

GTK - Entornos gráficos y programación orientada a eventos

Autor: Pablo D. Roca <pablodroca@gmail.com>
Revisión:1

GTK - Entornos gráficos y programación orientada a eventos	1
1. Introducción	1
Entornos Gráficos	2
Programación orientada a eventos	2
Cola de eventos (event queue)	2
Ciclo de eventos (event loop)	2
Manejadores de eventos (event handlers)	3
2. El esquema de GTK	3
Separación de capas	3
Xlib	4
Glib	4
GObject	4
Gdk	4
Cairo - Pango	4
Widgets	4
Callbacks	6
Eventos	6
Señales	6
El bucle principal	7
3. Programando en GTK	7
Programa de prueba	7
Compilación	8
Agregando una ventana	8
Empaquetamiento de Widgets	8
4. Programando en GTKMM	9
Programa sencillo	9
Dialogos	10
Manejo de memoria	11
Signals	11
ptr_fun	11
mem_fun	11
bind	12
5. Referencias	12

1. Introducción

La construcción de un aplicativo de software requiere cierto tiempo para la definición de su modo de uso y las capacidades que éste posee. Todo sistema tiene un objetivo y un requerimiento a satisfacer; por lo tanto, su construcción dependerá del mismo y estará estrechamente relacionada con su modo de uso.

Es así que nos encontramos con muy diversas arquitecturas para la construcción de un programa: procesamiento por lotes (conocido como *batch*), interactivos, orientados por eventos, de tiempo real (*realtime*).

Entornos Gráficos

Los sistemas con Entorno Gráfico conocidos también como GUIs (*graphical user interface*) son un tipo de sistema interactivo, orientado a eventos, donde el usuario interactúa con la computadora mediante gráficos. Una GUI brinda un marco de trabajo con recursos visuales que cumplen la función de otorgar información al usuario y recibir órdenes de su parte expresando las mismas por símbolos o gestos gráficos.

En el mercado de Entornos Gráficos se suelen utilizar los conceptos de ventanas, íconos, escritorios, etcétera para agrupar y presentar la información al usuario. De la misma forma, conceptos como botón, combo, cuadro de texto, suelen estar presente en toda operación de ingreso de datos.

Programación orientada a eventos

El paradigma de programación orientada por eventos se rige por una simple regla: todo acción que ejecute el sistema será en respuesta a los sucesos que acontezcan producto de acciones del usuario, del entorno en que se encuentra el sistema o del propio aplicativo que en su ejecución las produjo.

Bajo este esquema, todo suceso detectado por el sistema tendrá la posibilidad de ser un evento importante para el aplicativo, que merezca ser atendido y desencadene un procesamiento y posiblemente más eventos.

Cola de eventos (event queue)

Debido a la posibilidad de que los eventos ocurran en distintos contextos del sistema o incluso en simultaneo, se crea un mecanismo que permite controlar el orden que los mismos son atendidos.

Dada la simplicidad de la cola como estructura de ordenamiento y determinación de orden, se la escoge para definir un algoritmo simple que permita informar (o disparar) eventos sin importar el momento en que aparecen siendo que serán procesados de manera secuencial, sin poner en riesgo a los recursos.

Ciclo de eventos (event loop)

Es el encargado de procesar los eventos que se encuentran en la cola. Es importante aclarar que no todo evento requiere ser atendido pero por lo general, siempre es agregado a la cola de eventos. Luego al ser retirado de la cola, es necesario buscar el código manejador del evento y si este existe, ejecutarlo.

Un ejemplo simple de cola de eventos es el siguiente:

```
while (continuar)
{
    evento = retirar evento de cola
    if evento == salir
```

```
        continuar = false
    else if existe manejador para evento
        ejecutar manejador
}
```

La cola de eventos permite que los mismos sean agregados independientemente del contexto o momento en que se crearon, para que el sistema los atienda cuando posea los recursos (procesador, placa de video, etc) para hacerlo. Por otro lado, no es necesario que se declare la intención de manejar un evento de manera estática, pudiendo un manejador ser registrado y removido dinámicamente durante la ejecución del sistema.

