Tema VIII
Bibliotecas sobre XLib, Windows o Macintosh. (R-1.0)

Programación en Entornos Interactivos.

16 de febrero de 2012

Dpto. Lenguajes y Sistemas Informáticos
Universidad de Alicante

Resumen

Mecanismos no política: Toolkits. Algunos toolkits sobre X11. Xforms. Qt. Gtk+. Gtkmm. Libsigc. Libglade, Libglademm.

Mecanismos, no política: Toolkits. (I)

- El uso de X11 no impone una política de "apariencia" (aspecto de las ventanas, de los widgets, etc. . .), sino que proporciona unos mecanismos para utilizar el entorno gráfico.
- El nivel del "API" de X11 es "muy bajo".
- Esto favorece la aparición de bibliotecas por encima de X11, con un nivel lógico superior que hacen más fácil la programación con X11.
- Tradicionalmente a estas bibliotecas se les denomina toolkits.

Mecanismos, no política: Toolkits. (II)

Independientemente del toolkit que empleemos, el programa principal de nuestra aplicación constará siempre de tres fases:

Iniciación: En ella se 'inicia' la biblioteca o toolkit empleado y se analiza la línea de argumentos con la que ha sido llamada la aplicación.

Creación de widgets: En ella aprovechamos para crear (aunque no sean inicialmente visibles) todos aquellos widgets que necesitemos que estén creados justo al comenzar la aplicación.

Bucle de espera de eventos: En ella el toolkit empleado se dedica a esperar que se generen eventos de bajo nivel y los transforma en eventos de alto nivel.

Mecanismos, no política: Toolkits. (III)

- Los toolkits suelen proporcionar *elementos de interfaz de usuario* preconstruídos: botones, menús, listas de selección, etc. . .
- A los *elementos de interfaz de usuario* preconstruídos se les denomina: controles o también widgets.
- Un widget se caracteriza porque 'recoge' eventos de bajo nivel y los transforma en eventos de alto nivel. Los eventos de alto nivel existentes dependen del toolkit empleado.
- Un evento de alto nivel es aquel que está relacionado con un widget particular, por ejemplo: al hacer 'click' con el ratón sobre un 'widget botón' (evento de bajo nivel). . . éste widget 'captura' ese evento y lo transforma en uno de alto nivel: por ejemplo botón_pulsado.

Mecanismos, no política: Toolkits. (IV)

- Una vez hecha la conversión 'evento de bajo nivel' $\longleftrightarrow$ 'evento de alto nivel', los toolkits nos permiten, más o menos fácilmente, ejecutar código de nuestra aplicación en base al evento de alto nivel producido.
- Al código que se ejecuta se le denomina de forma general 'CallBack' y suele ser una función o un método de alguna clase invocado para un objeto concreto.
- La conexión entre los eventos de alto nivel de un widget y sus correspondientes callbacks se hace en la fase de inicio de la aplicación[1].

[1] De ahí el nombre de 'callback'.

Algunos toolkits sobre X11.

Son muchos los *toolkits* que se han desarrollado para facilitar la programación sobre X11, algunos de ellos son:

- WxWidgets, antes conocido como WxWindows[2]. El lenguaje de programación empleado con él es C++.
- Motif, el más utilizado entre las primeras aplicaciones gráficas en Unix. El lenguaje de programación empleado es C.
- FLTK, acrónimo de Fast Light Toolkit[3]. El lenguaje de programación empleado con él es C++.

Existen muchos otros, pero entre ellos destaca por su facilidad de uso y comprensión...

[2] Portabilidad del código: X11/Windows/Mac
[3] Portabilidad del código: X11/Windows/Mac

Xforms (I)

- Muy sencillo de usar:
  - Sólo consta de una biblioteca (libforms) y de una cabecera (forms.h) al más puro estilo 'C': datos y funciones que trabajan con ellos.
  - Un callback es una función de 'C' "normal y corriente".
  - Cada "widget" sólo puede tener asociado un "callback".
- Comenzó distribuyéndose sólo en formato binario.
- Ahora disponemos del código fuente: www.nongnu.org/xforms
- A partir de él evolucionó Fast Light Toolkit: www.fltk.org
- Incluye un constructor gráfico del interfaz de la aplicación: fdesign.

