_data);
"next-year"        void user_function (GtkCalendar *calendar, gpointer user_data);
```

### 12.16. Simplificación de las tareas

Cuando estamos haciendo un programa en GTK+, muchas veces necesitamos repetir la misma tarea muchas veces, lo que hará nuestra función principal muy grande. Podemos crear funciones que hagan cierta acción que se repita muchas veces, de esta manera ahorraremos líneas de código:

```
GtkWidget *append_menu_item( GtkWidget *menu, const gchar *text,
                             GtkSignalFunc callback, gpointer user_data)
{
    GtkWidget *menu_item;
    if ( text )
        menu_item = gtk_menu_item_new_with_label( text );
    else
        menu_item = gtk_menu_item_new();

    gtk_menu_shell_append(GTK_MENU_SHELL(menu), menu_item );
    gtk_widget_show( menu_item );
    if (callback)
        g_signal_connect( G_OBJECT(menu_item), "activate",
                          G_CALLBACK( callback ), user_data );
    return menu_item;
}
```

así, podremos llamar la función anterior de la siguiente manera:

```
change = append_menu_item( GTK_MENU(file_menu), "Change Message",
```



```

                                NULL, NULL );
append_menu_item( GTK_MENU(file_menu), NULL, NULL, NULL );
append_menu_item( GTK_MENU(file_menu), "Exit", print_and_quit, NULL );

```

## 12.17. Compartiendo la información

Muchas de las funciones que se implementarán en los programas de GTK+ llegan a necesitar una o más que la mayor parte de las veces han sido declaradas en alguna otra función del programa. La primera solución que se viene a la mente es declarar esas variables que son utilizadas en más de una función del programa como globales; sin embargo, el uso de variables globales nunca es recomendable ya que puede traer muchos conflictos cuando se está trabajando con muchos módulos.

Una segunda solución es crear un tipo de dato (es decir, una estructura) que contenga todas aquellas variables que se van a estar modificando en más de una función del programa. Posteriormente, en la función principal, se creará una variable de ese nuevo tipo de datos y se mandará la dirección de dicha variable a todas las llamadas de las funciones (*callbacks*) que necesiten utilizar las variables que están dentro de la estructura definida. En la función solamente se hará el cast correspondiente del tipo *gpointer* al tipo que se creó pudiendo de esta manera acceder a la variable deseada.

Otra solución sería agregar la información a los mismos widgets. Cada objeto en la clase de GTK+ puede guardar información por medio de la utilización de la función `gtk_object_set_data()`; de esta manera el *callback* que reciba la información deberá tener el ID del widget que tiene la información y el nombre que se asignó a la información que se agregó al widget y podrá recuperar la información por medio de la función `gtk_object_get_data()`. Los prototipos de las funciones anteriores son los siguientes:

```
void gtk_object_set_data(GtkObject *object, const gchar *key, gpointer data);
```

<i>Parámetro</i>	<i>Descripción</i>
<i>object</i>	El objeto al que se está agregando la información
<i>key</i>	La cadena con la que se identifica la información
<i>data</i>	La información a agregar

```
gpointer gtk_object_get_data(GtkObject *object, const gchar *key;
```

<i>Parámetro</i>	<i>Descripción</i>
<i>object</i>	El objeto al que se está agregando la información
<i>key</i>	La cadena con la que se identifica la información a recibir

## 12.18. Diálogos

En algunas plataformas de programación existe una gran cantidad de variaciones de diálogos, esto hace que su programación se vuelva tediosa.

GTK+ adoptó una solución más simple y flexible. Un diálogo es una ventana con un formato específico con los controles que contiene. Provee una función que nos permite hacer una ventan modal, esto significa que mientras la ventana se muestra el usuario es incapaz de trabajar con cualquier otra ventana en la misma aplicación. Esta es la funcionalidad que podemos esperar de un diálogo. En realidad esto no es tan cierto y lo veremos posteriormente.

Se provee un nuevo widget. `GtkDialog`, que contiene un par de cajas (contenedores) para hacernos más sencilla la programación. Además de las cajas, `GtkDialog` no tiene otros elementos de interfase. Para poner un botón de OK sólo se crea y se pone en el área del botón.

Para tareas más complejas, GTK+ provee diálogos preconstruidos para ese tipo de trabajos, por ejemplo la selección de archivos o de colores. Estos son únicamente ventanas con widgets insertados.

Por ejemplo, supongamos que queremos hacer un programa que contenga un menú "Acerca de..." y que al seleccionar esa opción aparezca una ventana con la información del programa.

Para ello, necesitaremos crear una ventana de *pop-up*, la cual aparecerá únicamente en el momento en que se seleccione la opción mencionada. Algunas clases de widgets, como `GtkDialog` están designadas con características de *pop-up*. El siguiente es el prototipo de la función para crear un widget de este tipo:

```
GtkWidget *gtk_dialog_new (void)
```

Si inmediatamente después de crear un widget de este tipo lo mostráramos en pantalla, lo que veríamos sería una ventana en blanco. Para crear algo con más sentido, lo que necesitamos sería agregar widgets a dicha ventana. Para hacer esto, necesitamos conocer más acerca del tipo `GtkDialog`. A continuación se describe la estructura que lo conforma:

```
typedef struct _GtkDialog GtkDialog;
struct _GtkDialog{
    GtkWidget window;
    GtkWidget *vbox;
    GtkWidget *action_area;
};
```

`window` se encuentra en la parte más alta de la jerarquía. `vbox` es del tipo `GtkVBox`, que se encarga de un widget del tipo `GtkSeparator` y de `action_area` (que es del tipo `GtkHBox`). El widget `separator` no forma parte de la estructura `GtkDialog`, porque nosotros como programadores no necesitamos conocer su ID.

Lo que tenemos que hacer es agregar widgets de información (como un `GtkLabel`) a `vbox` y `action items` (como `GtkButton`) a `action_area`. A continuación el código para ejemplificar lo anterior:

```
/* Asumimos que previamente se han declarado las variables del tipo GtkWidget:
 *about_dialog, *about_label, *about_ok */
about_dialog = gtk_dialog_new();
about_label = gtk_label_new("This is a simple sample program.\n"
                           "No acepta ninguna información, \n"
                           "pero estará listo en un futuro."
                           "Autor: Yo\n Versión 0.0.1\n");
about_ok = gtk_button_new_with_label("Ok");

/* Ponemos la etiqueta en el vbox, y el botón en action_area */
gtk_box_pack_start(GTK_BOX(GTK_DIALOG(about_dialog)->vbox), about_label),
                  FALSE, FALSE, NO_PADDING);
gtk_box_pack_start(GTK_BOX(GTK_DIALOG(about_dialog)->action_area),
                  about_ok, FALSE, FALSE, NO_PADDING);
gtk_widget_show(about_label);
gtk_widget_show(about_ok);

g_signal_connect_swapped(G_OBJECT(about_help), "activate",
                        G_CALLBACK( gtk_widget_show),
                        GTK_OBJECT(about_dialog));
```

En la última línea, estamos conectando la señal para que cuando se seleccione la opción "About my program..." aparezca la ventana de diálogo. Para ello estamos usando la función `g_signal_connect_swapped()`. Podemos utilizar la función anterior de dos maneras, la primera es crear una llamada a una función que llame a su vez a la función `gtk_widget_show()`, o podemos invocar directamente dicha función conectándola en la función `g_signal_connect_swapped()`, tal y como lo hicimos en nuestro ejemplo. A continuación se muestra el prototipo de dicha función:

```
guint g_signal_connect_swapped(GtkObject *object, const gchar *name
                                GCallback func, gpointer *slot_object)
```

<u>Parámetro</u>	<u>Descripción</u>
<i>object</i>	El GtkWidget (o su descendiente) al cual estamos adjuntando la llamada a la función.
<i>name</i>	La señal que estamos conectando a <i>func</i> .
<i>func</i>	La función que queremos que se ejecute.
<i>slot_object</i>	El parámetro a pasar a <i>func</i> cuando es llamada.

Ahora bien, ¿qué pasa si nosotros compilamos y corremos el programa del ejemplo anterior y hacemos click en el botón “Ok” de la ventana de diálogo? ¡Nada! Debido a que nosotros mostramos la ventana de diálogo cuando el usuario selecciona la opción “About my program...”, tenemos también que hacer que desaparezca cuando el usuario hace click en el botón “Ok”. Debido a que “Ok” es un GtkWidget, lo que tenemos que hacer es simplemente agregar una llamada a una función cuando la señal “clicked” en el GtkWidget “Ok” es detectada, mandando llamar a la función `gtk_widget_hide()`;

```
void gtk_widget_hide(GtkWidget *widget)
```

Para hacer la conexión de la señal tendríamos que hacer lo siguiente:

```
g_signal_connect_swapped(G_OBJECT( about_ok ), "clicked",
                        G_CALLBACK( gtk_widget_hide ), G_OBJECT( about_dialog ));
```

Con lo anterior, lo que estamos haciendo es que cuando el usuario cierra la ventana de diálogo, además de ocultarla la destruye, por lo tanto la siguiente vez que se quiera mostrar la ventana, se tendrá que volver a construir para poderse mostrar. Esto hará un programa poco eficiente si la ventana se necesita mostrar frecuentemente durante la ejecución del programa. Si nosotros queremos únicamente ocultar la ventana sin destruirla, tenemos que implementar un `delete_event` handler.