Si bien, el procesamiento de eventos dependiendo del contexto o en paralelo es posible en entornos multiprocesador, los lineamientos de este tipo de sistemas aparecen con las primeras workstations, donde no era pensado tener tales recursos. Hoy en día, sigue siendo muy frecuente el uso de recursos compartidos al utilizar GUIs por lo que el procesamiento de eventos se suele realizar de esta forma.

Manejadores de eventos (event handlers)

Conceptualmente es una sección del código que sabe como responder a la aparición de un evento. El manejador del evento puede requerir al evento en cuestión para obtener de él cierta información de contexto (dónde se disparó, bajo qué condiciones, qué información extra agregó el usuario, etc) y poder atender al evento de mejor manera. También existen manejadores genéricos para eventos o todo lo contrario, manejadores particulares que sólo cumplen una función y esperan a un evento determinado para ejecutarla.

2. El esquema de GTK

GTK[1] es un acrónimo para GIMP ToolKit, es al día de hoy una de las plataformas para creación de Entornos Gráficos más populares en sistemas Linux/Unix. GTK nace como librería de soporte para el proyecto GIMP (GNU Image Manipulation Program) intentando reemplazar a las viejas librerías de imágenes de el sistema X de ventanas. Luego de grandes avances y continuas mejoras, se gana un lugar como proyecto independiente bajo el nombre de GTK+.

Separación de capas

Uno de los pilares de GTK se basa en la separación de capas e incumbencias de sus componentes. Este esquema permite que cada capa sea autocontenida y se especialice en cumplir con un determinado trabajo. De esta forma, el avance de los sistemas operativos, hardware, conceptos de usabilidad, etc. deben ser atacados solamente por la capa que se encarga de ellos y no por toda la plataforma. Esto permitió un avance continuo e independiente de cada capa y la posibilidad de intercambiar capas para utilizar GTK en diversas plataformas.

Xlib

Otorga el acceso directo al protocolo del servidor X de ventanas (X windows system) de manera simplificada. Es una interfaz de muy bajo nivel.

Glib

Así como GTK surge como librería interna del GIMP, GLib tiene sus orígenes como librería interna de GTK. Posee utilidades y estructuras de datos no orientadas al manejo gráfico: strings, fechas, internacionalización, listas, estructuras de hash, caches, manejo de memoria, etc.

GObject

Provee programación soporte para POO en lenguaje C. Surge como proyecto interno a GLib del cual aún no se separó por completo.

Gdk

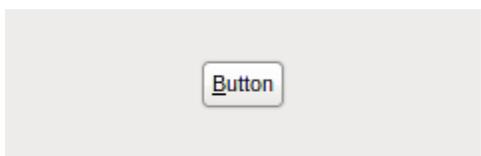
Acrónimo para GIMP DrawingKit, nace junto con GTK como capa intermedia entre éste y Xlib. Permite conceptualizar el acceso a la interfaz gráfica independientemente de la utilización de un entorno X o de los recursos de otro sistema operativo. Tanto es así que el intercambio de esta capa permite a GTK funcionar en entornos MS Windows sin mayores cambios.

Cairo - Pango

Proveen soporte para la manipulación vectorizada de gráficos (Cairo) y escritura de texto de alta calidad (Pango).

Widgets

Son los controles provistos por GTK. Encapsulan cierto comportamiento y estado proveyendo una abstracción para determinados conceptos generales de Entornos Gráficos. Ejemplos de widgets son: GtkButton, GtkEntry, GtkLabel, GtkWindow, GtkRuler, GtkTooltip, etc. Los widgets son codificados de tal forma que ocultan el manejo gráfico de bajo nivel y nos entregan una interfaz de utilización y extensión que es ajena a esos problemas.