Xforms (II)

- Es muy sencillo de instalar.
- Basta copiar libforms.{a,so} y forms.h a sus directorios respectivos (de bibliotecas y cabeceras).
- Opcionalment copiar fdesign a un directorio donde haya ejecutables.
- Y compilar suministrando al compilador las opciones -I (para cabeceras) y -l (para bibliotecas) oportunas.

Xforms (III)

Las tres fases de toda aplicación se crean en Xforms así:

Programa principal con Xforms

```
int main(int argc, char *argv[]) {
     FD_VentanaPrincipal *fd_VentanaPrincipal;
     ...
  1) fl_initialize(&argc, argv, 0, 0, 0);
  2) fd_VentanaPrincipal = create_form_VentanaPrincipal();
     ...
     fl_show_form(fd_VentanaPrincipal->VentanaPrincipal,
                  FL_PLACE_CENTER,FL_FULLBORDER,
                  "VentanaPrincipal");
  3) fl_do_forms();
   return 0;
}
```

Xforms (IV)

Hemos de tener en cuenta que de fl_do_forms:

- Sólo se sale cuando:
  - Termina la aplicación
  - El usuario interactúa con un widget sin callback asociado.
- Existe la variante fl_check_forms, la cual realiza los mismos cometidos que `fl_do_forms` pero llevando a cabo una sola pasada por el bucle de espera de eventos y devolviendo el control fuera de ella inmediatamante.
- Esta última posibilidad nos permite crear aplicaciones que den al usuario la sensación de que pueden realizar varias tareas de forma simultánea.

Xforms (V)

La estructura de datos que representa un widget es:

Estructura FL_OBJECT

```
typedef struct flobjs_ {
    struct forms_ *form;          struct flobjs_ *child;
    void *u_vdata;                int pushed;
    char *u_cdata;                int focus;
    long u_ldata;                 int belowmouse;
    int objclass;                 int active;
    int type;                     int input;
    int boxtype;                  ...
    FL_Coord x, y, w, h;          int visible;
    ...                           int clip;
    struct flobjs_ *prev;         ...
    struct flobjs_ *next;         char *tooltip;
    ...                           ...
    struct flobjs_ *parent;     } FL_OBJECT;
```

Xforms (VI)

Y la que representa un *formulario* es:

Estructura FL_FORM

```
typedef struct forms_
{
    void *fdui;                    void *flpixmap;
    void *u_vdata;                 ...
    char *u_cdata;                 unsigned long icon_pixmap;
    long u_ldata;                  unsigned long icon_mask;
    ...                            ...
    char *label;                   int use_pixmap;
    unsigned long window;          int frozen;
    FL_Coord x, y, w, h;           int visible;
    ...                            int wm_border;
    struct flobjs_ *first;        } FL_FORM;
    struct flobjs_ *last;
    struct flobjs_ *focusobj;
```

Qt (I)

- Inicialmente desarrollada por la empresa noruega `TrollTech`, la cual fue adquirida posteriormente por Nokia .
- Disponible en dos versiones: 'Libre' y 'Profesional/Comercial'. En ambos casos tenemos el código fuente de la biblioteca.
- Disponemos de una extensa documentación en formato electrónico que se puede consultar en línea.
- Incluye un constructor gráfico del interfaz de la aplicación: qt-designer .
- La instalación se realiza siguiendo los pasos del archivo `INSTALL` que acompaña al código fuente o, mejor aún, empleando los paquetes preconstruidos para nuestra distribución.
- Se trata de un producto 'maduro', podemos encontrarlo en las versiones 3.3.x y 4.x.y .

