Programming with gtkmm 3

Programming with *gtkmm* 3

1. Introduction

1.1. This book

This book explains key concepts of the *gtkmm* C++ API for creating user interfaces. It also introduces the main user interface elements ("widgets"). Although it mentions classes, constructors, and methods, it does not go into great detail. Therefore, for full API information you should follow the links into the reference documentation.

This book assumes a good understanding of C++, and how to create C++ programs.

We would very much like to hear of any problems you have learning *gtkmm* with this

document, and would appreciate input regarding improvements. Please see the Contributing section for further information.

1.2. gtkmm

gtkmm is a C++ wrapper for GTK+, a library used to create graphical user interfaces. It is licensed using the LGPL license, so you can develop open software, free software, or even commercial non-free software using *gtkmm* without purchasing licenses.

gtkmm was originally named gtk-- because GTK+ already has a + in the name. However, as -- is not easily indexed by search engines the package generally went by the name *gtkmm*, and that's what we stuck with.

1.2.1. Why use *gtkmm* instead of GTK+?

gtkmm allows you to write code using normal C++ techniques such as encapsulation, derivation, and polymorphism. As a C++ programmer you probably already realise that this leads to clearer and better organized code.

gtkmm is more type-safe, so the compiler can detect errors that would only be detected at run time when using C. This use of specific types also makes the API clearer because you can see what types should be used just by looking at a method's declaration.

Inheritance can be used to derive new widgets. The derivation of new widgets in GTK+ C code is so complicated and error prone that almost no C coders do it. As a C++ developer you know that derivation is an essential Object Orientated technique.

Member instances can be used, simplifying memory management. All GTK+ C widgets are dealt with by use of pointers. As a C++ coder you know that pointers should be avoided where possible.

gtkmm involves less code compared to GTK+, which uses prefixed function names and lots of cast macros.

1.2.2. *gtkmm* compared to Qt

Trolltech's Qt is the closest competition to *gtkmm*, so it deserves discussion.

gtkmm developers tend to prefer *gtkmm* to Qt because *gtkmm* does things in a more C++ way. Qt originates from a time when C++ and the standard library were not standardised or well supported by compilers. It therefore duplicates a lot of stuff that is now in the standard library, such as containers and type information. Most significantly, Trolltech modified the C++ language to provide signals, so that Qt classes cannot be used easily with non-Qt classes. *gtkmm* was able to use standard C++ to provide signals without changing the C++ language. See the FAQ for more detailed differences.

1.2.3. *gtkmm* is a wrapper

gtkmm is not a native C++ toolkit, but a C++ wrapper of a C toolkit. This separation of interface and implementation has advantages. The *gtkmm* developers spend most of their time talking about how *gtkmm* can present the clearest API, without awkward compromises due to obscure technical details. We contribute a little to the underlying GTK+ code base, but so do the C coders, and the Perl coders and the Python coders, etc. Therefore GTK+ benefits from a broader user base than language-specific toolkits - there are more implementers, more developers, more testers, and more users.

2. Installation

2.1. Dependencies

Before attempting to install *gtkmm* 3.0, you might first need to install these other packages.

- *libsigc++ 2.0*
- *GTK+ 3.0*
- *cairomm*
- *pangomm*
- *atkmm*

These dependencies have their own dependencies, including the following applications and libraries:

- *pkg-config*
- *glib*
- *ATK*
- *Pango*
- *cairo*
- *gdk-pixbuf*

2.2. Unix and Linux

2.2.1. Prebuilt Packages

Recent versions of *gtkmm* are packaged by nearly every major Linux distribution these days. So, if you use Linux, you can probably get started with *gtkmm* by installing the package from the official repository for your distribution. Distributions that include *gtkmm* in their repositories include Debian, Ubuntu, Red Hat, Fedora, Mandriva, Suse, and many others.

The names of the *gtkmm* packages vary from distribution to distribution (e.g. *libgtkmm3.0-dev* on Debian and Ubuntu or *gtkmm30-devel* on Red Hat Fedora), so check with your distribution's package management program for the correct package name and install it like you would any other package.

> The package names will not change when new API/ABI-compatible versions of *gtkmm* are released. Otherwise they would not be API/ABI-compatible. So don't be surprised, for instance, to find *gtkmm* 3.2 supplied by Debian's *libgtkmm3.0-dev* package.

2.2.2. Installing From Source

If your distribution does not provide a pre-built *gtkmm* package, or if you want to install a different version than the one provided by your distribution, you can also install *gtkmm* from source. The source code for *gtkmm* can be downloaded from http://www.gtkmm.org/.

After you've installed all of the dependencies, download the *gtkmm* source code, unpack it, and change to the newly created directory. *gtkmm* can be built and installed with the following sequence of commands:

```
# ./configure
# make
# make install
```

> Remember that on a Unix or Linux operating system, you will probably need to be `root` to install software. The `su` command will allow you to enter the `root` password and have `root` status temporarily.

The `configure` script will check to make sure all of the required dependencies are already installed. If you are missing any dependencies, it will exit and display an error.

By default, *gtkmm* will be installed under the `/usr/local` directory. On some systems you may need to install to a different location. For instance, on Red Hat Linux systems you might use the `--prefix` option with configure, like so:

```
# ./configure --prefix=/usr
```

> You should be very careful when installing to standard system prefixes such as `/usr`. Linux distributions install software packages to `/usr`, so installing a source package to this prefix could corrupt or conflict with software installed using your distribution's package-management system. Ideally, you should use a separate prefix for all software you install from source.

If you want to help develop *gtkmm* or experiment with new features, you can also install *gtkmm* from git. Most users will never need to do this, but if you're interested in helping with *gtkmm* development, see the Working with gtkmm's Source Code appendix.

2.3. Microsoft Windows

GTK+ and *gtkmm* were designed to work well with Microsoft Windows, and the developers encourage its use on the win32 platform. However, Windows has no standard installation system for development libraries. Please see the Windows Installation page for Windows-specific installation instructions and notes.

3. Basics

This chapter will introduce some of the most important aspects of *gtkmm* coding. These will be demonstrated with simple working example code. However, this is just a taster, so you need to look at the other chapters for more substantial information.

Your existing knowledge of C++ will help you with *gtkmm* as it would with any library. Unless we state otherwise, you can expect *gtkmm* classes to behave like any other C++ class, and you can expect to use your existing C++ techniques with *gtkmm* classes.

3.1. Simple Example

To begin our introduction to *gtkmm*, we'll start with the simplest program possible. This program will create an empty 200 x 200 pixel window.

Source Code

File: `base.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include <gtkmm.h>

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

  Gtk::Window window;

  Gtk::Main::run(window);

  return EXIT_SUCCESS;
}
```

We will now explain each line of the example

```
#include <gtkmm.h>
```

All *gtkmm* programs must include certain *gtkmm* headers; `gtkmm.h` includes the entire *gtkmm* kit. This is usually not a good idea, because it includes a megabyte or so of headers, but for simple programs, it suffices.

The next line:

```
Gtk::Main kit(argc, argv);
```

creates a `Gtk::Main` object. This is needed in all *gtkmm* applications. The constructor for this object initializes *gtkmm*, and checks the arguments passed to your application on the command line, looking for standard options such as `-display`. It takes these from the argument list, leaving anything it does not recognize for your application to parse or ignore. This ensures that all *gtkmm* applications accept the same set of standard arguments.

The next two lines of code create and display a window:

```
Gtk::Window window;
```

The last line shows the window and enters the *gtkmm* main processing loop, which will finish when the window is closed.

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

After putting the source code in `simple.cc` you can compile the above program with gcc using:

```
g++ simple.cc -o simple `pkg-config gtkmm-3.0 --cflags --libs`
```

Note that you must surround the `pkg-config` invocation with backquotes. Backquotes cause the shell to execute the command inside them, and to use the command's output as part of the command line.

3.2. Headers and Linking

Although we have shown the compilation command for the simple example, you really should use the automake and autoconf tools, as described in "Autoconf, Automake, Libtool", by G. V. Vaughan et al. The examples used in this book are included in the *gtkmm* package, with appropriate build files, so we won't show the build commands in future. You'll just need to find the appropriate directory and type `make`.

To simplify compilation, we use `pkg-config`, which is present in all (properly installed) *gtkmm* installations. This program 'knows' what compiler switches are needed to compile programs that use *gtkmm*. The `--cflags` option causes `pkg-config` to output a list of include directories for the compiler to look in; the `--libs` option requests the list of libraries for the compiler to link with and the directories to find them in. Try running it from your shell-prompt to see the results on your system.

However, this is even simpler when using the `PKG_CHECK_MODULES()` macro in a standard

configure.ac file with autoconf and automake. For instance:

```
PKG_CHECK_MODULES([MYAPP], [gtkmm-3.0 >= 3.0.0])
```

This checks for the presence of gtkmm and defines MYAPP_LIBS and MYAPP_CFLAGS for use in your Makefile.am files.

gtkmm-3.0 is the name of the current stable API. There was an older API called gtkmm-2-4 which installs in parallel when it is available. There were several versions of gtkmm-2.4, such as gtkmm 2.10 and there will be several versions of the gtkmm-3.0 API. Note that the API name does not change for every version because that would be an incompatible API and ABI break. Theoretically, there might be a future gtkmm-4.0 API which would install in parallel with gtkmm-3.0 without affecting existing applications.

Note that if you mention extra modules in addition to gtkmm-3.0, they should be separated by spaces, not commas.

Openismus has more basic help with automake and autoconf.

3.3. Widgets

gtkmm applications consist of windows containing widgets, such as buttons and text boxes. In some other systems, widgets are called "controls". For each widget in your application's windows, there is a C++ object in your application's code. So you just need to call a method of the widget's class to affect the visible widget.

Widgets are arranged inside container widgets such as frames and notebooks, in a hierarchy of widgets within widgets. Some of these container widgets, such as `Gtk::Grid`, are not visible - they exist only to arrange other widgets. Here is some example code that adds 2 `Gtk::Button` widgets to a `Gtk::VBox` container widgets:

```
m_box.pack_start(m_Button1);
m_box.pack_start(m_Button2);
```

and here is how to add the `Gtk::VBox`, containing those buttons, to a `Gtk::Frame`, which has a visible frame and title:

```
m_frame.add(m_box);
```

Most of the chapters in this book deal with specific widgets. See the Container Widgets section for more details about adding widgets to container widgets.

Although you can specify the layout and appearance of windows and widgets with C++ code, you will probably find it more convenient to design your user interfaces with `Glade` and load them at runtime with `Gtk::Builder`. See the Glade and Gtk::Builder chapter.

Although *gtkmm* widget instances have lifetimes and scopes just like those of other C++ classes, *gtkmm* has an optional time-saving feature that you will see in some of the examples. `Gtk::manage()` allows you to say that a child widget is owned by the container into which you place it. This allows you to `new` the widget, add it to the container and forget about deleting it. You can learn more about *gtkmm* memory management techniques in the Memory Management chapter.

3.4. Signals

gtkmm, like most GUI toolkits, is *event-driven*. When an event occurs, such as the press of a mouse button, the appropriate signal will be *emitted* by the Widget that was pressed. Each Widget has a different set of signals that it can emit. To make a button click result in an action, we set up a *signal handler* to catch the button's "clicked" signal.

gtkmm uses the libsigc++ library to implement signals. Here is an example line of code that connects a Gtk::Button's "clicked" signal with a signal handler called "on_button_clicked":

```
m_button1.signal_clicked().connect( sigc::mem_fun(*this,
  &HelloWorld::on_button_clicked) );
```

For more detailed information about signals, see the appendix.

For information about implementing your own signals rather than just connecting to the existing *gtkmm* signals, see the appendix.

3.5. Glib::ustring

You might be surprised to learn that *gtkmm* doesn't use `std::string` in it its interfaces. Instead it uses `Glib::ustring`, which is so similar and unobtrusive that you could actually pretend that each Glib::ustring is a `std::string` and ignore the rest of this section. But read on if you want to use languages other than English in your application.

std::string uses 8 bit per character, but 8 bits aren't enough to encode languages such as Arabic, Chinese, and Japanese. Although the encodings for these languages has now been specified by the Unicode Constortium, the C and C++ languages do not yet provide any standardised Unicode support. GTK+ and GNOME chose to implement Unicode using UTF-8, and that's what is wrapped by Glib::ustring. It provides almost exactly the same interface as std::string, along with automatic conversions to and from std::string.

One of the benefits of UTF-8 is that you don't need to use it unless you want to, so you don't need to retrofit all of your code at once. `std::string` will still work for 7-bit ASCII strings. But when you try to localize your application for languages like Chinese, for instance, you will start to see strange errors, and possible crashes. Then all you need to do is start using `Glib::ustring` instead.

Note that UTF-8 isn't compatible with 8-bit encodings like ISO-8859-1. For instance, German umlauts are not in the ASCII range and need more than 1 byte in the UTF-8 encoding. If your code contains 8-bit string literals, you have to convert them to UTF-8 (e.g. the Bavarian greeting "Grüß Gott" would be "Gr\xC3\xBC\xC3\x9F Gott").

You should avoid C-style pointer arithmetic, and functions such as strlen(). In UTF-8, each character might need anywhere from 1 to 6 bytes, so it's not possible to assume that the next byte is another character. `Glib::ustring` worries about the details of this for you so you can use methods such as Glib::ustring::substr() while still thinking in terms of characters instead of bytes.

Unlike the Windows UCS-2 Unicode solution, this does not require any special compiler options to process string literals, and it does not result in Unicode executables and libraries which are incompatible with ASCII ones.

Reference

See the Internationalization section for information about providing the UTF-8 string literals.

3.6. Intermediate types

Some API related to gtkmm uses intermediate data containers, such as `Glib::StringArrayHandle`, instead of a specific Standard C++ container such as `std::vector` or `std::list`, though *gtkmm* itself now uses just `std::vector` since *gtkmm* 3.0.

You should not declare these types yourself. You should instead use whatever Standard C++ container you prefer. glibmm will do the conversion for you. Here are some of these intermediate types:

- `Glib::StringArrayHandle` or `Glib::ArrayHandle<Glib::ustring>`: Use `std::vector<Glib::ustring>`, `std::list<Glib::ustring>`, `const char*[]`, etc.
- `Glib::ListHandle<Gtk::Widget*>`: Use `std::vector<Gtk::Widget*>`, `std::list<Gtk::Widget*>`, etc.
- `Glib::SListHandle<Gtk::Widget*>`: Use `std::vector<Gtk::Widget*>`, `std::list<Gtk::Widget*>`, etc.

3.7. Mixing C and C++ APIs

You can use C APIs which do not yet have convenient C++ interfaces. It is generally not a problem to use C APIs from C++, and *gtkmm* helps by providing access to the underlying C object, and providing an easy way to create a C++ wrapper object from a C object, provided that the C API is also based on the GObject system.

To use a *gtkmm* instance with a C function that requires a C GObject instance, use the `gobj()` function to obtain a pointer to the underlying GObject instance. For instance

```
Gtk::Button* button = new Gtk::Button("example");
gtk_button_do_something_new(button->gobj());
```

To obtain a *gtkmm* instance from a C GObject instance, use the Glib::wrap() function. For instance

```
GtkButton* cbutton = get_a_button();
Gtk::Button* button = Glib::wrap(cbutton);
```

3.8. Hello World in *gtkmm*

We've now learned enough to look at a real example. In accordance with an ancient tradition of computer science, we now introduce Hello World, a la *gtkmm*:

Source Code

File: `helloworld.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLE_HELLOWORLD_H
#define GTKMM_EXAMPLE_HELLOWORLD_H

#include <gtkmm/button.h>
#include <gtkmm/window.h>

class HelloWorld : public Gtk::Window
{

public:
  HelloWorld();
  virtual ~HelloWorld();

protected:
  //Signal handlers:
  void on_button_clicked();

  //Member widgets:
  Gtk::Button m_button;
};

#endif // GTKMM_EXAMPLE_HELLOWORLD_H
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "helloworld.h"
#include <gtkmm/main.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;
}
```

File: `helloworld.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "helloworld.h"
#include <iostream>
```

```
HelloWorld::HelloWorld()
: m_button("Hello World")   // creates a new button with label "Hello World".
{
  // Sets the border width of the window.
  set_border_width(10);

  // When the button receives the "clicked" signal, it will call the
  // on_button_clicked() method defined below.
  m_button.signal_clicked().connect(sigc::mem_fun(*this,
              &HelloWorld::on_button_clicked));

  // This packs the button into the Window (a container).
  add(m_button);

  // The final step is to display this newly created widget...
  m_button.show();
}

HelloWorld::~HelloWorld()
{
}

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

Try to compile and run it before going on. You should see something like this:

Figure 3-1 **Hello World**

Pretty thrilling, eh? Let's examine the code. First, the `HelloWorld` class:

```
class HelloWorld : public Gtk::Window
{

public:
  HelloWorld();
  virtual ~HelloWorld();

protected:
  //Signal handlers:
  virtual void on_button_clicked();

  //Member widgets:
  Gtk::Button m_button;
};
```

This class implements the "Hello World" window. It's derived from `Gtk::Window`, and has a single `Gtk::Button` as a member. We've chosen to use the constructor to do all of the initialisation work for the window, including setting up the signals. Here it is, with the comments omitted:

```
HelloWorld::HelloWorld()
:
  m_button ("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();
}
```

Notice that we've used an initialiser statement to give the `m_button` object the label "Hello World".

Next we call the Window's `set_border_width()` method. This sets the amount of space between the sides of the window and the widget it contains.

We then hook up a signal handler to `m_button`'s `clicked` signal. This prints our friendly

greeting to `stdout`.

Next, we use the Window's `add()` method to put `m_button` in the Window. (`add()` comes from `Gtk::Container`, which is described in the chapter on container widgets.) The `add()` method places the Widget in the Window, but it doesn't display the widget. *gtkmm* widgets are always invisible when you create them - to display them, you must call their `show()` method, which is what we do in the next line.

Now let's look at our program's `main()` function. Here it is, without comments:

```
int main(int argc, char** argv)
{
  Gtk::Main kit(argc, argv);

  HelloWorld helloworld;
  Gtk::Main::run(helloworld);

  return 0;
}
```

First we instantiate an object called `kit`. This is of type `Gtk::Main`. Every *gtkmm* program must have one of these. We pass our command-line arguments to its constructor. It takes the arguments it wants, and leaves you the rest, as we described earlier.

Next we make an object of our `HelloWorld` class, whose constructor takes no arguments, but it isn't visible yet. When we call Gtk::Main::run(), giving it the helloworld Window, it shows the Window and starts the *gtkmm event loop*. During the event loop *gtkmm* idles, waiting for actions from the user, and responding appropriately. When the user closes the Window, run() will return, causing the final line of our main() function be to executed. The application will then finish.

4. Changes in *gtkmm* 3

gtkmm-3.0 is a new version of the *gtkmm* API that installs in parallel with the older *gtkmm*-2.4 API. The last version of the *gtkmm*-2.4 API was *gtkmm* 2.24. *gtkmm* 3 has no major fundamental differences to *gtkmm* 2 but does make several small changes that were not possible while maintaining binary compatibility. If you never used the *gtkmm*-2.4 API then you can safely ignore this chapter.

gtkmm 3's library is called `libgtkmm-3.0` rather than `libgtkmm-2.4` and installs its headers in a similarly-versioned directory, so your pkg-config check should ask for `gtkmm-3.0` rather than `gtkmm-2.4`.

gtkmm 3 added some new classes:

1. `Gtk::AppChooser`, `Gtk::AppChooserButton`, `Gtk::AppChooserDialog` allow the user to select an installed application to open a particularl type of content.
2. `Gtk::Grid` is a new container widget that will eventually replace `Gtk::Box`. It arranges its children according to properties of those children rather than its own layout details.
3. `Gtk::Switch` displays On/Off states more explictly than `Gtk::CheckBox`. It may be useful, for instance, when allowing users to activate hardware.

gtkmm 3 also made several small changes to the API, which you will probably encounter when porting code that used *gtkmm*-2.4. Here is a short list:

1. `Gtk::CellLayout`, used by `Gtk::IconView`, `Gtk::TreeView::Column` and `Gtk::ComboBox`, now has a `Gtk::CellArea` which can be used to specify more details of how the `CellRenderer`s are arranged and aligned.
2. Gtk::ComboBox now derives from CellLayout, allowing easier layout and alignment of its `Gtk::CellRenderer`s.
3. `Gtk::Adjustment` and `IconSet` and `Gdk::Cursor` are now used via `Glib::RefPtr`.
4. `Gtk::Box`, `Gtk::ButtonBox`, `Gtk::IconView`, `Gtk::Paned`, `Gtk::ProgressBar`, `Gtk::ScaleButton`, `Gtk::ScrollBar` and `Gtk::Separator` now derive from `Gtk::Orientable`, allowing their orientation (vertical or horizontal) to be specified without requiring the use of a

derived class such as `Gtk::HBox`.

5. `Gtk::IconView`, `Gtk::TextView`, `Gtk::TreeView` and other widgets derive from Scrollable instead of having their own methods such as `get_vadjustment()` and instead of having their won set_scroll_adjustments signal.

6. `Gtk::Style` and `Gtk::Rc` were removed, replaced by `Gtk::StyleContext`, and `Gtk::StyleProviderS`, such as `Gtk::CssProvider`.

7. Widget::on_expose_event() was replaced by Widget::on_draw(), which assumes that cairomm is used for drawing, via the provided `Cairo::Context` and does not require you to call `Cairo::Context::clip()`.

8. `Gdk::RGBA` replaces `Color`, adding an alpha component for opacity. `Colormap` was removed, along with its awkward use to allocate colors.

9. `Gdk::Pixmap` and `Gdk::Bitmap` were removed in favour of `Gdk::Pixbuf`.

10. `Gdk::Drawable` was removed, with its methods moving into `Gdk::Window`.

11. We now use std::vector in several methods instead of the intermediate *Handle types to make the API clearer.

All deprecated API was removed in *gtkmm* 3.0, though there will be new deprecations in future versions.

As a first step to porting your source code to *gtkmm*-3.0; you should probably ensure that your application builds with the deprecated *gtkmm*-2.4; API disabled, by defining macro such as GTK_DISABLE_DEPRECATED. There are some autotools macros that can help with this by defining them optionally at build time. See the gtkmm 3 porting wiki page for more details.

5. Buttons

gtkmm provides four basic types of buttons:

Push-Buttons

`Gtk::Button`. Standard buttons, usually marked with a label or picture. Pushing one triggers an action. See the Button section.

Toggle buttons

`Gtk::ToggleButton`. Unlike a normal Button, which springs back up, a ToggleButton stays down until you press it again. It might be useful as an on/off switch. See the ToggleButton section.

Checkboxes

`Gtk::CheckButton`. These act like ToggleButtons, but show their state in small squares, with their label at the side. They should be used in most situations which require an on/off setting. See the CheckBox section.

Radio buttons

`Gtk::RadioButton`. Named after the station selectors on old car radios, these buttons are used in groups for options which are mutually exclusive. Pressing one causes all the others in its group to turn off. They are similar to CheckBoxes (a small widget with a label at the side), but usually look different. See the RadioButton section.

Note that, due to GTK+'s theming system, the appearance of these widgets will vary. In the case of checkboxes and radio buttons, they may vary considerably.

5.1. Button

5.1.1. Constructors

There are two ways to create a Button. You can specify a label string in the `Gtk::Button` constructor, or set it later with `set_label()`.

To define an accelerator key for keyboard navigation, place an underscore before one of the label's characters and specify `true` for the optional `mnemonic` parameter. For instance:

```
Gtk::Button* pButton = new Gtk::Button("_Something", true);
```

Wherever possible you should use Stock items, to ensure consistency with other applications, and to improve the appearance of your applications by using icons. For instance,

```
Gtk::Button* pButton = new Gtk::Button(Gtk::Stock::OK);
```

This will use standard text, in all languages, with standard keyboard accelerators, with a standard icon.

`Gtk::Button` is also a container so you could put any other widget, such as a `Gtk::Image` into it.

Reference

5.1.2. Example

This example creates a button with a picture and a label.

Figure 5-1 **buttons example**

Source Code

File: `buttons.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLE_BUTTONS_H
#define GTKMM_EXAMPLE_BUTTONS_H

#include <gtkmm/window.h>
#include <gtkmm/button.h>

class Buttons : public Gtk::Window
{
public:
  Buttons();
  virtual ~Buttons();

protected:
  //Signal handlers:
  void on_button_clicked();

  //Child widgets:
  Gtk::Button m_button;
};

#endif //GTKMM_EXAMPLE_BUTTONS_H
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "buttons.h"
#include <gtkmm/main.h>

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

  Buttons buttons;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(buttons);

  return 0;
}
```

File: `buttons.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "buttons.h"
#include <iostream>

Buttons::Buttons()
{
  m_button.add_pixlabel("info.xpm", "cool button");

  set_title("Pixmap'd buttons!");
  set_border_width(10);

  m_button.signal_clicked().connect( sigc::mem_fun(*this,
              &Buttons::on_button_clicked) );

  add(m_button);

  show_all_children();
}

Buttons::~Buttons()
{
}

void Buttons::on_button_clicked()
{
  std::cout << "The Button was clicked." << std::endl;
}
```

5.1.3. Signals

The `Gtk::Button` widget has the following signals, but most of the time you will just handle the `clicked` signal:

`pressed`
: Emitted when the button is pressed.

`released`
: Emitted when the button is released.

`clicked`
: Emitted when the button is pressed and released.