```
gboolean dialog_delete_handler(GtkWidget *widget, GdkEvent *event,
                               gpointer user_data)
{
    gtk_widget_hide(widget);
    return (TRUE) /* Prevent destruction */
}
```

y aquí sería la correspondiente llamada a la función:

```
g_signal_connect( G_OBJECT( about_dialog ), "delete_event",
                  G_CALLBACK( dialog_delete_handler), NULL );
```

Ver códigos `Filedlg.c` y `colordlg.c`

### **12.20.1.      Dialogos de Acerca de**

GTK+ provee de una manera fácil y rápida para crear ventanas de diálogo para proveer información acerca del programa:

```
GtkWidget *gtk_about_dialog_new(void);
```

Una vez creado, se le podrán agregar las diferentes características con las siguientes funciones:

```
gtk_about_dialog_set_name(GtkAboutDialog *about, gchar *nombre);

gtk_about_dialog_set_comments(GtkAboutDialog *about, gchar *comentarios);

gtk_about_dialog_set_authors(GtkAboutDialog *about, gchar **autores);
```

```
gtk_about_dialog_set_copyright(GtkAboutDialog *about, gchar *copyright);  
gtk_about_dialog_set_license(GtkAboutDialog *about, gchar *license);  
gtk_about_dialog_set_website(GtkAboutDialog *about, gchar *webPage);  
gtk_about_dialog_set_artists(GtkAboutDialog *about, gchar **artists);
```

Ejemplo:

```
GtkWidget *aboutDialog, *dialogBox, *botBox, *aboutLabel, *aboutOk;  
  
static const char *author[] = {"Andres Tortolero"};  
static const char *artist[] = {"Andres Tortolero"};  
  
aboutDialog = gtk_about_dialog_new();  
  
gtk_about_dialog_set_name(GTK_ABOUT_DIALOG(aboutDialog), "R u T e R o M e T r O");  
gtk_about_dialog_set_comments(GTK_ABOUT_DIALOG(aboutDialog), "Comentarios");  
gtk_about_dialog_set_authors(GTK_ABOUT_DIALOG(aboutDialog), author);  
gtk_about_dialog_set_copyright(GTK_ABOUT_DIALOG(aboutDialog), "Copyright text");  
gtk_about_dialog_set_license(GTK_ABOUT_DIALOG(aboutDialog), "License text");  
gtk_about_dialog_set_website(GTK_ABOUT_DIALOG(aboutDialog),  
                             "http://www.iec.uia.mx/acad/atortole");  
gtk_about_dialog_set_artists(GTK_ABOUT_DIALOG(aboutDialog), artist);  
gtk_widget_show_all(aboutDialog);
```

## 12.19. Diálogos y GtkWidget

La creación de una ventana se hace con:

```
window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
```

`GTK_WINDOW_TOPLEVEL` le dice a GTK que cree una ventana estandar en el primer nivel. La ventana se crea con un icono de cerrado y usualmente un icono para modificar el tamaño de la ventana.

## 12.20. Modalidad

Como ya se platicó anteriormente, la modalidad evita que el usuario pueda interactuar con otras ventanas diferentes a la de diálogo mientras esta última se encuentre presente en pantalla.

Nota: La ventana activa se dice que está en foco. Algunas veces sucederá cuando hagamos click en la ventana o con sólo posicionar el ratón encima de la ventana.

Este comportamiento no es el valor por omisión del diálogo y se requiere utilizar la siguiente función.

```
void gtk_window_set_modal(GtkWindow *window, gboolean modal);
```

El primer parámetro es el apuntador a la ventana de diálogo. El segundo parámetro es `TRUE` o `FALSE`.

### 12.21. Removiendo Ventanas

Cuando trabajamos con diálogos es importantne saber cómo podemos removerlos del área de trabajo. Las opciones que tenemos utilizan las siguientes funciones.

La primera es simplemente destruir la ventana:

```
void gtk_widget_destroy(GtkWidget *widget);
```

Se pasa el apuntador al widget y esto hace que la ventana desaparezca. Cuando se llama a esta función, la ventana envía la señal “destroy”, esta señal puede ser atrapada para que nosotros programemos una mayor funcionalidad a la ventana cuando es destruida.

También podemos ocultarla:

```
void gtk_widget_hide(GtkWidget *widget);
```

Esta función es la opuesa a `gtk_widget_show`. No se muestra pero aún está disponible en memoria para su uso. Siempre que mantengamos un apuntado al diálogo, claro está.

Al llamar a esta función se obtiene la señal “hide”, permitiéndonos trabajar con un manejador de esta señal, con lo que la ventana puede hacer alo útil mientras es ocultada.

Ver el ejercicio `dialog1.c`

En este programa se muestra una ventan y un botón, una vez que se hace click sobre el botón se muestra una ventana de diálogo. Si separamos el código para la ventana de diálogo a otro archivo fuente podemos tener acceso instantaneo a un diálogo en cualquier pate de la aplicación, haciendo una llamada a la función.

NOTA: Podemos tratar de cerrar la ventana inicial y veremos que sí es posible, siendo que la ventana de diálogo tiene su modalidad como verdadera.

Veamos el texto desplegado en la terminal, se hace la llamada a la función para creación del diálogo y se muestra el mensaje de regreso. El que un diálogo sea desplegado no significa que el código se detendrá y esperará hasta que el diálogo se cierre para continuar.

Si no queremos que el usuario interactúe con otras ventanas y que lo haga únicamente con la ventana de diálogo será necesario forzar al código que llama a la ventana del diálogo, para que espere hasta que el diálogo se cierre. Para hacer esto tenemos que examinar la función `gtk_main`.

### 12.22. `gtk_main`

Cuando se llama a `gtk_main` GTK produce un ciclo para esperar que algo ocurra. Ese algo es generalmente una señal emitida quizás como resultado de un evento del sistema operativo o una acción del usuario. En ese punto el ciclo de GTK transfiere el control al manejador de la señal y el ciclo de main se detiene.

Ahora debe ser más claro ver porque la aplicación no se detiene, realiza las acciones que se le indicaron y regresa a la función manejadora de la señal.

Hay dos soluciones, una elegante y la otra no, de momento cubriremos únicamente la mejor solución.

Las llamadas a `gtk_main` y `gtk_main_quit` pueden ser anidadas. Si llamamos a `gtk_main` tres veces y a `gtk_main_quit` dos, la aplicación continuará corriendo, ya que aún existe una instancia de `gtk_main` en ejecución. Aún más, si llamamos a `gtk_main` y entonces un manejador de eventos llama nuevamente a

`gtk_main`, la segunda llamada se ejecuta mientras que la primera (la que disparó al manejador del evento) se detiene.

Cuando el usuario teclee el botón OK en el diálogo, se llama a la función `gtk_widget_destroy`, esto quiere decir que se genera una señal “destroy”, que puede ser capturada, esto se hace ya en la función `CloseTheApp` que llama a `gtk_main`. Así solo tenemos que poner algunas líneas más en nuestra aplicación.

Modifiquemos `ShowMessage`

Después de la primer `g_signal_connect`

```
g_signal_connect(G_OBJECT(dialogwidget), "destroy", G_CALLBACK( CloseTheApp),
                dialogwindow);
```

Después de `gtk_widget_show_all`

```
gtk_main();
```

Recompilemos y probemos.

NOTA: Siguen existiendo algunos problemas, pues como existen dos instancias de `gtk_main` corriendo, es posible aún cerrar las ventanas.

### 12.23. Tool tips

Nos proveen una forma para dar al usuario una guía en el uso de una aplicación. Una vez que éstos se encuentran enlazados con un widget, todo lo que el usuario debe hacer es posicionar el ratón sobre el widget y un pequeño texto aparecerá en pantalla. También podemos fijar una descripción completa del widget para usarlo en un sistema de ayuda. Cuando se seleccione la opción de ayuda podemos extraer este texto y lo desplegaremos en un diálogo.

La creación de una herramienta de ayuda se compone de dos pasos.

El primer paso es crear un objeto `GtkTooltips` (en plural)

```
GtkTooltips *gtk_tooltips_new(void);
```

Nota: El tipo de retorno es el objeto en particular y no un `GtkWidget`. Una herramienta de ayuda no es un widget, se deriva de `GtkData`, un objeto para manipulación de datos.

Una vez creado el objeto se debe fijar el texto a un widget.

```
void gtk_tooltips_set_tip( GtkTooltips *tooltips, GtkWidget *widget,
                          const gchar *tip_text, const gchar *tip_private)
```

<i>Parámetro</i>	<i>Descripción</i>
<i>tooltips</i>	El objeto <code>GtkTooltips</code>
<i>widget</i>	El widget al que se le va a agregar el tooltip
<i>tip_text</i>	El texto que se quiere mostrar
<i>tip_private</i>	El texto adicional que se quiera mostrar. Por ahora usaremos NULL.

Ejemplo:

```
/* Asumimos que tenemos declarada una variable GtkTooltips *tooltips */
```

```
/* Crear Tooltips */
gtk_tooltips_new();

gtk_tooltips_set_tip(tooltips, file_item, "Left click to bring up", NULL);
gtk_tooltips_set_tip(tooltips, exit_item, "Click to exit", NULL);
```

Ver ejercicio `tooltips.c`

Ahora de qué manera podemos ver la descripción extendida de texto de la herramienta de ayuda. Eso lo podemos hacer a través de la función:

```
GtkTooltipsData *gtk_tooltips_data_get(GtkWidget *widget);
```

El tipo de retorno es un apuntador a una estructura llamada `GtkTooltipsData`

```
struct GtkTooltipsData{
    GtkTooltips *tooltips;
    GtkWidget *widget;
    gchar *tip_text;
    gchar *tip_private;
    GdkFont *font;
    gint width;
    GList *row;
}
```

Pongamos la siguiente función para manejar el evento de "click" del botón:

```
void ButtonClicked(GtkWidget *button, gpointer)
{
    GtkTooltipsdata *tipdata;

    tipdata = gtk_tooltips_data_get(button);
    g_print ("The tip itself was: %s \n", tipdata->tip_text);
    g_print("The private text was %s \n",tipdata->tip_private);
}
```

Tenemos que conectar la señal con la función manejadora

Después de `gtk_container_add`