Qt (II)

- El lenguaje de programación empleado es C++... con ciertos añadidos[4]: C* + +.
- La estructura interna de la biblioteca es la de una jerarquía de clases formada por varios árboles (distintas raíces) con herencia simple.
- Importante: al consultar la documentación lo que buscamos puede estar en alguna de las clases base de la actual.

[4] Para el soporte del mecanismo `signal/slot`.

Qt (III)

- Dado que la biblioteca extiende C++ con capacidades totalmente nuevas...
- Necesita 'procesarlas' para generar código C++ compatible con el estándar ISO del mismo.
- Esto se encarga de hacerlo la herramienta moc.
- Moc se genera como parte del proceso de obtención de la biblioteca, es decir, tenemos su código fuente también.
- Este paso de C*++ $\rightarrow$ C++ complica algo los `Makefile` que debemos construir, pero Qt proporciona la herramienta qmake que lo hace automático.

Qt Signal/Slot (I)

- Reemplaza al concepto de 'CallBack' que ya hemos visto.
- Proporciona:
  - Comprobación de tipos en tiempo de compilación.
  - Poder usar como función 'callback' un método de una clase además de una función normal.

Señal Es lo que emite un objeto cuando su estado cambia. El código de las señales lo genera automáticamente la aplicación moc.

Slot Es cómo se responde a la emisión de una señal, dicho de otro modo, el código de una función de nuestra aplicación que escribimos nosotros.

Qt Signal/Slot (II)

- La signatura de una señal -`signal`- debe coincidir con la del `slot` que se asocia.
- Una señal puede estar conectada a varios `slots`, y un `slot` puede estar conectado con varias `señales`.
- Incluso podemos conectar una `señal` con otra `señal`.

Qt Signal/Slot (III)

- Un `slot` puede pertenecer a la parte pública. privada o protegida de una clase.
- Para conectar una `señal` a un slot empleamos alguna de estas funciones:

Mecanismo Signal/Slot

```
bool QObject::connect(const QObject *sender, const char *signal,
                      const QObject *receiver, const char *member) [static]

bool QObject::connect(const QObject *sender, const char *signal,
                      const char *member ) const
```

- Veamos un ejemplo de cada una de ellas:

Mecanismo Signal/Slot: ejemplos

```
connect(scrollBar,SIGNAL(valueChanged(int)),lcdNumber,SLOT(display(int)));
connect(scrollBar,SIGNAL(valueChanged(int)),SIGNAL(anotherSignal(int)));
```

Qt Signal/Slot (IV)

Para emplear este mecanismo es necesario que la clase que emita señales y/o tenga slots:

- Derive directa o indirectamente de la clase QObject.
- Incluya al principio de su código la macro: Q_OBJECT.
- Cumpliendo estos requisitos podemos usar moc de esta forma: `moc fichero.h -o moc_fichero.cc`, y así obtener código C++ estándar.

Qt Signal/Slot (V)

Veamos ahora un ejemplo sencillo del uso de este mecanismo:

Signal/Slot: ejemplo sencillo

```
class StopWatch : public QObject {  |   void StopWatch::setValue( int v ) {
  Q_OBJECT;                         |     if ( v != m_val ) {
public:                             |        m_val = v;
  StopWatch() {m_val=0;}            |        emit valueChanged(v);
  int value() const {return m_val;} |     }
                                    |   }
public slots:
 void setValue(int);

signals:
 void valueChanged(int);

private:
  int  m_val;
};

//------- main() --------
StopWatch a, b;
QObject::connect( &a, SIGNAL(valueChanged(int)),
                  &b, SLOT(setValue(int))
                );
b.setValue( 11 );
a.setValue( 79 );
b.value();          // "b.m_val" ahora vale 79.
```

Qt Creación de widgets (I)