Los widgets de GTK poseen una estructura de clases jerárquica. Todo elemento hereda de GtkWidget y este a su vez hereda de GObject. Cada hijo agrega funcionalidad y estado a su padre. Cada nueva rama en la jerarquía muestra una separación de comportamiento con respecto a sus ramas hermanas.

Un esquema reducido de la jerarquía de widgets de GTK es el siguiente:

```
GObject
|
GtkWidget
+GtkWidget
| +GtkMisc
| | +GtkLabel
| | `GtkImage
| +GtkContainer
| | +GtkBin
| | | +GtkFrame
| | | +GtkButton
| | | | +GtkToggleButton
| | | | | `GtkCheckButton
| | | | | `GtkRadioButton
| | | | `GtkOptionMenu
| | | +GtkItem
| | | | +GtkMenuItem
| | | +GtkWindow
| | | | +GtkDialog
| | | | | +GtkColorSelectionDialog
| | | | | +GtkFileSelection
| | | | | +GtkFontSelectionDialog
| | | | | +GtkInputDialog
| | | | | `GtkMessageDialog
| | | +GtkScrolledWindow
| | +GtkTable
| | +GtkTextView
| | +GtkToolbar
| | `GtkTreeView
| +GtkCalendar
| +GtkDrawingArea
| +GtkEditable
| | +GtkEntry
| | `GtkSpinButton
| +GtkRuler
| +GtkRange
| +GtkSeparator
+GtkTooltips
```

Los widgets, como cualquier clase, poseen constructores, destructores, métodos y atributos definidos. Veamos un ejemplo de utilización:

```
GtkWidget* button = gtk_button_new_with_label("botón");  
gtk_button_set_alignment(GTK_BUTTON(button), 0.0, 1.0);  
gtk_widget_destroy(button);
```

Es importante notar el orden de los casteos y los prefijos de los métodos que responden a las clases que los poseen.

Callbacks

Se trata de funciones que se pueden pasar como argumento para otro bloque de código. Un callback es cualquier conjunto de instrucciones ejecutable desde otro punto del código mediante su invocación.

En C, los callbacks no son más que punteros a funciones, en este caso necesitarán información acerca del evento que están manejando e incluso del widget que lo disparó. Como ejemplo, las funciones callbacks suelen tener la siguiente firma, recibiendo al widget que generó el evento y la información de contexto que se indicó al registrar el callback:

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

Por supuesto, se podría optar por otro orden de argumentos:

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

Aunque no es tan común, esta opción también es utilizada.

Eventos

Son los eventos propios de la plataforma gráfica que son capturados por GDK. Los eventos de este estilo incluyen información de bajo nivel relacionada con el manejo de los dispositivos físicos de entrada-salida (`button_press_event`, `key_press_event`, `scroll_event`, `expose_event`) o bien funciones propias del manejo de los componentes gráficos básicos del entorno gráfico (`destroy_event`, `delete_event`, `property_notify_event`, etc).

Señales

Es la emisión de eventos propios de los objetos GTK, aquellos que son emitidos por la plataforma GTK y no por el sistema de ventanas. En general se basan en eventos disparados por el segundo que son luego traducidos al esquema de componentes de GTK donde se les otorga un significado más cercano a la utilización de controles GTK. Ejemplos de señales comunes son: `clicked`, `selection-changed`, `select-child`, `focus-in`, etc.

Las señales nos dan el soporte necesario para colocar nuestros manejadores de eventos. Admiten semántica de suscripción/desuscripción para el usuario de las mismas y le brindan la posibilidad al widget de emitirlas cuando lo crean necesario. Además, acumulan suscripciones que pasarán a ser invocadas una tras otra al momento de la emisión. Como usuario de las señales, el método más importante es el **connect**. Demos un ejemplo

de su firma:

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

Esta función recibe en primera instancia al objeto que posee la señal, luego al nombre de la señal a conectar, la función callback que manejará el evento y por último, un puntero a información de contexto que será pasada al callback en la invocación.