`enter`
: Emitted when the mouse pointer moves over the button's window.

`leave`
: Emitted when the mouse pointer leaves the button's window.

5.2. ToggleButton

`ToggleButton`s are like normal `Button`s, but when clicked they remain activated, or pressed, until clicked again.

To retrieve the state of the `ToggleButton`, you can use the `get_active()` method. This returns `true` if the button is "down". You can also set the toggle button's state, with `set_active()`. Note that, if you do this, and the state actually changes, it causes the "clicked" signal to be emitted. This is usually what you want.

You can use the `toggled()` method to toggle the button, rather than forcing it to be up or down: This switches the button's state, and causes the `toggled` signal to be emitted.

`Gtk::ToggleButton` is most useful as a base class for the `Gtk::CheckButton` and `Gtk::RadioButton` classes.

Reference

5.3. CheckButton

`Gtk::CheckButton` inherits from `Gtk::ToggleButton`. The only real difference between the two is `Gtk::CheckButton`'s appearance. You can check, set, and toggle a checkbox using the same member methods as for `Gtk::ToggleButton`.

Reference

5.3.1. Example

Figure 5-2 **CheckButton**

Source Code

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLE_BUTTONS_H
#define GTKMM_EXAMPLE_BUTTONS_H

#include <gtkmm/window.h>
#include <gtkmm/checkbutton.h>

class ExampleWindow : public Gtk::Window
{
public:
  ExampleWindow();
  virtual ~ExampleWindow();

protected:
  //Signal handlers:
  void on_button_clicked();

  //Child widgets:
  Gtk::CheckButton m_button;
};

#endif //GTKMM_EXAMPLE_BUTTONS_H
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <iostream>

ExampleWindow::ExampleWindow()
: m_button("something")
{
  set_title("checkbutton example");
  set_border_width(10);

  m_button.signal_clicked().connect(sigc::mem_fun(*this,
              &ExampleWindow::on_button_clicked) );

  add(m_button);

  show_all_children();
}

ExampleWindow::~ExampleWindow()
{
```

```
}

void ExampleWindow::on_button_clicked()
{
  std::cout << "The Button was clicked: state="
      << (m_button.get_active() ? "true" : "false")
      << std::endl;
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

5.4. RadioButton

Like checkboxes, radio buttons also inherit from `Gtk::ToggleButton`, but these work in groups, and only one RadioButton in a group can be selected at any one time.

5.4.1. Groups

There are two ways to set up a group of radio buttons. The first way is to create the buttons, and set up their groups afterwards. Only the first two constructors are used. In the following example, we make a new window class called `RadioButtons`, and then put three radio buttons in it:

```
class RadioButtons : public Gtk::Window
{
public:
    RadioButtons();

protected:
    Gtk::RadioButton m_rb1, m_rb2, m_rb3;
};

RadioButtons::RadioButtons()
  : m_rb1("button1"),
    m_rb2("button2"),
    m_rb3("button3")
{
    Gtk::RadioButton::Group group = m_rb1.get_group();
    m_rb2.set_group(group);
    m_rb3.set_group(group);
}
```

We told *gtkmm* to put all three `RadioButton`s in the same group by obtaining the group with `get_group()` and using `set_group()` to tell the other `RadioButton`s to share that group.

Note that you can't just do

```
m_rb2.set_group(m_rb1.get_group()); //doesn't work
```

because the group is modified by `set_group()` and therefore non-const.

The second way to set up radio buttons is to make a group first, and then add radio buttons to it. Here's an example:

```
class RadioButtons : public Gtk::Window
{
public:
    RadioButtons();
};

RadioButtons::RadioButtons()
{
    Gtk::RadioButton::Group group;
    Gtk::RadioButton *m_rb1 = Gtk::manage(
      new Gtk::RadioButton(group,"button1"));
    Gtk::RadioButton *m_rb2 = manage(
      new Gtk::RadioButton(group,"button2"));
      Gtk::RadioButton *m_rb3 = manage(
        new Gtk::RadioButton(group,"button3"));
}
```

We made a new group by simply declaring a variable, `group`, of type `Gtk::RadioButton::Group`. Then we made three radio buttons, using a constructor to make each of them part of `group`.

5.4.2. Methods

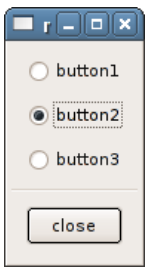
`RadioButtons` are "off" when created; this means that when you first make a group of them, they will all be off. Don't forget to turn one of them on using `set_active()`:

Reference

5.4.3. Example

The following example demonstrates the use of `RadioButtonS`:

Figure 5-3 **RadioButton**



Source Code

File: `radiobuttons.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLE_RADIOBUTTONS_H
#define GTKMM_EXAMPLE_RADIOBUTTONS_H

#include <gtkmm/box.h>
#include <gtkmm/window.h>
#include <gtkmm/radiobutton.h>
#include <gtkmm/separator.h>

class RadioButtons : public Gtk::Window
{
public:
  RadioButtons();
  virtual ~RadioButtons();

protected:
  //Signal handlers:
  void on_button_clicked();

  //Child widgets:
  Gtk::Box m_Box_Top, m_Box1, m_Box2;
  Gtk::RadioButton m_RadioButton1, m_RadioButton2, m_RadioButton3;
  Gtk::Separator m_Separator;
  Gtk::Button m_Button_Close;
};

#endif //GTKMM_EXAMPLE_RADIOBUTTONS_H
```

File: `radiobuttons.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "radiobuttons.h"


RadioButtons::RadioButtons() :
  m_Box_Top(Gtk::ORIENTATION_VERTICAL),
  m_Box1(Gtk::ORIENTATION_VERTICAL, 10),
  m_Box2(Gtk::ORIENTATION_VERTICAL, 10),
  m_RadioButton1("button1"),
  m_RadioButton2("button2"),
  m_RadioButton3("button3"),
  m_Button_Close("close")
{
  // Set title and border of the window
  set_title("radio buttons");
  set_border_width(0);

  // Put radio buttons 2 and 3 in the same group as 1:
  Gtk::RadioButton::Group group = m_RadioButton1.get_group();
  m_RadioButton2.set_group(group);
  m_RadioButton3.set_group(group);

  // Add outer box to the window (because the window
  // can only contain a single widget)
  add(m_Box_Top);

  //Put the inner boxes and the separator in the outer box:
  m_Box_Top.pack_start(m_Box1);
  m_Box_Top.pack_start(m_Separator);
  m_Box_Top.pack_start(m_Box2);

  // Set the inner boxes' borders
  m_Box2.set_border_width(10);
  m_Box1.set_border_width(10);

  // Put the radio buttons in Box1:
  m_Box1.pack_start(m_RadioButton1);
  m_Box1.pack_start(m_RadioButton2);
  m_Box1.pack_start(m_RadioButton3);

  // Set the second button active
  m_RadioButton2.set_active();

  // Put Close button in Box2:
  m_Box2.pack_start(m_Button_Close);

  // Make the button the default widget
  m_Button_Close.set_can_default();
  m_Button_Close.grab_default();

  // Connect the clicked signal of the button to
  // RadioButtons::on_button_clicked()
  m_Button_Close.signal_clicked().connect(sigc::mem_fun(*this,
              &RadioButtons::on_button_clicked) );

  // Show all children of the window
  show_all_children();
}

RadioButtons::~RadioButtons()
{
}

void RadioButtons::on_button_clicked()
{
  hide(); //to close the application.
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "radiobuttons.h"
#include <gtkmm/main.h>

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

  RadioButtons buttons;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(buttons);

  return 0;
}
```

6. Range Widgets

`Gtk::Scale` and `Gtk::Scrollbar` both inherit from `Gtk::Range` and share much functionality. They contain a "trough" and a "slider" (sometimes called a "thumbwheel" in other GUI environments). Dragging the slider with the pointer moves it within the trough, while clicking in the trough advances the slider towards the location of the click, either completely, or by a designated amount, depending on which mouse button is used. This should be familiar scrollbar behaviour.

As will be explained in the Adjustment section, all Range widgets are associated with a `Adjustment` object. To change the lower, upper, and current values used by the widget you need to use the methods of its `Adjustment`, which you can get with the `get_adjustment()` method. The `Range` widgets' default constructors create an `Adjustment` automatically, or you can specify an existing `Adjustment`, maybe to share it with another widget. See the Adjustments section for further details.

Reference

6.1. Scrollbar Widgets

These are standard scrollbars. They should be used only to scroll another widget, such as, a `Gtk::Entry`, or a `Gtk::Viewport`, though it's usually easier to use the `Gtk::ScrolledWindow` widget in most cases.

There are horizontal and vertical scrollbar classes - `Gtk::HScrollbar` and `Gtk::VScrollbar`.

Reference

6.2. Scale Widgets

`Gtk::Scale` widgets (or "sliders") allow the user to visually select and manipulate a value within a specific range. You might use one, for instance, to adjust the magnification level on a zoomed preview of a picture, or to control the brightness of a colour, or to specify the number of minutes of inactivity before a screensaver takes over the screen.

As with `Scrollbar`s, there are separate widget types for horizontal and vertical widgets - `Gtk::HScale` and `Gtk::VScale`. The default constructors create an `Adjustment` with all of its values set to `0.0`. This isn't useful so you will need to set some `Adjustment` details to get meaningful behaviour.

6.2.1. Useful methods

`Scale` widgets can display their current value as a number next to the trough. By default they show the value, but you can change this with the `set_draw_value()` method.

The value displayed by a scale widget is rounded to one decimal point by default, as

is the `value` field in its `Gtk::Adjustment`. You can change this with the `set_digits()` method.

Also, the value can be drawn in different positions relative to the trough, specified by the `set_value_pos()` method.

Reference

6.3. Update Policies

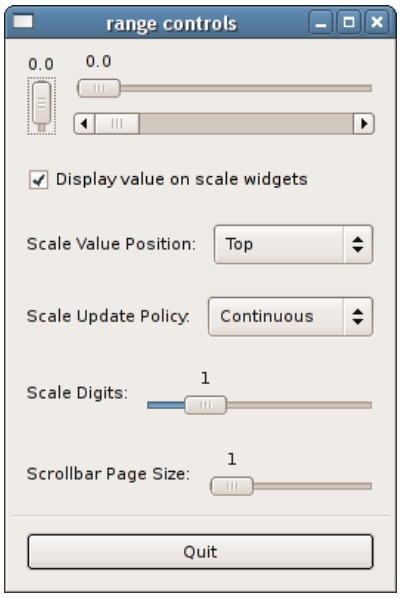
The *update policy* of a `Range` widget defines at what points during user interaction it will change the `value` field of its `Gtk::Adjustment` and emit the `value_changed` signal. The update policies, set with the `set_update_policy()` method, are:

- `Gtk::UPDATE_CONTINUOUS` - This is the default. The `value_changed` signal is emitted continuously, i.e. whenever the slider is moved by even the tiniest amount.
- `Gtk::UPDATE_DISCONTINUOUS` - The `value_changed` signal is only emitted once the slider has stopped moving and the user has released the mouse button.
- `Gtk::UPDATE_DELAYED` - The `value_changed` signal is emitted when the user releases the mouse button, or if the slider stops moving for a short period of time.

6.4. Example

This example displays a window with three range widgets all connected to the same adjustment, along with a couple of controls for adjusting some of the parameters mentioned above and in the section on adjustments, so you can see how they affect the way these widgets work for the user.

Figure 6-1 **Range Widgets**



Source Code

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLE_RANGEWIDGETS_H
#define GTKMM_EXAMPLE_RANGEWIDGETS_H

#include <gtkmm.h>

class ExampleWindow : public Gtk::Window
{
public:
  ExampleWindow();
  virtual ~ExampleWindow();

protected:
  //Signal handlers:
```

```
  void on_checkbutton_toggled();
  void on_menu_position(Gtk::PositionType type);
  void on_adjustment1_value_changed();
  void on_adjustment2_value_changed();
  void on_button_quit();

  //Child widgets:
  Gtk::Box m_VBox_Top, m_VBox2, m_VBox_HScale;
  Gtk::Box m_HBox_Scales, m_HBox_Digits, m_HBox_PageSize;

  Glib::RefPtr<Gtk::Adjustment> m_adjustment, m_adjustment_digits, m_adjustment_pagesize;

  Gtk::Scale m_VScale;
  Gtk::Scale m_HScale, m_Scale_Digits, m_Scale_PageSize;

  Gtk::Separator m_Separator;

  Gtk::CheckButton m_CheckButton;

  Gtk::Scrollbar m_Scrollbar;

  Gtk::Menu m_Menu_Position;

  Gtk::Button m_Button_Quit;
};

#endif //GTKMM_EXAMPLE_RANGEWIDGETS_H
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <iostream>

ExampleWindow::ExampleWindow()
:
  m_VBox_Top(Gtk::ORIENTATION_VERTICAL, 0),
  m_VBox2(Gtk::ORIENTATION_VERTICAL, 20),
  m_VBox_HScale(Gtk::ORIENTATION_VERTICAL, 10),
  m_HBox_Scales(Gtk::ORIENTATION_HORIZONTAL, 10),
  m_HBox_Digits(Gtk::ORIENTATION_HORIZONTAL, 10),
  m_HBox_PageSize(Gtk::ORIENTATION_HORIZONTAL, 10),

  // Value, lower, upper, step_increment, page_increment, page_size:
  // Note that the page_size value only makes a difference for
  // scrollbar widgets, and the highest value you'll get is actually
  // (upper - page_size).
  m_adjustment( Gtk::Adjustment::create(0.0, 0.0, 101.0, 0.1, 1.0, 1.0) ),
  m_adjustment_digits( Gtk::Adjustment::create(1.0, 0.0, 5.0, 1.0, 2.0) ),
  m_adjustment_pagesize( Gtk::Adjustment::create(1.0, 1.0, 101.0) ),

  m_VScale(m_adjustment, Gtk::ORIENTATION_VERTICAL),
  m_HScale(m_adjustment, Gtk::ORIENTATION_HORIZONTAL),
  m_Scale_Digits(m_adjustment_digits),
  m_Scale_PageSize(m_adjustment_pagesize),

  // A checkbutton to control whether the value is displayed or not:
  m_CheckButton("Display value on scale widgets", 0),

  // Reuse the same adjustment again.
  // Notice how this causes the scales to always be updated
  // continuously when the scrollbar is moved.
  m_Scrollbar(m_adjustment),

  m_Button_Quit("Quit")
{
  set_title("range controls");
  set_default_size(300, 350);

  //VScale:
  m_VScale.set_digits(1);
  m_VScale.set_value_pos(Gtk::POS_TOP);
  m_VScale.set_draw_value();
  m_VScale.set_inverted(); // highest value at top

  //HScale:
  m_HScale.set_digits(1);
  m_HScale.set_value_pos(Gtk::POS_TOP);
  m_HScale.set_draw_value();

  add(m_VBox_Top);
  m_VBox_Top.pack_start(m_VBox2);
  m_VBox2.set_border_width(10);
  m_VBox2.pack_start(m_HBox_Scales);

  //Put VScale and HScale (above scrollbar) side-by-side.
```

```
  m_HBox_Scales.pack_start(m_VScale);
  m_HBox_Scales.pack_start(m_VBox_HScale);

  m_VBox_HScale.pack_start(m_HScale);

  //Scrollbar:
  m_VBox_HScale.pack_start(m_Scrollbar);

  //CheckButton:
  m_CheckButton.set_active();
  m_CheckButton.signal_toggled().connect( sigc::mem_fun(*this,
    &ExampleWindow::on_checkbutton_toggled) );
  m_VBox2.pack_start(m_CheckButton, Gtk::PACK_SHRINK);

  //Menus:
  Gtk::MenuItem* item = Gtk::manage(new Gtk::MenuItem("Top"));
  item->signal_activate().connect(
    sigc::bind(sigc::mem_fun(*this,
      &ExampleWindow::on_menu_position), Gtk::POS_TOP));
  m_Menu_Position.append(*item);

  item = Gtk::manage(new Gtk::MenuItem("Bottom"));
  item->signal_activate().connect(
    sigc::bind(sigc::mem_fun(*this,
      &ExampleWindow::on_menu_position), Gtk::POS_BOTTOM));
  m_Menu_Position.append(*item);

  item = Gtk::manage(new Gtk::MenuItem("Left"));
  item->signal_activate().connect(
    sigc::bind(sigc::mem_fun(*this,
      &ExampleWindow::on_menu_position), Gtk::POS_LEFT));
  m_Menu_Position.append(*item);

  item = Gtk::manage(new Gtk::MenuItem("Right"));
  item->signal_activate().connect(
    sigc::bind(sigc::mem_fun(*this,
      &ExampleWindow::on_menu_position), Gtk::POS_RIGHT));
  m_Menu_Position.append(*item);

  //Digits:
  m_HBox_Digits.pack_start(
    *Gtk::manage(new Gtk::Label("Scale Digits:", 0)), Gtk::PACK_SHRINK);
  m_Scale_Digits.set_digits(0);
  m_adjustment_digits->signal_value_changed().connect(sigc::mem_fun(*this,
    &ExampleWindow::on_adjustment1_value_changed));
  m_HBox_Digits.pack_start(m_Scale_Digits);

  //Page Size:
  m_HBox_PageSize.pack_start(
    *Gtk::manage(new Gtk::Label("Scrollbar Page Size:", 0)),
    Gtk::PACK_SHRINK);
  m_Scale_PageSize.set_digits(0);
  m_adjustment_pagesize->signal_value_changed().connect(sigc::mem_fun(*this,
    &ExampleWindow::on_adjustment2_value_changed));
  m_HBox_PageSize.pack_start(m_Scale_PageSize);

  m_VBox2.pack_start(m_HBox_Digits, Gtk::PACK_SHRINK);
  m_VBox2.pack_start(m_HBox_PageSize, Gtk::PACK_SHRINK);
  m_VBox_Top.pack_start(m_Separator, Gtk::PACK_SHRINK);
  m_VBox_Top.pack_start(m_Button_Quit, Gtk::PACK_SHRINK);

  m_Button_Quit.set_can_default();
  m_Button_Quit.grab_default();
  m_Button_Quit.signal_clicked().connect(sigc::mem_fun(*this,
    &ExampleWindow::on_button_quit));
  m_Button_Quit.set_border_width(10);

  show_all_children();
}

ExampleWindow::~ExampleWindow()
{
}

void ExampleWindow::on_checkbutton_toggled()
{
  m_VScale.set_draw_value(m_CheckButton.get_active());
  m_HScale.set_draw_value(m_CheckButton.get_active());
}

void ExampleWindow::on_menu_position(Gtk::PositionType postype)
{
  m_VScale.set_value_pos(postype);
  m_HScale.set_value_pos(postype);
}

void ExampleWindow::on_adjustment1_value_changed()
{
```

```
  const double val = m_adjustment_digits->get_value();
  m_VScale.set_digits((int)val);
  m_HScale.set_digits((int)val);
}

void ExampleWindow::on_adjustment2_value_changed()
{
  const double val = m_adjustment_pagesize->get_value();
  m_adjustment->set_page_size(val);
  m_adjustment->set_page_increment(val);

  // Note that we don't have to emit the "changed" signal
  // because gtkmm does this for us.
}

void ExampleWindow::on_button_quit()
{
  hide();
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

7. Miscellaneous Widgets

7.1. Label

Labels are the main method of placing non-editable text in windows, for instance to place a title next to a `Entry` widget. You can specify the text in the constructor, or later with the `set_text()` or set_markup() methods.

The width of the label will be adjusted automatically. You can produce multi-line labels by putting line breaks ("\n") in the label string.

The label text can be justified using the `set_justify()` method. The widget is also capable of word-wrapping, which can be activated with `set_line_wrap()`.

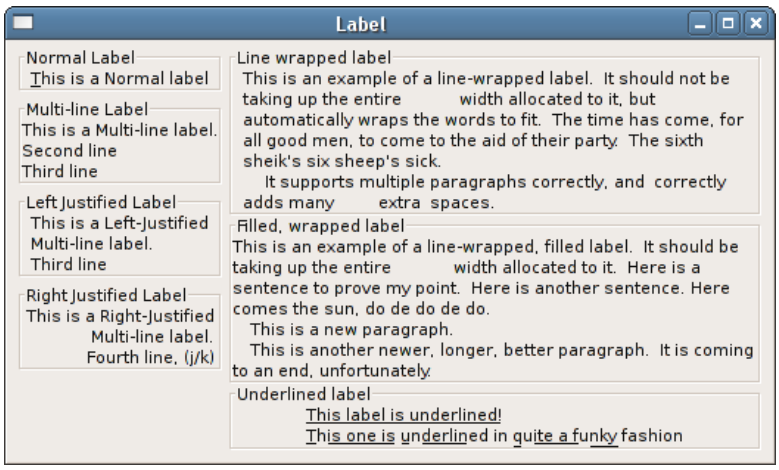
Gtk::Label support some simple formatting, for instance allowing you to make some text bold, colored, or larger. You can do this by providing a string to `set_markup()`, using the Pango Markup syntax. For instance, `<b>bold text</b> and <s>strikethrough text</s>`.

Reference

7.1.1. Example

Below is a short example to illustrate these functions. This example makes use of the Frame widget to better demonstrate the label styles. (The Frame widget is explained in the Frame section.)

Figure 7-1 **Label**



Source Code

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <gtkmm.h>

class ExampleWindow : public Gtk::Window
{
public:
  ExampleWindow();
  virtual ~ExampleWindow();

protected:

  //Child widgets:
  Gtk::Box m_HBox;
  Gtk::Box m_VBox, m_VBox2;
  Gtk::Frame m_Frame_Normal, m_Frame_Multi, m_Frame_Left, m_Frame_Right,
    m_Frame_LineWrapped, m_Frame_FilledWrapped, m_Frame_Underlined;
  Gtk::Label m_Label_Normal, m_Label_Multi, m_Label_Left, m_Label_Right,
    m_Label_LineWrapped, m_Label_FilledWrapped, m_Label_Underlined;
};

#endif //GTKMM_EXAMPLEWINDOW_H
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <iostream>

ExampleWindow::ExampleWindow()
:
  m_HBox(Gtk::ORIENTATION_HORIZONTAL, 5),
  m_VBox(Gtk::ORIENTATION_VERTICAL, 5),
  m_VBox2(Gtk::ORIENTATION_VERTICAL, 5),
  m_Frame_Normal("Normal Label"),
  m_Frame_Multi("Multi-line Label"),
  m_Frame_Left("Left Justified Label"),
  m_Frame_Right("Right Justified Label"),
  m_Frame_LineWrapped("Line wrapped label"),
  m_Frame_FilledWrapped("Filled, wrapped label"),
  m_Frame_Underlined("Underlined label"),
  m_Label_Normal("_This is a Normal label", true),
  m_Label_Multi("This is a Multi-line label.\nSecond line\nThird line"),
  m_Label_Left("This is a Left-Justified\nMulti-line label.\nThird line"),
  m_Label_Right("This is a Right-Justified\nMulti-line label.\nThird line"),
  m_Label_Underlined("This label is underlined!\n"
          "This one is underlined in quite a funky fashion")
{
  set_title("Label");
  set_border_width(5);

  add(m_HBox);
```

```
  m_HBox.pack_start(m_VBox, Gtk::PACK_SHRINK);

  m_Frame_Normal.add(m_Label_Normal);
  m_VBox.pack_start(m_Frame_Normal, Gtk::PACK_SHRINK);

  m_Frame_Multi.add(m_Label_Multi);
  m_VBox.pack_start(m_Frame_Multi, Gtk::PACK_SHRINK);

  m_Label_Left.set_justify(Gtk::JUSTIFY_LEFT);
  m_Frame_Left.add(m_Label_Left);
  m_VBox.pack_start(m_Frame_Left, Gtk::PACK_SHRINK);

  m_Label_Right.set_justify(Gtk::JUSTIFY_RIGHT);
  m_Frame_Right.add(m_Label_Right);
  m_VBox.pack_start(m_Frame_Right, Gtk::PACK_SHRINK);

  m_HBox.pack_start(m_VBox2, Gtk::PACK_SHRINK);

  m_Label_LineWrapped.set_text(
          "This is an example of a line-wrapped label.  It "
          /* add a big space to the next line to test spacing */
          "should not be taking up the entire             "
          "width allocated to it, but automatically "
          "wraps the words to fit.  "
          "The time has come, for all good men, to come to "
          "the aid of their party.  "
          "The sixth sheik's six sheep's sick.\n"
          "     It supports multiple paragraphs correctly, "
          "and  correctly   adds "
          "many          extra  spaces. ");
  m_Label_LineWrapped.set_line_wrap();
  m_Frame_LineWrapped.add(m_Label_LineWrapped);
  m_VBox2.pack_start(m_Frame_LineWrapped, Gtk::PACK_SHRINK);

  m_Label_FilledWrapped.set_text(
          "This is an example of a line-wrapped, filled label.  "
          "It should be taking "
          "up the entire              width allocated to it.  "
          "Here is a sentence to prove "
          "my point.  Here is another sentence. "
          "Here comes the sun, do de do de do.\n"
          "    This is a new paragraph.\n"
          "    This is another newer, longer, better "
          "paragraph.  It is coming to an end, "
          "unfortunately.");
  m_Label_FilledWrapped.set_justify(Gtk::JUSTIFY_FILL);
  m_Label_FilledWrapped.set_line_wrap();
  m_Frame_FilledWrapped.add(m_Label_FilledWrapped);
  m_VBox2.pack_start(m_Frame_FilledWrapped, Gtk::PACK_SHRINK);

  m_Label_Underlined.set_justify(Gtk::JUSTIFY_LEFT);
  m_Label_Underlined.set_pattern (
          "_________________________ _ _________ _ ______"
          "      __ _______ ___");
  m_Frame_Underlined.add(m_Label_Underlined);
  m_VBox2.pack_start(m_Frame_Underlined, Gtk::PACK_SHRINK);

  show_all_children();
}

ExampleWindow::~ExampleWindow()
{
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

7.2. Entry

7.2.1. Simple Use

Entry widgets allow the user to enter text. You can change the contents with the `set_text()` method, and read the current contents with the `get_text()` method.