- Un widget es una clase que nos proporciona una serie de `señales` y `slots` preconstruidos, es decir, aprovecha el mecanismo signal/slot de Qt.
- Lo normal es que empleemos el constructor gráfico del interfaz qt-designer y le dejemos generar el código fuente de la *vista*.
- Pero si necesitamos crear un widget nuevo desde el código fuente, podemos crear un objeto instancia de la clase del widget que nos interesa, por ejemplo:

```
QPushButton* quit;
QScrollBar* sBar;
```

Qt Creación de widgets (II)

- Pero también podemos crear una clase nueva que represente a un widget por derivación o composición:

Creación de widgets por derivación/composición

```
class MyWidget : public QWidget {
public:
  MyWidget( QWidget* parent=0, const char* name=0 );
protected:
  void resizeEvent( QResizeEvent * );
private:
  QPushButton* quit;
  QScrollBar*  sBar;
  QLCDNumber*  lcd;
} objeto_MyWidget;
```

- Este es el tipo de código que genera qt-designer.

Algunas Clases Importantes

- La clase QApplication sirve para crear un objeto -y sólo uno- que representa a la aplicación:
  `QApplication a(argc,argv)`.
- Disponemos de la variable global `qApp` para hacer referencia al 'objeto aplicación' en cualquier parte del código.
- La clase QWidget representa a la clase base de todos los objetos que son parte del interfaz de usuario de la aplicación.
- La clase QDialog representa la clase base de todas las ventanas que usa la aplicación para establecer un diálogo con el usuario. Estos diálogos pueden ser `modales` o `no-modales`.

Qt Fases de una aplicación

Las tres fases de toda aplicación se crean en Qt así:

Programa principal con Qt

```
#include <qapplication.h>
#include <qpushbutton.h>

int main( int argc, char **argv ) {
 1) QApplication a( argc, argv );

 2) QPushButton hello( "Hello world!", 0 );
    hello.resize( 100, 30 );

    a.setMainWidget( &hello );
    hello.show();

 3) return a.exec();
}
```

Gtk+/Gtkmm (I)

- Gtk+ se desarrolla como toolkit libre para la aplicación gimp de tratamiento de imágenes. Gtkmm es la adaptación oficial del API de Gtk+ a C++.
- Se distribuye con licencia LGPL.
- Disponemos de una extensa documentación para Gtk+ y para Gtkmm en formato electrónico que se puede consultar en línea.
- Incluye un constructor gráfico del interfaz de la aplicación: glade .
- Tanto Gtk+ como Gtkmm se instalan siguiendo los pasos del archivo `INSTALL` que acompaña al código fuente o, mejor aún, empleando los paquetes preconstruidos para nuestra distribución.

Gtk+/Gtkmm (II)

- Se trata de productos estables, Gtk+ se encuentra en la versión 2.24.x y Gtkmm en la 2.24.x. Las versiones estables de Gtkmm aparecen al poco tiempo de las de Gtk+.
- Gtkmm2 tiene dos ABI's representados por `gtkmm-2.0` y `gtkmm-2.4` respectivamente, cuando se llama a `pkg-config`.
- Tanto Gtk+ como Gtkmm están a punto de cambiar a la versión 3. Estas versiones no serán compatibles con sus homólogas de la serie 2, ni a nivel binario (ABI) ni a nivel de código fuente (API)

Gtk+/Gtkmm (III)

- Lo que denominamos de forma general Gtk+ es un compendio de una serie de bibliotecas: Glib, GdkPixbuf, Gdk, Gtk, Atk y Pango.
- En el caso de Gtkmm ocurre lo mismo, se nos proporciona una adaptación en forma de una biblioteca de cada una de las anteriores: Glibmm, Gdkmm, Gtkmm, etc. . .
- Conclusión: Si conocemos como usar Gtk+ sabemos como emplear Gtkmm. Desde este instante nos referiremos sólamente a Gtkmm como toolkit a emplear.