```
g_signal_connect(G_OBJECT(button), "clicked",
                 G_CALLBACK( ButtonClicked), NULL);
```

NOTA: Un solo objeto `GtkTooltips` es responsable de administrar todos los tips en la aplicación. Esto se hace por eficiencia.

Podemos fijar un retraso para que aparezca el mensaje de ayuda con la función

```
void gtk_tooltips_set_delay( GtkTooltips *tooltips, guint delay);
```

También podemos habilitar o deshabilitar la ayuda

```
void gtk_tooltips_enable (GtkTooltips *tooltips);
```

```
void gtk_tooltips_disable(GtkTooltips *tooltips);
```

ver ejercicio `bulktips.c`

## 12.24. Barras de Herramientas

Las barras de herramientas son usadas generalmente para agrupar un número determinado de widgets de tal manera que la interfase gráfica sea más amigable y fácil de manejar. Típicamente, una barra de herramientas se compone de una serie de botones, etiquetas y tooltips, sin embargo, cualquier widget puede ser agragado a ésta. Los elementos pueden ser organizados de manera horizontal o vertical y los botones pueden desplegarse con íconos, etiquetas o ambos.

Para poder crear una barra de herramientas, se deberá utilizar la siguiente función:

```
GtkWidget *gtk_toolbar_new( void );
```

Para poder agregar elementos a una barra de herraminentas, se podrá utilizar alguna 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,
                                   GtkSignalFunc callback,
                                   gpointer user_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 );

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 );
```

Las siguientes funciones permiten agregar espacios a una barra de herramientas:

```
void gtk_toolbar_append_space( GtkToolbar *toolbar );
void gtk_toolbar_prepend_space( GtkToolbar *toolbar );
void gtk_toolbar_insert_space( GtkToolbar *toolbar, gint position );
```

Para cambiar la orientación y estilo de una barra de herramientas, se pueden utilizar 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 );
```

En ,donde `orientation` puede ser: `GTK_ORIENTATION_HORIZONTAL` o `GTK_ORIENTATION_VERTICAL`. `style` es utilizado para fijar la apariencia de la barra de estado y puede tener alguno de los s valores: `GTK_TOOLBAR_ICONS`, `GTK_TOOLBAR_TEXT`, `GTK_TOOLBAR_BOTH`.

## 12.25. Barras de Estado

Generalmente es una barra que se muestra en la parte inferior de una ventana y provee información de retroalimentación al usuario.

La barra de estados, `GtkStatusbar`, usa un sistema de stack para desplegar los mensajes.



Para crear una barra de estados se utiliza la función:

```
GtkWidget *gtk_statusbar_new();
```

Para hacer un push y poner mensaje en la pila es necesario tener un valor de identificación.

```
guint gtk_statusbar_get_context_id (GtkStatusbar *statusbar,
                                   const gchar *context_description);
```

Una vez teniendo el número de identificación podemos hacer push y pop de los mensajes de la barra de estado.

```
guint gtk_statusbar_push(GtkStatusbar *statusbar, guint context_id,
                        const gchar *text);
void gtk_statusbar_pop(GtkStatusbar *statusbar, guint context_id);
```

ver statusbar.c

### 13. Añadiendo gráficos a la aplicación

Para poder dibujar líneas y gráficos en general dentro de una aplicación de GTK+, una de las primeras cosas que tenemos que hacer es determinar las características de los gráficos, tal como color de la línea y estilo (*Graphic Context, GC*).

```
void line_plot(GtkMenuItem *was_clicked, GtkWidget *where_to_draw)
{
    GdkGC *plot_gc;
    /* Obtiene el GC negro */
    plot_gc = GTK_WIDGET(where_to_draw)->style->black_gc;
    /* ..... */
}
```

#### 13.1. Puntos

Para dibujar un punto en una región de trabajo, usaremos la siguiente función:

```
void gdk_draw_point(GdkDrawable *drawable, GdkGC *gc, gint x, gint y)
```

<u>Parámetro</u>	<u>Descripción</u>
<i>drawable</i>	En donde se quiere dibujar el punto
<i>gc</i>	Las características del gráfico ( <i>Graphic context</i> )
<i>x, y</i>	El punto a dibujar. (Un punto es un simple pixel en tamaño.)

Ejemplo:

```
#define X 30
#define Y 45

void draw_point(GtkMenuItem *was_clicked, GtkWidget *where_to_draw)
{
    GdkGC *plot_gc;

    /* Obtiene el GC negro */
    plot_gc = GTK_WIDGET(where_to_draw)->style->black_gc;

    /* Dibuja */
    gdk_draw_point(GTK_WIDGET(where_to_draw)->window, plot_gc, X, Y);
}
```

```
}
```

## 13.2. Líneas

Para dibujar una línea, utilizaremos la siguiente función:

```
void gdk_draw_line(GdkDrawable *drawable, GdkGC *gc, gint x1, gint y1,  
                  gint x2, gint y2)
```

Parámetro	Descripción
<i>drawable</i>	En donde se quiere dibujar el punto
<i>gc</i>	Las características del gráfico ( <i>Graphic context</i> )
<i>x1, y1</i>	El inicio de la línea
<i>x2, y2</i>	El final de la línea.

Ejemplo:

```
#define X_AXIS 180  
#define Y_AXIS 175  
#define X_PAD 20  
#define Y_PAD 10  
#define NUM_PTS 10  
#define X 0  
#define Y 1  
  
void line_plot(GtkMenuItem *was_clicked, GtkWidget *where_to_draw)  
{  
    static gint to_plot[NUM_PTS][2] = { {0,0}, {15,20}, {30,80},  
                                          {45,80}, {60,25}, {75,90}, {90,120},  
                                          {105,170}, {130,100}, {135,0} };  
  
    GdkGC *plot_gc;  
    int pts;  
  
    /* Otbiene el gc negro */  
    plot_gc = GTK_WIDGET(where_to_draw)->style->black_gc;  
  
    /* Dibuja los ejes */  
    gdk_draw_line(GTK_WIDGET(where_to_draw)->window, plot_gc,  
                  X_PAD, Y_AXIS + Y_PAD, X_PAD, Y_PAD);  
    gdk_draw_line(GTK_WIDGET(where_to_draw)->window, plot_gc,  
                  X_PAD, Y_AXIS + Y_PAD, X_PAD + X_AXIS,  
                  Y_AXIS + Y_PAD);  
  
    /* Dibuja */  
    for(pts = 1; pts < NUM_PTS; pts++){  
        gdk_draw_line(GTK_WIDGET(where_to_draw)->window, plot_gc,  
                      to_plot[pts-1][X] + X_PAD, Y_AXIS + Y_PAD -  
                      to_plot[pts-1][Y], to_plot[pts][X] + X_PAD, Y_AXIS +  
                      +Y_PAD - to_plot[pts][Y] );  
    }  
}
```

Las líneas conectadas se pueden dibujar con la llamada a una sola función, constuyendo un arreglo de `GdkPoints` y usando la función `gdk_draw_lines()`:

```
struct GdkPoint{  
    gint16 x;  
    gint16 y;  
};
```

```
void gdk_draw_lines(GdkDrawable *drawable, GdkGC *gc, GdkPoint *points, gint
npoints)
```

<i>Parámetro</i>	<i>Descripción</i>
<i>drawable</i>	En donde se quieren dibujar las líneas
<i>gc</i>	Las características del gráfico ( <i>Graphic context</i> )
<i>points</i>	El arreglo de <i>GdkPoints</i> a dibujar
<i>npoints</i>	El número de líneas a dibujar.

Ejemplo:

```
#define X_AXIS 180
#define Y_AXIS 175
#define X_PAD 20
#define Y_PAD 10
#define NUM_PTS 10
#define X 0
#define Y 1

void plot_lines(GtkMenuItem *was_clicked, GtkWidget *where_to_draw)
{
    static int first_time_in = 1;
    static GdkPoint points_to_plot[NUM_PTS] = {
        { 0, 0}, { 15, 20}, {30, 80}, { 45, 30},
        { 60, 25}, { 75, 90}, {90, 120}, {105, 170},
        {120, 100}, {135, 0} };

    GdkGC *plot_gc;
    int pts;

    /* Ajuste de los puntos */

    if(first_time_in){
        for(pts = 0; pts<NUM_PTS; pts++){
            points_to_plot[pts].x += X_PAD;
            points_to_plot[pts].y += Y_PAD + Y_AXIS -
                points_to_plot[pts].y;
        }
        first_time_in = 0;
    }

    /* Obtiene el gc negro */
    plot_gc = GTK_WIDGET(where_to_draw)->style->black_gc;

    /* Dibuja los ejes */
    gdk_draw_line(GTK_WIDGET(where_to_draw)->window, plot_gc,
        X_PAD, Y_AXIS + Y_PAD, X_PAD, Y_PAD);
    gdk_draw_line(GTK_WIDGET(where_to_draw)->window, plot_gc,
        X_PAD, Y_AXIS + Y_PAD, Y_PAD + X_AXIS, Y_AXIS + Y_PAD);

    /* Dibuja */
    gdk_draw_lines(GTK_WIDGET(where_to_draw)->window, plot_gc,
        points_to_plot, NUM_PTS);
}
```

Si se quiere dibujar una serie de líneas que no están conectadas entre sí se utilizará la siguiente función:

```
void gdk_draw_segments(GdkDrawable *drawable, GdkGC *gc, GdkSegment *segs,
gint nsegs)
```

<i>Parámetro</i>	<i>Descripción</i>
<i>drawable</i>	En donde se quieren dibujar las líneas
<i>gc</i>	Las características del gráfico ( <i>Graphic context</i> )
<i>segs</i>	El arreglo de <i>GdkSegments</i> a dibujar
<i>nsegs</i>	El número de líneas a dibujar.

La definición de *GdkSegment* es la siguiente