Occasionally you might want to make an `Entry` widget read-only. This can be done by passing `false` to the `set_editable()` method.

For the input of passwords, passphrases and other information you don't want echoed on the screen, calling `set_visibility()` with `false` will cause the text to be hidden.

You might want to be notified whenever the user types in a text entry widget. `Gtk::Entry` provides two signals, `activate` and `changed`, for just this purpose. `activate` is emitted when the user presses the enter key in a text-entry widget; `changed` is emitted when the text in the widget changes. You can use these, for instance, to validate or filter the text the user types.

Reference

7.2.1.1. Simple Entry Example

This example uses `Gtk::Entry`. It also has two `CheckButton`s, with which you can toggle the editable and visible flags.

Figure 7-2 **Entry**

Source Code

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <gtkmm.h>

class ExampleWindow : public Gtk::Window
{
public:
  ExampleWindow();
  virtual ~ExampleWindow();

protected:
  //Signal handlers:
  void on_checkbox_editable_toggled();
  void on_checkbox_visibility_toggled();
  void on_button_close();

  //Child widgets:
  Gtk::Box m_HBox;
  Gtk::Box m_VBox;
  Gtk::Entry m_Entry;
  Gtk::Button m_Button_Close;
  Gtk::CheckButton m_CheckButton_Editable, m_CheckButton_Visible;
};

#endif //GTKMM_EXAMPLEWINDOW_H
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <iostream>

ExampleWindow::ExampleWindow()
: m_VBox(Gtk::ORIENTATION_VERTICAL),
  m_Button_Close("Close"),
  m_CheckButton_Editable("Editable"),
  m_CheckButton_Visible("Visible")
{
  set_size_request(200, 100);
  set_title("Gtk::Entry");

  add(m_VBox);

  m_Entry.set_max_length(50);
  m_Entry.set_text("hello");
  m_Entry.set_text(m_Entry.get_text() + " world");
  m_Entry.select_region(0, m_Entry.get_text_length());
  m_VBox.pack_start(m_Entry);

  // Note that add() can also be used instead of pack_xxx()
  m_VBox.add(m_HBox);

  m_HBox.pack_start(m_CheckButton_Editable);
  m_CheckButton_Editable.signal_toggled().connect( sigc::mem_fun(*this,
              &ExampleWindow::on_checkbox_editable_toggled) );
  m_CheckButton_Editable.set_active(true);

  m_HBox.pack_start(m_CheckButton_Visible);
  m_CheckButton_Visible.signal_toggled().connect( sigc::mem_fun(*this,
              &ExampleWindow::on_checkbox_visibility_toggled) );
  m_CheckButton_Visible.set_active(true);

  m_Button_Close.signal_clicked().connect( sigc::mem_fun(*this,
              &ExampleWindow::on_button_close) );
  m_VBox.pack_start(m_Button_Close);
  m_Button_Close.set_can_default();
  m_Button_Close.grab_default();

  show_all_children();
}

ExampleWindow::~ExampleWindow()
{
}

void ExampleWindow::on_checkbox_editable_toggled()
{
  m_Entry.set_editable(m_CheckButton_Editable.get_active());
}

void ExampleWindow::on_checkbox_visibility_toggled()
{
  m_Entry.set_visibility(m_CheckButton_Visible.get_active());
}

void ExampleWindow::on_button_close()
{
  hide();
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

7.2.2. Entry Completion

A `Entry` widget can offer a drop-down list of pre-existing choices based on the first few characters typed by the user. For instance, a search dialog could suggest text from previous searches.

To enable this functionality, you must create a `EntryCompletion` object, and provide it to the `Entry` widget via the `set_completion()` method.

The `EntryCompletion` may use a `TreeModel` containing possible entries, specified with `set_model()`. You should then call `set_text_column()` to specify which of your model columns should be used to match possible text entries.

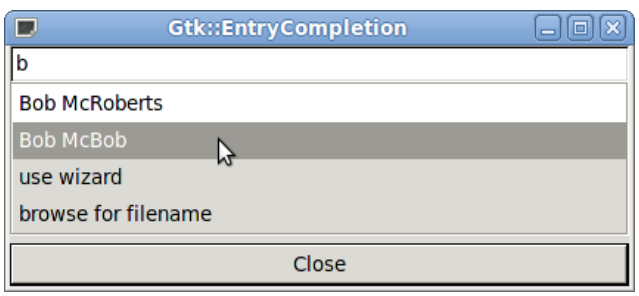
Alternatively, if a complete list of possible entries would be too large or too inconvenient to generate, a callback slot may instead be specified with `set_match_func()`. That callback function. This is also useful if you wish to match on a part of the string other than the start.

Reference

7.2.2.1. Entry Completion Example

This example creates a `Gtk::EntryCompletion` and associates it with a `Gtk::Entry` widget. The completion uses a `Gtk::TreeModel` of possible entries, and some additional actions.

Figure 7-3 **Entry Completion**



Source Code

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <gtkmm.h>

class ExampleWindow : public Gtk::Window
{
public:
  ExampleWindow();
  virtual ~ExampleWindow();

protected:
  //Signal handlers:
  void on_button_close();

  void on_completion_action_activated(int index);

  //See the comment in the implementation:
  //bool on_completion_match(const Glib::ustring& key, const Gtk::TreeModel::const_iterator& iter);


  //Tree model columns, for the EntryCompletion's filter model:
  class ModelColumns : public Gtk::TreeModel::ColumnRecord
  {
  public:

    ModelColumns()
    { add(m_col_id); add(m_col_name); }

    Gtk::TreeModelColumn<unsigned int> m_col_id;
    Gtk::TreeModelColumn<Glib::ustring> m_col_name;
  };
```

```
  ModelColumns m_Columns;

  typedef std::map<int, Glib::ustring> type_actions_map;
  type_actions_map m_CompletionActions;

  //Child widgets:
  Gtk::Box m_HBox;
  Gtk::Box m_VBox;
  Gtk::Entry m_Entry;
  Gtk::Label m_Label;
  Gtk::Button m_Button_Close;
};

#endif //GTKMM_EXAMPLEWINDOW_H
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <iostream>

ExampleWindow::ExampleWindow()
: m_VBox(Gtk::ORIENTATION_VERTICAL),
  m_Label("Press a or b to see a list of possible completions and actions."),
  m_Button_Close("Close")
{
  //set_size_request(200, 100);
  set_title("Gtk::EntryCompletion");

  add(m_VBox);
  m_VBox.pack_start(m_Entry, Gtk::PACK_SHRINK);

  m_VBox.pack_start(m_Label, Gtk::PACK_EXPAND_WIDGET);

  m_Button_Close.signal_clicked().connect( sigc::mem_fun(*this,
              &ExampleWindow::on_button_close) );
  m_VBox.pack_start(m_Button_Close, Gtk::PACK_SHRINK);
  m_Button_Close.set_can_default();
  m_Button_Close.grab_default();

  //Add an EntryCompletion:
  Glib::RefPtr<Gtk::EntryCompletion> completion =
      Gtk::EntryCompletion::create();
  m_Entry.set_completion(completion);

  //Create and fill the completion's filter model
  Glib::RefPtr<Gtk::ListStore> refCompletionModel =
      Gtk::ListStore::create(m_Columns);
  completion->set_model(refCompletionModel);

  // For more complex comparisons, use a filter match callback, like this.
  // See the comment below for more details:
  //completion->set_match_func( sigc::mem_fun(*this,
              //&ExampleWindow::on_completion_match) );

  //Fill the TreeView's model
  Gtk::TreeModel::Row row = *(refCompletionModel->append());
  row[m_Columns.m_col_id] = 1;
  row[m_Columns.m_col_name] = "Alan Zebedee";

  row = *(refCompletionModel->append());
  row[m_Columns.m_col_id] = 2;
  row[m_Columns.m_col_name] = "Adrian Boo";

  row = *(refCompletionModel->append());
  row[m_Columns.m_col_id] = 3;
  row[m_Columns.m_col_name] = "Bob McRoberts";

  row = *(refCompletionModel->append());
  row[m_Columns.m_col_id] = 4;
  row[m_Columns.m_col_name] = "Bob McBob";

  //Tell the completion what model column to use to
  //- look for a match (when we use the default matching, instead of
  //  set_match_func().
  //- display text in the entry when a match is found.
  completion->set_text_column(m_Columns.m_col_name);

  //Add actions to the completion:
  //These are just extra items shown at the bottom of the list of possible
  //completions.

  //Remember them for later.
  m_CompletionActions[0] = "Use Wizard";
  m_CompletionActions[1] = "Browse for Filename";
```

```
  for(type_actions_map::iterator iter = m_CompletionActions.begin();
          iter != m_CompletionActions.end(); ++iter)
  {
    int position = iter->first;
    Glib::ustring title = iter->second;
    completion->insert_action_text(title, position);
  }

  completion->signal_action_activated().connect( sigc::mem_fun(*this,
              &ExampleWindow::on_completion_action_activated) );

  show_all_children();
}

ExampleWindow::~ExampleWindow()
{
}

void ExampleWindow::on_button_close()
{
  hide();
}

/* You can do more complex matching with a handler like this.
 * For instance, you could check for substrings inside the string instead of the start,
 * or you could look for the key in extra model columns as well as the model column that will be displayed.
 * The code here is not actually more complex - it's a reimplementation of the default behaviour.
 *
bool ExampleWindow::on_completion_match(const Glib::ustring& key, const
        Gtk::TreeModel::const_iterator& iter)
{
  if(iter)
  {
    Gtk::TreeModel::Row row = *iter;

    Glib::ustring::size_type key_length = key.size();
    Glib::ustring filter_string = row[m_Columns.m_col_name];

    Glib::ustring filter_string_start = filter_string.substr(0, key_length);
    //The key is lower-case, even if the user input is not.
    filter_string_start = filter_string_start.lowercase();

    if(key == filter_string_start)
      return true; //A match was found.
  }

  return false; //No match.
}
*/

void ExampleWindow::on_completion_action_activated(int index)
{
  type_actions_map::iterator iter = m_CompletionActions.find(index);
  if(iter != m_CompletionActions.end()) //If it's in the map
  {
    Glib::ustring title = iter->second;
    std::cout << "Action selected: " << title << std::endl;
  }
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

7.2.3. Entry Icons

An `Entry` widget can show an icon at the start or end of the text area. The icon can be specifed by methods such as `set_icon_from_pixbuf()` or `set_icon_from_stock()`. An application can respond to the user pressing the icon by handling the `signal_icon_press`

signal.

7.2.3.1. Entry Icon Example

This example shows a `Gtk::Entry` widget with a stock search icon, and prints text to the terminal when the icon is pressed.

Figure 7-4 **Entry with Icon**

Source Code

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <gtkmm.h>

class ExampleWindow : public Gtk::Window
{
public:
  ExampleWindow();
  virtual ~ExampleWindow();

protected:
  //Signal handlers:
  void on_icon_pressed(Gtk::EntryIconPosition icon_pos, const GdkEventButton* event);
  void on_button_close();

  //Child widgets:
  Gtk::Box m_VBox;
  Gtk::Entry m_Entry;
  Gtk::Button m_Button_Close;
};

#endif //GTKMM_EXAMPLEWINDOW_H
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <iostream>

ExampleWindow::ExampleWindow()
: m_VBox(Gtk::ORIENTATION_VERTICAL),
  m_Button_Close("Close")
{
  set_title("Gtk::Entry");

  add(m_VBox);

  m_Entry.set_max_length(50);
  m_Entry.set_text("Hello world");
  m_VBox.pack_start(m_Entry, Gtk::PACK_SHRINK);

  m_Entry.set_icon_from_stock(Gtk::Stock::FIND);
  m_Entry.signal_icon_press().connect( sigc::mem_fun(*this, &ExampleWindow::on_icon_pressed) );


  m_Button_Close.signal_clicked().connect( sigc::mem_fun(*this,
              &ExampleWindow::on_button_close) );
  m_VBox.pack_start(m_Button_Close, Gtk::PACK_SHRINK);
  m_Button_Close.set_can_default();
  m_Button_Close.grab_default();

  show_all_children();
}

ExampleWindow::~ExampleWindow()
{
}

void ExampleWindow::on_icon_pressed(Gtk::EntryIconPosition /* icon_pos */, const GdkEventButton* /* event */)
{
  std::cout << "Icon pressed." << std::endl;
}
```

```
void ExampleWindow::on_button_close()
{
  hide();
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

7.2.4. Entry Progress

An `Entry` widget can show a progress bar inside the text area, under the entered text. The progress bar will be shown if the `set_progress_fraction()` or `set_progress_pulse_step()` methods are called.

7.2.4.1. Entry Progress Example

This example shows a `Gtk::Entry` widget with a progress bar.

Figure 7-5 **Entry with Progress Bar**

Source Code

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <gtkmm.h>

class ExampleWindow : public Gtk::Window
{
public:
  ExampleWindow();
  virtual ~ExampleWindow();

protected:
  //Signal handlers:
  bool on_timeout();
  void on_button_close();

  //Child widgets:
  Gtk::Box m_VBox;
  Gtk::Entry m_Entry;
  Gtk::Button m_Button_Close;
};

#endif //GTKMM_EXAMPLEWINDOW_H
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <iostream>

ExampleWindow::ExampleWindow()
: m_VBox(Gtk::ORIENTATION_VERTICAL),
```

```
  m_Button_Close("Close")
{
  set_title("Gtk::Entry");

  add(m_VBox);

  m_Entry.set_max_length(50);
  m_Entry.set_text("Hello world");
  m_VBox.pack_start(m_Entry, Gtk::PACK_SHRINK);

  //Change the progress fraction every 0.1 second:
  Glib::signal_timeout().connect(
    sigc::mem_fun(*this, &ExampleWindow::on_timeout),
    100
  );

  m_Button_Close.signal_clicked().connect( sigc::mem_fun(*this,
              &ExampleWindow::on_button_close) );
  m_VBox.pack_start(m_Button_Close, Gtk::PACK_SHRINK);
  m_Button_Close.set_can_default();
  m_Button_Close.grab_default();

  show_all_children();
}

ExampleWindow::~ExampleWindow()
{
}

bool ExampleWindow::on_timeout()
{
  static double fraction = 0;
  m_Entry.set_progress_fraction(fraction);

  fraction += 0.01;
  if(fraction > 1)
    fraction = 0;

  return true;
}

void ExampleWindow::on_button_close()
{
  hide();
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

7.3. SpinButton

A `SpinButton` allows the user to select a value from a range of numeric values. It has an Entry widget with up and down arrow buttons at the side. Clicking the buttons causes the value to 'spin' up and down across the range of possible values. The `Entry` widget may also be used to enter a value directly.

The value can have an adjustable number of decimal places, and the step size is configurable. `SpinButton`s have an 'auto-repeat' feature as well: holding down one of the arrows can optionally cause the value to change more quickly the longer the arrow is held down.

`SpinButton`s use an Adjustment object to hold information about the range of values. These Adjustment attributes are used by the Spin Button like so:

- `value`: value for the Spin Button

- `lower`: lower range value
- `upper`: upper range value
- `step_increment`: value to increment/decrement when pressing mouse button 1 on a button
- `page_increment`: value to increment/decrement when pressing mouse button 2 on a button
- `page_size`: unused

Additionally, mouse button 3 can be used to jump directly to the `upper` or `lower` values.

The `SpinButton` can create a default `Adjustment`, which you can access via the `get_adjustment()` method, or you can specify an existing `Adjustment` in the constructor.

7.3.1. Methods

The number of decimal places can be altered using the `set_digits()` method.

You can set the spinbutton's value using the `set_value()` method, and retrieve it with `get_value()`.

The `spin()` method 'spins' the `SpinButton`, as if one of its arrows had been clicked. You need to specify a `Gtk::SpinType` to specify the direction or new position.

To prevent the user from typing non-numeric characters into the entry box, pass `true` to the `set_numeric()` method.

To make the `SpinButton` 'wrap' between its upper and lower bounds, use the `set_wrap()` method.

To force it to snap to the nearest `step_increment`, use `set_snap_to_ticks()`.

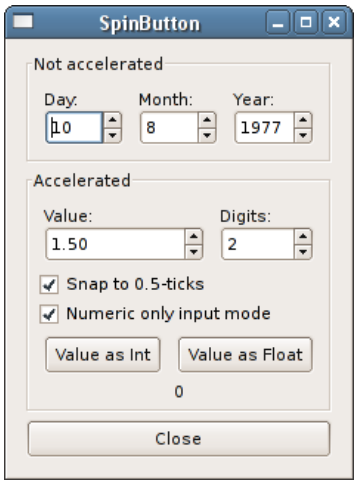
You can modify the update policy using the `set_update_policy()` method, specifying either `Gtk::UPDATE_ALWAYS` or `Gtk::UPDATE_IF_VALID`. `Gtk::UPDATE_ALWAYS` causes the `SpinButton` to ignore errors encountered while converting the text in the entry box to a numeric value. This setting also therefore allows the `SpinButton` to accept non-numeric values. You can force an immediate update using the `update()` method.

Reference

7.3.2. Example

Here's an example of a `SpinButton` in action:

Figure 7-6 **SpinButton**



Source Code

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <gtkmm.h>

class ExampleWindow : public Gtk::Window
{
public:
  ExampleWindow();
  virtual ~ExampleWindow();

protected:
  //Signal handlers:
  void on_checkbutton_snap();
  void on_checkbutton_numeric();
  void on_spinbutton_digits_changed();
  void on_button_close();

  enum enumValueFormats
  {
    VALUE_FORMAT_INT,
    VALUE_FORMAT_FLOAT
  };
  void on_button_getvalue(enumValueFormats display);

  //Child widgets:
  Gtk::Frame m_Frame_NotAccelerated, m_Frame_Accelerated;
  Gtk::Box m_HBox_NotAccelerated, m_HBox_Accelerated,
    m_HBox_Buttons;
  Gtk::Box m_VBox_Main, m_VBox, m_VBox_Day, m_VBox_Month, m_VBox_Year,
    m_VBox_Accelerated, m_VBox_Value, m_VBox_Digits;
  Gtk::Label m_Label_Day, m_Label_Month, m_Label_Year,
    m_Label_Value, m_Label_Digits,
    m_Label_ShowValue;
  Glib::RefPtr<Gtk::Adjustment> m_adjustment_day, m_adjustment_month, m_adjustment_year,
    m_adjustment_value, m_adjustment_digits;
  Gtk::SpinButton m_SpinButton_Day, m_SpinButton_Month, m_SpinButton_Year,
    m_SpinButton_Value, m_SpinButton_Digits;
  Gtk::CheckButton m_CheckButton_Snap, m_CheckButton_Numeric;
  Gtk::Button m_Button_Int, m_Button_Float, m_Button_Close;
};

#endif //GTKMM_EXAMPLEWINDOW_H
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <iostream>
#include <cstdio>

ExampleWindow::ExampleWindow()
:
  m_Frame_NotAccelerated("Not accelerated"),
  m_Frame_Accelerated("Accelerated"),
  m_VBox_Main(Gtk::ORIENTATION_VERTICAL, 5),
  m_Label_Day("Day: "),
  m_Label_Month("Month: "),
  m_Label_Year("Year: "),
  m_Label_Value("Value: "),
  m_Label_Digits("Digits: "),
  m_adjustment_day( Gtk::Adjustment::create(1.0, 1.0, 31.0, 1.0, 5.0, 0.0) ),
  m_adjustment_month( Gtk::Adjustment::create(1.0, 1.0, 12.0, 1.0, 5.0, 0.0) ),
  m_adjustment_year( Gtk::Adjustment::create(1998.0, 0.0, 2100.0, 1.0, 100.0, 0.0) ),
  m_adjustment_value( Gtk::Adjustment::create(0.0, -10000.0, 10000.0, 0.5, 100.0, 0.0) ),
  m_adjustment_digits( Gtk::Adjustment::create(2.0, 1.0, 5.0, 1.0, 1.0, 0.0) ),
  m_SpinButton_Day(m_adjustment_day),
  m_SpinButton_Month(m_adjustment_month),
  m_SpinButton_Year(m_adjustment_year),
  m_SpinButton_Value(m_adjustment_value, 1.0, 2),
  m_SpinButton_Digits(m_adjustment_digits),
  m_CheckButton_Snap("Snap to 0.5-ticks"),
  m_CheckButton_Numeric("Numeric only input mode"),
  m_Button_Int("Value as Int"),
  m_Button_Float("Value as Float"),
  m_Button_Close("Close")
{
  set_title("SpinButton");

  m_VBox_Main.set_border_width(10);
```

```
  add(m_VBox_Main);

  m_VBox_Main.pack_start(m_Frame_NotAccelerated);

  m_VBox.set_border_width(5);
  m_Frame_NotAccelerated.add(m_VBox);

  /* Day, month, year spinners */

  m_VBox.pack_start(m_HBox_NotAccelerated, Gtk::PACK_EXPAND_WIDGET, 5);

  m_Label_Day.set_alignment(Gtk::ALIGN_START);
  m_VBox_Day.pack_start(m_Label_Day);

  m_SpinButton_Day.set_wrap();

  m_VBox_Day.pack_start(m_SpinButton_Day);

  m_HBox_NotAccelerated.pack_start(m_VBox_Day, Gtk::PACK_EXPAND_WIDGET, 5);

  m_Label_Month.set_alignment(Gtk::ALIGN_START);
  m_VBox_Month.pack_start(m_Label_Month);

  m_SpinButton_Month.set_wrap();
  m_VBox_Month.pack_start(m_SpinButton_Month);

  m_HBox_NotAccelerated.pack_start(m_VBox_Month, Gtk::PACK_EXPAND_WIDGET, 5);

  m_Label_Year.set_alignment(Gtk::ALIGN_START);
  m_VBox_Year.pack_start(m_Label_Year);

  m_SpinButton_Year.set_wrap();
  m_SpinButton_Year.set_size_request(55, -1);
  m_VBox_Year.pack_start(m_SpinButton_Year);

  m_HBox_NotAccelerated.pack_start(m_VBox_Year, Gtk::PACK_EXPAND_WIDGET, 5);

  //Accelerated:
  m_VBox_Main.pack_start(m_Frame_Accelerated);

  m_VBox_Accelerated.set_border_width(5);
  m_Frame_Accelerated.add(m_VBox_Accelerated);

  m_VBox_Accelerated.pack_start(m_HBox_Accelerated, Gtk::PACK_EXPAND_WIDGET, 5);

  m_HBox_Accelerated.pack_start(m_VBox_Value, Gtk::PACK_EXPAND_WIDGET, 5);

  m_Label_Value.set_alignment(Gtk::ALIGN_START);
  m_VBox_Value.pack_start(m_Label_Value);

  m_SpinButton_Value.set_wrap();
  m_SpinButton_Value.set_size_request(100, -1);
  m_VBox_Value.pack_start(m_SpinButton_Value);

  m_HBox_Accelerated.pack_start(m_VBox_Digits, Gtk::PACK_EXPAND_WIDGET, 5);

  m_Label_Digits.set_alignment(Gtk::ALIGN_START);
  m_VBox_Digits.pack_start(m_Label_Digits);

  m_SpinButton_Digits.set_wrap();
  m_adjustment_digits->signal_value_changed().connect( sigc::mem_fun(*this,
              &ExampleWindow::on_spinbutton_digits_changed) );

  m_VBox_Digits.pack_start(m_SpinButton_Digits);


  //CheckButtons:
  m_VBox_Accelerated.pack_start(m_CheckButton_Snap);
  m_CheckButton_Snap.set_active();
  m_CheckButton_Snap.signal_clicked().connect( sigc::mem_fun(*this,
              &ExampleWindow::on_checkbutton_snap) );

  m_VBox_Accelerated.pack_start(m_CheckButton_Numeric);
  m_CheckButton_Numeric.set_active();
  m_CheckButton_Numeric.signal_clicked().connect( sigc::mem_fun(*this,
              &ExampleWindow::on_checkbutton_numeric) );


  //Buttons:
  m_VBox_Accelerated.pack_start (m_HBox_Buttons, Gtk::PACK_SHRINK, 5);

  m_Button_Int.signal_clicked().connect( sigc::bind( sigc::mem_fun(*this,
                  &ExampleWindow::on_button_getvalue), VALUE_FORMAT_INT) );
  m_HBox_Buttons.pack_start(m_Button_Int, Gtk::PACK_EXPAND_WIDGET, 5);

  m_Button_Float.signal_clicked().connect( sigc::bind( sigc::mem_fun(*this,
                  &ExampleWindow::on_button_getvalue), VALUE_FORMAT_FLOAT) );
  m_HBox_Buttons.pack_start(m_Button_Float, Gtk::PACK_EXPAND_WIDGET, 5);
```

```
  m_VBox_Accelerated.pack_start(m_Label_ShowValue);
  m_Label_ShowValue.set_text("0");

  //Close button:
  m_Button_Close.signal_clicked().connect( sigc::mem_fun(*this,
              &ExampleWindow::on_button_close) );
  m_VBox_Main.pack_start(m_Button_Close, Gtk::PACK_SHRINK);

  show_all_children();
}

ExampleWindow::~ExampleWindow()
{
}


void ExampleWindow::on_button_close()
{
  hide();
}

void ExampleWindow::on_checkbutton_snap()
{
  m_SpinButton_Value.set_snap_to_ticks( m_CheckButton_Snap.get_active() );
}

void ExampleWindow::on_checkbutton_numeric()
{
  m_SpinButton_Value.set_numeric( m_CheckButton_Numeric.get_active() );
}

void ExampleWindow::on_spinbutton_digits_changed()
{
  m_SpinButton_Value.set_digits( m_SpinButton_Digits.get_value_as_int() );
}

void ExampleWindow::on_button_getvalue(enumValueFormats display)
{
  gchar buf[32];

  if (display == VALUE_FORMAT_INT)
    sprintf (buf, "%d", m_SpinButton_Value.get_value_as_int());
  else
    sprintf (buf, "%0.*f", m_SpinButton_Value.get_digits(),
             m_SpinButton_Value.get_value());

  m_Label_ShowValue.set_text(buf);
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

7.4. ProgressBar

Progress bars are used to show the status of an ongoing operation. For instance, a `ProgressBar` can show how much of a task has been completed.