Gtkmm (I)

- El lenguaje de programación empleado es C++... con soporte del mecanismo `signal/slot`[5].
- La estructura interna de la biblioteca es la de una jerarquía de clases formada por varios árboles (distintas raíces) con herencia simple.
- Estos árboles representan a cada una de las bibliotecas que hemos visto antes (glib, gdk, gtk, etc...). Cada uno de ellos reside en su propio espacio de nombres.
- Importante: al consultar la documentación lo que buscamos puede estar en alguna de las clases base de la actual.

[5] Sin alterar el lenguaje: emplea la biblioteca `libsigc`.

Gtkmm Signal/Slot (I)

- La idea de este mecanismo es la misma que la vista con Qt.
- Diferencia con Qt: no se añaden construcciones nuevas a C++, en su lugar se emplea la biblioteca libsigc .
- En libsigc los componentes 'signal' y 'slot' son clases que hay que instanciar ya que se trata de clases genéricas.
- La clase 'signal' es una clase parametrizada. Los parámetros del `template` son la 'signatura' de la función a invocar: el primero se corresponde con el tipo de resultado y los siguientes con el tipo de cada uno de los parámetros de la función o método a invocar.
- Gtkmm $<$ 2.4.x emplea `libsigc 1.2.x`, mientras que Gtkmm $>=$ 2.4.x usa `libsigc 2.0.x`...veamos un ejemplo de su uso.

Gtkmm Signal/Slot (II)

Ejemplo libsigc-1.2

```
#include <sigc++/sigc++.h>
#ifdef SIGC_CXX_NAMESPACES
using namespace std;
using namespace SigC;
#endif

int foo1(int i) { cout << "f("<<i<<");"
                       << endl; return 1;}
// (*) Debemos derivar de from SigC::Object.
struct A : public Object {
  int foo(int i)   { cout << "A::f("<<i<<");"
                          << endl; return 1;}
  void foov(int i) { cout << "A::fv("<<i<<");"
                     << endl;}
  A() {}
};
```

Gtkmm Signal/Slot (III)

Ejemplo libsigc-1.2

```
int main() {
  A a;
  // Lets declare a few signals.
  Signal1<int,int> sig1;    // int sig1(int);
  Signal1<int,int> sig2;    // int sig2(int);
  // The return type is allowed to be void.
  Signal1<void,int> sig1v;  // void sig(int);
  Signal1<void,int> sig2v;  // void sig2(int);

  // Connect to function foo.
  sig1.connect(slot(foo1));
  // Connect to method foo of object a.
  sig1.connect(slot(a,&A::foo));
  // Connect to signal 1 to signal 2.
  sig1.connect(sig2.slot());
```

Gtkmm Signal/Slot (IV)

Ejemplo libsigc-1.2

```
// We can do the same for the void signals.
sig1v.connect(slot(foo1v));
sig1v.connect(slot(a,&A::foov));
sig1v.connect(sig2v.slot());

sig2.connect(slot(foo2));
sig2v.connect(slot(foo2v));
...
```

Gtkmm Signal/Slot (V)

En el caso de `libsigc++-2.0` ha habido cambios, veamos los dos más importantes:

Ejemplo libsigc-2.0 : conexión con Funcion

```
#include <iostream>
#include <string>
#include <sigc++/sigc++.h>

void on_print(const std::string& str) {std::cout << str;}
int main() {
  sigc::signal<void, const std::string&> signal_print;

  signal_print.connect( std::ptr_fun(&on_print) );
  signal_print.emit("hello world\n");
  return 0;
}
```

Gtkmm Signal/Slot (VI)

Ejemplo libsigc-2.0 : conexión con Método

```
class Something : public sigc::trackable {
public:
  Something();
protected:
  virtual void on_print(int a);
  typedef sigc::signal<void, int> type_signal_print;
  type_signal_print signal_print;

};

Something::Something() {
  type_signal_print::iterator iter = signal_print
                                     .connect(sigc::mem_fun(this, &Something::on_print)

  signal_print.emit(2);
  //This isn't necessary - it's just to demonstrate how to disconnect:
  iter->disconnect();
  signal_print.emit(3); //Prove that it is no longer connected.
}

void Something::on_print(int a) {
  std::cout << "on_print recieved: " << a << std::endl;
}
```