```
struct GdkSegment{
    gint16 x1;
    gint16 y1;
    gint16 x2;
    gint16 y2
};
```

El siguiente ejemplo dibuja un tablero del juego de gato:

```
void tictactoe_grid( GtkWidget *was_clicked, GtkWidget *where_to_draw )
{
    GdkSegment grid[4] = { { 60,20 , 60,100}, { 90,20 , 90,100},
                          { 35,50 , 115,50}, { 35,70 , 115,70} };

    GdkGC *plot_gc;

    /* Obtiene un GC negro */
    plot_gc = GTK_WIDGET(where_to_draw)->style->black_gc ;

    /* Dibuja */
    gdk_draw_segments(GTK_WIDGET(where_to_draw)->window, plot_gc, grid , 4);
}
```

### 13.3. Rectángulos

Para dibujar rectángulos utilizaremos la siguiente función:

```
void gdk_draw_rectangle(GdkDrawable *drawable, GdkGC *gc, gint *filled,
                       gint x, gint y, gint width, gint height)
```

<i>Parámetro</i>	<i>Descripción</i>
<i>drawable</i>	En donde se quieren dibujar las líneas
<i>gc</i>	Las características del gráfico ( <i>Graphic context</i> )
<i>filled</i>	Especifica si el rectángulo es relleno y no. TRUE o FALSE
<i>x, y,</i>	La posición de la esquina superior izquierda del rectángulo.
<i>width, height</i>	El tamaño del rectángulo.

Ejemplo:

```
void little_house(GtkWidget *where_to_draw, GdkGC *gc, gint x, gint y )
{
    GdkGC *house_gc;

    house_gc = where_to_draw->style->black_gc ;

    /* Techo */
    gdk_draw_rectangle( where_to_draw->window, house_gc, FALSE,
                        x, y, 70, 5 );

    /* Chimenea */
    gdk_draw_rectangle( where_to_draw->window, house_gc, FALSE,
                        x+40, y-10, 5, 10 );
}
```

```

/* Casa */
gdk_draw_rectangle( where_to_draw->window, house_gc, FALSE,
                    x+10, y+5, 50, 50 );

/* Ventanas */
gdk_draw_rectangle( where_to_draw->window, house_gc, FALSE,
                    x+15, y+20, 10, 13 );
gdk_draw_rectangle( where_to_draw->window, house_gc, FALSE,
                    x+45, y+20, 10, 13 );

/* Puerta */
gdk_draw_rectangle( where_to_draw->window, house_gc, FALSE,
                    x+30, y+35, 12, 20 );

gdk_draw_string(  where_to_draw->window,  where_to_draw->style->font,  house_gc,
25, 125, "Rectangles");
}

```

### 13.4. Polígonos

Para dibujar polígonos utilizaremos la siguiente función:

```

void gdk_draw_polygon(GdkDrawable *drawable, GdkGC *gc, gint *filled,
                    GdkPoint *points, gint npoints)

```

<u>Parámetro</u>	<u>Descripción</u>
<i>drawable</i>	En donde se quieren dibujar las líneas
<i>gc</i>	Las caracterísicas del gráfico ( <i>Graphic context</i> )
<i>filled</i>	Especifica si el rectángulo es relleno y no. TRUE o FALSE
<i>points</i>	El arreglo de <code>GdkPoints</code> que contiene los puntos del polígono.
<i>npoints</i>	El número de puntos que harán el polígono.

Ejemplo:

```

void little_star( GtkWidget *where_to_draw, GdkGC *gc )
{
    GdkPoint vertices[5] = { { 140, 56 }, { 122, 104}, { 164, 74 },
                             { 116, 74 }, { 152, 104} };

    /* Dibuja una estrella de 5 picos*/
    gdk_draw_polygon( where_to_draw->window, gc, TRUE,
                      vertices , 5 );

    gdk_draw_string(  where_to_draw->window,  where_to_draw->style->font,  gc,  115,
125, "Polygon");
}

```

### 13.5. Arcos

Para dibujar arcos usaremos la siguiente función:

```

void gdk_draw_arc(GdkDrawable *drawable, GdkGC *gc, gint *filled,
                 gint x, gint y, gint width, gint height,
                 gint angle1, gint angle2);

```

<u>Parámetro</u>	<u>Descripción</u>
<i>drawable</i>	En donde se quieren dibujar las líneas
<i>gc</i>	Las caracterísicas del gráfico ( <i>Graphic context</i> )
<i>filled</i>	Especifica si el rectángulo es relleno y no. TRUE o FALSE
<i>x, y</i>	La esquina superior izquierda del rectángulo dentro del cual se dibujará el arco.

<i>width, height</i>	El tamaño del rectángulo dentro del cual se dibujará el arco
<i>angle1</i>	El comienzo del arco, en 64° de grado.
<i>angle2</i>	El final del arco, en 64° de grado.

Ejemplo:

```
void girl( GtkWidget *was_clicked, GtkWidget *where_to_draw )
{
    GdkGC *plot_gc;

    /* Obtiene el GC negro */
    plot_gc = GTK_WIDGET(where_to_draw)->style->black_gc ;

    /* Dibuja la cara */
    gdk_draw_arc( GTK_WIDGET(where_to_draw)->window, plot_gc, TRUE,
                  15, 35, 40, 50, 0, 360*64 );
    /* Dibuja el moño del cabello */
    gdk_draw_arc( GTK_WIDGET(where_to_draw)->window, plot_gc, TRUE,
                  10, 12, 50, 35, -5*64, 45*64 );
    gdk_draw_arc( GTK_WIDGET(where_to_draw)->window, plot_gc, TRUE,
                  10, 12, 50, 35, 165*64, 45*64 );

    /* Oídos */
    gdk_draw_arc( GTK_WIDGET(where_to_draw)->window, plot_gc, TRUE,
                  10, 55, 8, 8, 0, 360*64 );
    gdk_draw_arc( GTK_WIDGET(where_to_draw)->window, plot_gc, TRUE,
                  50, 55, 8, 8, 0, 360*64 );
}
```

### 13.6. Dibujar texto

Para dibujar texto se utilizará la siguiente función:

```
void gdk_draw_string(GdkDrawable *drawable, GdkFont *font, GdkGC *gc,
                    gint x, gint y, const gchar *string)
```

<u>Parámetro</u>	<u>Descripción</u>
<i>drawable</i>	En donde se quieren dibujar las líneas
<i>font</i>	El tipo de letra a usar
<i>gc</i>	Las características del gráfico ( <i>Graphic context</i> )
<i>x, y</i>	La posición de la cadena a escribir
<i>string</i>	El texto a escribir.

Ejemplo:

```
void draw_pie_slice( GtkWidget *where, GdkGC *gc, guint start, guint percent, gchar
*label, guint pie_x, guint pie_y )
{
    static gint label_number=0;
    gint string_width, string_height;
    gint change_in_y;

    if (start == 0)
        label_number = 0;
    gdk_draw_arc( where->window, gc, TRUE, 0, 0, pie_x, pie_y,
                  start*360*64/100, percent*360*64/100 );
    string_width = gdk_string_width( where->style->font, label );
    string_height = gdk_string_height( where->style->font, label );
```

```

change_in_y = 1.3*label_number*string_height;
gdk_draw_rectangle( where->window, gc, TRUE, 20, 40+pie_y+change_in_y,
                    10, string_height );
gdk_draw_string( where->window, where->style->font, gc,
                 35,40+pie_y+string_height+change_in_y, label );
label_number++;
}

```

La función `gdk_font_load` nos permite obtener un tipo válido de fuentes:

```
GdkFont *gdk_font_load(const gchar *font_name)
```

<u>Parámetro</u>	<u>Descripción</u>
<i>font_name</i>	El nombre del tipo de fuente a cargar

También existe la función `gdk_draw_text`, el cual, en lugar de recibir una cadena de caracteres, recibe la cadena y un número, que indica cuántos caracteres de dicha cadena se imprimirán:

```
void gdk_draw_string(GdkDrawable *drawable, GdkFont *font, GdkGC *gc,
                    gint x, gint y, const gchar *string, gint text_length)
```

<u>Parámetro</u>	<u>Descripción</u>
<i>drawable</i>	En donde se quieren dibujar las líneas
<i>font</i>	El tipo de letra a usar
<i>gc</i>	Las caracterísicas del gráfico ( <i>Graphic context</i> )
<i>x, y</i>	La posición de la cadena a escribir
<i>string</i>	El texto a escribir.
<i>length</i>	El número de caracteres de string a escribir.

Para determinar el tamaño de una cadena se pueden utilizar las siguientes funciones:

