DESARROLLO MULTIPLATAFORMA CON WXWINDOWS

Israel Herraiz Tabernero

israel.herraiz@hispalinux.es

RESUMEN

wxWindows es una biblioteca de clases para C++ y *Python*, que permite el desarrollo de aplicaciones con interfaces gráficas de usuario de una manera rápida y sencilla. Su principal característica es que es multiplataforma. Desde su concepción se procuró que fuera completamente independiente del sistema operativo y del compilador. En la actualidad, las versiones para *Windows* y para *Linux* son completamente estables, y la versión para *MacOS* 8/9 y *MacOS* X está en una fase muy avanzada de desarrollo. Todas estas versiones se pueden conseguir en `www.wxwindows.org`. En este artículo crearemos un ejemplo "¡Hola mundo!" con la biblioteca para C++, y lo compilaremos en *Windows* y *Linux*.

1. INTRODUCCIÓN

Julian Smart comenzó a desarrollar *wxWindows* en 1992 en el *Artificial Intelligence Applications Institute*, de la Universidad de Edimburgo. *wxWindows* se distribuye bajo licencia *wxWindows Library License*, que es similar a la *GNU Library General Public License* pero que además permite usar la biblioteca para desarrollos comerciales (ya sean aplicaciones o modificaciones de la propia biblioteca), siempre y cuando estos desarrollo comerciales no usen ningún código distribuido bajo alguna licencia GNU.

Una característica importante de *wxWindows* es que intenta usar los controles nativos en cada plataforma. Si no existen controles de algún tipo en alguna plataforma, los emula (por ejemplo, el control *wxSpinCtrl* en *Linux + GTK+*). De este modo, se logra un aspecto homogéneo en cada plataforma (al contrario, por ejemplo, de lo que ocurre con las aplicaciones desarrolladas con *GTK* en *Windows*). En la actualidad, está soportadas las siguientes plataformas:

- *Windows* 3.1, *Windows* 95/98/NT/2000/XP
- La mayoría de las variantes de *UNIX* que usen *Motif*
- La mayoría de las variantes de *UNIX* que usen *GTK+*
- Mac

Además de las clases para el desarrollo de *GUI's*, *wxWindows* consta de una parte denominada *wxBase* que incluye clases como *wxString*, clases para el manejo de archivos y directorios de manera independiente del sistema operativo, tipos de datos independientes de la arquitectura (32 ó 16 bits), etc.

A la hora de compilar *wxWindows* podemos elegir que se enlace de manera estática con las aplicaciones, ó crear una biblioteca de enlace dinámico. Además, podemos compilar en modo *FINAL*, o en modo *DEBUG*. En cualquiera de los casos, los ejecutables obtenidos tienen un tamaño relativamente pequeño (que podemos disminuir hasta en un 70 % si usamos un compresor de ejecutables como *UPX*, disponible en `upx.sourceforge.net`).

2. WXHTML. UN MOTOR HTML PARA WXWINDOWS

Otra característica destacable de *wxWindows* es que incluye clases para visualizar archivos HTML. Por el momento, no implementa el estándar completo de HTML, pero en cualquier caso tiene una funcionalidad suficiente como para poder mostrar archivos sencillos, usarlo como editor de textos con coloreado de sintaxis, visor de archivos de ayuda (existen clases específicas para ello), etc.

Las clases *wxHTML* permiten además extender el conjunto de *tags* mediante etiquetas personalizadas, que el motor se encarga de interpretar y visualizar. Esta característica va más allá de simplemente mostrar texto o imágenes de una manera personalizada, sino que nos permite interactuar con el archivo HTML que se está mostrando. En definitiva, podemos implementar una aplicación cuya interfaz con el usuario sea un archivo HTML. *wxHTML* se encargan de generar los eventos necesarios para interacción del usuario con el archivo mostrado.

3. CREACIÓN Y VISUALIZACIÓN DE IMÁGENES

La clase *wxImage* puede cargar y crear varios formatos de imagen diferentes. Incluso, se puede extender la funcionalidad de esta clase si creamos el *handler* para un formato nuevo. En este momento, existen *handlers* para los formatos:

- TIFF, usando libTIFF
- JPEG, usando la biblioteca del *Independent JPEG Group*

- BMP
- GIF (sólo para visualizar, no para crear)
- PNG
- PCX
- PNM

Además también se pueden cargar iconos en formato *ICO* o en formato *XPM*. Es importante señalar que todos los formatos de imagen y todos los de iconos se pueden usar en cualquier sistema operativo. Por ejemplo, perfectamente puedes usar iconos en *XPM* para la barra de herramientas de una aplicación *Windows*.