Gtkmm Signal/Slot (VII)

- Al igual que ocurre con Qt las clases de Gtkmm nos proporcionan una serie de señales predefinidas. . . con las que podemos conectar nuestro código:

Ejemplo Gtkmm/libsigc-2.0

```
#include <gtkmm/button.h>
void on_button_clicked() {
  std::cout << "Hello World" << std::endl;
}

int main() {
  ...
  Gtk::Button button("Hello World");
  button.signal_clicked()
        .connect(sigc::ptr_fun(&on_button_clicked));
  return 0;
}
```

Gtkmm Fases de una aplicación

Las tres fases de toda aplicación se crean en Gtkmm así:

Programa principal con Gtkmm

```
#include <gtkmm.h>

int main(int argc, char *argv[])
{
  1) Gtk::Main kit(argc, argv);

  2) Gtk::Window window;
     window.show();

  3) Gtk::Main::run(window);

  return 0;
}
```

Gtkmm pkg-config (I)

- Para compilar una aplicación Gtkmm empleamos la herramienta pkg-config.
- `pkg-config` nos proporciona las opciones que debemos suministrar al preprocesador/compilador y al enlazador cuando trabajamos con una biblioteca.
- Para ello empleamos las opciones (juntas o por separado): –cflags y –libs, las cuales acompañan al nombre de la biblioteca.
- Ejemplo: `pkg-config [opciones] bibliotecas`.

Gtkmm pkg-config (II)

- Por ejemplo: `pkg-config --cflags gtkmm-2.0` proporciona:

  -DXTHREADS -I/usr/include/gtkmm-2.0 -I/usr/lib/gtkmm-2.0/include -I/usr/include/gtk-2.0 -I/usr/lib/sigc++-1.2/include -I/usr/include/sigc++-1.2 -I/usr/include/glib-2.0 -I/usr/lib/glib-2.0/include -I/usr/lib/gtk-2.0/include -I/usr/X11R6/include -I/usr/include/pango-1.0 -I/usr/include/freetype2 -I/usr/include/atk-1.0

- Mientras que `pkg-config --libs gtkmm-2.0` proporciona:

  -Wl,–export-dynamic -lgtkmm-2.0 -lgdkmm-2.0 -latkmm-1.0 -lgtk-x11-2.0 -lpangomm-1.0 -lglibmm-2.0 -lsigc-1.2 -lgdk-x11-2.0 -latk-1.0 -lgdk_pixbuf-2.0 -lm -lpangoxft-1.0 -lpangox-1.0 -lpango-1.0 -lgobject-2.0 -lgmodule-2.0 -ldl -lglib-2.0

- Se puede emplear `pkg-config` con más de una biblioteca simultáneamente. Podemos ver las que lo emplean con la opción –list-all.
- Recuerda que el parámetro de `pkg-config` `gtkmm-2.0` es para el ABI de *gtkmm2 <= v2.2.0*. Para versiones de *gtkmm2 >= 2.4.0* este parámetro de `pkg-config` se llama `gtkmm-2.4`.
- Su antecesor fue `gtk-config`.

Libglade/Libglademm (I)

- Libglade (y su adaptación a C++: Libglademm) es una biblioteca que permite 'leer' en tiempo de ejecución el interfaz de usuario de nuestra aplicación. . . también permite conectar las señales que hayamos especificado en glade.
- Se llama así porque es lo que emplea glade para leer sus archivos, los archivos XML que contienen la descripción del interfaz de usuario de la aplicación.
- Esto tiene una serie de ventajas:
  - No es necesario generar código fuente, desde la aplicación leemos directamente la definición del interfaz.
  - Al ser una representación textual, se puede comprimir mucho y ocupar poco espacio.
  - Podemos cambiar aspectos del interfaz sin tener que recompilar la aplicación.