To change the value shown, use the `set_fraction()` method, passing a double between 0 and 1 to provide the new percentage.

where `percentage` is a number, from 0 to 1, indicating what fraction of the bar should be filled.

A `ProgressBar`is horizontal and left-to-right by default, but you can change it to a vertical progress bar by using the `set_orientation()` method.

Reference

7.4.1. Activity Mode

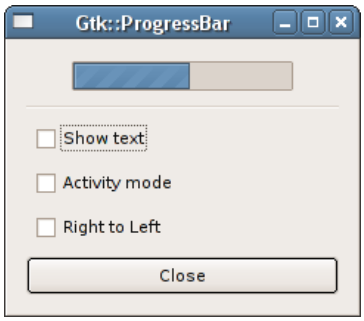
Besides indicating the amount of progress that has occured, the progress bar can also be used to indicate that there is some activity; this is done by placing the progress bar in *activity mode*. In this mode, the progress bar displays a small rectangle which moves back and forth. Activity mode is useful in situations where the progress of an operation cannot be calculated as a value range (e.g., receiving a file of unknown length).

To do this, you need to call the `pulse()` method at regular intervals. You can also choose the step size, with the `set_pulse_step()` method.

When in continuous mode, the progress bar can also display a configurable text string within its trough, using the `set_text()` method.

7.4.2. Example

Figure 7-7 **ProgressBar**



Source Code

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <gtkmm.h>

class ExampleWindow : public Gtk::Window
{
public:
  ExampleWindow();
  virtual ~ExampleWindow();

protected:
  //Signal handlers:
  void on_checkbutton_text();
  void on_checkbutton_activity();
  void on_checkbutton_inverted();
  virtual bool on_timeout();
  void on_button_close();

  //Child widgets:
  Gtk::Box m_VBox;
  Gtk::Alignment m_Alignment;
  Gtk::Table m_Table;
  Gtk::ProgressBar m_ProgressBar;
  Gtk::Separator m_Separator;
  Gtk::CheckButton m_CheckButton_Text, m_CheckButton_Activity, m_CheckButton_Inverted;
  Gtk::Button m_Button_Close;

  int m_connection_id_timeout;
  bool m_bActivityMode;
};

#endif //GTKMM_EXAMPLEWINDOW_H
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <iostream>

ExampleWindow::ExampleWindow()
: m_VBox(Gtk::ORIENTATION_VERTICAL, 5),
  m_Alignment(0.5, 0.5, 0, 0),
  m_Table(2, 2, true),
  m_CheckButton_Text("Show text"),
  m_CheckButton_Activity("Activity mode"),
  m_CheckButton_Inverted("Right to Left"),
  m_Button_Close("Close"),
  m_bActivityMode(false)
{
  set_resizable();
  set_title("Gtk::ProgressBar");

  m_VBox.set_border_width(10);
  add(m_VBox);

  m_VBox.pack_start(m_Alignment, Gtk::PACK_SHRINK, 5);
  m_Alignment.add(m_ProgressBar);
  m_ProgressBar.set_text("some text");
  m_ProgressBar.set_show_text(false);

  //Add a timer callback to update the value of the progress bar:
  m_connection_id_timeout = Glib::signal_timeout().connect(sigc::mem_fun(*this,
              &ExampleWindow::on_timeout), 50 );

  m_VBox.pack_start(m_Separator, Gtk::PACK_SHRINK);
  m_VBox.pack_start(m_Table);

  //Add a check button to select displaying of the trough text:
  m_Table.attach(m_CheckButton_Text, 0, 1, 0, 1, Gtk::EXPAND | Gtk::FILL,
         Gtk::EXPAND | Gtk::FILL, 5, 5);
  m_CheckButton_Text.signal_clicked().connect(sigc::mem_fun(*this,
              &ExampleWindow::on_checkbutton_text) );

  //Add a check button to toggle activity mode:
  m_Table.attach(m_CheckButton_Activity, 0, 1, 1, 2, Gtk::EXPAND | Gtk::FILL,
         Gtk::EXPAND | Gtk::FILL, 5, 5);
  m_CheckButton_Activity.signal_clicked().connect(sigc::mem_fun(*this,
              &ExampleWindow::on_checkbutton_activity) );

  //Add a check button to select growth from left to right or from right to left:
  m_Table.attach(m_CheckButton_Inverted, 0, 1, 2, 3, Gtk::EXPAND | Gtk::FILL,
         Gtk::EXPAND | Gtk::FILL, 5, 5);
  m_CheckButton_Inverted.signal_clicked().connect(sigc::mem_fun(*this,
              &ExampleWindow::on_checkbutton_inverted) );

  //Add a button to exit the program.
  m_VBox.pack_start(m_Button_Close, Gtk::PACK_SHRINK);
  m_Button_Close.signal_clicked().connect(sigc::mem_fun(*this,
              &ExampleWindow::on_button_close) );
  m_Button_Close.set_can_default();
  m_Button_Close.grab_default();

  show_all_children();
}

ExampleWindow::~ExampleWindow()
{
}

void ExampleWindow::on_checkbutton_text()
{
  const bool show_text = m_CheckButton_Text.get_active();
  m_ProgressBar.set_show_text(show_text);
}

void ExampleWindow::on_checkbutton_activity()
{
  m_bActivityMode = m_CheckButton_Activity.get_active();

  if(m_bActivityMode)
    m_ProgressBar.pulse();
  else
    m_ProgressBar.set_fraction(0.0);
}

void ExampleWindow::on_checkbutton_inverted()
{
  const bool inverted = m_CheckButton_Inverted.get_active();
  m_ProgressBar.set_inverted(inverted);
}
```

```
void ExampleWindow::on_button_close()
{
  hide();
}

/* Update the value of the progress bar so that we get
 * some movement */
bool ExampleWindow::on_timeout()
{
  if(m_bActivityMode)
    m_ProgressBar.pulse();
  else
  {
    double new_val = m_ProgressBar.get_fraction() + 0.01;

    if(new_val > 1.0)
      new_val = 0.0;

    //Set the new value:
    m_ProgressBar.set_fraction(new_val);
  }

  //As this is a timeout function, return true so that it
  //continues to get called
  return true;
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

7.5. InfoBar

An `InfoBar` may show small items of information or to ask brief questions. Unlike a `Dialog`, it appears at the top of the current window instead of opening a new window. Its API is very similar to the Gtk::Dialog API.

Reference

7.5.1. Example

Figure 7-8 **InfoBar**

Source Code

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <gtkmm.h>

class ExampleWindow : public Gtk::Window
{
public:
  ExampleWindow();
  virtual ~ExampleWindow();

protected:
  //Signal handlers:
  void on_infobar_response(int response);
  void on_button_quit();
  void on_button_clear();
  void on_textbuffer_changed();

  //Child widgets:
  Gtk::Box m_VBox;

  Gtk::ScrolledWindow m_ScrolledWindow;
  Gtk::TextView m_TextView;

  Glib::RefPtr<Gtk::TextBuffer> m_refTextBuffer;

  Gtk::InfoBar m_InfoBar;
  Gtk::Label m_Message_Label;

  Gtk::ButtonBox m_ButtonBox;
  Gtk::Button m_Button_Quit, m_Button_Clear;
};

#endif //GTKMM_EXAMPLEWINDOW_H
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"

ExampleWindow::ExampleWindow()
: m_VBox(Gtk::ORIENTATION_VERTICAL, 6),
  m_Button_Quit(Gtk::Stock::QUIT),
  m_Button_Clear("Clear")
{
  set_title("Gtk::InfoBar example");
  set_border_width(6);
  set_default_size(400, 200);

  add(m_VBox);

  // Add the message label to the InfoBar:
  Gtk::Container* infoBarContainer =
    dynamic_cast<Gtk::Container*>(m_InfoBar.get_content_area());
  if (infoBarContainer)
    infoBarContainer->add(m_Message_Label);

  // Add an ok button to the InfoBar:
  m_InfoBar.add_button(Gtk::Stock::OK, 0);

  // Add the InfoBar to the vbox:
  m_VBox.pack_start(m_InfoBar, Gtk::PACK_SHRINK);

  // Create the buffer and set it for the TextView:
  m_refTextBuffer = Gtk::TextBuffer::create();
  m_TextView.set_buffer(m_refTextBuffer);

  // Add the TreeView, inside a ScrolledWindow:
  m_ScrolledWindow.add(m_TextView);

  // Show the scrollbars only when they are necessary:
  m_ScrolledWindow.set_policy(Gtk::POLICY_AUTOMATIC, Gtk::POLICY_AUTOMATIC);

  m_VBox.pack_start(m_ScrolledWindow);

  // Add button box:
  m_VBox.pack_start(m_ButtonBox, Gtk::PACK_SHRINK);

  m_ButtonBox.pack_start(m_Button_Clear, Gtk::PACK_SHRINK);
  m_ButtonBox.pack_start(m_Button_Quit, Gtk::PACK_SHRINK);
  m_ButtonBox.set_spacing(6);
  m_ButtonBox.set_layout(Gtk::BUTTONBOX_END);
```

```
  // Connect signals:
  m_InfoBar.signal_response().connect(sigc::mem_fun(*this,
              &ExampleWindow::on_infobar_response) );
  m_Button_Quit.signal_clicked().connect(sigc::mem_fun(*this,
              &ExampleWindow::on_button_quit) );
  m_Button_Clear.signal_clicked().connect(sigc::mem_fun(*this,
              &ExampleWindow::on_button_clear) );
  m_refTextBuffer->signal_changed().connect(sigc::mem_fun(*this,
              &ExampleWindow::on_textbuffer_changed) );

  show_all_children();

  // Keep the InfoBar hidden until a message needs to be shown:
  m_InfoBar.hide();

  // Make the clear button insensitive until text is typed in the buffer.  When
  // the button is sensitive and it is pressed, the InfoBar is displayed with a
  // message.
  m_Button_Clear.set_sensitive(false);
}

ExampleWindow::~ExampleWindow()
{
}

void ExampleWindow::on_infobar_response(int)
{
  // Clear the message and hide the info bar:
  m_Message_Label.set_text("");
  m_InfoBar.hide();
}

void ExampleWindow::on_button_quit()
{
  hide();
}

void ExampleWindow::on_button_clear()
{
  m_refTextBuffer->set_text("");
  m_Message_Label.set_text("Cleared the text.");
  m_InfoBar.set_message_type(Gtk::MESSAGE_INFO);
  m_InfoBar.show();
}

void ExampleWindow::on_textbuffer_changed()
{
  m_Button_Clear.set_sensitive(m_refTextBuffer->size() > 0);
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

7.6. Tooltips

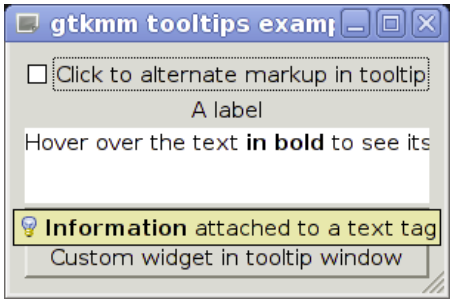
Tooltips are the little information windows that pop up when you leave your pointer over a widget for a few seconds. Use `set_tooltip_text()` to set a text string as a tooltip on any `Widget`. `Gtk::ToolItem`s are not `Widget`s, but have the same method for convenience. `Gtk::Tooltip` is used for more advanced tooltip usage, such as showing an image as well as text.

Widget Reference

Tooltip Reference

7.6.1. Example

Figure 7-9 **Tooltip**



Source Code

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <gtkmm.h>

class ExampleWindow : public Gtk::Window
{
public:

  ExampleWindow();
  virtual ~ExampleWindow();

protected:

  //Methods:
  void prepare_textview();
  void connect_signals();

  //Signal handlers:
  void on_markup_checkbutton_click();
  bool on_textview_query_tooltip(int x, int y, bool keyboard_tooltip, const Glib::RefPtr<Gtk::Tooltip>& tooltip);
  bool on_button_query_tooltip(int x, int y, bool keyboard_tooltip, const Glib::RefPtr<Gtk::Tooltip>& tooltip);

  //Child widgets:
  Gtk::Box m_vbox;

  Gtk::CheckButton m_checkbutton;
  Gtk::Label m_label;

  Gtk::ScrolledWindow m_scrolled_window;
  Gtk::TextView m_text_view;
  Glib::RefPtr<Gtk::TextBuffer> m_ref_text_buffer;
  Glib::RefPtr<Gtk::TextTag> m_ref_bold_tag;

  Gtk::Button m_button;
  Gtk::Window m_button_tooltip_window;

};

#endif // GTKMM_EXAMPLEWINDOW_H
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"

#include <vector>

const Glib::ustring app_title = "gtkmm tooltips example";
const Glib::ustring non_markedup_tip = "A tooltip without markup.";
const Glib::ustring markedup_tip = "<i>Markup</i> in a tooltip.";

ExampleWindow::ExampleWindow()
  :
  m_vbox(Gtk::ORIENTATION_VERTICAL, 3),
  m_checkbutton("Click to alternate markup in tooltip"),
  m_label("A label"),
  m_button("Custom widget in tooltip window"),
```

```
  m_button_tooltip_window(Gtk::WINDOW_POPUP)
{
  //Set up window and the top-level container:
  set_title(app_title);
  set_border_width(10);

  add(m_vbox);

  //Check button with markup in tooltip:
  m_checkbutton.set_tooltip_text(non_markedup_tip);
  m_vbox.pack_start(m_checkbutton, Gtk::PACK_SHRINK);

  //Label:
  m_label.set_tooltip_text("Another tooltip");
  m_vbox.pack_start(m_label, Gtk::PACK_SHRINK);

  //Textview:
  prepare_textview();

  //Button:
  // set_tooltip_window(), like set_tooltip_text(),
  // will call set_has_tooltip() for us.
  m_button.set_tooltip_window(m_button_tooltip_window);
  m_vbox.pack_start(m_button, Gtk::PACK_SHRINK);

  //Button's custom tooltip window:
  m_button_tooltip_window.set_default_size(250, 30);
  Gtk::Label* label =
    Gtk::manage(new Gtk::Label("A label in a custom tooltip window"));
  label->show();
  m_button_tooltip_window.add(*label);

  connect_signals();

  show_all_children();
}

ExampleWindow::~ExampleWindow()
{
}

void ExampleWindow::prepare_textview()
{
  Gtk::TextIter iter;
  std::vector< Glib::RefPtr<Gtk::TextTag> > tags;

  //Set up a scrolled window:
  m_scrolled_window.add(m_text_view);
  m_scrolled_window.set_policy(Gtk::POLICY_AUTOMATIC, Gtk::POLICY_AUTOMATIC);
  m_vbox.pack_start(m_scrolled_window);

  //Create a text buffer with some text:
  m_ref_text_buffer = Gtk::TextBuffer::create();

  iter = m_ref_text_buffer->end();
  m_ref_text_buffer->insert(iter, "Hover over the text ");

  //Insert some text with a tag.
  //In the tooltip signal handler below, we will show a tooltip
  //when mouse pointer is above this tagged text.
  m_ref_bold_tag = m_ref_text_buffer->create_tag("bold");
  m_ref_bold_tag->set_property("weight", Pango::WEIGHT_BOLD);

  tags.push_back(m_ref_bold_tag);

  iter = m_ref_text_buffer->end();
  m_ref_text_buffer->insert_with_tags(iter, "in bold", tags);

  iter = m_ref_text_buffer->end();
  m_ref_text_buffer->insert(iter, " to see its' tooltip");

  m_text_view.set_buffer(m_ref_text_buffer);

  m_text_view.set_size_request(320, 50);

  //When only connecting to the query-tooltip signal, and not using any
  //of set_tooltip_text(), set_tooltip_markup() or set_tooltip_window(),
  //we need to explicitly tell GTK+ that the widget has a tooltip which
  //we'll show.
  m_text_view.set_has_tooltip();
}

void ExampleWindow::connect_signals()
{
  m_checkbutton.signal_clicked().connect(
    sigc::mem_fun(*this, &ExampleWindow::on_markup_checkbutton_click));

  m_text_view.signal_query_tooltip().connect(
```

```
    sigc::mem_fun(*this, &ExampleWindow::on_textview_query_tooltip));

  m_button.signal_query_tooltip().connect(
    sigc::mem_fun(*this, &ExampleWindow::on_button_query_tooltip));
}

void ExampleWindow::on_markup_checkbutton_click()
{
  if (m_checkbutton.get_active() == true)
  {
    m_checkbutton.set_tooltip_markup(markedup_tip);
  }
  else
  {
    m_checkbutton.set_tooltip_markup(non_markedup_tip);
  }
}

bool ExampleWindow::on_textview_query_tooltip(int x, int y, bool keyboard_tooltip, const Glib::RefPtr<Gtk::Tooltip>& tooltip)
{
  Gtk::TextIter iter;

  if (keyboard_tooltip)
  {
    int offset = m_ref_text_buffer->property_cursor_position().get_value();
    iter = m_ref_text_buffer->get_iter_at_offset(offset);
  }
  else
  {
    int mouse_x, mouse_y, trailing;
    m_text_view.window_to_buffer_coords(Gtk::TEXT_WINDOW_TEXT,
                                        x, y, mouse_x, mouse_y);
    m_text_view.get_iter_at_position(iter, trailing, mouse_x, mouse_y);
  }

  //Show a tooltip if the cursor or mouse pointer is over the text
  //with the specific tag:
  if (iter.has_tag(m_ref_bold_tag))
  {
    tooltip->set_markup("<b>Information</b> attached to a text tag");
    tooltip->set_icon_from_stock (Gtk::Stock::INFO, Gtk::ICON_SIZE_MENU);
  }
  else
  {
    return false;
  }

  return true;
}

bool ExampleWindow::on_button_query_tooltip(int, int, bool, const Glib::RefPtr<Gtk::Tooltip>&)
{
  //We already have a custom window ready, just return true to show it:
  return true;
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

8. Container Widgets

All container widgets derive from `Gtk::Container`, not always directly. Some container widgets, such as `Gtk::Table` can hold many child widgets, so these typically have more complex interfaces. Others, such as `Gtk::Frame` contain only one child widget.

8.1. Single-item Containers

The single-item container widgets derive from `Gtk::Bin`, which provides the `add()` and `remove()` methods for the child widget. Note that `Gtk::Button` and `Gtk::Window` are technically single-item containers, but we have discussed them already elsewhere.

We also discuss the `Gtk::Paned` widget, which allows you to divide a window into two separate "panes". This widget actually contains two child widgets, but the number is fixed so it seems appropriate.

8.1.1. Frame

Frames can enclose one or a group of widgets within a box, optionally with a title. For instance, you might place a group of `RadioButton`s or `CheckButton`s in a `Frame`.

Reference

8.1.1.1. Example

Figure 8-1 **Frame**

Source Code

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <gtkmm.h>

class ExampleWindow : public Gtk::Window
{
public:
  ExampleWindow();
  virtual ~ExampleWindow();

protected:

  //Child widgets:
  Gtk::Frame m_Frame;
};

#endif //GTKMM_EXAMPLEWINDOW_H
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"

ExampleWindow::ExampleWindow()
{
 /* Set some window properties */
  set_title("Frame Example");
  set_size_request(300, 300);

  /* Sets the border width of the window. */
  set_border_width(10);

  add(m_Frame);

  /* Set the frames label */
  m_Frame.set_label("Gtk::Frame Widget");

  /* Align the label at the right of the frame */
  //m_Frame.set_label_align(Gtk::ALIGN_END, Gtk::ALIGN_START);

  /* Set the style of the frame */
  m_Frame.set_shadow_type(Gtk::SHADOW_ETCHED_OUT);

  show_all_children();
}

ExampleWindow::~ExampleWindow()
{
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

8.1.2. Paned

Panes divide a widget into two halves, separated by a moveable divider. There are two such widgets: `Gtk::HPaned` adds a horizontal divider, and `Gtk::VPaned` adds a vertical one. Other than the names and the orientations, there's no difference between the two.

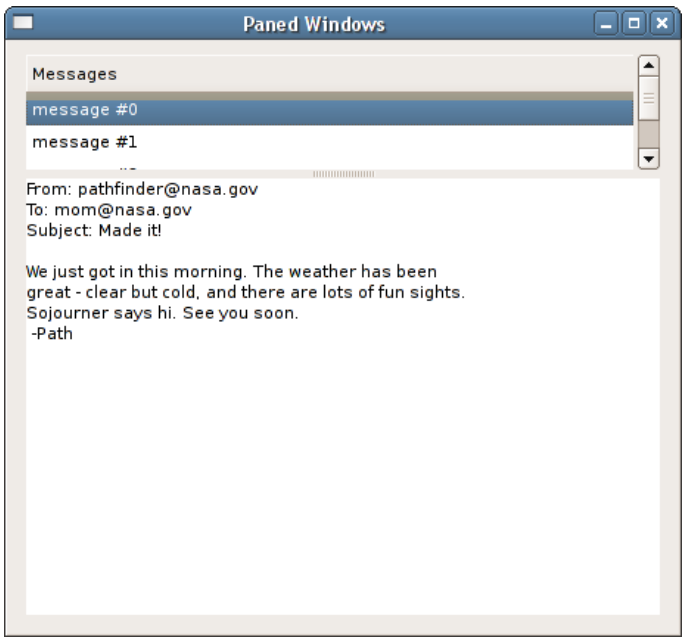
Unlike the other widgets in this chapter, pane widgets contain not one but two child widgets, one in each pane. Therefore, you should use `add1()` and `add2()` instead of the `add()` method.

You can adjust the position of the divider using the `set_position()` method, and you will probably need to do so.

Reference

8.1.2.1. Example

Figure 8-2 **Paned**



Source Code

File: `messageslist.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLE_MESSAGESLIST_H
#define GTKMM_EXAMPLE_MESSAGESLIST_H

#include <gtkmm.h>

class MessagesList: public Gtk::ScrolledWindow
{
public:
  MessagesList();
  virtual ~MessagesList();

  class ModelColumns : public Gtk::TreeModel::ColumnRecord
  {
  public:

    ModelColumns()
    { add(m_col_text); }

    Gtk::TreeModelColumn<Glib::ustring> m_col_text;
  };

  ModelColumns m_Columns;

protected:
  Glib::RefPtr<Gtk::ListStore> m_refListStore; //The Tree Model.
  Gtk::TreeView m_TreeView; //The Tree View.
};
#endif //GTKMM_EXAMPLE_MESSAGESLIST_H
```

File: `messagetext.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLE_MESSAGETEXT_H
```

```
#define GTKMM_EXAMPLE_MESSAGETEXT_H

#include <gtkmm.h>

class MessageText : public Gtk::ScrolledWindow
{
public:
  MessageText();
  virtual ~MessageText();

  void insert_text();

protected:
  Gtk::TextView m_TextView;
};

#endif //GTKMM_EXAMPLE_MESSAGETEXT_H
```

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include "messageslist.h"
#include "messagetext.h"
#include <gtkmm.h>

class ExampleWindow : public Gtk::Window
{
public:
  ExampleWindow();
  virtual ~ExampleWindow();

protected:

  //Child widgets:
  Gtk::Paned m_VPaned;
  MessagesList m_MessagesList;
  MessageText m_MessageText;
};

#endif //GTKMM_EXAMPLEWINDOW_H
```

File: `messagetext.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "messagetext.h"

MessageText::MessageText()
{
  set_policy(Gtk::POLICY_AUTOMATIC, Gtk::POLICY_AUTOMATIC);

  add(m_TextView);
  insert_text();

  show_all_children();
}

MessageText::~MessageText()
{
}

void MessageText::insert_text()
{
  Glib::RefPtr<Gtk::TextBuffer> refTextBuffer = m_TextView.get_buffer();

  Gtk::TextBuffer::iterator iter = refTextBuffer->get_iter_at_offset(0);
  refTextBuffer->insert(iter,
    "From: pathfinder@nasa.gov\n"
    "To: mom@nasa.gov\n"
    "Subject: Made it!\n"
    "\n"
    "We just got in this morning. The weather has been\n"
    "great - clear but cold, and there are lots of fun sights.\n"
    "Sojourner says hi. See you soon.\n"
    " -Path\n");
}
```

File: `messageslist.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "messageslist.h"
```

```
#include <sstream>

MessagesList::MessagesList()
{
  /* Create a new scrolled window, with scrollbars only if needed */
  set_policy(Gtk::POLICY_AUTOMATIC, Gtk::POLICY_AUTOMATIC);

  add(m_TreeView);

  /* create list store */
  m_refListStore = Gtk::ListStore::create(m_Columns);

  m_TreeView.set_model(m_refListStore);

  /* Add some messages to the window */
  for(int i = 0; i < 10; ++i)
  {
    std::ostringstream text;
    text << "message #" << i;

    Gtk::TreeModel::Row row = *(m_refListStore->append());
    row[m_Columns.m_col_text] = text.str();
  }

  //Add the Model's column to the View's columns:
  m_TreeView.append_column("Messages", m_Columns.m_col_text);

  show_all_children();
}

MessagesList::~MessagesList()
{
}
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"

ExampleWindow::ExampleWindow()
: m_VPaned(Gtk::ORIENTATION_VERTICAL)
{
  set_title ("Paned Windows");
  set_border_width(10);
  set_default_size(450, 400);

  /* Add a vpaned widget to our toplevel window */
  add(m_VPaned);

  /* Now add the contents of the two halves of the window */
  m_VPaned.add1(m_MessagesList);
  m_VPaned.add2(m_MessageText);

  show_all_children();
}

ExampleWindow::~ExampleWindow()
{
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

8.1.3. ScrolledWindow

`ScrolledWindow` widgets create a scrollable area. You can insert any type of widget into a `ScrolledWindow` window, and it will be accessible regardless of its size by using the scrollbars. Note that `ScrolledWindow` is not a `Gtk::Window` despite the slightly misleading

name.