4. CONTEXTUALIZACIÓN (LOCALIZACIÓN)

wxWindows incluye macros que permiten la traducción automática de una aplicación a diferentes idiomas. Los catálogos de traducción necesarios se crean con *GNU GetText* a partir del código fuente de la aplicación. Para que una cadena sea traducida, debe crearse con la macro `_(cadena)`. Por ejemplo:

```
wxMessageBox(
       _("Este texto se podrá traducir"),
       "Este no");
```

Para generar el catálogo, hay que usar la siguiente instrucción:

```
xgettext -C -k_ *.h *.cpp -o mi_catálogo.po
```

Para poder usar el catálogo de traducción, primero hay que compilarlo (generar el binario *mo*), con la instrucción (por supuesto, después de haber realizado las traducciones):

```
msgfmt mi_catálogo.po -o salida.mo
```

Después, en el código de nuestra aplicación, creamos un objeto *wxLocale* antes de crear la ventana principal:

```
wxLocale m_traductor;
m_traductor.Init("Español","es","C");
m_traductor.AddCatalog("mi_catálogo");
```

El primer parámetro del método `Init()` es una mera etiqueta que se usará en los mensajes de *wxWindows* que lo requieran (por ejemplo, los mensajes de error si no encuentra el catálogo). El segundo parámetro es la identificación del idioma que vamos a usar. Es muy importante, ya que el catálogo debe encontrarse en un subdirectorio que se llame como esta identificación. La tercera cadena indica las características culturales que se deben aplicar en la localización (es igual que el parámetro de la función `setlocale()` de la biblioteca estándar de C[1]).

Nuestra aplicación buscará el archivo `mi_catálogo.mo` dentro del subdirectorio *es* (en relación al directorio donde está el ejecutable). También buscará el directorio *es* en otros directorios predeterminados (los contenidos en la variable de entorno *PATH*, *Mis documentos* en *Windows* y nuestro directorio *home* en *UNIX*). Además de nuestro catálogo, también buscará uno llamado `wxstd.mo`, que contiene las traducciones de todos los mensajes que emite la biblioteca.

5. BIBLIOTECA ESTÁNDAR DE C++

wxWindows no emplea ninguna clase de la biblioteca estándar de plantillas, ni *namespaces* ni plantillas. La razón es que si emplearan se reduciría drásticamente el número de compiladores que se podrían usar con wxWindows, ya que no todos los compiladores soportan perfectamente estas características (e incluso, no todos tienen el mismo API para la biblioteca estándar de plantillas).

6. DOCUMENTACIÓN

6.1. Nuevos usuarios

El principal inconveniente de la documentación de *wxWindows* es que no incluye ningún tutorial para principiantes. Se ha concebido más como una documentación de referencia (eso sí, muy útil para el trabajo diario con *wxWindows*) que como una introducción a la biblioteca.

Si se han tenido contactos con otras bibliotecas de clases como *Qt* ó *MFC*, es fácil adaptarse a *wxWindows*, pero si no se tiene ninguna experiencia con alguna biblioteca similar, puede ser algo sufrido comenzar a programar con *wxWindows*. Por el momento, la mejor alternativa es plantearse una aplicación, e intentar desarrollarla con *wxWindows*. Cuando no sepamos cómo implementar exactamente una característica, podemos recurrir a la extensa colección de ejemplos, o incluso al propio código fuente de la biblioteca.

Está previsto el desarrollo de un libro de *wxWindows* por parte de los autores de la biblioteca. Por ahora, el libro está algo avanzado, pero todavía quedan bastantes puntos por escribir.

6.2. Usuarios experimentados

Una vez que ya tengamos experiencia con *wxWindows*, la documentación será una valiosa ayuda para el desarrollo de nuestras aplicaciones.

La referencia alfabética incluye la documentación de todos los métodos de cada clase, de los eventos que genera (y de cómo capturarlos), una breve descripción de cada clase y enlaces a otros puntos de interés dentro de la misma biblioteca (artículos u otras clases relacionadas).

Además, se incluyen una serie de artículos que ilustran sobre determinadas características de la biblioteca. Estos artículos incluyen explicaciones detalladas, código de ejemplo y enlaces a otros artículos relacionados y a la documentación de las clases implicadas.

