Graphic user interfaces with Gtk

GTK IN SHEEP'S CLOTHING

TOBY PETERS

When developing graphical user interfaces, object-oriented programming can really demonstrate its strengths. The widely-used Gimp Toolkit is object-oriented in structure even though it is written in C. However, in order to make best use of it with C++ a wrapper is needed, such as Gtk--.

Anyone wanting to write a program with a graphical user interface for the X Window system is faced with the question of which library to use. Even if you have already decided upon C++ as the implementation language, there are still many different libraries to choose from. Several of the better-known C++ libraries are listed in Table 1, but many more exist. In this article we will introduce the library Gtk-- version 1.2.2. (Version 1.2.0 should not be used, incidentally, since this could cause problems at the next update).

Gtk-- is a C++ wrapper round the Gimp Toolkit Gtk+. In other words, this library provides the application programmer with C++ interfaces but uses the C library Gtk+ to do the real work such as drawing objects on the screen or waiting for events. Gtk+ has been used to create the Gnome Desktop. There is also a C++ wrapper around the Gnome widget set called Gnome-- which is built on Gtk--. In this article, however, we shall only be looking at Gtk-- itself.

Of libraries and wrappers

Why are we supposed to use a wrapper round a C library anyway? Because it is compatible with C it is possible to use any C library within a C++ application, and in this case, therefore, also Gtk+! A number of C++ applications do this, *Abiword* being a well-known example. When using a C library, however, there are a whole series of disadvantages and traps waiting for the unwary programmer.

To register callbacks a C library like Gtk+ expects a pointer to the function that is intended to be called. This means that the C++ programmer is restricted when using callbacks to global functions and static methods because only they are compatible with pointers to C functions. In a C++ application, however, we also want a method of registering a specific object as a callback. To achieve this, an adapter function must be used which will allow itself to be registered as a callback and then calls up the appropriate method of the object required. The adapter function must know to which object the method belongs and which it is supposed to call up.

Gtk+ can store any desired pointer to *void* as the attribute of a callback and can pass this pointer as a parameter when the callback function is invoked. In other words, we have to cast a pointer to the object initially as *void ** in order to be able to store it temporarily in the library during the registration of the callback. The pointer, which is received from the library when the callback is invoked, is now of the type *void **, and that leads to the next disadvantage.

Type checking must be disabled during compilation. In order to make use in the callback function of the object which is referenced by the pointer just mentioned, it has to be cast back and a check made in the process as to whether the referenced object also has the expected type, for example with *dynamic cast<T*>*. Type tests will therefore not take place until run time.

Another disadvantage of C libraries – and a real trap with wrappers – are exceptions, because exceptions cannot be thrown by the C library. If an exception is thrown in a callback, this exception must be caught again before it reaches a function of the C library, because exceptions can't go any further in functions written in C.

The small program from Listing 1 consists of two files, one C source file *c_exception.c*, which simulates a C library function, and a C++ source file *c_exception.cc*, which takes on the role of a C++ application. This program demonstrates the difficulties with exceptions in this arrangement:

```
gcc -c c exception.c
g++ -o c exception c exception.cc c exception.o
```

If it is compiled in this way, the text 0 caught"

Listing 1: Exceptions and C

```
/* File c exception.c */
void c func(void(*cxx func)(void)) {
 cxx func(); // calls up the transferred
}                      // function


/* File c exception.cc */
#include <iostream>
extern "C" {
  void c_func(void(*)(void));
}
void thrower (void) {
  throw int(0);
}
int main(int, char**) {
  try{
    c_func(&thrower);
  }
  catch(int i) {
   cout << i >> " caught." << endl ;
  }
}
```

Table 1: Some GUI Libraries for C++

Library	License	Platforms	Note
Gtk--	LGPL	Unix, Windows	C++ code wrapper for Gtk+
Vdk	LGPL	Unix	code wrapper for Gtk+
wxWindows	LGPL or wxWindows Library License, Version 3	Unix, Windows, Macintosh	code wrapper for Gtk+ and Motif
Qt	GPL or comercial licence	Unix, Windows	Very good documentation, basis of KDE