```
gint gdk_string_width(GdkFont *font, const gchar *string)
gint gdk_string_height(GdkFont *font, const gchar *string)
```

<u>Parámetro</u>	<u>Descripción</u>
<i>font</i>	El tipo de letra a usar.
<i>string</i>	La cadena a la que se quiere cambiar el tamaño.

## 14. Manejando GDK Events

La diferencia entre GTK+ y GDK es que éste último trabaja a un nivel más bajo que GTK+. GDK trabaja con `GdkEvents`, no con señales. Un `GdkEvent` es recibido por un programa de GDK cuando el usuario realiza alguna acción dentro de una ventana, tal y como hacer clic en un botón del ratón, presionar alguna tecla del teclado, etc. Los eventos que son generados en respuesta a alguna acción del usuario son guardados dentro de una cola de eventos de GDK para que éste pueda procesarlos.

Los eventos de GDK permiten al programador procesar acciones realizadas por el usuario que pueden llegar a ser de interés pero que no están específicamente ligadas a algún widget en particular.

Existen muchas maneras de procesar un evento de GDK en una aplicación. Afortunadamente, GTK+ tiene incluido un modelo de procesamiento de eventos de GDK que hace la programación de este tipo de eventos más fácil. En lugar de estar monitoreando y recibiendo desde un programa en GTK+ los eventos desde la cola de eventos de GDK, GTK+ realiza esa acción por nosotros. Los eventos de bajo nivel de GDK están disponibles en los programas de GTK+ como señales. Los prototipos de las funciones que manejarán dichas señales difieren un poco de las funciones manejadoras estándar. Por ejemplo, los prototipos de la mayoría de las funciones que

manejan las señales emitidas por un evento de GDK incluyen un valor de retorno de tipo `gboolean`; la función regresará un valor `TRUE` si el evento fue manejado satisfactoriamente o `FALSE` si se decide que sea manejado el evento directamente por el sistema. Lo anterior ya se había realizado anteriormente cuando se utilizó la señal `"delete_event"`.

Supongamos que queremos desplegar las coordenadas en donde se encuentra el ratón en un *drawing area* cuando el usuario hace clic con el botón izquierdo del ratón (o botón 1 en algunos dispositivos de entrada gráficos). El widget de tipo `GdkDrawingArea` no tiene asociado a él ningún tipo de señal `activate` o `click`. Sin embargo, la aplicación será ciertamente capaz de preguntar y de recibir la información acerca de cuándo se hizo clic en un botón del ratón dentro de una ventana que tiene un widget de tipo `GdkDrawingArea`. Lo anterior se puede hacer proceando un evento de GDK, específicamente el evento `GDK_BUTTON_PRESS`.

GTK+ sigue básicamente el mismo modelo conocido hasta este momento de las señales cuando está trabajando con los eventos de GDK de emitir una señal en respuesta a alguna acción del usuario, por lo tanto se seguirá conectando las llamadas a las funciones que atienden las señales declaradas con la función `g_signal_connect()`.

Las señales que se emiten como consecuencia de un evento de GDK están disponibles a través de la clase `GtkWidget`. A continuación se enlistan todos los eventos disponibles:

- `event`
- `button_press_event`
- `button_release_event`
- `scroll_event`
- `motion_notify_event`
- `delete_event`
- `destroy_event`
- `expose_event`
- `key_press_event`
- `key_release_event`
- `enter_notify_event`
- `leave_notify_event`
- `configure_event`
- `focus_in_event`
- `focus_out_event`
- `map_event`
- `unmap_event`
- `property_notify_event`
- `selection_clear_event`
- `selection_request_event`
- `selection_notify_event`
- `proximity_in_event`
- `proximity_out_event`
- `visibility_notify_event`
- `client_event`
- `no_expose_event`
- `window_state_event`
- `selection_received`
- `selection_get`
- `drag_begin_event`
- `drag_end_event`
- `drag_data_delete`
- `drag_motion`
- `drag_drop`
- `drag_data_get`
- `drag_data_received`

Como se dijo anteriormente, en ocasiones anteriores ya se había trabajado con la señal `delete_event`. Dicha señal es emitida, de hecho, como respuesta a un evento `GDK_DELETE`.

Como se recordará, el prototipo de la función que debe atender la función `delete_event` es el siguiente:

```
gint user_function (GtkWidget *widget, GdkEvent *event, gpointer user_data)
```

El segundo parámetro (`GdkEvent *event`) es un apuntador a una unión que contiene la información correspondiente al evento `GDK_DELETE` que es de mucho interés para la función. A continuación se muestra la definición de dicha unión:

```
union GdkEvent{  
    GdkEventType type;
```



```
GdkEventAny any;  
GdkEventExpose expose;  
GdkEventNoExpose no_expose;  
GdkEventVisibility visiblility;  
GdkEventMotion motion;  
GdkEventButton button;  
GdkEventKey key;  
GdkEventCrossing crossing;  
GdkEventFocus focus_change;  
GdkEventConcigure configure;  
GdkEventProperty property;  
GdkEventSelecion selection;  
GdkEventProximity proximity;  
GdkEventClient client;  
GdkEventDND dnd;  
};
```

`GdkEvent` es una unión de las posibles estructuras del evento que pueden ser regresadas, dependiendo de la acción realizada por el usuario. El primer elemento en la unión indica el tipo del evento que ha ocurrido y, por lo tanto, el campo de la unión que es el válido. Los posibles valores del tipo son los siguientes:

- |                       |                         |
|-----------------------|-------------------------|
| • GDK_NOTHING         | • GDK_SELECTION_CLEAR   |
| • GDK_DELETE          | • GDK_SELECTION_REQUEST |
| • GDK_DESTROY         | • GDK_SELECTION_NOTIFY  |
| • GDK_EXPOSE          | • GDK_PROXIMITY_IN      |
| • GDK_MOTION_NOTIFY   | • GDK_PROXIMITY_OUT     |
| • GDK_BUTTON_PRESS    | • GDK_DRAG_ENTER        |
| • GDK_2BUTTON_PRESS   | • GDK_DRAG_LEAVE        |
| • GDK_3BUTTON_PRESS   | • GDK_DRAG_MOTION       |
| • GDK_BUTTON_RELEASE  | • GDK_DRAG_STATUS       |
| • GDK_KEY_PRESS       | • GDK_DROP_START        |
| • GDK_KEY_RELEASE     | • GDK_DROP_FINISHED     |
| • GDK_ENTER_NOTIFY    | • GDK_CLIENT_EVENT      |
| • GDK_LEAVE_NOTIFY    | • GDK_VISIBILITY_NOTIFY |
| • GDK_FOCUS_CHANGE    | • GDK_NO_EXPOSE         |
| • GDK_CONFIGURE       | • GDK_SCROLL            |
| • GDK_MAP             | • GDK_WINDOW_STATE      |
| • GDK_UNMAP           | • GDK_SETTING           |
| • GDK_PROPERTY_NOTIFY |                         |

Los demás campos de la unión representan diferentes estructuras de eventos específicos usadas para regresar información a la aplicación, tal y como en qué botón del ratón se hizo clic o qué ventana cambió de tamaño. El segundo campo `GdkEventAny any;`, es una estructura genérica que contiene información proporcionada por todos los eventos. Con la excepción de `GdkEventAny`, sólo un evento es válido al mismo tiempo, de ahí que lo que se use sea una unión.

Cuando se llama un manejador del evento `delete_event`, el campo `type` del `GdkEvent` es `GDK_DELETE`, el cual no tiene ninguna estructura específica; toda la información necesaria se encuentra en la estructura `GdkEventAny`, la cual se compone de la manera siguiente:

```
struct GdkEventAny{  
    GdkEventType type;  
    GdkWindow *window;  
    gint8 send_event;  
}
```

## 14.1. La estructura `GdkEventButton`

Las demás estructuras que se encuentran en la unión `GdkEvent` contienen información específica; por ejemplo, la estructura `GdkEventButton` mostrada a continuación indica qué botón fue seleccionado:

```
struct GdkEventButton{
    GdkEventType type;
    GdkWindow *window;
    gint8 send_event;
    guint32 time;
    gdouble x;
    gdouble y;
    gdouble pressure;
    gdouble xtilt;
    gdouble ytilt;
    guint state;
    guint button;
    GdkInputSource source;
    guint32 deviceid;
    gdouble x_root, y_root;
}
```

El valor del campo `type` para la estructura `GdkEventButton` es `GDK_BUTTON_PRESS` o `GDK_BUTTON_RELEASE` porque ambas acciones generan la misma información. Adicionalmente, GDK maneja los valores `GDK_2BUTTON_PRESS` y `GDK_3BUTTON_PRESS` que son tipos válidos utilizados para ayudar a detectar clics dobles y triples. El campo `window` indica en qué ventana ocurrió el evento del botón. El campo `time` indica la hora en la que sucedió el evento (en milisegundos), las líneas 5 y 6 indican la posición del ratón cuando el evento ocurrió. Las líneas 7 a 9 se utilizan para dispositivos de entrada gráficos especiales. La línea 10 muestra el estado de los posibles modificadores y máscaras, tales como la tecla `Alt` o `Ctrl`. La línea 11 indica qué botón fue presionado. Las líneas 12 – 13 son usadas cuando se está trabajando con múltiples dispositivos de entrada. La línea 14 muestra la posición del cursor cuando el evento ocurrió.

Una pregunta común podría ser: ¿qué botón fue presionado cuando se emite una señal `clicked`? La respuesta es el botón 1 (botón izquierdo del ratón) ya que si se intenta hacer clic con el botón derecho del ratón sobre algún widget, no pasará nada; sin embargo, lo anterior no es un impedimento para implementar alguna acción cuando se hace clic con el botón de en medio o con el botón derecho del ratón. Para lo anterior, se deberá procesar la señal `button-press-event`. A continuación se muestra el prototipo de la función que se debe implementar para atender dicha señal:

```
static gint button_press_handler (GtkWidget *widget, GdkEvent *event,
                                  gpointer user_data)
```

La función deberá regresar un `TRUE` si el evento es manejado satisfactoriamente y un `FALSE` en cualquier otro caso. Por ejemplo, a continuación se muestra el código de una función que imprime qué botón fue presionado cuando se hace clic en un `GtkButton`:

```
gboolean que_boton (GtkWidget *widget, GdkEvent *event,
                    gpointer user_data)
{
    g_print("El boton %d fue presionado\n",event->button.button);
    return TRUE;
}
```

Debido a que la función `que_boton` maneja el evento, regresa un valor de `TRUE`. La conexión de la señal se tendría que hacer de la siguiente manera:

```
g_signal_connect(G_OBJECT(boton_salir), "button-press-event",
                G_CALLBACK( que_boton), NULL);
```

## 14.2. Seleccionando Eventos

Una aplicación de GTK de manera automática recibe las señales emitidas por los widgets ya que los mismos widgets se aseguran de ello. Algunas señales de eventos están también ya seleccionadas por GTK+, como la señal `delete_event`. Un `GtkButton` puede procesar también la señal `button-press-event` debido a que un clic es un `GDK_BUTTON_PRESS` seguido de un `GDK_BUTTON_RELEASE`. Sin embargo, una aplicación de GTK+ no recibe automáticamente todos los eventos que ocurren dentro de una ventana. La función utilizada para notificar a un widget qué señales adicionales se quieren procesar para él es la siguiente:

```
void gtk_widget_set_events(GtkWidget *widget, gint *events)
```

<i>Parámetro</i>	<i>Descripción</i>
<i>widget</i>	El widget al que estamos agregando la señal
<i>events</i>	La lista de máscaras de eventos que queremos procesar, pueden enunciarse por separado o separados por el operador OR. Por ejemplo: GDK_BUTTON_PRESS_MASK   GDK_BUTTON_RELEASE_MASK

Las máscaras de los eventos son utilizadas para indicar en qué eventos estamos interesados. La siguiente tabla muestra una lista de las máscaras disponibles, así como sus correspondientes señales:

<b>Máscara del Evento</b>	<b>Señal Emitida</b>	<b>Descripción</b>
GDK_BUTTON_MOTION_MASK	motion-notify-event	Movimiento del ratón mientras un botón es presionado
GDK_BUTTON1_MOTION_MASK GDK_BUTTON2_MOTION_MASK GDK_BUTTON3_MOTION_MASK	motion-notify-event	Movimiento del ratón mientras un botón específico es presionado
GDK_BUTTON_PRESS_MASK	button-press-event	Apretar un botón del ratón
GDK_BUTTON_RELEASE_MASK	button-release-event	Soltar un botón del ratón
GDK_ENTER_NOTIFY_MASK	enter-notify-event	El puntero del ratón ha entrado a una ventana
GDK_EXPOSURE_MASK	expose-event	Una porción de una ventana se ha hecho visible
GDK_FOCUS_CHANGE_MASK	focus-in-event focus-out-event	El objetivo del teclado ha cambiado entre ventanas
GDK_KEY_PRESS_MASK	key-press-event	Una tecla ha sido presionada
GDK_KEY_RELEASE_MASK	key-release-event	Una tecla ha sido soltada
GDK_LEAVE_NOTIFY_MASK	leave-notify-event	El puntero del ratón ha salido de una ventana
GDK_PROPERTY_CHANGE_MASK	property-notify-event	Ha cambiado una propiedad de una ventana
GDK_VISIBILITY_NOTIFY_MASK	visibility-notify-event	Un cambio ha ocurrido en la visibilidad de una ventana

## 15. Cambiar un texto en una etiqueta

```
void gtk_label_set_text(GtkLabel *label, const gchar *str)
```

<i>Parámetro</i>	<i>Descripción</i>
<i>label</i>	La etiqueta ya creada a la que se quiere cambiar el texto
<i>str</i>	El nuevo texto que se quiere desplegar en la etiqueta

## 16. Hacer aparecer un widget

```
void gtk_widget_popup(GtkWidget *widget, gint x, gint y)
```

<u>Parámetro</u>	<u>Descripción</u>
<i>widget</i>	El widget que se quiere aparecer
<i>x, y</i>	La posición en donde quiere que se aparezca el widget.

## 17. Más sobre listas

Para remover elementos de una lista, usaremos la siguiente función:

```
void gtk_list_remove_items(GtkList *list, GList *items)
```

<u>Parámetro</u>	<u>Descripción</u>
<i>list</i>	La lista de la que se quieren remover los elementos.
<i>items</i>	El elemento que se quiere eliminar.

Para remover un conjunto de elementos:

```
void gtk_list_clear_items(GtkList *list, gint *start, gint end)
```

<u>Parámetro</u>	<u>Descripción</u>
<i>list</i>	La lista de la que se quieren remover los elementos.
<i>start</i>	La posición del primer elemento que se quiere eliminar.
<i>end</i>	La posición del último elemento que se quiere eliminar.

## 18. Más sobre Botones

Además de los botones que ya se han visto, GTK+ provee de diferentes funciones que permiten crear otros tipos de botones. Además de las funciones:

```
GtkWidget *gtk_button_new(void);  
GtkWidget *gtk_button_new_with_label(gchar *label);
```

que se explicaron a detalle anteriormente, GTK+ provee de otras funciones de gran utilidad para la creación de botones, a saber:

```
GtkWidget *gtk_button_new_with_mnemonic (const gchar *label);
```

```
GtkWidget *gtk_button_new_from_stock(const gchar *stock_id);
```

La primera función permite crear un botón agregándole en su nombre un acceso directo. Para esto, se tendrá que preceder alguno de los caracteres de la cadena que recibe como argumento con un guión bajo (\_) para que dicho caracter sea considerado como el acceso directo.

La segunda función permite crear un botón con un diseño prediseñado. Algunas de las opciones válidas para el parámetro *stock\_id* son:

GTK_STOCK_CANCEL	GTK_STOCK_NO
GTK_STOCK_CLOSE	GTK_STOCK_OK
GTK_STOCK_COPY	GTK_STOCK_QUIT
GTK_STOCK_CUT	GTK_STOCK_SAVE
GTK_STOCK_DELETE	GTK_STOCK_SAVE_AS
GTK_STOCK_FLOPPY	GTK_STOCK_STOP
GTK_STOCK_NEW	GTK_STOCK_UNDEL

## 18.1. Toggle buttons

```
GtkWidget *gtk_toggle_button_new_with_label(const gchar *label)
```

<u>Parámetro</u>	<u>Descripción</u>
<i>label</i>	La etiqueta del botón

```
GtkWidget *gtk_toggle_button_new(void)
```

```
GtkWidget *gtk_toggle_button_new_with_mnemonic( const gchar *label );
```

<u>Parámetro</u>	<u>Descripción</u>
<i>label</i>	La etiqueta del botón que contendrá el caracter subrayado para tener un acceso directo. Dicho caracter deberá estar precedido por un guión bajo (_)

Para saber si un toggle button está activo o no, se utiliza la siguiente función:

```
gboolean gtk_toggle_button_get_active(GtkToggleButton *toggle_button)
```

<u>Parámetro</u>	<u>Descripción</u>
<i>toggle_button</i>	El botón del que se quiere conocer el estado

Para fijar el estado del botón:

```
void gtk_toggle_button_set_active(GtkToggleButton *toggle_button,  
                                gboolean is_active)
```

<u>Parámetro</u>	<u>Descripción</u>
<i>toggle_button</i>	El botón del que se quiere fijar el estado.
<i>is_active</i>	El estado que se quiere fijar al botón.

Cuando un *toggle button* es seleccionado, emite la señal *toggled* la cual mandará llamar a una función cuyo prototipo es el siguiente:

```
void user_function(GtkToggleButton *toggle_button, gpointer user_data)
```

## 18.2. Check buttons

```
GtkWidget *gtk_check_button_new_with_label(const gchar *label)
```

<u>Parámetro</u>	<u>Descripción</u>
<i>label</i>	La etiqueta del botón

```
GtkWidget *gtk_ check_button_new(void)
```

```
GtkWidget *gtk_check_button_new_with_mnemonic ( const gchar *label );
```

<u>Parámetro</u>	<u>Descripción</u>
<i>label</i>	La etiqueta del botón que contendrá el caracter subrayado para tener un acceso directo. Dicho caracter deberá estar precedido por un guión bajo (_)

Un *check button* emite la misma señal que un *toggle button*.

### 18.3. Radio buttons

```
GtkWidget *gtk_radio_button_new_with_label(GList *group,  
                                           const gchar *label)
```

<u>Parámetro</u>	<u>Descripción</u>
<i>group</i>	El grupo de botones del cual es miembro el botón.
<i>label</i>	La etiqueta del botón.

La primera vez que se crea un botón, el parámetro *group* puede ser NULL, por cada siguiente botón que se vaya a crear y que se asigne al mismo grupo, será necesario pasar en el parámetro *group* el ID de dicho grupo. Esto se obtiene con la siguiente función:

```
GsList *gtk_radio_button_get_group(GtkRadioButton *radio_button)
```

<u>Parámetro</u>	<u>Descripción</u>
<i>radio_button</i>	El primer botón que se creó.

Ejemplo:

```
GtkWidget *clear_radio, *line_radio, *bar_radio;  
/* ..... */  
/* Crea los radio buttons */  
clear_radio = gtk_radio_button_new_with_label(NULL, "Clear Area");  
gtk_box_pack_start_defaults(GTK_BOX(box), clear_radio);  
  
line_radio = gtk_radio_button_new_with_label(  
    gtk_radio_button_get_group(GTK_RADIO_BUTTON(clear_radio)),  
    "Line Plot");  
gtk_box_pack_start_defaults(GTK_BOX(box), line_radio);  
  
bar_radio = gtk_radio_button_new_with_label(  
    gtk_radio_button_get_group(GTK_RADIO_BUTTON(line_radio)),  
    "Bar Chart");  
gtk_box_pack_start_defaults(GTK_BOX(box), bar_radio);  
  
g_signal_connect(G_OBJECT(clear_radio), "toggled",  
    G_CALLBACK( clear_area), drawing);  
g_signal_connect(G_OBJECT(line_radio), "toggled",  
    G_CALLBACK( line_plot), drawing);  
g_signal_connect(G_OBJECT(bar_radio), "toggled",  
    G_CALLBACK( bar_chart), drawing);
```

En un grupo de *radio buttons*, si ninguno se fija como activo, por default se activará el primer botón del grupo.

### 19. Combo Box

Un Combo Box es un menú con opciones predefinidas dentro del cual el usuario puede seleccionar una de éstas. Un Combo Box se compone básicamente de dos elementos: Un Entry y una lista. Para crearlo se debe usar la siguiente función:

```
GtkWidget *gtk_combo_new( void );
```

Si por alguna razón se quiere fijar el texto que se despliega en un combo, se podrá usar la siguiente función:

```
gtk_entry_set_text (GTK_ENTRY (GTK_COMBO (combo)->entry), "Texto");
```

Para fijar los valores dentro de un Combo Box, se deberá utilizar la siguiente función:

```
void gtk_combo_set_popdown_strings( GtkCombo *combo, GList *strings );
```

Antes de poder hacer lo anterior, se debe crear una lista (GList) del texto que se desea desplegar en el Combo, lo anterior se puede hacer de la siguiente manera:

```
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);
```

Existen otras funciones que ayudan a manejar el comportamiento de un Combo Box:

```
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 );
```

Para poder obtener el texto que está actualmente seleccionado en un Combo Box, se tendrá que hacer lo siguiente:

```
gchar *string;

string = gtk_entry_get_text (GTK_ENTRY (GTK_COMBO (combo)->entry));
```

El Combo Box emite una señal "activated" cuando es seleccionado.

Ejemplo:

```
#include <gtk/gtk.h>

/* Callbacks */
GtkWidget *append_menu_item( GtkMenu *menu, const gchar *text,
                             GtkSignalFunc callback, gpointer user_data)
{
    GtkWidget *menu_item;
    if ( text )
        menu_item = gtk_menu_item_new_with_label( text );
    else
        menu_item = gtk_menu_item_new();

    gtk_menu_shell_append(GTK_MENU_SHELL(menu), menu_item );
    gtk_widget_show( menu_item );
    if (callback)
        g_signal_connect( G_OBJECT(menu_item), "activate",
                           G_CALLBACK( callback ), user_data );
    return menu_item;
}

gboolean delete_event_handler (GtkWidget *widget, GdkEvent *event,
                               gpointer user_data)
{
    /* Received closure from the window manager */
    g_print("The window manager is asking to close this application\n");
    return(FALSE); /* FALSE - do no prevent closure */
}
```

```
void print_and_quit( GtkWidget *was_clicked, gpointer user_data )
{
    /* Use glibs printf equivalent to print a message */
    g_print( "Thank you for using this program.\n" );
    gtk_main_quit();
}

int main( int argc, char *argv[] )
{
    GtkWidget *top_widget, *vbox, *label, *menu_box;
    GtkWidget *drink, *entree, *side;
    GList *drink_list = NULL, *entree_list = NULL, *side_list = NULL;

    /* Initialize the environment */
    gtk_init( &argc, &argv );

    /* Create toplevel widget */
    top_widget = gtk_window_new( GTK_WINDOW_TOPLEVEL );
    gtk_window_set_title( GTK_WINDOW( top_widget ), "Order" );
    g_signal_connect( G_OBJECT( top_widget ), "delete_event",
                     G_CALLBACK( delete_event_handler ), NULL );
    g_signal_connect( G_OBJECT( top_widget ), "destroy",
                     G_CALLBACK( print_and_quit ), NULL );

    /* Create managing vertical box widget */
    vbox = gtk_vbox_new( FALSE, 10 );
    gtk_container_add( GTK_CONTAINER( top_widget ), vbox );

    /* Create label widget */
    label = gtk_label_new( "Customer 1" );
    gtk_box_pack_start_defaults( GTK_BOX( vbox ), label );

    /* Create managing horizontal box widget */
    menu_box = gtk_hbox_new( FALSE, 20 );
    gtk_box_pack_start_defaults( GTK_BOX( vbox ), menu_box );

    /* Create Drink option menu and associated submenu */
    drink = gtk_combo_new();

    /* Create Drink menu items */
    drink_list = g_list_append( drink_list, "Select Drink" );
    drink_list = g_list_append( drink_list, "Iced Tea" );
    drink_list = g_list_append( drink_list, "Cola" );
    drink_list = g_list_append( drink_list, "Lemonade" );

    gtk_combo_set_popdown_strings( GTK_COMBO( drink ), drink_list );

    gtk_box_pack_start_defaults( GTK_BOX( menu_box ), drink );

    /* Create Entree option menu and associated submenu */
    entree = gtk_combo_new();

    /* Create Entree menu items */
    entree_list = g_list_append( entree_list, "Tacos" );
    entree_list = g_list_append( entree_list, "Enchiladas" );
    entree_list = g_list_append( entree_list, "Burritos" );
    entree_list = g_list_append( entree_list, "Stuffed Sopapilla" );

    gtk_combo_set_popdown_strings( GTK_COMBO( entree ), entree_list );

    gtk_box_pack_start_defaults( GTK_BOX( menu_box ), entree );
```



```
/* Create Side Dish option menu and associated submenu */
side = gtk_combo_new();

/* Create Side Dish menu items */
side_list = g_list_append (side_list, "Tacos");
side_list = g_list_append (side_list, "Enchiladas");
side_list = g_list_append (side_list, "Burritos");
side_list = g_list_append (side_list, "Stuffed Sopapilla");

gtk_combo_set_popdown_strings (GTK_COMBO (side), side_list);

gtk_box_pack_start_defaults( GTK_BOX( menu_box ), side );

/* Show the widgets */
gtk_widget_show_all( top_widget );

/* Processing Loop */
gtk_main();
g_print( "Bye!\n");
return 0;
}
```

## 20. Más opciones para un menú

Hasta ahora, los menus implementados solamente han tenido menu items. Existen básicamente tres elementos que también pueden agregarse a un menú, los cuales son:

- GtkTearOffMenuItem
- GtkCheckMenuItem
- GtkRadioMenuItem

### 20.1. GtkTearOffMenuItem

Este widget generalmente se pone en la parte superior de un menú y permite que el usuario separe dicho menú de la aplicación para tenerlo siempre visible, esto ocasiona que el menú aparezca con su propio administrador de ventanas que permite que el menú pueda moverse, maximizarse y minimizarse si así lo quiere el usuario. Para crear este widget se tiene que utilizar la función:

```
GtkWidget *gtk_tearoff_menu_item_new(void);
```

Normalmente, el GtkTearOffMenuItem es el primer elemento que se pone dentro de un menú; sin embargo, se puede poner en cualquier posición dentro de un menú.

### 20.2. GtkRadioMenuItem

El GtkRadioMenuItem es el componente equivalente a un GtkRadioButton, solamente que estará adentro de un menú. Para crearlo, se deberá utilizar la función:

```
GtkWidget *gtk_radio_menu_item_new_with_label(GSList *group, gchar *label);
```

<i>Parámetro</i>	<i>Descripción</i>
<i>group</i>	El grupo al que pertenecerá este widget.
<i>label</i>	El texto que aparecerá junto al widget.

Al igual que en los check buttons, cuando se está creando el primer elemento se deberá pasar NULL para el parámetro group, para los siguientes widgets, se necesitará pasar el grupo por medio de la función:

```
GSList *gtk_radio_menu_item_group(GtkRadioMenuItem *radio_menu_item);
```

<u>Parámetro</u>	<u>Descripción</u>
<i>radio_menu_item</i>	El <i>radio menu</i> que se quiere agregar al grupo.

El GtkRadioMenuItem emite una señal "toggled" cuando es seleccionado.

### 20.3. GtkCheckMenuItem

El GtkCheckMenuItem es el componente equivalente a un GtkCheckButton, solamente que estará adentro de un menú. Para crearlo, se deberá utilizar la función:

```
GtkWidget *gtk_check_menu_item_new_with_label(gchar *label);
```

<u>Parámetro</u>	<u>Descripción</u>
<i>label</i>	El texto que aparecerá junto al widget.

El GtkCheckMenuItem emite una señal "toggled" cuando es seleccionado.

## 21. Más sobre contenedores

Hasta este momento hemos ya trabajado con contenedores tales como GtkHBox, GtkVBox y GtkTable que, a diferencia del GtkWindow, permiten tener a más de un widget dentro de ellos. Sin embargo existen otros tipos de contenedores que también permiten tener dentro de ellos a más de un widget. Dichos contenedores son:

1. GtkNoteBook widgets: Permiten tener múltiples "páginas" de widgets que comparten el mismo espacio en la interfase gráfica.
2. GtkFixed widgets: Permiten asignar una posición específica a los widgets y moverlos a diferentes lados
3. GtkLayout widgets: Permiten crear un área con barras de desplazamiento en la cual se pueden poner los widgets en diferentes posiciones
4. Paned widgets: Permiten poner dos widgets uno seguido del otro y cambiarles el tamaño a cada uno indistintamente

### 21.1 GtkNoteBook widgets

Cuando queremos tener múltiples widgets compartiendo el mismo espacio, se utilizarán este tipo de widgets.