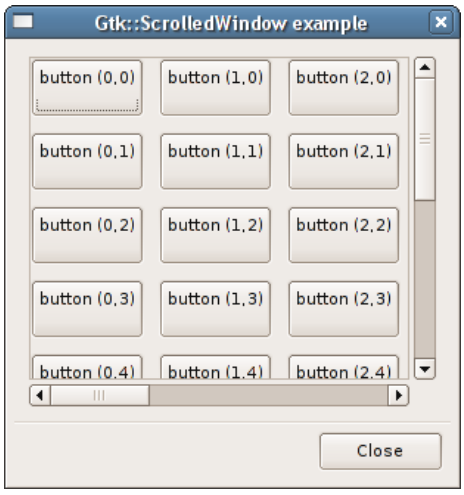
Scrolled windows have *scrollbar policies* which determine whether the `Scrollbar`s will be displayed. The policies can be set with the `set_policy()` method. The policy may be one of `Gtk::POLICY_AUTOMATIC` or `Gtk::POLICY_ALWAYS`. `Gtk::POLICY_AUTOMATIC` will cause the scrolled window to display the scrollbar only if the contained widget is larger than the visible area. `Gtk::POLICY_ALWAYS` will cause the scrollbar to be displayed always.

Reference

8.1.3.1. Example

Here is a simple example that packs 100 toggle buttons into a ScrolledWindow. Try resizing the window to see the scrollbars react.

Figure 8-3 **ScrolledWindow**



Source Code

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <gtkmm.h>

class ExampleWindow : public Gtk::Dialog
{
public:
  ExampleWindow();
  virtual ~ExampleWindow();

protected:
  //Signal handlers:
  void on_button_close();

  //Child widgets:
  Gtk::ScrolledWindow m_ScrolledWindow;
  Gtk::Table m_Table;
  Gtk::Button m_Button_Close;
};

#endif //GTKMM_EXAMPLEWINDOW_H
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <cstdio>

ExampleWindow::ExampleWindow()
```

```
: m_Table(10, 10),
  m_Button_Close("Close")
{
  set_title("Gtk::ScrolledWindow example");
  set_border_width(0);
  set_size_request(300, 300);

  m_ScrolledWindow.set_border_width(10);

  /* the policy is one of Gtk::POLICY AUTOMATIC, or Gtk::POLICY_ALWAYS.
   * Gtk::POLICY_AUTOMATIC will automatically decide whether you need
   * scrollbars, whereas Gtk::POLICY_ALWAYS will always leave the scrollbars
   * there.  The first one is the horizontal scrollbar, the second,
   * the vertical. */
  m_ScrolledWindow.set_policy(Gtk::POLICY_AUTOMATIC, Gtk::POLICY_ALWAYS);

  get_content_area()->pack_start(m_ScrolledWindow);

  /* set the spacing to 10 on x and 10 on y */
  m_Table.set_row_spacings(10);
  m_Table.set_col_spacings(10);

  /* pack the table into the scrolled window */
  m_ScrolledWindow.add(m_Table);

  /* this simply creates a grid of toggle buttons on the table
   * to demonstrate the scrolled window. */
  for(int i = 0; i < 10; i++)
  {
     for(int j = 0; j < 10; j++)
     {
        char buffer[32];
        sprintf(buffer, "button (%d,%d)\n", i, j);
        Gtk::Button* pButton = Gtk::manage(new Gtk::ToggleButton(buffer));
        m_Table.attach(*pButton, i, i + 1, j, j + 1);
     }
  }

  /* Add a "close" button to the bottom of the dialog */
  m_Button_Close.signal_clicked().connect( sigc::mem_fun(*this,
              &ExampleWindow::on_button_close));

  /* this makes it so the button is the default. */
  m_Button_Close.set_can_default();

  Gtk::Box* pBox = get_action_area();
  if(pBox)
    pBox->pack_start(m_Button_Close);

  /* This grabs this button to be the default button. Simply hitting
   * the "Enter" key will cause this button to activate. */
  m_Button_Close.grab_default();

  show_all_children();
}

ExampleWindow::~ExampleWindow()
{
}

void ExampleWindow::on_button_close()
{
  hide();
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

8.1.4. AspectFrame

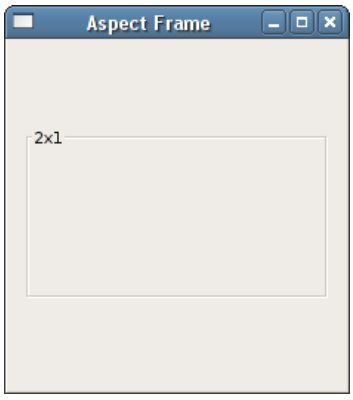
The `AspectFrame` widget looks like a `Frame` widget, but it also enforces the *aspect ratio* (the ratio of the width to the height) of the child widget, adding extra space if necessary. For instance, this would allow you to display a photograph without allowing the user to distort it horizontally or vertically while resizing.

Reference

8.1.4.1. Example

The following program uses a `Gtk::AspectFrame` to present a drawing area whose aspect ratio will always be 2:1, no matter how the user resizes the top-level window.

Figure 8-4 **AspectFrame**



Source Code

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <gtkmm.h>

class ExampleWindow : public Gtk::Window
{
public:
  ExampleWindow();
  virtual ~ExampleWindow();

protected:

  //Child widgets:
  Gtk::AspectFrame m_AspectFrame;
  Gtk::DrawingArea m_DrawingArea;
};

#endif //GTKMM_EXAMPLEWINDOW_H
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"

ExampleWindow::ExampleWindow()
: m_AspectFrame("2x1", /* label */
    Gtk::ALIGN_CENTER, /* center x */
    Gtk::ALIGN_CENTER, /* center y */
    2.0, /* xsize/ysize = 2 */
    false /* ignore child's aspect */)
{
  set_title("Aspect Frame");
  set_border_width(10);

  // Add a child widget to the aspect frame */
  // Ask for a 200x200 window, but the AspectFrame will give us a 200x100
  // window since we are forcing a 2x1 aspect ratio */
```

```
  m_DrawingArea.set_size_request(200, 200);
  m_AspectFrame.add(m_DrawingArea);

  // Add the aspect frame to our toplevel window:
  add(m_AspectFrame);

  show_all_children();
}

ExampleWindow::~ExampleWindow()
{
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

8.1.5. Alignment

The `Alignment` widget allows you to place a widget at a position and size relative to the size of the `Alignment` widget itself. For instance, it might be used to center a widget.

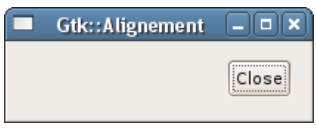
You need to specify the `Alignment`'s characteristics to the constructor, or to the `set()` method. In particular, you won't notice much effect unless you specify a number other than 1.0 for the `xscale` and `yscale` parameters, because 1.0 simply means that the child widget will expand to fill all available space.

Reference

8.1.5.1. Example

This example right-aligns a button in a window by using an `Alignment` widget.

Figure 8-5 **Alignment**



Source Code

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <gtkmm.h>

class ExampleWindow : public Gtk::Window
{
public:
  ExampleWindow();
  virtual ~ExampleWindow();

protected:
  //Signal handlers:
  void on_button_clicked();
```

```
  //Child widgets:
  Gtk::Alignment m_Alignment;
  Gtk::Button m_Button;
};

#endif //GTKMM_EXAMPLEWINDOW_H
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"

ExampleWindow::ExampleWindow()
: m_Alignment(Gtk::ALIGN_END, Gtk::ALIGN_CENTER, 0.0, 0.0),
  m_Button("Close")
{
  set_title("Gtk::Alignement");
  set_border_width(10);
  set_default_size(200, 50);

  add(m_Alignment);

  m_Alignment.add(m_Button);

  m_Button.signal_clicked().connect( sigc::mem_fun(*this,
              &ExampleWindow::on_button_clicked) );

  show_all_children();
}

ExampleWindow::~ExampleWindow()
{
}

void ExampleWindow::on_button_clicked()
{
  hide();
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

See the ProgressBar section for another example that uses an `Alignment`.

8.2. Multiple-item widgets

Multiple-item widgets inherit from `Gtk::Container`; just as with `Gtk::Bin`, you use the `add()` and `remove()` methods to add and remove contained widgets. Unlike `Gtk::Bin::remove()`, however, the `remove()` method for `Gtk::Container` takes an argument, specifiying which widget to remove.

8.2.1. Packing

You've probably noticed that *gtkmm* windows seem "elastic" - they can usually be stretched in many different ways. This is due to the *widget packing* system.

Many GUI toolkits require you to precisely place widgets in a window, using absolute positioning, often using a visual editor. This leads to several problems:

- The widgets don't rearrange themselves when the window is resized. Some widgets are hidden when the window is made smaller, and lots of useless space appears when the window is made larger.
- It's impossible to predict the amount of space necessary for text after it has been translated to other languages, or displayed in a different font. On Unix it is also impossible to anticipate the effects of every theme and window manager.
- Changing the layout of a window "on the fly", to make some extra widgets appear, for instance, is complex. It requires tedious recalculation of every widget's position.

gtkmm uses the packing system to solve these problems. Rather than specifying the position and size of each widget in the window, you can arrange your widgets in rows, columns, and/or tables. *gtkmm* can size your window automatically, based on the sizes of the widgets it contains. And the sizes of the widgets are, in turn, determined by the amount of text they contain, or the minimum and maximum sizes that you specify, and/or how you have requested that the available space should be shared between sets of widgets. You can perfect your layout by specifying padding distance and centering values for each of your widgets. *gtkmm* then uses all this information to resize and reposition everything sensibly and smoothly when the user manipulates the window.

gtkmm arranges widgets hierarchically, using *containers*. A Container widget contains other widgets. Most *gtkmm* widgets are containers. Windows, Notebook tabs, and Buttons are all container widgets. There are two flavours of containers: single-child containers, which are all descendants of `Gtk::Bin`, and multiple-child containers, which are descendants of `Gtk::Container`. Most widgets in *gtkmm* are descendants of `Gtk::Bin`, including `Gtk::Window`.

Yes, that's correct: a Window can contain at most one widget. How, then, can we use a window for anything useful? By placing a multiple-child container in the window. The most useful container widgets are `Gtk:Grid`, `Gtk:VBox`, `Gtk::HBox`, and `Gtk::Table`.

- `Gtk::Grid` arranges its child widgets in rows and columns. Use `attach()`, `attach_next_to()` and `add()` to insert child widgets.
- `Gtk::VBox` and `Gtk::HBox` arrange their child widgets vertically and horizontally, respectively. Use `pack_start()` and `pack_end()` to insert child widgets.
- `Gtk::Table` arranges its widgets in a grid. Use `attach()` to insert widgets.

There are several other containers, which we will also discuss.

If you've never used a packing toolkit before, it can take some getting used to. You'll probably find, however, that you don't need to rely on visual form editors quite as much as you might with other toolkits.

8.2.2. An improved Hello World

Let's take a look at a slightly improved `helloworld`, showing what we've learnt.

Figure 8-6 **Hello World 2**

Source Code

File: `helloworld.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLE_HELLOWORLD_H
```

```
#define GTKMM_EXAMPLE_HELLOWORLD_H

#include <gtkmm/box.h>
#include <gtkmm/button.h>
#include <gtkmm/window.h>

class HelloWorld : public Gtk::Window
{
public:
  HelloWorld();
  virtual ~HelloWorld();

protected:

  // Signal handlers:
  // Our new improved on_button_clicked(). (see below)
  void on_button_clicked(Glib::ustring data);

  // Child widgets:
  Gtk::Box m_box1;
  Gtk::Button m_button1, m_button2;
};

#endif // GTKMM_EXAMPLE_HELLOWORLD_H
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "helloworld.h"
#include <gtkmm/main.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;
}
```

File: `helloworld.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "helloworld.h"
#include <iostream>

HelloWorld::HelloWorld()
: m_button1("Button 1"),
  m_button2("Button 2")
{
  // This just sets the title of our new window.
  set_title("Hello Buttons!");

  // sets the border width of the window.
  set_border_width(10);

  // put the box into the main window.
  add(m_box1);

  // Now when the button is clicked, we call the "on_button_clicked" function
  // with a pointer to "button 1" as it's argument
  m_button1.signal_clicked().connect(sigc::bind<Glib::ustring>(
              sigc::mem_fun(*this, &HelloWorld::on_button_clicked), "button 1"));

  // instead of gtk_container_add, we pack this button into the invisible
  // box, which has been packed into the window.
  // note that the pack_start default arguments are Gtk::EXPAND | Gtk::FILL, 0
  m_box1.pack_start(m_button1);

  // always remember this step, this tells GTK that our preparation
  // for this button is complete, and it can be displayed now.
  m_button1.show();

  // call the same signal handler with a different argument,
  // passing a pointer to "button 2" instead.
  m_button2.signal_clicked().connect(sigc::bind<-1, Glib::ustring>(
              sigc::mem_fun(*this, &HelloWorld::on_button_clicked), "button 2"));

  m_box1.pack_start(m_button2);

  // Show the widgets.
  // They will not really be shown until this Window is shown.
  m_button2.show();
```

```
  m_box1.show();
}

HelloWorld::~HelloWorld()
{
}

// Our new improved signal handler.  The data passed to this method is
// printed to stdout.
void HelloWorld::on_button_clicked(Glib::ustring data)
{
  std::cout << "Hello World - " << data << " was pressed" << std::endl;
}
```

After building and running this program, try resizing the window to see the behaviour. Also, try playing with the options to `pack_start()` while reading the Boxes section.

8.2.3. Boxes

Most packing uses boxes as in the above example. These are invisible containers into which we can pack our widgets. When packing widgets into a horizontal box, the objects are inserted horizontally from left to right or right to left depending on whether `pack_start()` or `pack_end()` is used. In a vertical box, widgets are packed from top to bottom or vice versa. You may use any combination of boxes inside or beside other boxes to create the desired effect.

8.2.3.1. Adding widgets

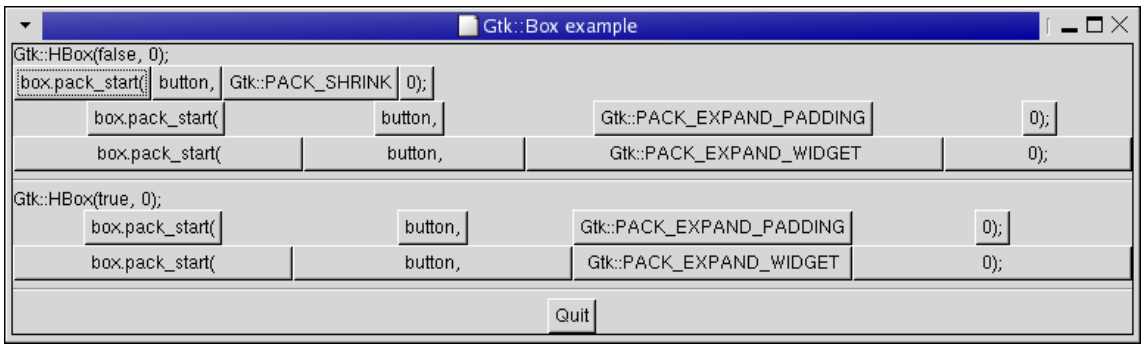
8.2.3.1.1. Per-child packing options

The `pack_start()` and `pack_end()` methods place widgets inside these containers. The `pack_start()` method will start at the top and work its way down in a `VBox`, or pack left to right in an `HBox`. `pack_end()` will do the opposite, packing from bottom to top in a `VBox`, or right to left in an `HBox`. Using these methods allows us to right justify or left justify our widgets. We will use `pack_start()` in most of our examples.

There are several options governing how widgets are to be packed, and this can be confusing at first. If you have difficulties then it is sometimes a good idea to play with the *glade* GUI designer to see what is possible. You might even decide to use the *Gtk::Builder* API to load your GUI at runtime.

There are basically five different styles, as shown in this picture:

Figure 8-7 **Box Packing 1**



Each line contains one horizontal box (`HBox`) with several buttons. Each of the buttons on a line is packed into the `HBox` with the same arguments to the `pack_start()` method).

This is the declaration of the `pack_start()` method:

```
void pack_start(Gtk::Widget& child,
                PackOptions options = PACK_EXPAND_WIDGET,
                guint padding = 0);
```

The first argument is the widget you're packing. In our example these are all `Button`s.

The `options` argument can take one of these three options:

- `PACK_SHRINK`: Space is contracted to the child widget size. The widget will take up just-enough space and never expand.
- `PACK_EXPAND_PADDING`: Extra space is filled with padding. The widgets will be spaced out evenly, but their sizes won't change - there will be empty space between the widgets instead.
- `PACK_EXPAND_WIDGET`: Extra space is taken up by increasing the child widget size, without changing the amount of space between widgets.

The `padding` argument specifies the width of an extra border area to leave around the packed widget.

Reference

8.2.3.1.2. Per-container packing options

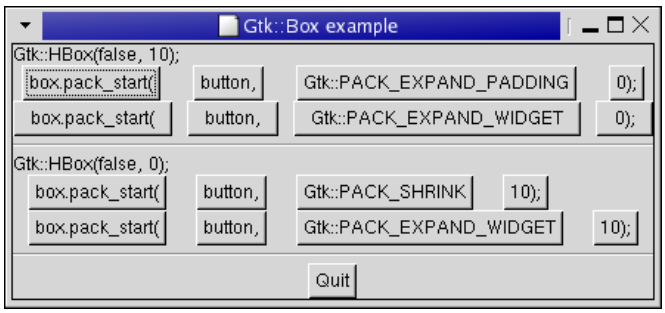
Here's the constructor for the box widgets:

```
Gtk::Box(bool homogeneous = false, int spacing = 0);
```

Passing `true` for `homogeneous` will cause all of the contained widgets to be the same size. `spacing` is a (minimum) number of pixels to leave between each widget.

What's the difference between spacing (set when the box is created) and padding (set when elements are packed)? Spacing is added between objects, and padding is added on either side of a widget. The following figure should make it clearer:

Figure 8-8 **Box Packing 2**



8.2.3.2. Example

Here is the source code for the example that produced the screenshots above. When you run this example, provide a number between 1 and 3 as a command-line option, to see different packing options in use.

Source Code

File: `packbox.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLE_PACKBOX_H
#define GTKMM_EXAMPLE_PACKBOX_H

#include <gtkmm.h>

class PackBox : public Gtk::Box
```

```
{
public:
  PackBox(bool homogeneous, int spacing, Gtk::PackOptions, int padding = 0);
  virtual ~PackBox();

protected:
  Gtk::Button m_button1, m_button2, m_button3;
  Gtk::Button* m_pbutton4;

  char padstr[80];
};

#endif //GTKMM_EXAMPLE_PACKBOX_H
```

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <gtkmm.h>
#include "packbox.h"

class ExampleWindow : public Gtk::Window
{
public:
  ExampleWindow(int which);
  virtual ~ExampleWindow();

protected:
  //Signal handlers:
  void on_button_quit_clicked();

  //Child widgets:
  Gtk::Button m_button;
  Gtk::Box m_box1;
  Gtk::Box m_boxQuit;
  Gtk::Button m_buttonQuit;

  Gtk::Label m_Label1, m_Label2;

  Gtk::Separator m_seperator1, m_seperator2, m_seperator3, m_seperator4, m_seperator5;
};

#endif //GTKMM_EXAMPLEWINDOW_H
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include <iostream>
#include "examplewindow.h"

ExampleWindow::ExampleWindow(int which)
: m_box1(Gtk::ORIENTATION_VERTICAL),
  m_buttonQuit("Quit")
{
  set_title("Gtk::Box example");

  PackBox *pPackBox1, *pPackBox2, *pPackBox3, *pPackBox4, *pPackBox5;

  switch(which)
  {
    case 1:
    {
      m_Label1.set_text("Gtk::Box(Gtk::ORIENTATION_HORIZONTAL, 0);");

      // Align the label to the left side.  We'll discuss this function and
      // others in the section on Widget Attributes.
      m_Label1.set_alignment(Gtk::ALIGN_START, Gtk::ALIGN_START);

      // Pack the label into the vertical box (vbox box1).  Remember that
      // widgets added to a vbox will be packed one on top of the other in
      // order.
      m_box1.pack_start(m_Label1, Gtk::PACK_SHRINK);

      // Create a PackBox - homogeneous = false, spacing = 0,
      // options = Gtk::PACK_SHRINK, padding = 0
      pPackBox1 = Gtk::manage(new PackBox(false, 0, Gtk::PACK_SHRINK));
      m_box1.pack_start(*pPackBox1, Gtk::PACK_SHRINK);

      // Create a PackBox - homogeneous = false, spacing = 0,
      // options = Gtk::PACK_EXPAND_PADDING, padding = 0
      pPackBox2 = Gtk::manage(new PackBox(false, 0, Gtk::PACK_EXPAND_PADDING));
      m_box1.pack_start(*pPackBox2, Gtk::PACK_SHRINK);
```

```
  // Create a PackBox - homogeneous = false, spacing = 0,
  // options = Gtk::PACK_EXPAND_WIDGET, padding = 0
  pPackBox3 = Gtk::manage(new PackBox(false, 0, Gtk::PACK_EXPAND_WIDGET));
  m_box1.pack_start(*pPackBox3, Gtk::PACK_SHRINK);

  // pack the separator into the vbox.  Remember each of these
  // widgets are being packed into a vbox, so they'll be stacked
  // vertically.
  m_box1.pack_start(m_seperator1, Gtk::PACK_SHRINK, 5);

  // create another new label, and show it.
  m_Label2.set_text("Gtk::Box(Gtk::ORIENTATION_HORIZONTAL, 0);");
  m_Label2.set_alignment(Gtk::ALIGN_START, Gtk::ALIGN_START);
  m_box1.pack_start(m_Label2, Gtk::PACK_SHRINK);

  // Args are: homogeneous, spacing, options, padding
  pPackBox4 = Gtk::manage(new PackBox(true, 0, Gtk::PACK_EXPAND_PADDING));
  m_box1.pack_start(*pPackBox4, Gtk::PACK_SHRINK);

  // Args are: homogeneous, spacing, options, padding
  pPackBox5 = Gtk::manage(new PackBox(true, 0, Gtk::PACK_EXPAND_WIDGET));
  m_box1.pack_start(*pPackBox5, Gtk::PACK_SHRINK);

  m_box1.pack_start(m_seperator2, Gtk::PACK_SHRINK, 5);

  break;
}

case 2:
{

  m_Label1.set_text("Gtk::Box(Gtk::ORIENTATION_HORIZONTAL, 10);");
  m_Label1.set_alignment(Gtk::ALIGN_START, Gtk::ALIGN_START);
  m_box1.pack_start(m_Label1, Gtk::PACK_SHRINK);

  pPackBox1 = Gtk::manage(new PackBox(false, 10, Gtk::PACK_EXPAND_PADDING));
  m_box1.pack_start(*pPackBox1, Gtk::PACK_SHRINK);

  pPackBox2 = Gtk::manage(new PackBox(false, 10, Gtk::PACK_EXPAND_WIDGET));
  m_box1.pack_start(*pPackBox2, Gtk::PACK_SHRINK);

  m_box1.pack_start(m_seperator1, Gtk::PACK_SHRINK, 5);


  m_Label2.set_text("Gtk::Box(Gtk::ORIENTATION_HORIZONTAL, 0);");
  m_Label2.set_alignment(Gtk::ALIGN_START, Gtk::ALIGN_START);
  m_box1.pack_start(m_Label2, Gtk::PACK_SHRINK);

  pPackBox3 = Gtk::manage(new PackBox(false, 0, Gtk::PACK_SHRINK, 10));
  m_box1.pack_start(*pPackBox3, Gtk::PACK_SHRINK);

  pPackBox4 = Gtk::manage(new PackBox(false, 0, Gtk::PACK_EXPAND_WIDGET, 10));
  m_box1.pack_start(*pPackBox4, Gtk::PACK_SHRINK);

  m_box1.pack_start(m_seperator2, Gtk::PACK_SHRINK, 5);

  break;
}

case 3:
{
  // This demonstrates the ability to use Gtk::Box::pack_end() to
  // right justify widgets.  First, we create a new box as before.
  pPackBox1 = Gtk::manage(new PackBox(false, 0, Gtk::PACK_SHRINK));

  // create the label that will be put at the end.
  m_Label1.set_text("end");

  // pack it using pack_end(), so it is put on the right side
  // of the PackBox.
  pPackBox1->pack_end(m_Label1, Gtk::PACK_SHRINK);

  m_box1.pack_start(*pPackBox1, Gtk::PACK_SHRINK);

  // this explicitly sets the separator to 400 pixels wide by 5 pixels
  // high.  This is so the hbox we created will also be 400 pixels wide,
  // and the "end" label will be separated from the other labels in the
  // hbox.  Otherwise, all the widgets in the hbox would be packed as
  // close together as possible.
  m_seperator1.set_size_request(400, 5);

  // pack the separator into ourselves
  m_box1.pack_start(m_seperator1, Gtk::PACK_SHRINK, 5);

  break;
}

default:
```

```
    {
      std::cerr << "Unexpected command-line option." << std::endl;
      break;
    }
  }

  // Connect the signal to hide the window:
  m_buttonQuit.signal_clicked().connect( sigc::mem_fun(*this,
              &ExampleWindow::on_button_quit_clicked) );

  // pack the button into the quitbox.
  // The last 2 arguments to Box::pack_start are: options, padding.
  m_boxQuit.pack_start(m_buttonQuit, Gtk::PACK_EXPAND_PADDING);
  m_box1.pack_start(m_boxQuit, Gtk::PACK_SHRINK);

  // pack the vbox (box1) which now contains all our widgets, into the
  // main window.
  add(m_box1);

  show_all_children();
}

ExampleWindow::~ExampleWindow()
{
}

void ExampleWindow::on_button_quit_clicked()
{
  hide();
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>
#include <iostream>
#include <cstdlib>

using std::atoi;

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

  if(argc != 2)
  {
    std::cerr << "usage: packbox num, where num is 1, 2, or 3." << std::endl;
    return 1;
  }

  ExampleWindow window( atoi(argv[1]) );
  Gtk::Main::run(window); //Shows the window and returns when it is closed.

  return 0;
}
```

File: `packbox.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "packbox.h"
#include <cstdio> //For sprintf().