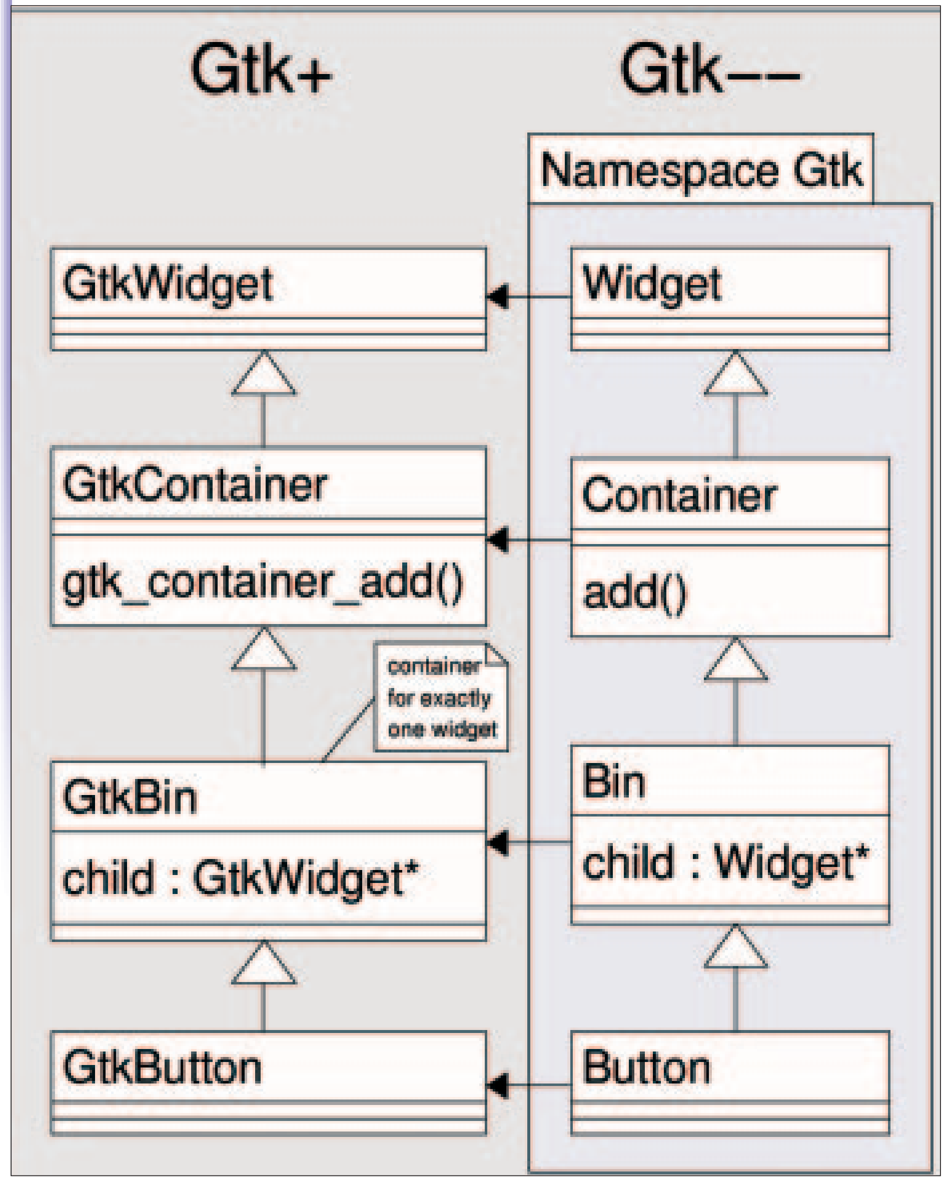

Fig. 1: Extract from the propagation hierarchies

will not be output. Instead, the program will be interrupted if the exception reaches the C function. Because a wrapper calls the functions of the wrapped library it cannot eliminate this problem either. This is a clear disadvantage of wrappers in relation to pure C++ libraries.

There are two ways of getting round this problem. Either you recompile the C library yourself, from scratch, with support for C++ exceptions (using special compiler settings) or you must allow for them in the program design. To use the former method with the GNU C compiler you use the switch *-fexceptions*.

To use this method with the program in Listing **1**:

```
gcc -fexceptions -c_c_exception.c;  g++ \
-o c_exception c_exception.cc c_exception.o
```

Compiled using this command, the exception is caught in *main*. If this method is used, however, users of the program must also recompile the C library, which is not something they would normally be expected to do. There are other disadvantages, too. After the exception, the C library doesn't release its resources with the result that the next time it is called it is in an unstable state.