Libglade/Libglademm (II)

- Veamos un ejemplo desde C:

Ejemplo de Libglade en C

```
#include <gtk/gtk.h>
#include <glade/glade.h>
void sig_handler(GtkWidget *widget, gpointer user_data) {
  g_printf("HOLA\n");
}

int main(int argc, char *argv[]) {
    GladeXML *xml;

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

    /* load the interface */
    xml = glade_xml_new("ejemplo_lg.glade", NULL, NULL);
    /* connect the signals in the interface */
    glade_xml_signal_autoconnect(xml);
    /* start the event loop */
    gtk_main();
    return 0;
}
```

Libglade/Libglademm (III)

- Mientras que parte del contenido del archivo '`.glade`' sería este:

ejemplo_lg.glade

```
<?xml version="1.0" standalone="no"?> <!--*- mode: xml -*-->
<!DOCTYPE glade-interface SYSTEM "http://glade.gnome.org/glade-2.0.dtd">

<glade-interface>

<widget class="GtkWindow" id="window1">
  <property name="border_width">10</property>
  <property name="visible">True</property>
  <property name="title" translatable="yes">Testing</property>
  <property name="type">GTK_WINDOW_TOPLEVEL</property>
  <property name="window_position">GTK_WIN_POS_NONE</property>
  <property name="modal">False</property>
  <property name="resizable">True</property>
  <property name="destroy_with_parent">False</property>

  <child>
    <widget class="GtkVBox" id="vbox1">
      <property name="visible">True</property>
      <property name="homogeneous">False</property>
      <property name="spacing">5</property>

      <child>
        <widget class="GtkLabel" id="label1">
```

Libglade/Libglademm (IV)

- La compilación se haría de este modo:
  `gcc ejemplo_lg.c -o ejemplo_lg ‘pkg-config --cflags --libs gtk+-2.0 libglade-2.0‘`
- Mientras que en C++ lo haríamos así:
  `g++ basic.cc -o basic ‘pkg-config --cflags --libs gtkmm-2.0 libglademm-2.0‘`
- `libglademm` también tiene dos ABI's, una más reciente que la `2.0`, de manera que si tenemos instalada esta mas nueva, llamaremos a `pkg-config` con el parámetro `libglademm-2.4`

Libglade/Libglademm (V)

- Veamos un ejemplo desde C++ con Libglademm:

Ejemplo de Libglade en C++

```
#include <libglademm/xml.h>
#include <gtkmm.h>
int main (int argc, char **argv){
  Gtk::Main kit(argc, argv);

  //Load the Glade file and instiate its widgets:
  Glib::RefPtr<Gnome::Glade::Xml> refXml;
  try
  {
    refXml = Gnome::Glade::Xml::create("basic.glade");
  }
  catch(const Gnome::Glade::XmlError& ex)
  ...
  //Get the Glade-instantiated Dialog:
  Gtk::Dialog* pDialog = 0;
  refXml->get_widget("DialogBasic", pDialog);
  if(pDialog)
  {
    //Get the Glade-instantiated Button, and connect a signal handler:
    Gtk::Button* pButton = 0;
    refXml->get_widget("quit_button", pButton);
    if(pButton)
    {
      pButton->signal_clicked().connect(SigC::slot(*pDialog, &Gtk::Dialog::hide));
  ...
```

Mas allá de libglade/libglademm

- Libglade/libglademm es una biblioteca 'externa' a Gtk+/Gtkmm.
- Se ha incorporado al conjunto de bibliotecas estandar con el nombre de GtkBuilder en 'C' y Gtk::Builder en 'C++'.
- Versiones recientes de Gtk+/Gtkmm aconsejan el uso de GtkBuilder en lugar de libglade/libglademm.
- Hay ligeras diferencias en los archivos XML de libglade y GtkBuilder. Existe un conversor: `gtk-builder-convert`.
- Versiones recientes de Glade permiten guardar el interfaz en uno u otro formato.