PackBox::PackBox(bool homogeneous, int spacing, Gtk::PackOptions options,
        int padding)
: Gtk::Box(Gtk::ORIENTATION_HORIZONTAL, spacing),
  m_button1("box.pack_start("),
  m_button2("button,"),
  m_button3((options == Gtk::PACK_SHRINK) ? "Gtk::PACK_SHRINK" :
            ((options == Gtk::PACK_EXPAND_PADDING) ?
             "Gtk::PACK_EXPAND_PADDING" : "Gtk::PACK_EXPAND_WIDGET"))
{
  set_homogeneous(homogeneous);

  pack_start(m_button1, options, padding);
  pack_start(m_button2, options, padding);
  pack_start(m_button3, options, padding);

  sprintf(padstr, "%d);", padding);

  m_pbutton4 = new Gtk::Button(padstr);
  pack_start(*m_pbutton4, options, padding);
}
```

```
PackBox::~PackBox()
{
  delete m_pbutton4;
}
```

8.2.4. ButtonBoxes

Button boxes are a convenient way to quickly arrange a group of buttons. They come in both horizontal (`Gtk::HButtonBox`) and vertical (`Gtk::VButtonBox`) flavours. They are exactly alike, except in name and orientation.

`ButtonBox`es help to make applications appear consistent because they use standard settings, such as inter-button spacing and packing.

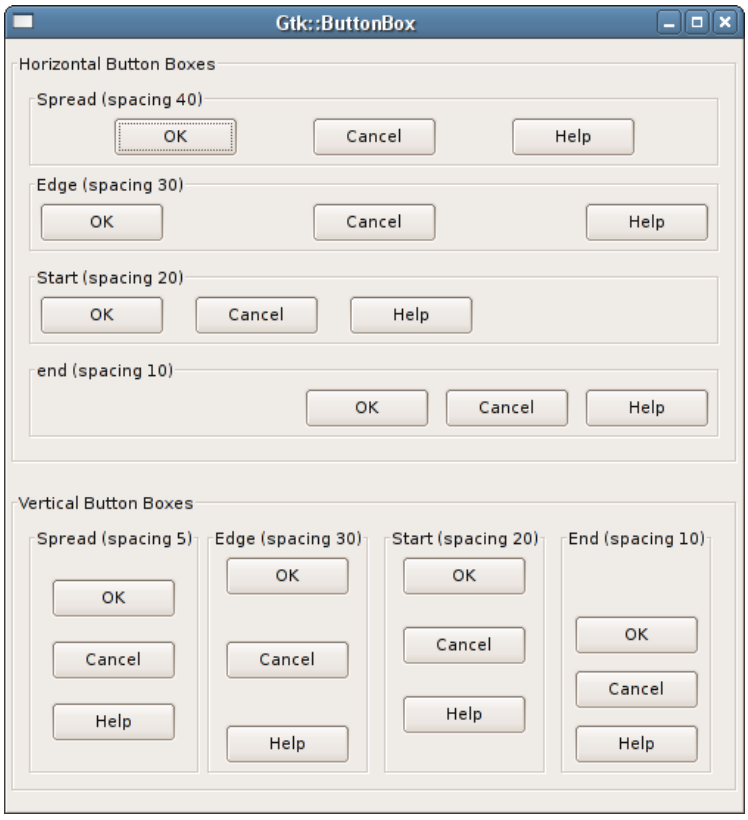
Buttons are added to a `ButtonBox` with the `add()` method.

Button boxes support several layout styles. The style can be retrieved and changed using `get_layout()` and `set_layout()`.

Reference

8.2.4.1. Example

Figure 8-9 **ButtonBox**



Source Code

File: `examplebuttonbox.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLE_BUTTONBOX_H
#define GTKMM_EXAMPLE_BUTTONBOX_H

#include <gtkmm.h>

class ExampleButtonBox : public Gtk::Frame
```

```
{
public:
  ExampleButtonBox(bool horizontal,
       const Glib::ustring& title,
       gint spacing,
       Gtk::ButtonBoxStyle layout);

protected:
  Gtk::Button m_Button_OK, m_Button_Cancel, m_Button_Help;
};

#endif //GTKMM_EXAMPLE_BUTTONBOX_H
```

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <gtkmm.h>

class ExampleWindow : public Gtk::Window
{
public:
  ExampleWindow();
  virtual ~ExampleWindow();

protected:
  //Signal handlers:
  void on_button_clicked();

  //Child widgets:
  Gtk::Box m_VBox_Main, m_VBox;
  Gtk::Box m_HBox;
  Gtk::Frame m_Frame_Horizontal, m_Frame_Vertical;
};

#endif //GTKMM_EXAMPLEWINDOW_H
```

File: `examplebuttonbox.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplebuttonbox.h"

ExampleButtonBox::ExampleButtonBox(bool horizontal,
       const Glib::ustring& title,
       gint spacing,
       Gtk::ButtonBoxStyle layout)
: Gtk::Frame(title),
  m_Button_OK("OK"),
  m_Button_Cancel("Cancel"),
  m_Button_Help("Help")
{
  Gtk::ButtonBox* bbox = 0;

  if(horizontal)
    bbox = Gtk::manage( new Gtk::ButtonBox(Gtk::ORIENTATION_HORIZONTAL) );
  else
    bbox = Gtk::manage( new Gtk::ButtonBox(Gtk::ORIENTATION_VERTICAL) );

  bbox->set_border_width(5);

  add(*bbox);

  /* Set the appearance of the Button Box */
  bbox->set_layout(layout);
  bbox->set_spacing(spacing);

  bbox->add(m_Button_OK);
  bbox->add(m_Button_Cancel);
  bbox->add(m_Button_Help);
}
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include "examplebuttonbox.h"

ExampleWindow::ExampleWindow()
: m_VBox_Main(Gtk::ORIENTATION_VERTICAL),
  m_VBox(Gtk::ORIENTATION_VERTICAL),
```

```
  m_Frame_Horizontal("Horizontal Button Boxes"),
  m_Frame_Vertical("Vertical Button Boxes")
{
  set_title("Gtk::ButtonBox");
  add(m_VBox_Main);

  m_VBox_Main.pack_start(m_Frame_Horizontal, Gtk::PACK_EXPAND_WIDGET, 10);

  //The horizontal ButtonBoxes:
  m_VBox.set_border_width(10);
  m_Frame_Horizontal.add(m_VBox);

  m_VBox.pack_start(*Gtk::manage(
              new ExampleButtonBox(true, "Spread (spacing 40)", 40,
                  Gtk::BUTTONBOX_SPREAD)),
          Gtk::PACK_EXPAND_WIDGET, 0);

  m_VBox.pack_start(*Gtk::manage(
              new ExampleButtonBox(true, "Edge (spacing 30)", 30,
                  Gtk::BUTTONBOX_EDGE)),
          Gtk::PACK_EXPAND_WIDGET, 5);

  m_VBox.pack_start(*Gtk::manage(
              new ExampleButtonBox(true, "Start (spacing 20)", 20,
                  Gtk::BUTTONBOX_START)),
          Gtk::PACK_EXPAND_WIDGET, 5);

  m_VBox.pack_start(*Gtk::manage(
              new ExampleButtonBox(true, "end (spacing 10)", 10,
                  Gtk::BUTTONBOX_END)),
          Gtk::PACK_EXPAND_WIDGET, 5);


  //The vertical ButtonBoxes:
  m_VBox_Main.pack_start(m_Frame_Vertical, Gtk::PACK_EXPAND_WIDGET, 10);

  m_HBox.set_border_width(10);
  m_Frame_Vertical.add(m_HBox);

  m_HBox.pack_start(*Gtk::manage(
              new ExampleButtonBox(false, "Spread (spacing 5)", 5,
                  Gtk::BUTTONBOX_SPREAD)),
          Gtk::PACK_EXPAND_WIDGET, 0);

  m_HBox.pack_start(*Gtk::manage(
              new ExampleButtonBox(false, "Edge (spacing 30)", 30,
                  Gtk::BUTTONBOX_EDGE)),
          Gtk::PACK_EXPAND_WIDGET, 5);

  m_HBox.pack_start(*Gtk::manage(
              new ExampleButtonBox(false, "Start (spacing 20)", 20,
                  Gtk::BUTTONBOX_START)),
          Gtk::PACK_EXPAND_WIDGET, 5);

  m_HBox.pack_start(*Gtk::manage(new ExampleButtonBox(false, "End (spacing 10)",
                  10, Gtk::BUTTONBOX_END)),
          Gtk::PACK_EXPAND_WIDGET, 5);

  show_all_children();
}

ExampleWindow::~ExampleWindow()
{
}

void ExampleWindow::on_button_clicked()
{
  hide();
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

8.2.5. Grid

A `Grid` dynamically lays out child widgets in rows and columns. The dimensions of the grid do not need to be specified in the constructor.

Child widgets can span multiple rows or columns, using `attach()`, or added next to an existing widget inside the grid with `attach_next_to()`. Individual rows and columns of the grid can be set to have uniform height or width with `set_row_homogeneous()` and `set_column_homogeneous()`.

You can set the *margin* and *expand* properties of the child `Widget`s to control their spacing and their behaviour when the Grid is resized.

Reference

8.2.5.1. Example

This example creates a window with three buttons in a grid. The first two buttons are in the upper row, from left to right. A third button is attached underneath the first button, in a new lower row, spanning two columns.

Figure 8-10 **Grid**

Source Code

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <gtkmm.h>

class ExampleWindow : public Gtk::Window
{
public:
  ExampleWindow();
  virtual ~ExampleWindow();

private:
  // Signal handlers:
  void on_button_quit();
  void on_button_numbered(const Glib::ustring& data);

  // Child widgets:
  Gtk::Grid m_grid;
  Gtk::Button m_button_1, m_button_2, m_button_quit;
};

#endif /* GTKMM_EXAMPLEWINDOW_H */
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include <iostream>
#include "examplewindow.h"

ExampleWindow::ExampleWindow()
: m_button_1("button 1"),
  m_button_2("button 2"),
  m_button_quit("Quit")
{
  set_title("Gtk::Grid");
  set_border_width(12);

  add(m_grid);
```

```
  m_grid.add(m_button_1);
  m_grid.add(m_button_2);
  m_grid.attach_next_to(m_button_quit, m_button_1, Gtk::POS_BOTTOM, 2, 1);

  m_button_1.signal_clicked().connect(
    sigc::bind<Glib::ustring>( sigc::mem_fun(*this,
      &ExampleWindow::on_button_numbered), "button 1") );
  m_button_2.signal_clicked().connect(
    sigc::bind<Glib::ustring>( sigc::mem_fun(*this,
      &ExampleWindow::on_button_numbered), "button 2") );

  m_button_quit.signal_clicked().connect(sigc::mem_fun(*this,
    &ExampleWindow::on_button_quit) );

  show_all_children();
}

ExampleWindow::~ExampleWindow()
{
}

void ExampleWindow::on_button_quit()
{
  hide();
}

void
ExampleWindow::on_button_numbered(const Glib::ustring& data)
{
  std::cout << data << " was pressed" << std::endl;
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;
  // Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

8.2.6. Table

Tables allows us to place widgets in a grid, similar to `Gtk::Grid`.

8.2.6.1. Constructor

The grid's dimensions need to be specified in the constructor:

```
Gtk::Table(int rows = 1, int columns = 1, bool homogeneous = false);
```

The first argument is the number of rows to make in the table, while the second, obviously, is the number of columns. If `homogeneous` is `true`, the table cells will all be the same size (the size of the largest widget in the table).

The rows and columns are indexed starting at 0. If you specify `rows` = 2 and `columns` = 2, the layout would look something like this:

```
 0         1         2
0+---------+---------+
 |         |         |
```

```
1+----------+----------+
 |          |          |
2+----------+----------+
```

Note that the coordinate system starts in the upper left hand corner.

8.2.6.2. Adding widgets

To place a widget into a box, use the following method:

```
void Gtk::Table::attach(Gtk::Widget& child,
                        guint left_attach, guint right_attach,
                        guint top_attach, guint bottom_attach,
                        guint xoptions = Gtk::FILL | Gtk::EXPAND,
                        guint yoptions = Gtk::FILL | Gtk::EXPAND,
                        guint xpadding = 0, guint ypadding = 0);
```

The first argument is the widget you wish to place in the table.

The `left_attach` and `right_attach` arguments specify where to place the widget, and how many boxes to use. For example, if you want a button in the lower-right cell of a 2 x 2 table, and want it to occupy that cell *only*, then `left_attach` would be 1, `right_attach` 2, `top_attach` 1, and `bottom_attach` 2. If, on the other hand, you wanted a widget to take up the entire top row of our 2 x 2 table, you'd set `left_attach` = 0, `right_attach` = 2, `top_attach` = 0, and `bottom_attach` = 1.

`xoptions` and `yoptions` are used to specify packing options and may be bitwise ORed together to allow multiple options. These options are:

`Gtk::FILL`

If the table box is larger than the widget, and `Gtk::FILL` is specified, the widget will expand to use all the room available.

`Gtk::SHRINK`

If the table widget is allocated less space than it requested (because the user resized the window), then the widgets will normally just disappear off the bottom of the window. If `Gtk::SHRINK` is specified, the widgets will shrink with the table.

`Gtk::EXPAND`

This will cause the table to expand to use up anyremaining space in the window.

The padding arguments work just as they do for `pack_start()`.

8.2.6.3. Other methods

`set_row_spacing()` and `set_col_spacing()` set the spacing between the rows at the specified row or column. Note that for columns, the space goes to the right of the column, and for rows, the space goes below the row.

You can also set a consistent spacing for all rows and/or columns with `set_row_spacings()` and `set_col_spacings()`. Note that with these calls, the last row and last column do not get any spacing.

Reference

8.2.6.4. Example

In the following example, we make a window with three buttons in a 2 x 2 table. The first two buttons will be placed in the upper row. A third button is placed in the lower row, spanning both columns.

Figure 8-11 **Table**

Source Code

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <gtkmm.h>

class ExampleWindow : public Gtk::Window
{
public:
  ExampleWindow();
  virtual ~ExampleWindow();

protected:
  //Signal handlers:
  void on_button_quit();
  void on_button_numbered(Glib::ustring data);

  //Child widgets:
  Gtk::Table m_Table;
  Gtk::Button m_Button_1, m_Button_2, m_Button_Quit;

};

#endif //GTKMM_EXAMPLEWINDOW_H
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include <iostream>
#include "examplewindow.h"

ExampleWindow::ExampleWindow()
: m_Table(2, 2, true),
  m_Button_1("button 1"),
  m_Button_2("button 2"),
  m_Button_Quit("Quit")
{
  set_title("Gtk::Table");
  set_border_width(20);

  add(m_Table);

  m_Table.attach(m_Button_1, 0, 1, 0, 1);
  m_Table.attach(m_Button_2, 1, 2, 0, 1);
  m_Table.attach(m_Button_Quit, 0, 2, 1, 2);

  m_Button_1.signal_clicked().connect(
          sigc::bind<Glib::ustring>( sigc::mem_fun(*this,
                  &ExampleWindow::on_button_numbered), "button 1") );
  m_Button_2.signal_clicked().connect(
          sigc::bind<Glib::ustring>( sigc::mem_fun(*this,
                  &ExampleWindow::on_button_numbered), "button 2") );

  m_Button_Quit.signal_clicked().connect(sigc::mem_fun(*this,
              &ExampleWindow::on_button_quit) );

  show_all_children();
}

ExampleWindow::~ExampleWindow()
{
}

void ExampleWindow::on_button_quit()
{
  hide();
}

void
ExampleWindow::on_button_numbered(Glib::ustring data)
```

```
{
  std::cout << "Hello again - " << data << " was pressed" << std::endl;
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

8.2.7. Notebook

A `Notebook` has a set of stacked `pages`, each of which contains widgets. Labelled `tabs` allow the user to select the pages. `Notebook`s allow several sets of widgets to be placed in a small space, by only showing one page at a time. For instance, they are often used in preferences dialogs.

Use the `append_page()`, `prepend_page()` and `insert_page()` methods to add tabbed pages to the `Notebook`, supplying the child widget and the name for the tab.

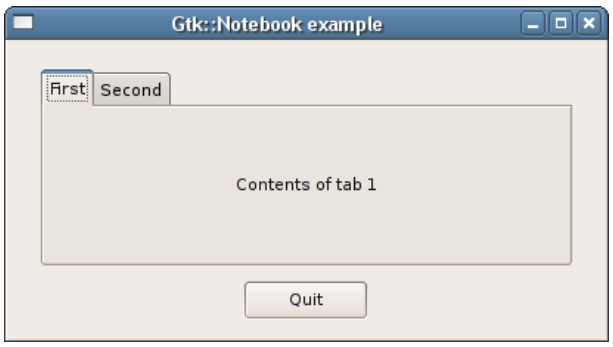
To discover the currently visible page, use the `get_current_page()` method. This returns the page number, and then calling `get_nth_page()` with that number will give you a pointer to the actual child widget.

To programmatically change the selected page, use the `set_current_page()` method.

Reference

8.2.7.1. Example

Figure 8-12 **Notebook**



Source Code

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <gtkmm.h>

class ExampleWindow : public Gtk::Window
{
public:
```

```
  ExampleWindow();
  virtual ~ExampleWindow();

protected:
  //Signal handlers:
  void on_button_quit();
  void on_notebook_switch_page(Gtk::Widget* page, guint page_num);

  //Child widgets:
  Gtk::Box m_VBox;
  Gtk::Notebook m_Notebook;
  Gtk::Label m_Label1, m_Label2;

  Gtk::ButtonBox m_ButtonBox;
  Gtk::Button m_Button_Quit;
};

#endif //GTKMM_EXAMPLEWINDOW_H
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include <iostream>
#include "examplewindow.h"

ExampleWindow::ExampleWindow()
: m_VBox(Gtk::ORIENTATION_VERTICAL),
  m_Label1("Contents of tab 1"),
  m_Label2("Contents of tab 2"),
  m_Button_Quit("Quit")
{
  set_title("Gtk::Notebook example");
  set_border_width(10);
  set_default_size(400, 200);


  add(m_VBox);

  //Add the Notebook, with the button underneath:
  m_Notebook.set_border_width(10);
  m_VBox.pack_start(m_Notebook);
  m_VBox.pack_start(m_ButtonBox, Gtk::PACK_SHRINK);

  m_ButtonBox.pack_start(m_Button_Quit, Gtk::PACK_SHRINK);
  m_Button_Quit.signal_clicked().connect(sigc::mem_fun(*this,
              &ExampleWindow::on_button_quit) );

  //Add the Notebook pages:
  m_Notebook.append_page(m_Label1, "First");
  m_Notebook.append_page(m_Label2, "Second");

  m_Notebook.signal_switch_page().connect(sigc::mem_fun(*this,
              &ExampleWindow::on_notebook_switch_page) );

  show_all_children();
}

ExampleWindow::~ExampleWindow()
{
}

void ExampleWindow::on_button_quit()
{
  hide();
}

void ExampleWindow::on_notebook_switch_page(Gtk::Widget* /* page */, guint page_num)
{
  std::cout << "Switched to tab with index " << page_num << std::endl;

  //You can also use m_Notebook.get_current_page() to get this index.
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;
  //Shows the window and returns when it is closed.
```

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

  return 0;
}
```

8.2.8. Assistant

An `Assistant` splits a complex operation into steps. Each step is a page, containing a header, a child widget and an action area. The Assistant's action area has navigation buttons which update automatically depending on the type of the page, set with `set_page_type()`.

Use the `append_page()`, `prepend_page` and `insert_page()` methods to add pages to the `Assistant`, supplying the child widget for each page.

To determine the currently-visible page, use the `get_current_page()` method, and pass the result to `get_nth_page()`, which returns a pointer to the actual widget. To programmatically change the current page, use the `set_current_page()` method.

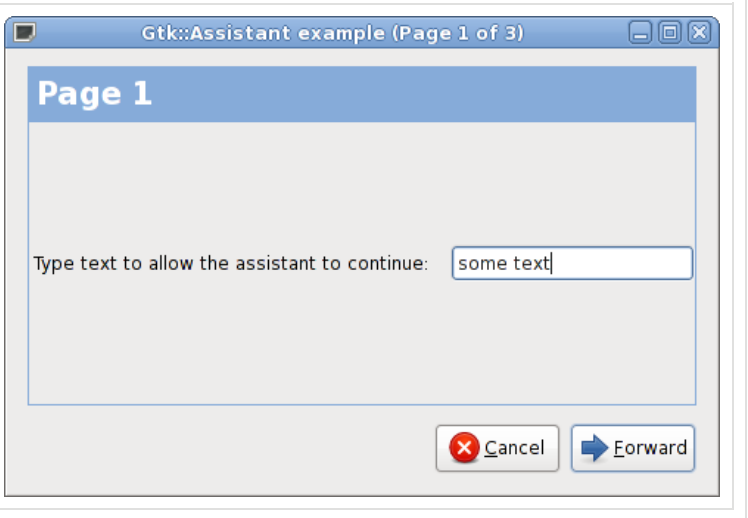
To set the title of a page, use the `set_page_title()` method. The header and side images of a page can be set with the `set_page_header_image()` and `set_page_side_image()` methods.

To add widgets to the action area, use the `add_action_widget()` method. They will be packed alongside the default buttons. Use the `remove_action_widget()` method to remove widgets.

Reference

8.2.8.1. Example

Figure 8-13 **Assistant**



Source Code

File: `exampleassistant.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEASSISTANT_H
#define GTKMM_EXAMPLEASSISTANT_H

#include <gtkmm.h>

class ExampleAssistant : public Gtk::Assistant
{
public:
  ExampleAssistant();
  virtual ~ExampleAssistant();

  void get_result(bool& check_state, Glib::ustring& entry_text);
```

```
private:
  // Signal handlers:
  void on_assistant_apply();
  void on_assistant_cancel();
  void on_assistant_close();
  void on_assistant_prepare(Gtk::Widget* widget);
  void on_entry_changed();

  // Member functions:
  void print_status();

  // Child widgets:
  Gtk::Box m_box;
  Gtk::Label m_label1, m_label2;
  Gtk::CheckButton m_check;
  Gtk::Entry m_entry;
};

#endif /* GTKMM_EXAMPLEASSISTANT_H */
```

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include "exampleassistant.h"
#include <gtkmm.h>

class ExampleWindow : public Gtk::Window
{
public:
  ExampleWindow();
  virtual ~ExampleWindow();

private:
  // Signal handlers:
  void on_button_clicked();
  void on_assistant_apply();

  // Child widgets:
  Gtk::Table m_table;
  Gtk::Button m_button;
  Gtk::Label m_label1, m_label2;
  Gtk::CheckButton m_check;
  Gtk::Entry m_entry;
  ExampleAssistant m_assistant;
};