It is therefore better to be flexible. Either do without exceptions altogether or take meticulous care in catching all exceptions before one of them reaches a function of the C library or the wrapper.

The basis: Gtk+

Gtk+ is a GUI library written in C but still object-oriented. The developers of Gtk+ have had a hell of a job to implement a class system with support for inheritance and virtual methods in C. Let's take a quick look at how inheritance works in Gtk+, and what consequences this has when using these classes.

Classes in Gtk+ are structs which contain as the first element the struct of the parent class:

```
struct Child {
  struct Parent parent;
  /* ... attributes ... */
};
```

A pointer to an instance of Child can be cast as a pointer to a Parent, because both pointers point to the same address in memory. But during inheritance there is something more to be taken into account: the list of virtual methods. Because of this there is no quick way to derive a new class from an existing class in the library.

An extract from the class hierarchy of Gtk+ widgets (GUI elements) and the corresponding hierarchy of Gtk– are shown in Figure 1. The methods of the Gtk+ classes are conventional C functions with the name and signature convention.

```
gtk <class name> <action>(Gtk<classname> *,
<Arguments>)
```

They are always called after their class and expect, as the first argument, a pointer to the object to which the method belongs. For example:

```
gtk_container_add(GtkContainer*, GtkWidget*)
```

which is a method of the class GtkContainer which packs any other desired widget into the container widget.

A button is also a container in Gtk, and can contain another widget. In this way a button can contain a label (text), a graphic or another container which combines both. Here's an example using Gtk+ of how a label can be integrated into a button:

```
GtkButton * button = /* ... */;
GtkLabel * label   = /* ... */;

gtk_container_add(GTK CONTAINER(button),
                               GTK_WIDGET(label));
```

This should now persuade every C++ programmer to use Gtk-- rather than Gtk+ directly. The widgets would in this case have to be explicitly cast using macros to the anticipated type, because otherwise the program cannot be compiled. A check on whether the casts are permissible at all will, however, only take place at run time.

In addition, it should be borne in mind that the method *add* was introduced as a method of GtkContainer, and not, for example, as a method of GtkBin (see the class hierarchy in Figure 1). The class in which a method was introduced will influence the name of the function which is to be called and the cast which is to be applied to the object.

Now the good news. Using Gtk-- the same code extract looks like this:

```
Gtk::Button button;
Gtk::Label  label(/* ... */);

button.add(label);
```

Aims of Gtk--

Gtk-- tries to wrap Gtk+ entirely. For each Gtk+ widget there is a C++ class. The class hierarchy of the C++ classes in Gtk-- corresponds exactly to the

libsigc++: Type-secure dynamic callbacks in Standard C++

Thanks to the compile-time type safety checks performed by the C++ compiler a great many program errors can be detected by the compiler. This type checking should also be applied to callbacks. To maximise reusability, classes from different parts of a program should be decoupled from one another as far as possible.

Type security and decoupling are two aims of program development that tend to conflict with one another. But with the aid of libsigc++, both can be achieved. Libsigc++ generates binder elements in standard C++ between the sender and receiver of a message, which the compiler can then check for compatibility. Sender and receiver need know nothing of each other; the only consideration is that the type of message (signature) which they send and receive must be identical.

The binding element on the sender side is the signal: SigC::Signal#<return type, parameter type1,…, parameter type#>. In this situation, the #-sign stands for the number of call-up parameters. Such a signal is a normal data type, an instance of which can be deposited in a class which is intended to send information.

The binding element on the receiver side is the slot. Slots treat function and method calls in the same way, and are generated by a "slot factory."

We have, then, the two sides, sender and receiver, who know nothing of one another. From a third place in the program, from which both the sender (Signal#<201>) as well as the receiver function or method are visible, the connection can now be made:

```
sig.connect(SigC::slot(&function)); sig.connect(SigC::slot(&object,&class:µethod));
```

SigC::slot is the "slot factory" mentioned. A class whose methods are to be called up in this way must be derived from SigC::Object. The connect calls are only allowed by the compiler if the signatures of the signal and slot match. The slots are called by the sender using the signal instance:

```
 r = sig(params);
```

Several slots can be connected using the same signal, or none at all. Several slots may be called up one after another: the return value of the last slot is then returned.