Toda la documentación está disponible en formato HTML, WinHelp, MS HTML Help, PDF, PS y código fuente LaTeX. El único inconveniente es que la documentación sólo está disponible en inglés.

7. ¡HOLA MUNDO!

Después de un breve recorrido por algunas de las características de *wxWindows*, vamos a crear un pequeño ejemplo para mostrar cómo se crea una aplicación, cómo se contextualiza (localiza) y cómo se manejan los eventos.

Compilaremos la aplicación en *Windows* y *Linux* usando el compilador `gcc` (en *Windows* usaremos el compilador *Mingw*, disponible en `www.mingw.org`).

En primer lugar, crearemos el archivo con las interfaces de las diferentes clases (*hola_mundo.h*). Lo iremos comentando a la vez que mostramos el código:

```
// Archivo hola_mundo.h
#ifndef _HOLA_MUNDO_H_
#define _HOLA_MUNDO_H_

#ifndef TRUE
#define TRUE 1
#endif

#ifndef FALSE
#define FALSE 0
#endif
```

Definimos una constante para que no se incluya el archivo más de una vez (que es más que suficiente). También definimos las macros `TRUE` y `FALSE` para usar con las variables de tipo `bool`.

```
// Para los compiladores que soporten
// archivos 'h' precompilados
#include <wx/wxprec.h>

// Para todos los demás
#ifndef WX_PRECOMP
    #include <wx/wx.h>
#endif
```

Aquí simplemente incluimos los archivos de *wxWindows* necesarios. Si el compilador soporta archivos de cabecera precompilados, podemos usarlos para disminuir el tiempo de compilación (por ahora, sólo funciona con Borland C++ 5 y Visual C++ 6). Si no es así (es decir, no está definida la macro `WX_PRECOMP`, usamos un archivo de encabezado de texto plano.

```
// Para que funcione con
// el compilador de Borland
#ifdef __BORLANDC__
    #pragma hdrstop
#endif
```

Este es un extraño requerimiento del compilador Borland C++. Es un *pragma* (es decir, una opción específica del compilador), sin el cual no se puede compilar *wxWindows* ni ninguna aplicación que use esta biblioteca.

```
// Otros archivos requeridos
// Para la contextualización
#include <wx/intl.h>
```

En este archivo se define la clase *wxLocale*, que usaremos para traducir la aplicación a diferentes idiomas.

```
// Clase derivada de wxApp
class MiAplicacion : public wxApp
```

Toda aplicación tiene que crear una clase derivada de *wxApp*. Esta clase es especial, ya que no crearemos ninguna instancia de ella (de ello se encarga el *framework*).

```
{
public:
   // El método OnInit() actuará de función main()
   virtual bool OnInit();

protected:

   // Instancia de wxLocale para traducir
   // la aplicación
   wxLocale m_traductor;
};
```

En la interfaz de la clase *MiAplicacion* definimos el método *OnInit()*. Este método actuará como si fuera la función *main()* del programa, ya que como veremos después, las aplicaciones desarrolladas con *wxWindows* no tienen función *main()*. El atributo *m_traductor* se encargará de encontrar el catálogo de traducción (según nosotros le indiquemos), y de forma totalmente transparente para nosotros, traducirá todas las cadenas de texto que hayamos creado con la macro `_(cadena)`. Además, si junto a nuestro catálogo encuentra el de *wxWindows* (`wxstd.mo`), traducirá todos los mensajes de la biblioteca (que originalmente están en inglés).

Hay que señalar que hemos obviado el constructor, ya que no crearemos directamente ninguna instancia de esta clase. Todo el código de inicialización de la aplicación debe estar contenido en el método *OnInit()*.

```
// Ventana principal
class MiVentana : public wxFrame
{
public:
   // Constructor
   MiVentana(
      const wxString& titulo,
      const wxPoint& posicion,
      const wxSize& tam);
```

La ventana principal tiene que ser una instancia de una clase que derive de *wxFrame* (o de alguna clase derivada de *wxFrame*, como *wxDocMDIParentFrame*). El constructor que hemos definido acepta tres parámetros, que pasaremos al constructor de *wxFrame*. Los parámetros se han definido como constantes para cumplir con el *principio de menor privilegio* (si el constructor no necesita modificar estos parámetros, no debe tener la potestad de hacerlo), y se pasan por referencia para que no se haga una copia de los mismos cuando creemos una instancia de esta clase (de este

modo, se ahorra algo de memoria). Como se han definido como constantes, no existe peligro de que se modifiquen las variable originales que se pasan al constructor.

```
private:
   // Maneja el evento de la opción Salir
   void OnQuit(wxCommandEvent& event);
   // Maneja el evento de la opción Acerca de
   void OnAbout(wxCommandEvent& event);

   // Esta es una función virtual que hemos
   // sobreescrito (para mostrar un mensaje
   // antes de cerrar la ventana)
   void OnCloseWindow(wxCommandEvent& event);
```