```
GtkWidget *gtk_notebook_new();
```

Para fijar cada una de las propiedades de este widget, existen las siguientes funciones:

```
void gtk_notebook_set_show_tabs(GtkNotebook *notebook, GtkPositionType pos);
```

en donde *pos* puede ser alguno de los siguientes valores:

```
GTK_POS_LEFT  
GTK_POS_RIGHT  
GTK_POS_TOP  
GTK_POS_BOTTOM
```

```
void gtk_notebook_set_homogenous_tabs(GtkNotebook *notebook,  
                                     gboolean homogenous);
```

```
void gtk_notebook_set_tab_pos(GtkNotebook *notebook, GtkPositionType pos);

void gtk_notebook_set_scrollable(GtkNotebook *notebook,
                                gboolean scrollable);

void gtk_notebook_set_tab_border(GtkNotebook *notebook,
                                guint border_width);

void gtk_notebook_set_tab_hborder(GtkNotebook *notebook,
                                guint tab_hborder);

void gtk_notebook_set_tab_vborder(GtkNotebook *notebook,
                                guint tab_vborder);
```

Este tipo de widget permite cambiar a las diferentes páginas mediante un menú de pop-up que aparecerá al hacer click con el botón derecho del mouse. Por default este menú está desactivado, pero se pueden utilizar alguna de las siguientes funciones para activarlo o desactivarlo:

```
void gtk_notebook_popup_disable(GtkNotebook *notebook);
void gtk_notebook_popup_enable(GtkNotebook *notebook);
```

Para agregar páginas:

```
void gtk_notebook_append_page(GtkNotebook *notebook, GtkWidget *child,
                              GtkWidget *tab_label);

void gtk_notebook_append_page_menu(GtkNotebook *notebook, GtkWidget *child,
                                   GtkWidget *tab_label, GtkWidget *menu_label);

void gtk_notebook_prepend_page(GtkNotebook *notebook, GtkWidget *child,
                               GtkWidget *tab_label);

void gtk_notebook_prepend_page_menu(GtkNotebook *notebook,
                                     GtkWidget *child, GtkWidget *tab_label, GtkWidget *menu_label);

void gtk_notebook_insert_page(GtkNotebook *notebook, GtkWidget *child,
                              GtkWidget *tab_label, gint position);

void gtk_notebook_insert_page_menu(GtkNotebook *notebook, GtkWidget *child,
                                   GtkWidget *tab_label, GtkWidget *menu_label,
                                   gint position);
```

Para saber qué página está activa en cierto momento:

```
gint gtk_notebook_get_current_page(GtkNotebook *notebook);
```

Para cambiar de página:

```
void gtk_notebook_set_current_page(GtkNotebook *notebook, gint page_num);

void gtk_notebook_next_page(GtkNotebook *notebook);
void gtk_notebook_prev_page(GtkNotebook *notebook);
```

Se pueden borrar páginas:

```
void gtk_notebook_remove_page(GtkNotebook *notebook, gint page_num);
```

La señal emitida cuando es cambiada una página en un GtkNotebook widget es switch-page. El prototipo de la función que deberá ser implementada para atender dicha señal es el siguiente:

```
void user_function(GtkNoteBook *notebook, GtkNotebookPage *page,  
                  gint page_num, gpointer data);
```

## 21.2 GtkFixed Widgets

Un widget del tipo GtkFixed es simplemente un área en blanco en donde podemos poner otros widgets. El desarrollador puede determinar en dónde serán puestos los widgets determinando su posición exacta dentro del contenedor. Este tipo de contenedores también trae como consecuencia que si cambia de tamaño, los widgets que estén dentro de él no lo harán. Para crear este tipo de widgets, se deberá llamar a la siguiente función:

```
GtkWidget *gtk_fixed_new(void)
```

Para agregar cada uno de los widgets al contenedor, se deberá llamar la función:

```
void gtk_fixed_put(GtkFixed *fixed, GtkWidget *child, gint16 x, gint16 y)
```

Parámetro	Descripción
<i>fixed</i>	El widget GtkFixed en donde se quiere agregar el widget.
<i>child</i>	El widget que se quiere agregar el widget.
<i>x</i>	La posición en x, en pixeles, del widget hijo, relativa a la esquina superior izquierda del GtkFixed widget.
<i>y</i>	La posición en y, en pixeles, del widget hijo, relativa a la esquina superior izquierda del GtkFixed widget.

Una vez que se han agregado elementos al contenedor, éstos podrán moverse utilizando la función:

```
void gtk_fixed_move(GtkFixed *fixed, GtkWidget *child, gint16 x, gint16 y)
```

Parámetro	Descripción
<i>fixed</i>	El widget GtkFixed en donde se quiere agregar el widget.
<i>child</i>	El widget que se quiere agregar el widget.
<i>x</i>	La posición en x, en pixeles, del widget hijo, relativa a la esquina superior izquierda del GtkFixed widget.
<i>y</i>	La posición en y, en pixeles, del widget hijo, relativa a la esquina superior izquierda del GtkFixed widget.

## 21.3 GtkLayout Widgets

Un GtkLayout widget es muy parecido a un GtkFixed widget. Ambos consisten en un área en blanco dentro de la cual se pueden agregar otros widgets por medio de dos coordenadas que están expresadas en pixeles. La diferencia básica entre ambos contenedores es que si el widget hijo se sale del área visible de GtkLayout, éste tendrá barras de desplazamiento que permitirán poner al widget hijo dentro de la zona visible de la interfase gráfica.

Para crear un GtkLayout widget se utilizar la siguiente función:

```
GtkWidget *gtk_layout_new(GtkAdjustment *hadjustment, GtkAdjustment *vadjustment);
```

Parámetro	Descripción
<i>hadjustment</i>	El apuntador al objeto GtkAdjustment que se utilizará para controlar el desplazamiento horizontal. Puede ser NULL.
<i>vadjustment</i>	El apuntador al objeto GtkAdjustment que se utilizará para controlar

el desplazamiento vertical. Puede ser `NULL`.

Para agregar y mover los widgets dentro del contenedor, se utilizarán las siguientes funciones:

Para agregar cada uno de los widgets al contenedor, se deberá llamar la función:

```
void gtk_layout_put(GtkLayout *layout, GtkWidget *child, gint16 x, gint16 y)
```

<i>Parámetro</i>	<i>Descripción</i>
<i>layout</i>	El widget <code>GtkFixed</code> en donde se quiere agregar el widget.
<i>child</i>	El widget que se quiere agregar el widget.
<i>x</i>	La posición en x, en pixeles, del widget hijo, relativa a la esquina superior izquierda del <code>GtkFixed</code> widget.
<i>y</i>	La posición en y, en pixeles, del widget hijo, relativa a la esquina superior izquierda del <code>GtkFixed</code> widget.

Una vez que se han agregado elementos al contenedor, éstos podrán moverse utilizando la función:

```
void gtk_layout_move(GtkLayout *layout, GtkWidget *child, gint16 x, gint16 y)
```

<i>Parámetro</i>	<i>Descripción</i>
<i>layout</i>	El widget <code>GtkFixed</code> en donde se quiere agregar el widget.
<i>child</i>	El widget que se quiere agregar el widget.
<i>x</i>	La posición en x, en pixeles, del widget hijo, relativa a la esquina superior izquierda del <code>GtkFixed</code> widget.
<i>y</i>	La posición en y, en pixeles, del widget hijo, relativa a la esquina superior izquierda del <code>GtkFixed</code> widget.

## 21.4 Paned Widgets

Cuando queremos particionar nuestra área de trabajo en diferentes secciones las cuales contengan cada una un widget diferente, podemos usar un widget del tipo `Paned`.

```
GtkWidget *gtk_hpaned_new();
GtkWidget *gtk_vpaned_new();
```

Para fijar los bordes entre los hijos de un `Paned` widget y los bordes para manipular el tamaño de los widgets hijos existen las siguientes funciones:

```
void gtk_paned_set_gutter_size(GtkPaned *paned, guint16 size);
void gtk_paned_set_handle_size(GtkPaned *paned, guint16 size);
```

Para agregar cada uno de los elementos a un `Paned` widget se usan las siguientes funciones:

```
void gtk_paned_pack1(GtkPaned *paned, GtkWidget *child,
                    gboolean resize, gboolean shrink);

void gtk_paned_pack2(GtkPaned *paned, GtkWidget *child,
                    gboolean resize, gboolean shrink);

void gtk_paned_add1(GtkPaned *paned, GtkWidget *child);
void gtk_paned_add2(GtkPaned *paned, GtkWidget *child);
```

Para cambiar el tamaño de un `Paned`:

```
void gtk_paned_set_position(GtkPaned *paned, gint *pos);
```

## 22. Añadiendo imágenes a una aplicación

Para poder agregar imágenes a un programa desarrollado en GTK 1.2, éstas deberán estar en formato “.xpm”. Una vez que se haya hecho esto, se deberán declarar tres variables:

```
GtkWidget *picwidget;  
GdkBitmap *mask;  
GdkPixmap *pixmap;
```

Posteriormente, se deberán ejecutar las siguientes instrucciones:

```
pixmap = gdk_pixmap_colormap_create_from_xpm(NULL,  
gtk_widget_get_default_colormap(), &mask, NULL, "imagen.xpm");  
  
picwidget = gtk_pixmap_new(pixmap, mask);
```

Las cuales permiten crear un widget (picwidget en este caso) a partir de la imagen “imagen.xpm”. Una vez realizado lo anterior se puede manipular el widget “picwidget” igual que cualquier otro widget.

## 23. Funciones especiales

En GTK 1.2 existe la función `gtk_timeout_add()` que permite llamar a una función determinada cada cierto tiempo. Para remover la acción anterior, se utiliza la función `gtk_timeout_remove()`. El código siguiente ejemplifica lo anterior:

```
id = gtk_timeout_add(2500, salir, NULL);  
gtk_main();  
gtk_timeout_remove(id);
```

Lo anterior especifica que se va a mandar llamar la función “salir” cada 2.5 segundos. El prototipo de la función llamada debe ser como el siguiente:

```
gint nombre(gpointer data)
```

## B I B L I O G R A F Í A

- Kernighan, Brian, Ritchie Dennis; The C Programming Language; 2nd Edition; Prentice Hall. 1988.
- Oualline Steve, Practical C Programming, O'Reilly & Associates, Inc. 1997
- Loudon Kyle Mastering Algorithms with C, O'Reilly & Associates, Inc. 1999
- Deitel, H.M, Deitel, P.J.; Como Programar en C/C++; Prentice Hall; Segunda Edición
- Wright, Peter; Beginning GTK+/GNOME Programming; Wrox; 2000.
- Martin Donna; GTK+ Programming in 21 Days; Sams Publishing; 2000.
- Loukides Mike, Oram Andy; Programming with GNU Software; O'Reilly & Associates, Inc. 1997
- <http://www.gtk.org>
- <http://developer.gnome.org/doc/API/gtk/index.html>
- <http://www.cs.jhu.edu/~goodrich/dsa/trees/btree.html>