If this procedure isn't satisfactory it can be changed: The signals are additionally parameterised to the call-up signature using a marshaller class. An object of such a marshaller class is fed with the return values of the slots when they are called up, and from this generates a total return value. In addition to this, this object can decide after each slot call-up whether further slots are to be called up, or whether the call-up of further slots is to be suppressed. The marshaller class installed and used as standard in libsigc++ will ensure that all slots are called up and that the return value of the last slot is used as a total return value. If there is no slot at all connected to a signal it returns an instance of the return type created using the default constructor.

If another procedure is desired you can create your own marshaller class and use this as the last template argument of the signal instance.

The connections established using connect() can also be released if the return value of connect() is stored and its method SigC::Connection::disconnect() is called. Also, if the signal instance or the object is destroyed, the connection will be released automatically.

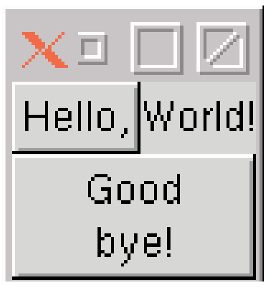

Fig. 2: The program from Listing 2

Listing 2: hello.cc -- "Hello World" for Gtk--

```
/* hello.cc
A "Hallo World" program for Gtk--.

This program initially defines a HelloWindow class derived from
Gtk::Window, which arranges two buttons and a label in a window
in the following way:

+--------+-------+
| Button | Label |
+--------+-------+  (To determine the arrangement,
|    Button     |   an HBox and a VBox are used)
+----------------+

In main () an instance of this class is generated and used.
*/

#include <iostream>

// Headerfile for Gtk::Window, a main window for X11 applications:
#include <gtk--/window.h>

// Headerfile for Gtk::HBox, Gtk::VBox, two containers, which arr➘
ange
// any desired widgets next to or on top of one another:
#include <gtk--/box.h>

#include <gtk--/label.h>   // Gtk::Label
#include <gtk--/button.h>  // Gtk::Button

// Headerfile for Gtk::Main.  Each Gtk-- application must instanc➘
e an
// object of this type precisely.
#include <gtk--/main.h>

// Instead of the individual header files, we would also have
// been able to write only #include <gtk--.h>.  This will allow
// all the header files to be read in.  For larger projects, howev➘
er,
// this is not allowed because of the extended translation time.

// The new class which is inherited from an empty application wind➘
ow:
class HelloWindow : public Gtk:Window {
  Gtk::VBox   vbox;
  Gtk::HBox   hbox;
  Gtk::Button hello_button; // the button top left
  Gtk::Label  world_label;  // the label top right
  Gtk::Button bye_button;   // the bottom button

protected:
  // We overwrite a virtual function which is called up automatica➘
lly
  // if it is intended that the application window should be closed
  // by the window manager:
  virtual gint
  delete event impl(GdkEventAny*);

public:
  // The constructor receives three strings as arguments, which
  // are indicated by the buttons (strings 1 and 3) or the label.
  HelloWindow(const string & hello string,
              const string & world string,
              const string & bye string);

  // Callback method, which should be called up
  // when the top left-hand button is clicked.
  void say hello(void); // Issues "Hello!" on the terminal.

  // Callback method, which should be called up
  // when the bottom button is clicked.
  voidsay goodbye(void); // Issues "Good Bye" and ends the program.
};

HelloWindow::HelloWindow(const string & hello string,
             const string & world string,
             const string & bye string)
// The HBox and the VBox are initialised by the default
// constructor. The buttons and the label are initialised with
// the strings which are passed over. (Button has a comfort
// constructor, which produces a label and inserts it in the butto➘
n.)
: hello button(hello string),
  world label (world string),
  bye button  (bye string
{
  // Insert the top left button into the HBox:
  hbox.add(hello button);
  // All widgets must be made visible with show():
  hello_button.show();

  // Insert the top right-hand label into the HBox next to the but➘
ton:
  hbox.add(world label);
  world_label.show();

  // Insert the HBox into the VBox:
  vbox.add(hbox);
  hbox.show();

  // Insert the bottom button beneath the HBox, which contains th➘
e other button and the label:
  vbox.add(bye button);
  bye_button.show();

  // Insert the VBox into the application window (*this):
  add(vbox);
  vbox.show();

  // Connect the buttons with the callbacks (see box about libsig➘
c++):
  hello_button.clicked.connect(SigC::slot(this,&HelloWindow::sa➘
y hello));
  bye_button.clicked.connect(SigC::slot(this,&HelloWindow::say g➘
oodbye));
}


void HelloWindow::say hello(void) {
  cout << "Hello!" << endl;
}

void HelloWindow::say goodbye(void) {
  cout << "Good bye!" << endl;
  Gtk::Main::quit(); // Exit event loop
}

// The window is to be closed by the window manager:
gint HelloWindow::delete event impl(GdkEventAny*) {
  Gtk::Main::quit(); // Exit event loop

  // Return value true prevents the window being destroyed immedia➘
tely
  // after the return. (In this case, this would not actually matt➘
er,
  // because the program will then be ended in any event.
  return true;
}

int main(int argc, char ** argv) {
  // An instance of Main is created.  It seeks through the comman➘
d line
  // elements and removes those which are processed by the Toolkit.
  Gtk::Main kit(argc, argv);

  // Create and display an instance of the application window.
  HelloWindow hello("Hello,", "World!","Good" "\n" "bye!");
  hello.show();

  // Event loop.  The callbacks are called up from this function.
  kit.run();

  return 0;
}
```