El parámetro de retorno es el identificador de la conexión realizada que servirá en caso de pretender desconectar la señal en algún momento.

```
g_signal_connect (G_OBJECT(button), "clicked",  
G_CALLBACK(clicked_handler), (gpointer)"extra argument");
```

Conecta la señal **clicked** del botón con el callback **clicked_handler** pasando como información de contexto **"extra argument"**.

El bucle principal

Antes de colocar al sistema en su bucle de procesamiento de eventos, es necesario inicializar el framework GTK. Para ello ejecutamos la siguiente instrucción:

```
gtk_init(&argc, &argv);
```

Una vez realizado esto, podremos iniciar el ciclo de procesamiento de eventos mediante:

```
gtk_main();
```

Luego de esta instrucción, el control de la ejecución está en manos de GTK hasta que se le indique explícitamente que deseamos salir del bucle:

```
gtk_main_quit();
```

Es importante notar que toda operación de inicialización de widgets que deben ser mostrados, así como el disparo de nuevos hilos de debe realizar antes de llamar a `gtk_main` para que existan al momento en que GTK toma el control. Igualmente, es posible agregar nuevos widgets o cambiar a los ya existentes en tiempo de ejecución, pero mínimamente debe existir el widget básico que todo programa posee: la ventana.

3. Programando en GTK

Programa de prueba

El primer programa será el más básico que puedo realizar en GTK, simplemente inicializamos el framework y pasamos a esperar eventos. Este programa no posee forma de cerrar por lo que deberá ser abortado.

```
#include <gtk/gtk.h>
int main( int argc, char *argv[] )
{
    gtk_init(&argc, &argv);
    gtk_main();
    return 0;
}
```

Compilación

La compilación se realiza con una gran cantidad de flags y parámetros. Para facilitar esta tarea utilizamos el programa pkg-config [2] disponible en la mayoría de las distribuciones de Linux:

```
gcc prueba.c `pkg-config --libs --cflags gtk+-2.0`
```

Agregando una ventana

En este caso, modificamos el programa anterior para construir una ventana vacía y mostrarla al usuario:

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

int main(int argc, char* argv[])
{
    /* almacena un puntero a un widget */
    GtkWidget* ventana;
    /* inicia gtk, usando parámetros de la línea de comandos */
    gtk_init(&argc, &argv);
    /* creo a una nueva ventana */
    ventana = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    /* muestra la ventana */
    gtk_widget_show(ventana);
    /* comienza el loop de eventos */
    gtk_main();
    return 0;
}
```

Empaquetamiento de Widgets

Todo widget visual debe pertenecer a otro, y guardar una relación de parentesco con éste. La excepción a la regla son los objetos del tipo Window que sirven para contener a los distintos widgets. La vinculación de widgets no es automática, debe ser realizada de forma explícita mediante distintos métodos de distintas clases:

```
gtk_container_add(GtkContainer*, GtkWidget*)
gtk_box_pack_start(GtkBox*, GtkWidget*, gboolean expand, gboolean fill, guint padding);
```

```
gtk_box_pack_end(GtkBox*, GtkWidget*, gboolean expand, gboolean fill, guint  
padding);  
gtk_table_new(guint rows, guint columns, gboolean homogeneous)  
gtk_table_attach_defaults(GtkTable*, GtkWidget*, guint left_attach, guint right, guint  
top, guint bottom)
```

4. Programando en GTKMM

La programación en GTKMM[3] fue adaptada para contar con las ventajas semánticas y sintácticas que ofrece C++.

Programa sencillo

A continuación daremos un ejemplo de una simple ventana bajo GTKMM:

```
#include <gtkmm/main.h>  
#include "helloworld.h"  
int main (int argc, char *argv[])  
{  
    Gtk::Main kit(argc, argv);  
    HelloWorld helloworld;  
    //Shows the window and returns when it is closed.  
    Gtk::Main::run(helloworld);  
    return 0;  
}  
  