Estos tres métodos se encargan de recibir los eventos que señalemos en la tabla de eventos. Los dos primeros simplemente responden a dos opciones del menú que crearemos. El tercer método, lo llamaremos cuando se vaya a cerrar a la ventana, con el fin de mostrar un mensaje de despedida. El método *OnCloseWindow()* es un método virtual de *wxWindow*, de la que deriva *wxFrame*. *wxFrame* ya tiene definido un método *OnCloseWindow()* con un comportamiento predeterminado (cierra la ventana ;-). Nosotros sobrescribiremos este comportamiento, añadiendo un mensaje de despedida.

```
   // Tabla de eventos
   // (no se incluye el punto y coma al
   // final porque es una macro)
   DECLARE_EVENT_TABLE()
};
```

La manera más cómoda de conectar eventos con métodos es mediante tablas de eventos. Si queremos usar una tabla de eventos, hay que llamar a esta macro dentro del cuerpo de la interfaz de la clase que quiera manejar los eventos. Posteriormente veremos cómo se implementa la tabla de eventos. Hay que señalar que no hay que incluir un punto y coma después de esta macro, ya que la macro no es más que un trozo de código que ya incluye su propio punto y coma.

```
#endif // end of _HOLA_MUNDO_H_
```

Y por fin terminamos el archivo de encabezado de nuestra pequeña aplicación.

Ahora comenzaremos la implementación de las clases definidas:

```
// Archivo hola_mundo.cpp

#include "hola_mundo.h"

// ID's para las opciones
// del menú
#define ID_ABOUT 100
#define ID_QUIT 101

// o también (cuestión de gustos)
// enum
// {
// ID_ABOUT = 100,
// ID_QUIT
// };

// Primer requerimiento:
IMPLEMENT_APP(MiAplicacion)
```

Esta macro hace que el método *OnInit()* de *MiAplicacion* se comporte como si fuera la función *main()* del programa. Dentro de este método deberemos crear y mostrar la ventana principal, e iniciar la instancia de *wxLocale* para que se traduzca la aplicación.

```
// Inicializamos la aplicación
bool MiAplicacion::OnInit()
{
    m_traductor.Init("Español","es","C");
    m_traductor.AddCatalog("messages");

    MiVentana *ventana = new MiVentana(
                            _("Hello world!"),
                            wxDefaultPosition,
                            wxDefaultSize);

   ventana->Show(TRUE);
   return TRUE;
}
```

Ya hemos dicho que este método juega el mismo papel que la función *main()* en otros programas. Lo primero que tenemos que hacer es iniciar la instancia de *wxLocale* (por supuesto, si no queremos usar traducciones, no hay que crear ninguna instancia, ni inicializarla). Después le pasamos el nombre del catálogo de las traducciones. En este caso, buscará el archivo `messages.mo` dentro del subdirectorio *es*. También intentará buscar el archivo `wxstd.mo` para traducir los mensajes de la biblioteca.

Después, creamos la ventana principal, instancia de *MiVentana*. Le pasamos el título de la ventana (está en inglés, y lo traduciremos al español), y la posición y el tamaño de la misma (en este caso, tamaño y posición por defecto). Por último mostramos la ventana y devolvemos `TRUE` para indicar que todo ha ido bien.

```
// Implementación de la tabla de eventos
BEGIN_EVENT_TABLE(MiVentana,wxFrame)
   EVT_MENU(ID_ABOUT,MiVentana::OnAbout)
   EVT_MENU(ID_QUIT,MiVentana::OnQuit)
   EVT_CLOSE(MiVentana::OnCloseWindow)
END_EVENT_TABLE()
```

Esta es la tabla de eventos. La comenzamos con la macro `BEGIN_EVENT_TABLE`, que acepta dos parámetros. El primero es el nombre de la clase que quiere manejar los eventos. El segundo parámetro es el nombre de la clase de la que nuestra clase hereda el comportamiento. En este caso, *MiVentana* hereda de *wxFrame*.