#endif /* GTKMM_EXAMPLEWINDOW_H */
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include "exampleassistant.h"

ExampleWindow::ExampleWindow()
: m_table(3, 2),
  m_button("Show the assistant"),
  m_label1("State of assistant checkbutton:"),
  m_label2("Contents of assistant entry:")
{
  set_title("Gtk::Assistant example");
  set_border_width(12);

  m_table.attach(m_button, 0, 2, 0, 1, Gtk::FILL, Gtk::EXPAND);
  m_table.attach(m_label1, 0, 1, 1, 2, Gtk::FILL, Gtk::EXPAND);
  m_table.attach(m_label2, 0, 1, 2, 3, Gtk::FILL, Gtk::EXPAND);
  m_table.attach(m_check, 1, 2, 1, 2);
  m_table.attach(m_entry, 1, 2, 2, 3);
  add(m_table);

  m_label1.set_alignment(0.0, 0.5);
  m_label2.set_alignment(0.0, 0.5);

  m_button.signal_clicked().connect(sigc::mem_fun(*this,
    &ExampleWindow::on_button_clicked));
  m_assistant.signal_apply().connect(sigc::mem_fun(*this,
    &ExampleWindow::on_assistant_apply));

  m_check.set_sensitive(false);
  m_entry.set_sensitive(false);

  show_all_children();
```

```
}

ExampleWindow::~ExampleWindow()
{
}

void ExampleWindow::on_assistant_apply()
{
  bool check_state;
  Glib::ustring entry_text;

  m_assistant.get_result(check_state, entry_text);
  m_check.set_active(check_state);
  m_entry.set_text(entry_text);
}

void ExampleWindow::on_button_clicked()
{
  m_assistant.show();
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;
  // Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

File: `exampleassistant.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include <iostream>
#include "exampleassistant.h"

ExampleAssistant::ExampleAssistant()
: m_box(Gtk::ORIENTATION_HORIZONTAL, 12),
  m_label1("Type text to allow the assistant to continue:"),
  m_label2("Confirmation page"),
  m_check("Optional extra information")
{
  set_title("Gtk::Assistant example");
  set_border_width(12);
  set_default_size(400, 300);

  m_box.pack_start(m_label1);
  m_box.pack_start(m_entry);

  append_page(m_box);
  append_page(m_check);
  append_page(m_label2);

  set_page_title(*get_nth_page(0), "Page 1");
  set_page_title(*get_nth_page(1), "Page 2");
  set_page_title(*get_nth_page(2), "Confirmation");

  set_page_complete(m_check, true);
  set_page_complete(m_label2, true);

  set_page_type(m_box, Gtk::ASSISTANT_PAGE_INTRO);
  set_page_type(m_label2, Gtk::ASSISTANT_PAGE_CONFIRM);

  signal_apply().connect(sigc::mem_fun(*this,
    &ExampleAssistant::on_assistant_apply));
  signal_cancel().connect(sigc::mem_fun(*this,
    &ExampleAssistant::on_assistant_cancel));
  signal_close().connect(sigc::mem_fun(*this,
    &ExampleAssistant::on_assistant_close));
  signal_prepare().connect(sigc::mem_fun(*this,
    &ExampleAssistant::on_assistant_prepare));

  m_entry.signal_changed().connect(sigc::mem_fun(*this,
    &ExampleAssistant::on_entry_changed));

  show_all_children();
}
```

```
ExampleAssistant::~ExampleAssistant()
{
}

void ExampleAssistant::get_result(bool& check_state, Glib::ustring& entry_text)
{
  check_state = m_check.get_active();
  entry_text = m_entry.get_text();
}

void ExampleAssistant::on_assistant_apply()
{
  std::cout << "Apply was clicked";
  print_status();
}

void ExampleAssistant::on_assistant_cancel()
{
  std::cout << "Cancel was clicked";
  print_status();
  hide();
}

void ExampleAssistant::on_assistant_close()
{
  std::cout << "Assistant was closed";
  print_status();
  hide();
}

void ExampleAssistant::on_assistant_prepare(Gtk::Widget* /* widget */)
{
  set_title(Glib::ustring::compose("Gtk::Assistant example (Page %1 of %2)",
    get_current_page() + 1, get_n_pages()));
}

void ExampleAssistant::on_entry_changed()
{
  // The page is only complete if the entry contains text.
  if(m_entry.get_text_length())
    set_page_complete(m_box, true);
  else
    set_page_complete(m_box, false);
}

void ExampleAssistant::print_status()
{
  std::cout << ", entry contents: \"" << m_entry.get_text()
    << "\", checkbutton status: " << m_check.get_active() << std::endl;
}
```

9. The TreeView widget

The `Gtk::TreeView` widget can contain lists or trees of data, in columns.

9.1. The Model

Each `Gtk::TreeView` has an associated `Gtk::TreeModel`, which contains the data displayed by the `TreeView`. Each `Gtk::TreeModel` can be used by more than one `Gtk::TreeView`. For instance, this allows the same underlying data to be displayed and edited in 2 different ways at the same time. Or the 2 Views might display different columns from the same Model data, in the same way that 2 SQL queries (or "views") might show different fields from the same database table.

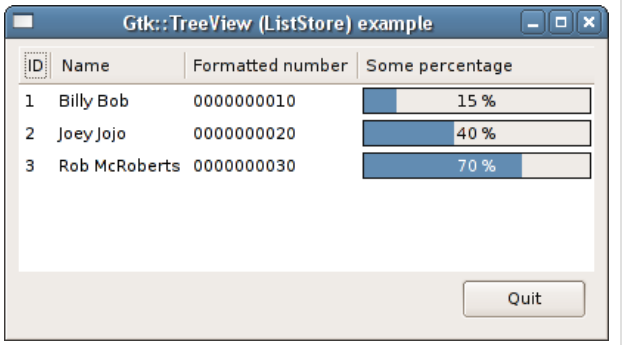
Although you can theoretically implement your own Model, you will normally use either the `ListStore` or `TreeStore` model classes.

Reference

9.1.1. ListStore, for rows

The `ListStore` contains simple rows of data, and each row has no children.

Figure 9-1 **TreeView - ListStore**



Reference

9.1.2. TreeStore, for a hierarchy

The `TreeStore` contains rows of data, and each row may have child rows.

Figure 9-2 **TreeView - TreeStore**

Reference

9.1.3. Model Columns

The `TreeModelColumnRecord` class is used to keep track of the columns and their data types. You add `TreeModelColumn` instances to the `ColumnRecord` and then use those `TreeModelColumns` when getting and setting the data in model rows. You will probably find it convenient to derive a new `TreeModelColumnRecord` which has your `TreeModelColumn` instances as member data.

```
class ModelColumns : public Gtk::TreeModelColumnRecord
{
public:

  ModelColumns()
    { add(m_col_text); add(m_col_number); }
```

```
  Gtk::TreeModelColumn<Glib::ustring> m_col_text;
  Gtk::TreeModelColumn<int> m_col_number;
};

ModelColumns m_Columns;
```

You specify the `ColumnRecord` when creating the Model, like so:

```
Glib::RefPtr<Gtk::ListStore> refListStore =
    Gtk::ListStore::create(m_Columns);
```

Note that the instance (such as m_Columns here) should usually not be static, because it often needs to be instantiated after glibmm has been instantiated.

9.1.4. Adding Rows

Add rows to the model with the `append()`, `prepend()`, or `insert()` methods.

```
Gtk::TreeModel::iterator iter = m_refListStore->append();
```

You can dereference the iterator to get the Row:

```
Gtk::TreeModel::Row row = *iter;
```


9.1.4.1. Adding child rows

`Gtk::TreeStore` models can have child items. Add them with the `append()`, `prepend()`, or `insert()` methods, like so:

```
Gtk::TreeModel::iterator iter_child =
    m_refListStore->append(row.children());
```

9.1.5. Setting values

You can use the `operator[]` override to set the data for a particular column in the row, specifying the `TreeModelColumn` used to create the model.

```
row[m_Columns.m_col_text] = "sometext";
```

9.1.6. Getting values

You can use the `operator[]` override to get the data in a particular column in a row, specifiying the `TreeModelColumn` used to create the model.

```
Glib::ustring strText = row[m_Columns.m_col_text];
int number = row[m_Columns.m_col_number];
```

The compiler will complain if you use an inappropriate type. For instance, this would generate a compiler error:

```
//compiler error - no conversion from ustring to int.
int number = row[m_Columns.m_col_text];
```

9.1.7. "Hidden" Columns

You might want to associate extra data with each row. If so, just add it as a Model column, but don't add it to the View.

9.2. The View

The View is the actual widget (`Gtk::TreeView`) that displays the model (`Gtk::TreeModel`) data and allows the user to interact with it. The View can show all of the model's columns, or just some, and it can show them in various ways.

Reference

9.2.1. Using a Model

You can specify a `Gtk::TreeModel` when constructing the `Gtk::TreeView`, or you can use the `set_model()` method, like so:

```
m_TreeView.set_model(m_refListStore);
```

9.2.2. Adding View Columns

You can use the `append_column()` method to tell the View that it should display certain Model columns, in a certain order, with a certain column title.

```
m_TreeView.append_column("Messages", m_Columns.m_col_text);
```

When using this simple `append_column()` override, the `TreeView` will display the model data with an appropriate `CellRenderer`. For instance, strings and numbers are shown in a simple `Gtk::Entry` widget, and booleans are shown in a `Gtk::CheckButton`. This is usually what you need. For other column types you must either connect a callback that converts your type into a string representation, with `TreeViewColumn::set_cell_data_func()`, or derive a custom `CellRenderer`. Note that (unsigned) short is not supported by default - You could use (unsigned) int or (unsigned) long as the column type instead.

9.2.3. More than one Model Column per View Column

To render more than one model column in a view column, you need to create the `TreeView::Column` widget manually, and use `pack_start()` to add the model columns to it.

Then use `append_column()` to add the view Column to the View. Notice that `Gtk::View::append_column()` is overridden to accept either a prebuilt `Gtk::View::Column` widget, or just the `TreeModelColumn` from which it generates an appropriate `Gtk::View::Column` widget.

Here is some example code from `demos/gtk-demo/example_stockbrowser.cc`, which has a pixbuf icon and a text name in the same column:

```
Gtk::TreeView::Column* pColumn =
    Gtk::manage( new Gtk::TreeView::Column("Symbol") );

// m_columns.icon and m_columns.symbol are columns in the model.
// pColumn is the column in the TreeView:
pColumn->pack_start(m_columns.icon, false); //false = don't expand.
pColumn->pack_start(m_columns.symbol);

m_TreeView.append_column(*pColumn);
```

9.2.4. Specifying CellRenderer details

The default `CellRenderers` and their default behaviour will normally suffice, but you might occasionally need finer control. For instance, this example code from `demos/gtk-demo/example_treestore.cc`, manually constructs a `Gtk::CellRenderer` widget and instructs it to render the data from various model columns through various aspects of its appearance.

```
Gtk::CellRendererToggle* pRenderer =
    Gtk::manage( new Gtk::CellRendererToggle() );
int cols_count = m_TreeView.append_column("Alex", *pRenderer);
Gtk::TreeViewColumn* pColumn = m_TreeView.get_column(cols_count-1);
if(pColumn)
{
  pColumn->add_attribute(pRenderer->property_active(),
      m_columns.alex);
  pColumn->add_attribute(pRenderer->property_visible(),
      m_columns.visible);
  pColumn->add_attribute(pRenderer->property_activatable(),
      m_columns.world);
```

You can also connect to `CellRenderer` signals to detect user actions. For instance:

```
Gtk::CellRendererToggle* pRenderer =
    Gtk::manage( new Gtk::CellRendererToggle() );
pRenderer->signal_toggled().connect(
    sigc::bind( sigc::mem_fun(*this,
        &Example_TreeView_TreeStore::on_cell_toggled), m_columns.dave)
);
```

9.2.5. Editable Cells

9.2.5.1. Automatically-stored editable cells.

Cells in a `TreeView` can be edited in-place by the user. To allow this, use the `Gtk::TreeView` `insert_column_editable()` and `append_column_editable()` methods instead of `insert_column()` and `append_column()`. When these cells are edited the new values will be stored immediately in the Model. Note that these methods are templates which can only be instantiated for simple column types such as `Glib::ustring`, int, and long.

9.2.5.2. Implementing custom logic for editable cells.

However, you might not want the new values to be stored immediately. For instance, maybe you want to restrict the input to certain characters or ranges of values.

To achieve this, you should use the normal `Gtk::TreeView` `insert_column()` and `append_column()` methods, then use `get_column_cell_renderer()` to get the `Gtk::CellRenderer` used by that column.

You should then cast that `Gtk::CellRenderer*` to the specific `CellRenderer` that you expect, so you can use specific API.

For instance, for a CellRendererText, you would set the cell's *editable* property to true, like so:

```
cell.property_editable() = true;
```

For a CellRendererToggle, you would set the *activatable* property instead.

You can then connect to the appropriate "edited" signal. For instance, connect to `Gtk::CellRendererText::signal_edited()`, or `Gtk::CellRendererToggle::signal_toggled()`. If the column contains more than one `CellRenderer` then you will need to use `Gtk::TreeView::get_column()` and then call `get_cell_renderers()` on that view Column.

In your signal handler, you should examine the new value and then store it in the Model if that is appropriate for your application.

9.3. Iterating over Model Rows

`Gtk::TreeModel` provides a C++ Standard Library-style container of its children, via the `children()` method. You can use the familiar `begin()` and `end()` methods iterator incrementing, like so:

```
typedef Gtk::TreeModel::Children type_children; //minimise code length.
type_children children = refModel->children();
for(type_children::iterator iter = children.begin();
    iter != children.end(); ++iter)
{
  Gtk::TreeModel::Row row = *iter;
  //Do something with the row - see above for set/get.
}
```


9.3.1. Row children

When using a `Gtk::TreeStore`, the rows can have child rows, which can have their own children in turn. Use `Gtk::TreeModel::Row::children()` to get the container of child `Row`s:

```
Gtk::TreeModel::Children children = row.children();
```

9.4. The Selection

To find out what rows the user has selected, get the `Gtk::TreeView::Selection` object from the `TreeView`, like so:

```
Glib::RefPtr<Gtk::TreeSelection> refTreeSelection =
    m_TreeView.get_selection();
```


9.4.1. Single or multiple selection

By default, only single rows can be selected, but you can allow multiple selection by setting the mode, like so:

```
refTreeSelection->set_mode(Gtk::SELECTION_MULTIPLE);
```

9.4.2. The selected rows

For single-selection, you can just call `get_selected()`, like so:

```
TreeModel::iterator iter = refTreeSelection->get_selected();
if(iter) //If anything is selected
{
  TreeModel::Row row = *iter;
  //Do something with the row.
}
```

For multiple-selection, you need to define a callback, and give it to `selected_foreach()`, `selected_foreach_path()`, or `selected_foreach_iter()`, like so:

```
refTreeSelection->selected_foreach_iter(
    sigc::mem_fun(*this, &TheClass::selected_row_callback) );

void TheClass::selected_row_callback(
    const Gtk::TreeModel::iterator& iter)
{
  TreeModel::Row row = *iter;
  //Do something with the row.
}
```

9.4.3. The "changed" signal

To respond to the user clicking on a row or range of rows, connect to the signal like so:

```
refTreeSelection->signal_changed().connect(
    sigc::mem_fun(*this, &Example_StockBrowser::on_selection_changed)
);
```

9.4.4. Preventing row selection

Maybe the user should not be able to select every item in your list or tree. For instance, in the gtk-demo, you can select a demo to see the source code, but it doesn't make any sense to select a demo category.

To control which rows can be selected, use the `set_select_function()` method, providing a `sigc::slot` callback. For instance:

```
m_refTreeSelection->set_select_function( sigc::mem_fun(*this,
    &DemoWindow::select_function) );
```

and then

```
bool DemoWindow::select_function(
    const Glib::RefPtr<Gtk::TreeModel>& model,
    const Gtk::TreeModel::Path& path, bool)
{
  const Gtk::TreeModel::iterator iter = model->get_iter(path);
  return iter->children().empty(); // only allow leaf nodes to be selected
}
```

9.4.5. Changing the selection

To change the selection, specify a `Gtk::TreeModel::iterator` or `Gtk::TreeModel::Row`, like so:

```
Gtk::TreeModel::Row row = m_refModel->children()[5]; //The fifth row.
if(row)
  refTreeSelection->select(row);
```

or

```
Gtk::TreeModel::iterator iter = m_refModel->children().begin()
if(iter)
  refTreeSelection->select(iter);
```

9.5. Sorting

The standard tree models (`TreeStore` and `ListStore` derive from `TreeSortable`, so they offer sorting functionality. For instance, call `set_sort_column()`, to sort the model by the specified column. Or supply a callback function to `set_sort_func()` to implement a more complicated sorting algorithm.

TreeSortable Reference

9.5.1. Sorting by clicking on columns

So that a user can click on a `TreeView`'s column header to sort the `TreeView`'s contents, call `Gtk::TreeViewModel::set_sort_column()`, supplying the model column on which model should be sorted when the header is clicked. For instance:

```
Gtk::TreeView::Column* pColumn = treeview.get_column(0);
if(pColumn)
  pColumn->set_sort_column(m_columns.m_col_id);
```

9.5.2. Independently sorted views of the same model

The `TreeView` already allows you to show the same `TreeModel` in two `TreeView` widgets. If you need one of these TreeViews to sort the model differently than the other then you should use a `TreeModelSort` instead of just, for instance, `Gtk::TreeViewModel::set_sort_column()`. `TreeModelSort` is a model that contains another model, presenting a sorted version of that model. For instance, you might add a sorted version of a model to a `TreeView` like so:

```
Glib::RefPtr<Gtk::TreeModelSort> sorted_model =
    Gtk::TreeModelSort::create(model);
sorted_model->set_sort_column(columns.m_col_name, Gtk::SORT_ASCENDING);
treeview.set_model(sorted_model);
```

Note, however, that the TreeView will provide iterators to the sorted model. You must convert them to iterators to the underlying child model in order to perform actions on that model. For instance:

```
void ExampleWindow::on_button_delete()
{
  Glib::RefPtr<Gtk::TreeSelection> refTreeSelection =
      m_treeview.get_selection();
  if(refTreeSelection)
  {
    Gtk::TreeModel::iterator sorted_iter =
        m_refTreeSelection->get_selected();
    if(sorted_iter)
    {
      Gtk::TreeModel::iterator iter =
          m_refModelSort->convert_iter_to_child_iter(sorted_iter);
      m_refModel->erase(iter);
    }
  }
}
```

TreeModelSort Reference

9.6. Drag and Drop

`Gtk::TreeView` already implments simple drag-and-drop when used with the `Gtk::ListStore` or `Gtk::TreeStore` models. If necessary, it also allows you to implement more complex behaviour when items are dragged and dropped, using the normal Drag and Drop API.

9.6.1. Reorderable rows

If you call `Gtk::TreeView::set_reorderable()` then your TreeView's items can be moved within the treeview itself. This is demonstrated in the `TreeStore` example.

However, this does not allow you any control of which items can be dragged, and where they can be dropped. If you need that extra control then you might create a derived `Gtk::TreeModel` from `Gtk::TreeStore` or `Gtk::ListStore` and override the `Gtk::TreeDragSource::row_draggable()` and `Gdk::TreeDragDest::row_drop_possible()` virtual methods. You can examine the `Gtk::TreeModel::Path`s provided and allow or disallow dragging or dropping by return `true` or `false`.

This is demonstrated in the drag_and_drop example.

9.7. Popup Context Menu

Lots of people need to implement right-click context menus for `TreeView`'s so we will explain how to do that here to save you some time. Apart from one or two points, it's

much the same as a normal context menu, as described in the menus chapter.

9.7.1. Handling `button_press_event`

To detect a click of the right mouse button, you need to handle the `button_press_event` signal, and check exactly which button was pressed. Because the `TreeView` normally handles this signal completely, you need to either override the default signal handler in a derived `TreeView` class, or use `connect_nofify()` instead of `connect()`. You probably also want to call the default handler before doing anything else, so that the right-click will cause the row to be selected first.

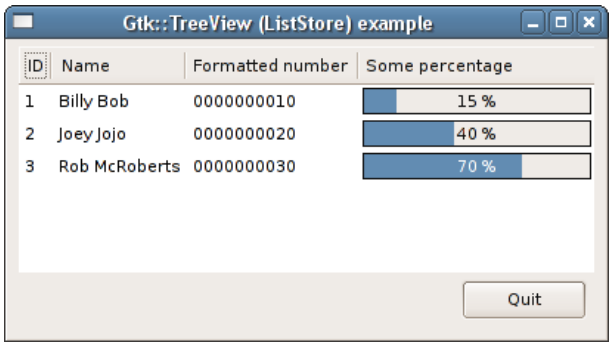
This is demonstrated in the Popup Custom Menu example.

9.8. Examples

9.8.1. ListStore

This example has a `Gtk::TreeView` widget, with a `Gtk::ListStore` model.

Figure 9-3 **TreeView - ListStore**



Source Code

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <gtkmm.h>

class ExampleWindow : public Gtk::Window
{
public:
  ExampleWindow();
  virtual ~ExampleWindow();

protected:
  //Signal handlers:
  void on_button_quit();

  //Tree model columns:
  class ModelColumns : public Gtk::TreeModel::ColumnRecord
  {
  public:

    ModelColumns()
    { add(m_col_id); add(m_col_name); add(m_col_number); add(m_col_percentage);}

    Gtk::TreeModelColumn<unsigned int> m_col_id;
```

```
    Gtk::TreeModelColumn<Glib::ustring> m_col_name;
    Gtk::TreeModelColumn<short> m_col_number;
    Gtk::TreeModelColumn<int> m_col_percentage;
  };

  ModelColumns m_Columns;

  //Child widgets:
  Gtk::Box m_VBox;

  Gtk::ScrolledWindow m_ScrolledWindow;
  Gtk::TreeView m_TreeView;
  Glib::RefPtr<Gtk::ListStore> m_refTreeModel;

  Gtk::ButtonBox m_ButtonBox;
  Gtk::Button m_Button_Quit;
};

#endif //GTKMM_EXAMPLEWINDOW_H
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include <iostream>
#include "examplewindow.h"

ExampleWindow::ExampleWindow()
: m_VBox(Gtk::ORIENTATION_VERTICAL),
  m_Button_Quit("Quit")
{
  set_title("Gtk::TreeView (ListStore) example");
  set_border_width(5);
  set_default_size(400, 200);

  add(m_VBox);

  //Add the TreeView, inside a ScrolledWindow, with the button underneath:
  m_ScrolledWindow.add(m_TreeView);

  //Only show the scrollbars when they are necessary:
  m_ScrolledWindow.set_policy(Gtk::POLICY_AUTOMATIC, Gtk::POLICY_AUTOMATIC);

  m_VBox.pack_start(m_ScrolledWindow);
  m_VBox.pack_start(m_ButtonBox, Gtk::PACK_SHRINK);

  m_ButtonBox.pack_start(m_Button_Quit, Gtk::PACK_SHRINK);
  m_ButtonBox.set_border_width(5);
  m_ButtonBox.set_layout(Gtk::BUTTONBOX_END);
  m_Button_Quit.signal_clicked().connect( sigc::mem_fun(*this,
              &ExampleWindow::on_button_quit) );

  //Create the Tree model:
  m_refTreeModel = Gtk::ListStore::create(m_Columns);
  m_TreeView.set_model(m_refTreeModel);

  //Fill the TreeView's model
  Gtk::TreeModel::Row row = *(m_refTreeModel->append());
  row[m_Columns.m_col_id] = 1;
  row[m_Columns.m_col_name] = "Billy Bob";
  row[m_Columns.m_col_number] = 10;
  row[m_Columns.m_col_percentage] = 15;

  row = *(m_refTreeModel->append());
  row[m_Columns.m_col_id] = 2;
  row[m_Columns.m_col_name] = "Joey Jojo";
  row[m_Columns.m_col_number] = 20;
  row[m_Columns.m_col_percentage] = 40;

  row = *(m_refTreeModel->append());
  row[m_Columns.m_col_id] = 3;
  row[m_Columns.m_col_name] = "Rob McRoberts";
  row[m_Columns.m_col_number] = 30;
  row[m_Columns.m_col_percentage] = 70;

  //Add the TreeView's view columns:
  //This number will be shown with the default numeric formatting.
  m_TreeView.append_column("ID", m_Columns.m_col_id);
  m_TreeView.append_column("Name", m_Columns.m_col_name);

  m_TreeView.append_column_numeric("Formatted number", m_Columns.m_col_number,
          "%010d" /* 10 digits, using leading zeroes. */);

  //Display a progress bar instead of a decimal number:
  Gtk::CellRendererProgress* cell = Gtk::manage(new Gtk::CellRendererProgress);
  int cols_count = m_TreeView.append_column("Some percentage", *cell);
  Gtk::TreeViewColumn* pColumn = m_TreeView.get_column(cols_count - 1);
```

```
  if(pColumn)
  {
    pColumn->add_attribute(cell->property_value(), m_Columns.m_col_percentage);
  }

  //Make all the columns reorderable:
  //This is not necessary, but it's nice to show the feature.
  //You can use TreeView::set_column_drag_function() to more
  //finely control column drag and drop.
  for(guint i = 0; i < 2; i++)
  {
    Gtk::TreeView::Column* pColumn = m_TreeView.get_column(i);
    pColumn->set_reorderable();
  }

  show_all_children();
}

ExampleWindow::~ExampleWindow()
{
}

void ExampleWindow::on_button_quit()
{
  hide();
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

9.8.2. TreeStore

This example is very similar to the `ListStore` example, but uses a `Gtk::TreeStore` model instead, and adds children to the rows.

Figure 9-4 **TreeView - TreeStore**

Source Code

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <gtkmm.h>
```

```
class ExampleWindow : public Gtk::Window
{
public:
  ExampleWindow();
  virtual ~ExampleWindow();

protected:
  //Signal handlers:
  void on_button_quit();
  void on_treeview_row_activated(const Gtk::TreeModel::Path& path, Gtk::TreeViewColumn* column);

  //Tree model columns:
  class ModelColumns : public Gtk::TreeModel::ColumnRecord
  {
  public:

    ModelColumns()
    { add(m_col_id); add(m_col_name); }

    Gtk::TreeModelColumn<int> m_col_id;
    Gtk::TreeModelColumn<Glib::ustring> m_col_name;
  };

  ModelColumns m_Columns;

  //Child widgets:
  Gtk::Box m_VBox;

  Gtk::ScrolledWindow m_ScrolledWindow;
  Gtk::TreeView m_TreeView;
  Glib::RefPtr<Gtk::TreeStore> m_refTreeModel;

  Gtk::ButtonBox m_ButtonBox;
  Gtk::Button m_Button_Quit;
};

#endif //GTKMM_EXAMPLEWINDOW_H
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include <iostream>
#include "examplewindow.h"

ExampleWindow::ExampleWindow()
: m_VBox(Gtk::ORIENTATION_VERTICAL),
  m_Button_Quit("Quit")
{
  set_title("Gtk::TreeView (TreeStore) example");
  set_border_width(5);
  set_default_size(400, 200);

  add(m_VBox);

  //Add the TreeView, inside a ScrolledWindow, with the button underneath:
  m_ScrolledWindow.add(m_TreeView);

  //Only show the scrollbars when they are necessary:
  m_ScrolledWindow.set_policy(Gtk::POLICY_AUTOMATIC, Gtk::POLICY_AUTOMATIC);

  m_VBox.pack_start(m_ScrolledWindow);
  m_VBox.pack_start(m_ButtonBox, Gtk::PACK_SHRINK);

  m_ButtonBox.pack_start(m_Button_Quit, Gtk::PACK_SHRINK);
  m_ButtonBox.set_border_width(5);
  m_ButtonBox.set_layout(Gtk::BUTTONBOX_END);
  m_Button_Quit.signal_clicked().connect(sigc::mem_fun(*this,
              &ExampleWindow::on_button_quit) );

  //Create the Tree model:
  m_refTreeModel = Gtk::TreeStore::create(m_Columns);
  m_TreeView.set_model(m_refTreeModel);

  //All the items to be reordered with drag-and-drop:
  m_TreeView.set_reorderable();

  //Fill the TreeView's model
  Gtk::TreeModel::Row row = *(m_refTreeModel->append());
  row[m_Columns.m_col_id] = 1;
  row[m_Columns.m_col_name] = "Billy Bob";

  Gtk::TreeModel::Row childrow = *(m_refTreeModel->append(row.children()));
  childrow[m_Columns.m_col_id] = 11;
  childrow[m_Columns.m_col_name] = "Billy Bob Junior";

  childrow = *(m_refTreeModel->append(row.children()));
```

```
  childrow[m_Columns.m_col_id] = 12;
  childrow[m_Columns.m_col_name] = "Sue Bob";

  row = *(m_refTreeModel->append());
  row[m_Columns.m_col_id] = 2;
  row[m_Columns.m_col_name] = "Joey Jojo";


  row = *(m_refTreeModel->append());
  row[m_Columns.m_col_id] = 3;
  row[m_Columns.m_col_name] = "Rob McRoberts";

  childrow = *(m_refTreeModel->append(row.children()));
  childrow[m_Columns.m_col_id] = 31;
  childrow[m_Columns.m_col_name] = "Xavier McRoberts";

  //Add the TreeView's view columns:
  m_TreeView.append_column("ID", m_Columns.m_col_id);
  m_TreeView.append_column("Name", m_Columns.m_col_name);

  //Connect signal:
  m_TreeView.signal_row_activated().connect(sigc::mem_fun(*this,
              &ExampleWindow::on_treeview_row_activated) );

  show_all_children();
}

ExampleWindow::~ExampleWindow()
{
}

void ExampleWindow::on_button_quit()
{
  hide();
}

void ExampleWindow::on_treeview_row_activated(const Gtk::TreeModel::Path& path,
        Gtk::TreeViewColumn* /* column */)
{
  Gtk::TreeModel::iterator iter = m_refTreeModel->get_iter(path);
  if(iter)
  {
    Gtk::TreeModel::Row row = *iter;
    std::cout << "Row activated: ID=" << row[m_Columns.m_col_id] << ", Name="
        << row[m_Columns.m_col_name] << std::endl;
  }
}
```