#include <gtkmm/button.h>  
#include <gtkmm/window.h>  
class HelloWorld : public Gtk::Window  
{  
public:  
    HelloWorld();  
protected:  
    void on_button_clicked();  
private:  
    Gtk::Button m_button;  
};  
  
#include "helloworld.h"  
#include <iostream>  
HelloWorld::HelloWorld()  
: m_button("Hello World") // creates a new button with label "Hello  
World".  
{
```

```
        set_border_width(10);
        m_button.signal_clicked().connect(sigc::mem_fun(*this,
&HelloWorld::on_button_clicked));
        add(m_button);
        m_button.show();
    }

void HelloWorld::on_button_clicked()
{
    std::cout << "Hello World" << std::endl;
}
```

Widgets propios

En GTKMM es posible utilizar widgets existentes y extenderlos de forma sencilla. Esto también era posible en GTK, pero el tiempo de codificación resultaba altísimo debido a la falta de soporte de orientación a objetos que ofrece el lenguaje.

```
#include <gtkmm/button.h>
class OverriddenButton : public Gtk::Button
{
protected:
    virtual void on_clicked();
}

void OverriddenButton::on_clicked()
{
    std::cout << "Hello World" << std::endl;

    // call the base class's version of the method:
    Gtk::Button::on_clicked();
}
```

Dialogos

Los dialogos son ventanas de mensaje, donde el usuario puede ser notificado sobre sucesos del sistema o consultado sobre distintas decisiones básicas. Ej.:

```
Gtk::MessageDialog dialog(*this, "This is a QUESTION MessageDialog",
false /* use_markup */, Gtk::MESSAGE_QUESTION, Gtk::BUTTONS_OK_CANCEL);
dialog.set_secondary_text("And this is the secondary text that explains
things.");

int result = dialog.run();
switch(result)
{
    case(Gtk::RESPONSE_OK) :
```

Manejo de memoria

El manejo de la memoria de los widgets puede ser explícito o implícito dependiendo del esquema elegido al crear el widget. Las distintas posibilidades son:

- Class Scope
- Function Scope
- new / delete
- Managed Widgets:

```
Gtk::Button* pButton = manage(new Gtk::Button("Test"));  
add(*pButton); //add aButton to MyWidget
```

Signals

El manejo de señales es muy similar al utilizado en GTK pero modificada para ser orientada a objetos. Las señales pasan a ser objetos que reciben functors [4] con el callback a ejecutar. Los tipos más importantes de functors son: ptr_fun, mem_fun y bind, aunque existen otros.

ptr_fun

Almacena la referencia a una función global. Ej.:

```
void on_button_clicked();  
  
int main()  
{  
    Gtk::Button button("Hello World");  
    button.signal_clicked().connect(sigc::ptr_fun(&on_button_clicked));  
}
```

mem_fun

Almacena la referencia a una función miembro de un objeto. Requiere la función y la instancia del objeto sobre el que será invocada. Ej.:

```
class some_class  
{  
    void on_button_clicked();  
};  
  
...  
some_class some_object;  
button.signal_clicked().connect( sigc::mem_fun(some_object,  
&some_class::on_button_clicked) );
```

bind

Transforma un functor que no requiere argumentos de ejecución en otro que sí los requiere. Permite indicar estos argumentos al momento de crear el functor. Ej.:

```
m_button1.signal_clicked().connect( sigc::bind<Glib::ustring>(
    sigc::mem_fun(*this, &HelloWorld::on_button_clicked), "button 1" ) );
virtual void on_button_clicked(Glib::ustring data);
```

5. Referencias

- [1] <http://www.gtk.org/>
- [2] <http://pkg-config.freedesktop.org/wiki/>
- [3] <http://www.gtkmm.org/>
- [4] <http://www.sgi.com/tech/stl/functors.html>