En la tabla de eventos hemos definido dos *handlers* para eventos de menú. Cuando se elija la opción del menú etiquetada con el valor `ID_ABOUT`, se llamará al *handler OnAbout*, que pertenece a la clase *MiVentana*. El otro evento de menú es idéntico.

La macro EVT_CLOSE se usa para capturar el evento que se produce cuando se va a cerrar la ventana (por ejemplo, cuando el usuario hace clic sobre el botón de cerrar la ventana, o cuando pulsa la combinación ALT+F4, ó cuando llamamos al método *Close()* de *wxFrame*). Como sólo hay un posible emisor del evento (la ventana que maneja los eventos) no hay que pasar la identificación del control emisor. El evento será manipulado por *OnCloseWindow* de la clase *MiVentana*.

```
// Constructor de MiVentana
MiVentana::MiVentana(const wxString& titulo,
                     const wxPoint &posicion,
                     const wxSize &tam)
          : wxFrame(NULL,-1,titulo,posicion,tam)
```

El primer parámetro del constructor de *wxFrame* es la ventana padre. Debe pasarse un puntero a un objeto del tipo *wxWindow*. Como en este caso es la ventana principal, no tiene ventana padre, le pasamos la macro NULL (un puntero a nada). No es necesario hacer un casting a wxWindow*, ya que la biblioteca incorpora ya todos los castings necesarios para que el puntero NULL sea del tipo wxWindow*.

El segundo parámetro es la identificación de la ventana. Debe ser un objeto del tipo *wxWindowID*, que a todos los efectos lo podemos tomar igual que un entero. Como en este caso no tenemos necesidad de asignarle ninguna identificación a la ventana, le pasamos -1, y la biblioteca le asignará una identificación por defecto.

El resto de parámetros son el título que aparecerá en la barra de título de la ventana, la posición y el tamaño.

```
{
wxMenu *mi_menu = new wxMenu(_("Today's menu"),
                             wxMENU_TEAROFF);
wxMenuBar *mi_barra_de_menu =
                  new wxMenuBar(wxMB_DOCKABLE);

mi_menu->Append(ID_ABOUT,
                _("About SuperApp"),
                _("Show about SuperApp dialog"));

mi_menu->Append(ID_QUIT,
            _("Close SuperApp"),
            _("Close this amazing application"));

mi_barra_de_menu->Append(mi_menu,
                         _("Today's menu"));

this->SetMenuBar(mi_barra_de_menu);
this->CreateStatusBar(2);
this->SetStatusText(_("Welcome to wxWindows!"));
}
```

Dentro del constructor, creamos un menú con su título, y el estilo que indica que podremos desvincular el menú de la ventana, para crear un menú flotante (este estilo sólo es válido en *wxGTK*, que es la versión para *Linux* de *wxWindows* usando la biblioteca *GTK*). Después creamos una barra de menú, que también podremos hacerla flotante (de nuevo, sólo vale en *wxGTK*).

Añadimos dos opciones al menú, con su identificación, su título, y el mensaje que aparecerá en la barra de status (que después creamos, con dos campos). También añadimos el menú a la barra de menú.

```
// Maneja el evento de la opción Salir
void MiVentana::OnQuit(wxCommandEvent& event)
{
this->Close(TRUE);
}
```

Cuando elegimos salir en el menú, se llama a este método que cierra la ventana. El método *Close()* no cierra la ventana, sino que genera un evento *wxCloseEvent*, que de manera predeterminada cierra la ventana. Pero como hemos sobreimplementado la función virtual *OnCloseWindow()*, si no cerramos nosotros explícitamente la ventana, nunca se cerrará. Eso es lo que vamos a hacer a continuación:

```
// Esta es una función virtual que hemos
// sobreescrito (para mostrar un mensaje
// antes de cerrar la ventana)
void MiVentana::OnCloseWindow(wxCommandEvent& event)
{
wxMessageBox(_("Goodbye world!"),
             _("Bye bye!"),
             wxOK|wxICON_INFORMATION,
             this);

this->Destroy();
}
```

En este método, mostramos un diálogo con un mensaje de despedida. La primera cadena es el texto del diálogo, la segunda el título del diálogo. El tercer parámetro es el estilo del diálogo. En este caso, tendrá un botón *Aceptar* y un icono de información. El último parámetro es la ventana padre (un puntero a *wxWindow*). En este caso, la ventana padre es la ventana principal (this). Después de mostrar el mensaje, cerramos la ventana. El método *Destroy()*, a diferencia de *Close()*, no genera ningún evento, sino que cierra definitivamente la ventana.