class hierarchy of Gtk+. Your own classes can be derived from these classes by inheritance with no problem.

Type safety at compile time is a declared aim of Gtk--. For callbacks the signal-slot system of the library *libsigc++* (see box) guarantees type safety. This system was initially developed as a part of Gtk-- and can now be used independently of it.

It is intended that Gtk-- should remain as slim as possible, but it should not be too thin. If the API of Gtk+ can support a C++ specific improvement, this will be implemented. Examples of this are the consistent use of *std::string* instead of *char **, and the possible use of the Standard Template Library's container API for every type of container in Gtk.

Hello World

It's time for the obligatory example program. Listing 2 shows a complete "Hello World" program for Gtk--. A screen shot of this can be seen in Figure 2.

The program is compiled using:

```
g++ gtkmm-config --cflags \
  --libs  -o hello hello.cc
```

There are a number of points of interest in the source code:

Packed in the container

At no point in the source code is the size of widgets (in pixels) specified. In Gtk, container widgets are always just sufficiently large enough to accommodate the widgets contained in them. The widgets located in a container in turn, as a rule, expand so that they take up all the space which the container provides for them.

In the example, this means: "The button with the text 'Hello' is just large enough to be able to accommodate the label, whose size is determined from the font and the contents. The HBox which contains this button and the label next to it must now be at least as high as the button. The button fills the entire height of the HBox and is placed at the centre of it. The width of the HBox is determined from the widths of the button and the label. Because this HBox is inserted into a VBox, the VBox must also have at least the same width. The button with the inscription "Good bye!" contains a two-line label and is therefore taller than the other widgets. It occupies the full width of the VBox.

If the user now enlarges the window displayed by the program, the widgets automatically take the additional space. There are still a few refinements which can be done differently when packing the container in order to achieve specific effects, but there isn't the space to go into greater detail here. Refer instead to the Gtk-- Tutorial on the Web, which describes in detail how widgets are packed in boxes and presents all the other containers.

Dialogs

In Listing 2 all the widgets of the program are defined as data members of one class, this class being derived from the Window class. Because it is easy in C++ to derive a new class, this is the recommended method for creating a dialog. All the widgets of a dialog, or, if the dialog becomes too complex, all the widgets of a logical sub-unit of the dialog, are contained in one class. This class is derived from the container widget which contains all the other widgets.

For a complete dialog, then, you can derive a class from a *Gtk::Window*. It is possible, for example, to derive part of a dialog based on an index card widget (*Gtk::Notebook*) from a box or a table (*Gtk::Table*) if all the other widgets are located within this box or table. In this way you can create specialised dialog widgets with the desired functionality. The file selection dialog supplied with Gtk, for example, was created in this way (although at the level of Gtk+.)

If the new classes contain all the other widgets as data members, as in the example program, they can become very large. The size can be problematic in low-memory situations if the address space of the process is fragmented and there is no sufficiently large single memory area available to create an instance of the class. One way out of this is to work with pointers to the widgets instead of with the widgets themselves, creating these dynamically in the constructor and destroying them again in the destructor.