Una característica importante es que al cerrar una ventana (o en general al destruir cualquier instancia de una clase de *wxWindow*, o de alguna derivada) el comportamiento predeterminado es cerrar todos los objetos hijos que existen en esa instancia. Esta es la razón por la que no hemos implementado un destructor para nuestra clase, ya que *wxWindows* se encarga de realizar el *trabajo sucio*, y ordenadamente destruir todos los objetos. Si no existiera esta característica, y cerráramos el objeto padre sin haber destruido antes los hijos (con un destructor), obtendríamos probablemente un error de violación de segmento.

```
// Muestra el mensaje 'Hello world!'
void MiVentana::OnAbout(wxCommandEvent& event)
{
wxMessageBox(_("Hello world!"),
             _("Welcome to wxWindows!"),
```

```
            wxOK|wxICON_INFORMATION,
            this);
}
```

Este método muestra el mensaje “Hello world!”.

7.1. Compilación en Windows

La mejor manera de compilarlo en Windows es mediante un *makefile*:

```
WXDIR = $(WXWIN)

TARGET=hola_mundo
OBJECTS = $(TARGET).o

include $(WXDIR)/src/makeprog.g95
```

Tiene que estar definida la variable de entorno `WXWIN`, que contiene el directorio donde está instalada la biblioteca. Para compilar, es tan simple como:

```
make -f mi_makefile
```

Por supuesto, hemos supuesto que tenemos configurado bien el compilador (que se encuentra en la variable `PATH`, que hemos definido las variables de entorno `C_INCLUDE_PATH`, `CPP_INCLUDE_PATH` y `LIBRARY_PATH`) y que hemos compilado ya la biblioteca. Es importante, en cualquier caso, leer la documentación de *wxWindows*, ya que al compilador *Mingw* le faltan algunos archivos para poder compilar *wxWindows*. Se pueden encontrar en un archivo llamado `extra.zip`, en la página de *wxWindows*. Además, hay que crear un archivo de recursos (`hola_mundo.rc`) que contenga la línea

```
#include "wx/msw/wx.rc"
```

7.2. Compilación en Linux

Para compilar en Linux, es tan sencillo como escribir (cuidado, las tildes son hacia la derecha):

```
gcc -c ‘wx-config --cflags‘ hola_mundo.cpp
gcc ‘wx-config --libs‘ -o hola_mundo hola_mundo.o
```

7.3. Generación del catálogo de traducción

Si tenemos instalado *GetText*, escribimos en el directorio de las fuentes:

```
xgettext -C -k_ *.h *.cpp
```

Una vez traducidas las cadenas de texto, lo compilamos con:

```
msgfmt messages.po -o messages.mo
```

Copiamos el archivo generado al subdirectorio *es* dentro de nuestro directorio *home*, y ejecutamos el programa. Todas las cadenas que hayamos traducido en el catálogo, estarán traducidas en la aplicación. Opcionalmente también podemos compilar el catálogo `es.po` del subdirectorio `locale` de *wxWindows* y copiarlo junto a nuestro catálogo.

7.4. Ejecución y pantallazos

Si no creamos los catálogos de traducción, se generará un aviso de que no se han encontrado, y los textos aparecerán tal y como están en el código fuente. Aquí van algunos pantallazos (todos obtenidos con la versión *GTK*):

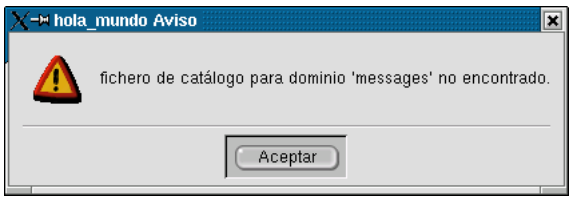

Fig. 1. Mensaje de error (en este caso, *wxGTK* se compiló en español porque la variable `LANG` estaba definida como *es*, y por tanto este es su idioma por defecto)

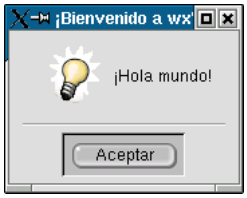

Fig. 2. Opción ID_ABOUT (traducida)

Fig. 3. Mensaje mostrado antes de salir (traducido)

8. BIBLIOGRAFÍA

[1] Gerardo Aburruzaga García, Inmaculada Medina Bulo, y Francisco Palomo Lozano, *La biblioteca estándar de C*. Servicio de Publicaciones de la Universidad de Cádiz, 1998.