Gtk-- (or libsigc++) provides a mechanism which makes the handling of the dynamically created widgets much easier: the function *manage()*. Many widgets, once they have been created and placed in a container, are no longer referred to in the source code of the program. In this case, instead of storing a pointer to the widget just in order to be able to destroy it again later, this task can be left to the library. If you give the function *manage()* a pointer to a widget, it will be marked in such a way that it will automatically be deleted by the library when the

container which holds the widget is destroyed. *manage* is defined in *$PREFIX/include/sigc++/object.h* (read in automatically from the Gtk– header files) as *template <class T> T* manage(T*)*, so there may be no need to store a pointer to the object created temporarily in the constructor.

```
somecontainer.add(*manage  (new Gtk::SomeWi➘
dget(/*…*/)));
```

inserts the newly created widget directly into the container. A version of the "Hello World" program, which uses *manage* is provided on the CD with this issue of the magazine.

Callbacks

For callbacks, the signal-slot system of the library libsigc++ was developed (see the box "libsigc++"). The widgets react to user interaction by emitting signals. The button, for example, contains a signal such as *Signal10<void> clicked*, which is sent when the user clicks the button using the left-hand mouse button. In Listing 2 in the constructor of HelloWindow, the last two statements connect these *clicked* signals with other methods of the class.

There is yet another way of calling up callbacks: virtual methods. Each widget in Gtk-- which can emit a signal also contains a virtual method which is called up each time this signal is emitted. The name convention for these methods is *<signalname> impl*.

If you are interested in a signal from a widget from which you are deriving a new class it may be simpler to override this virtual method in order to obtain the desired functionality. Unlike with signals and slots, event and reaction are then securely coupled to one another. This might impair the reusability of the derived class. In the example program the method *delete event impl* is overridden in this way such that the program is terminated when the user closes the window.

If methods are overridden it is important to take account of the fact that important functions of Gtk-- are carried out in these methods. It may therefore be necessary to call the corresponding method of the parent class explicitly from an overridden method in order for everything to work as it is supposed to. If you don't want to look into this any further, take it as a general rule of thumb to end the names of methods with ** event impl* in order to call the method of the parent class.

With style: Styles

An object of type *Gtk::Style* is allocated to each widget, which determines the appearance of the widget. As a rule, several if not all the widgets share the same style object. You should never change the style of a widget if you don't know which other widgets are also using this style.

New styles can be created by copying an existing style using *Gtk::Style * Gtk::Style:Copy() const* or creating standard style using *static Gtk::Style * Gtk::Style:Create()*. In the new style you can then change colours and fonts before assigning it to other widgets.

The colours which a style defines are principally foreground colour ("fg"; for widgets with editable text "base") and background colour ("bg"; for widgets with editable text "text"). A widget can adopt different states. For each state, each of the colours can be different. GTK_STATE_NORMAL denotes the basic state; with GTK_STATE_PRELIGHT the mouse pointer is located above the widget, as is the case with GTK_STATE_ACTIVE, although in this case the left-hand mouse button is also pressed. In addition to this, there is GTK_STATE_INSENSITIVE and GTK_STATE_SELECTED.

It is possible to play around with styles, colours and fonts in the program. This would mean that the colours and fonts would be hard-coded. There is a simple mechanism in Gtk which allows the appearance of the widgets to be determined at run-time: *gtkrc* files. Using these files, users are given the ability to change the look of programs to suit themselves.

Self-determination for all!

In gtkrc files you can define "styles" and then assign them to a group of widgets. Styles are defined here as follows:

```
style "<name>" [ = "<optional other sty
le name>"]
{ … }
```

where … is a selection of instructions:

```
bg[<state>] = {<r>,<g>,<b>}
fg[<state>] = {<r>,<g>,<b>}
base[<state>] = {<r>,<g>,<b>}
text[<state>] = {<r>,<g>,<b>}
bg pixmap[<state>] = "<xpm filename>"
font = "<fontname>"
fontset = "<fontnames>"
```

<state> designates one of the widget states; possible values are NORMAL, PRELIGHT, ACTIVE, INSENSITIVE and SELECTED. <r>,<g>,<b> are colour values between 0.0 and 1.0. It's important to ensure that a decimal point appears in the values otherwise the value will not be recognized and zero will be used (i.e. write 1.0 and not 1). Xpm-files are sought in directories specified by:

```
pixmap path = "<directory1:directory2:dir➘
ectory3...>"
```

at the start of the *gtkrc* file. Finally, styles defined in this way are allocated to a group of widgets. The following assignments apply:

```
class         <classname>  style  <stylename>
widget class  <class path> style  <stylename>
widget        <path>       style  <stylename>
```

In the above, <stylename> is the name of a defined style and <classname> is the name of a class to which a *is a* designation pertains.

Examples of class names are "Button" or "Widget". <class path> is a list of exact class names, separated by dots, of the containers in which a widget in this sequence is contained (for example "Window.HBox.Button.Label"). A <class path> usually starts with "Window.*". <path> is the same as <classpath>, unless a name has been given to one of the widgets using the method *set name()*, in which case this name should be used in the <path> instead.

The class names "Button", "Widget", etc. are a feature of Gtk--. The corresponding class names of the Gtk+ widgets in a C program are "GtkButton", "GtkWidget" and so on.

<class name>, <class path>, and <path> may contain the wildcards * and ?.

Putting it to use

If you intend that a widget should be able to be configured in this way, a name which is as unambiguous as possible should be allocated to it in the program, for example:

```
my button.set name("Hello Button");
```

In the *gtkrc* file, which is provided with the program, the desired style is defined and the button allocated to it:

```
style "hello button style" {
  bg[PRELIGHT] = { 0.9, 0.2, 0.2 }
}
widget "*Hello Button" style "hello button st↵
yle"
```

The asterisk here is a wildcard for all containers which might contain the button. A style is allocated to a widget at the moment at which its *set name()* method is called, or when it is packed into a container. Before this happens, the program must read in the desired *gtkrc* file:

Of course, the documentation for the application will need to specify which widgets have which names so that the user knows what to change in the *gtkrc* file in order to obtain the desired result. Refer to the GNOME developer web site for more information on *gtkrc* files.

More documentation

There are more than 80 widget classes in Gtk– and the Tutorial makes an effort to present all of them. In this article that is unfortunately not possible. Anyone whose appetite has been whetted by this article should read this Tutorial as the next step.

What other documentation is there? First, there are the reference pages which are automatically produced when installing Gtk--. This reference is unfortunately still very incomplete. After installation it is found under <gtkmm-source>/docs/gtk/class index.html. There is also an example program for almost every widget which demonstrates its use. The example programs are located under <gtkmm-source>/examples.

Because Gtk-- is just a wrapper it is possible to make something of a start with the documentation for Gtk+. The book "Gtk+/Gnome Application Development" by Havoc Pennington is also worth having. It can be downloaded or read on the Net, though we would recommend buying a copy of the printed version.

The book "Developing Linux Applications with Gtk+ and Gdk" by Eric Harlow is rather heavy going but there are a number of low-level situations which it explains well. Before buying a copy, be sure to leaf through it carefully to get an overall impression, because the start of the book is really intended for C programmers and has nothing of direct relevance to Gtk+.

It is often helful to take a look at existing applications which use Gtk--. Terraform is a program for creating fractal landscapes and cardwords is a crossword puzzle game. Other application examples can be found on the Web.

Gtk-- is not only usable, it is being used. The project has been around for almost as long as Gtk+ itself. It has come a long way from its origins as a simple mapping of all the C functions of Gtk+ to become a sophisticated wrapper without which you will never want to use Gtk with C++ again. ■

Info

Homepage of Gtk--: *http//gtkmm.sourceforge.net/*
Homepage of Vdk: *http://vdkbuilder.sourceforge.net/vdk.html*
Homepage of wxWindows: *http://www.freiburg.linux.de/-wxxt/*
Homepage of V: *http://www.objectcentral.com/vgui/vgui.htm*
Homepage of Qt: *http://www.trolltech.com/products/qt.html*
For compatibility of QPL and GPL:
http://www.gnu.org/philosophy/license-list.html
Homepage of Fltk: *http://www.fltk.org/*
List of X11 libraries:
http//sal.kachinatech.com/F/5/index.shtml
List of GUI libraries: *http://atai.org/guitool/*
Homepage of Abiword: *http://www.abisource.com/*
Homepage of libsigc++: *http://libsigc.sourceforge.net/*
Tutorial for Gtk--:
http://gtkmm.sourceforge.net/tutorial/tutorial.html
gtkrc:
http://developer.gnome.org/doc/API/gtk/gtk-resource-files.html
Havoc Pennington: "Gtk+/Gnome Application Development"
http://developer.gnome.org/doc/GGAD/GGAD.tar.gz
Homepage of Terraform: *http://terraform.sourceforge.net/*
Homepage of cardwords: *http://www.crosswinds.net/-cardwords*
Other example applications:
http://gtkmm.sourceforge.net/extra.html#applications

■