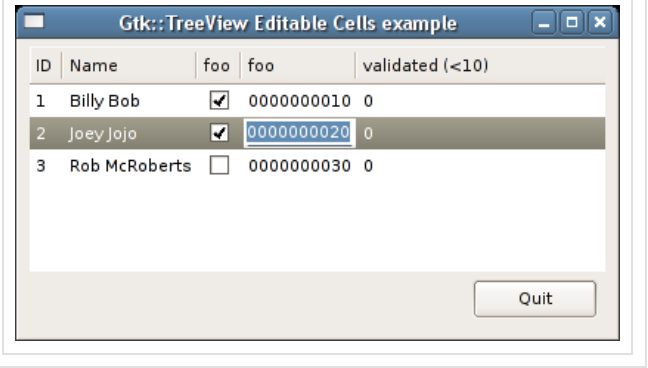
File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

9.8.3. Editable Cells

This example is identical to the `ListStore` example, but it uses `TreeView::append_column_editable()` instead of `TreeView::append_column()`.

Figure 9-5 **TreeView - Editable Cells**



Source Code

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <gtkmm.h>

class ExampleWindow : public Gtk::Window
{
public:
  ExampleWindow();
  virtual ~ExampleWindow();

protected:
  //Signal handlers:
  void on_button_quit();

  void treeviewcolumn_validated_on_cell_data(Gtk::CellRenderer* renderer, const Gtk::TreeModel::iterator& iter);
  void cellrenderer_validated_on_editing_started(Gtk::CellEditable* cell_editable, const Glib::ustring& path);
  void cellrenderer_validated_on_edited(const Glib::ustring& path_string, const Glib::ustring& new_text);

  //Tree model columns:
  class ModelColumns : public Gtk::TreeModel::ColumnRecord
  {
  public:

    ModelColumns()
    { add(m_col_id); add(m_col_name); add(m_col_foo); add(m_col_number); add(m_col_number_validated); }

    Gtk::TreeModelColumn<unsigned int> m_col_id;
    Gtk::TreeModelColumn<Glib::ustring> m_col_name;
    Gtk::TreeModelColumn<bool> m_col_foo;
    Gtk::TreeModelColumn<int> m_col_number;
    Gtk::TreeModelColumn<int> m_col_number_validated;
  };

  ModelColumns m_Columns;

  //Child widgets:
  Gtk::Box m_VBox;

  Gtk::ScrolledWindow m_ScrolledWindow;
  Gtk::TreeView m_TreeView;
  Glib::RefPtr<Gtk::ListStore> m_refTreeModel;

  Gtk::ButtonBox m_ButtonBox;
  Gtk::Button m_Button_Quit;

  //For the validated column:
  //You could also use a CellRendererSpin or a CellRendererProgress:
  Gtk::CellRendererText m_cellrenderer_validated;
  Gtk::TreeView::Column m_treeviewcolumn_validated;
  bool m_validate_retry;
  Glib::ustring m_invalid_text_for_retry;
};

#endif //GTKMM_EXAMPLEWINDOW_H
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include <iostream>
#include <cstdio>
```

```
#include <cstdlib>
#include "examplewindow.h"

using std::sprintf;
using std::strtol;

ExampleWindow::ExampleWindow()
: m_VBox(Gtk::ORIENTATION_VERTICAL),
  m_Button_Quit("Quit"),
  m_validate_retry(false)
{
  set_title("Gtk::TreeView Editable Cells example");
  set_border_width(5);
  set_default_size(400, 200);


  add(m_VBox);

  //Add the TreeView, inside a ScrolledWindow, with the button underneath:
  m_ScrolledWindow.add(m_TreeView);

  //Only show the scrollbars when they are necessary:
  m_ScrolledWindow.set_policy(Gtk::POLICY_AUTOMATIC, Gtk::POLICY_AUTOMATIC);

  m_VBox.pack_start(m_ScrolledWindow);
  m_VBox.pack_start(m_ButtonBox, Gtk::PACK_SHRINK);

  m_ButtonBox.pack_start(m_Button_Quit, Gtk::PACK_SHRINK);
  m_ButtonBox.set_border_width(5);
  m_ButtonBox.set_layout(Gtk::BUTTONBOX_END);
  m_Button_Quit.signal_clicked().connect( sigc::mem_fun(*this,
              &ExampleWindow::on_button_quit) );

  //Create the Tree model:
  m_refTreeModel = Gtk::ListStore::create(m_Columns);
  m_TreeView.set_model(m_refTreeModel);

  //Fill the TreeView's model
  Gtk::TreeModel::Row row = *(m_refTreeModel->append());
  row[m_Columns.m_col_id] = 1;
  row[m_Columns.m_col_name] = "Billy Bob";
  row[m_Columns.m_col_foo] = true;
  row[m_Columns.m_col_number] = 10;

  row = *(m_refTreeModel->append());
  row[m_Columns.m_col_id] = 2;
  row[m_Columns.m_col_name] = "Joey Jojo";
  row[m_Columns.m_col_foo] = true;
  row[m_Columns.m_col_number] = 20;

  row = *(m_refTreeModel->append());

  row[m_Columns.m_col_id] = 3;
  row[m_Columns.m_col_name] = "Rob McRoberts";
  row[m_Columns.m_col_foo] = false;
  row[m_Columns.m_col_number] = 30;

  //Add the TreeView's view columns:
  //We use the *_editable convenience methods for most of these,
  //because the default functionality is enough:
  m_TreeView.append_column_editable("ID", m_Columns.m_col_id);
  m_TreeView.append_column_editable("Name", m_Columns.m_col_name);
  m_TreeView.append_column_editable("foo", m_Columns.m_col_foo);
  m_TreeView.append_column_numeric_editable("foo", m_Columns.m_col_number,
          "%010d");


  //For this column, we create the CellRenderer ourselves, and connect our own
  //signal handlers, so that we can validate the data that the user enters, and
  //control how it is displayed.
  m_treeviewcolumn_validated.set_title("validated (<10)");
  m_treeviewcolumn_validated.pack_start(m_cellrenderer_validated);
  m_TreeView.append_column(m_treeviewcolumn_validated);

  //Tell the view column how to render the model values:
  m_treeviewcolumn_validated.set_cell_data_func(m_cellrenderer_validated,
          sigc::mem_fun(*this,
              &ExampleWindow::treeviewcolumn_validated_on_cell_data) );

  //Make the CellRenderer editable, and handle its editing signals:
  m_cellrenderer_validated.property_editable() = true;

  m_cellrenderer_validated.signal_editing_started().connect(
          sigc::mem_fun(*this,
        &ExampleWindow::cellrenderer_validated_on_editing_started) );

  m_cellrenderer_validated.signal_edited().connect( sigc::mem_fun(*this,
              &ExampleWindow::cellrenderer_validated_on_edited) );
```

```
  //If this was a CellRendererSpin then you would have to set the adjustment:
  //m_cellrenderer_validated.property_adjustment() = m_spin_adjustment;

  show_all_children();
}

ExampleWindow::~ExampleWindow()
{
}

void ExampleWindow::on_button_quit()
{
  hide();
}

void ExampleWindow::treeviewcolumn_validated_on_cell_data(
        Gtk::CellRenderer* /* renderer */,
        const Gtk::TreeModel::iterator& iter)
{
  //Get the value from the model and show it appropriately in the view:
  if(iter)
  {
    Gtk::TreeModel::Row row = *iter;
    int model_value = row[m_Columns.m_col_number_validated];

    //This is just an example.
    //In this case, it would be easier to use append_column_editable() or
    //append_column_numeric_editable()
    char buffer[32];
    sprintf(buffer, "%d", model_value);

    Glib::ustring view_text = buffer;
    m_cellrenderer_validated.property_text() = view_text;
  }
}

void ExampleWindow::cellrenderer_validated_on_editing_started(
        Gtk::CellEditable* cell_editable, const Glib::ustring& /* path */)
{
  //Start editing with previously-entered (but invalid) text,
  //if we are allowing the user to correct some invalid data.
  if(m_validate_retry)
  {
    //This is the CellEditable inside the CellRenderer.
    Gtk::CellEditable* celleditable_validated = cell_editable;

    //It's usually an Entry, at least for a CellRendererText:
    Gtk::Entry* pEntry = dynamic_cast<Gtk::Entry*>(celleditable_validated);
    if(pEntry)
    {
      pEntry->set_text(m_invalid_text_for_retry);
      m_validate_retry = false;
      m_invalid_text_for_retry.clear();
    }
  }

}

void ExampleWindow::cellrenderer_validated_on_edited(
        const Glib::ustring& path_string,
        const Glib::ustring& new_text)
{
  Gtk::TreePath path(path_string);

  //Convert the inputed text to an integer, as needed by our model column:
  char* pchEnd = 0;
  int new_value = strtol(new_text.c_str(), &pchEnd, 10);

  if(new_value > 10)
  {
    //Prevent entry of numbers higher than 10.

    //Tell the user:
    Gtk::MessageDialog dialog(*this,
            "The number must be less than 10. Please try again.",
            false, Gtk::MESSAGE_ERROR);
    dialog.run();

    //Start editing again, with the bad text, so that the user can correct it.
    //A real application should probably allow the user to revert to the
    //previous text.

    //Set the text to be used in the start_editing signal handler:
    m_invalid_text_for_retry = new_text;
    m_validate_retry = true;

    //Start editing again:
```

```
    m_TreeView.set_cursor(path, m_treeviewcolumn_validated,
            m_cellrenderer_validated, true /* start_editing */);
  }
  else
  {
    //Get the row from the path:
    Gtk::TreeModel::iterator iter = m_refTreeModel->get_iter(path);
    if(iter)
    {
      Gtk::TreeModel::Row row = *iter;

      //Put the new value in the model:
      row[m_Columns.m_col_number_validated] = new_value;
    }
  }
}
```

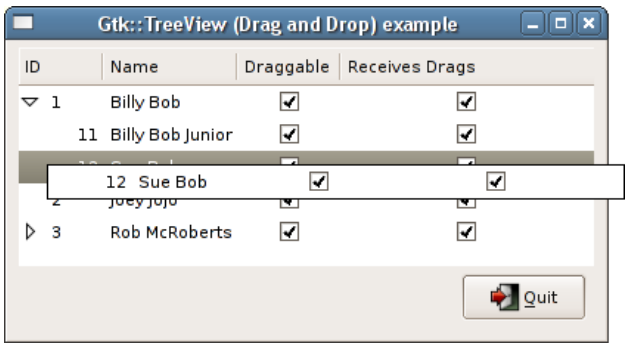
File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

9.8.4. Drag and Drop

This example is much like the `TreeStore` example, but has 2 extra columns to indicate whether the row can be dragged, and whether it can receive drag-and-dropped rows. It uses a derived `Gtk::TreeStore` which overrides the virtual functions as described in the TreeView Drag and Drop section..

Figure 9-6 **TreeView - Drag And Drop**



Source Code

File: `treemodel_dnd.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLE_TREEMODEL_DND_H
#define GTKMM_EXAMPLE_TREEMODEL_DND_H

#include <gtkmm.h>

class TreeModel_Dnd : public Gtk::TreeStore
{
protected:
  TreeModel_Dnd();

public:

  //Tree model columns:
  class ModelColumns : public Gtk::TreeModel::ColumnRecord
```

```
  {
  public:

    ModelColumns()
    { add(m_col_id); add(m_col_name); add(m_col_draggable); add(m_col_receivesdrags); }

    Gtk::TreeModelColumn<int> m_col_id;
    Gtk::TreeModelColumn<Glib::ustring> m_col_name;
    Gtk::TreeModelColumn<bool> m_col_draggable;
    Gtk::TreeModelColumn<bool> m_col_receivesdrags;
  };

  ModelColumns m_Columns;

  static Glib::RefPtr<TreeModel_Dnd> create();

protected:
  //Overridden virtual functions:
  virtual bool row_draggable_vfunc(const Gtk::TreeModel::Path& path) const;
  virtual bool row_drop_possible_vfunc(const Gtk::TreeModel::Path& dest, const Gtk::SelectionData& selection_data) const;

};

#endif //GTKMM_EXAMPLE_TREEMODEL_DND_H
```

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <gtkmm.h>
#include "treemodel_dnd.h"


class ExampleWindow : public Gtk::Window
{
public:
  ExampleWindow();
  virtual ~ExampleWindow();

protected:
  //Signal handlers:
  void on_button_quit();


  //Child widgets:
  Gtk::Box m_VBox;

  Gtk::ScrolledWindow m_ScrolledWindow;
  Gtk::TreeView m_TreeView;
  Glib::RefPtr<TreeModel_Dnd> m_refTreeModel;

  Gtk::ButtonBox m_ButtonBox;
  Gtk::Button m_Button_Quit;
};

#endif //GTKMM_EXAMPLEWINDOW_H
```

File: `treemodel_dnd.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "treemodel_dnd.h"
#include <iostream>

TreeModel_Dnd::TreeModel_Dnd()
{
  //We can't just call Gtk::TreeModel(m_Columns) in the initializer list
  //because m_Columns does not exist when the base class constructor runs.
  //And we can't have a static m_Columns instance, because that would be
  //instantiated before the gtkmm type system.
  //So, we use this method, which should only be used just after creation:
  set_column_types(m_Columns);
}

Glib::RefPtr<TreeModel_Dnd> TreeModel_Dnd::create()
{
  return Glib::RefPtr<TreeModel_Dnd>( new TreeModel_Dnd() );
}

bool
TreeModel_Dnd::row_draggable_vfunc(const Gtk::TreeModel::Path& path) const
{
  // Make the value of the "draggable" column determine whether this row can
  // be dragged:
```

```
  //TODO: Add a const version of get_iter to TreeModel:
  TreeModel_Dnd* unconstThis = const_cast<TreeModel_Dnd*>(this);
  const_iterator iter = unconstThis->get_iter(path);
  //const_iterator iter = get_iter(path);
  if(iter)
  {
    Row row = *iter;
    bool is_draggable = row[m_Columns.m_col_draggable];
    return is_draggable;
  }

  return Gtk::TreeStore::row_draggable_vfunc(path);
}

bool
TreeModel_Dnd::row_drop_possible_vfunc(const Gtk::TreeModel::Path& dest,
        const Gtk::SelectionData& selection_data) const
{
  //Make the value of the "receives drags" column determine whether a row can be
  //dragged into it:

  //dest is the path that the row would have after it has been dropped:
  //But in this case we are more interested in the parent row:
  Gtk::TreeModel::Path dest_parent = dest;
  bool dest_is_not_top_level = dest_parent.up();
  if(!dest_is_not_top_level || dest_parent.empty())
  {
    //The user wants to move something to the top-level.
    //Let's always allow that.
  }
  else
  {
    //Get an iterator for the row at this path:
    //We must unconst this. This should not be necessary with a future version
    //of gtkmm.
    //TODO: Add a const version of get_iter to TreeModel:
    TreeModel_Dnd* unconstThis = const_cast<TreeModel_Dnd*>(this);
    const_iterator iter_dest_parent = unconstThis->get_iter(dest_parent);
    //const_iterator iter_dest_parent = get_iter(dest);
    if(iter_dest_parent)
    {
      Row row = *iter_dest_parent;
      bool receives_drags = row[m_Columns.m_col_receivesdrags];
      return receives_drags;
    }
  }

  //You could also examine the row being dragged (via selection_data)
  //if you must look at both rows to see whether a drop should be allowed.
  //You could use
  //TODO: Add const version of get_from_selection_data(): Glib::RefPtr<const
  //Gtk::TreeModel> refThis = Glib::RefPtr<const Gtk::TreeModel>(this);
  //
  //Glib::RefPtr<Gtk::TreeModel> refThis =
  //Glib::RefPtr<Gtk::TreeModel>(const_cast<TreeModel_Dnd*>(this));
  //refThis->reference(); //, true /* take_copy */)
  //Gtk::TreeModel::Path path_dragged_row;
  //Gtk::TreeModel::Path::get_from_selection_data(selection_data, refThis,
  //path_dragged_row);

  return Gtk::TreeStore::row_drop_possible_vfunc(dest, selection_data);
}
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include <iostream>
#include "examplewindow.h"

ExampleWindow::ExampleWindow()
: m_VBox(Gtk::ORIENTATION_VERTICAL),
  m_Button_Quit(Gtk::Stock::QUIT)
{
  set_title("Gtk::TreeView (Drag and Drop) example");
  set_border_width(5);
  set_default_size(400, 200);

  add(m_VBox);

  //Add the TreeView, inside a ScrolledWindow, with the button underneath:
  m_ScrolledWindow.add(m_TreeView);

  //Only show the scrollbars when they are necessary:
  m_ScrolledWindow.set_policy(Gtk::POLICY_AUTOMATIC, Gtk::POLICY_AUTOMATIC);
```

```
  m_VBox.pack_start(m_ScrolledWindow);
  m_VBox.pack_start(m_ButtonBox, Gtk::PACK_SHRINK);

  m_ButtonBox.pack_start(m_Button_Quit, Gtk::PACK_SHRINK);
  m_ButtonBox.set_border_width(5);
  m_ButtonBox.set_layout(Gtk::BUTTONBOX_END);
  m_Button_Quit.signal_clicked().connect(sigc::mem_fun(*this,
              &ExampleWindow::on_button_quit) );

  //Create the Tree model:
  //Use our derived model, which overrides some Gtk::TreeDragDest and
  //Gtk::TreeDragSource virtual functions:
  //The columns are declared in the overridden TreeModel.
  m_refTreeModel = TreeModel_Dnd::create();
  m_TreeView.set_model(m_refTreeModel);

  //Enable Drag-and-Drop of TreeView rows:
  //See also the derived TreeModel's *_vfunc overrides.
  m_TreeView.enable_model_drag_source();
  m_TreeView.enable_model_drag_dest();

  //Fill the TreeView's model
  Gtk::TreeModel::Row row = *(m_refTreeModel->append());
  row[m_refTreeModel->m_Columns.m_col_id] = 1;
  row[m_refTreeModel->m_Columns.m_col_name] = "Billy Bob";
  row[m_refTreeModel->m_Columns.m_col_draggable] = true;
  row[m_refTreeModel->m_Columns.m_col_receivesdrags] = true;

  Gtk::TreeModel::Row childrow = *(m_refTreeModel->append(row.children()));
  childrow[m_refTreeModel->m_Columns.m_col_id] = 11;
  childrow[m_refTreeModel->m_Columns.m_col_name] = "Billy Bob Junior";
  childrow[m_refTreeModel->m_Columns.m_col_draggable] = true;
  childrow[m_refTreeModel->m_Columns.m_col_receivesdrags] = true;

  childrow = *(m_refTreeModel->append(row.children()));
  childrow[m_refTreeModel->m_Columns.m_col_id] = 12;
  childrow[m_refTreeModel->m_Columns.m_col_name] = "Sue Bob";
  childrow[m_refTreeModel->m_Columns.m_col_draggable] = true;
  childrow[m_refTreeModel->m_Columns.m_col_receivesdrags] = true;

  row = *(m_refTreeModel->append());
  row[m_refTreeModel->m_Columns.m_col_id] = 2;
  row[m_refTreeModel->m_Columns.m_col_name] = "Joey Jojo";
  row[m_refTreeModel->m_Columns.m_col_draggable] = true;
  row[m_refTreeModel->m_Columns.m_col_receivesdrags] = true;

  row = *(m_refTreeModel->append());
  row[m_refTreeModel->m_Columns.m_col_id] = 3;
  row[m_refTreeModel->m_Columns.m_col_name] = "Rob McRoberts";
  row[m_refTreeModel->m_Columns.m_col_draggable] = true;
  row[m_refTreeModel->m_Columns.m_col_receivesdrags] = true;

  childrow = *(m_refTreeModel->append(row.children()));
  childrow[m_refTreeModel->m_Columns.m_col_id] = 31;
  childrow[m_refTreeModel->m_Columns.m_col_name] = "Xavier McRoberts";
  childrow[m_refTreeModel->m_Columns.m_col_draggable] = true;
  childrow[m_refTreeModel->m_Columns.m_col_receivesdrags] = true;

  //Add the TreeView's view columns:
  m_TreeView.append_column("ID", m_refTreeModel->m_Columns.m_col_id);
  m_TreeView.append_column("Name", m_refTreeModel->m_Columns.m_col_name);
  m_TreeView.append_column_editable("Draggable",
          m_refTreeModel->m_Columns.m_col_draggable);
  m_TreeView.append_column_editable("Receives Drags",
          m_refTreeModel->m_Columns.m_col_receivesdrags);

  show_all_children();
}

ExampleWindow::~ExampleWindow()
{
}

void ExampleWindow::on_button_quit()
{
  hide();
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

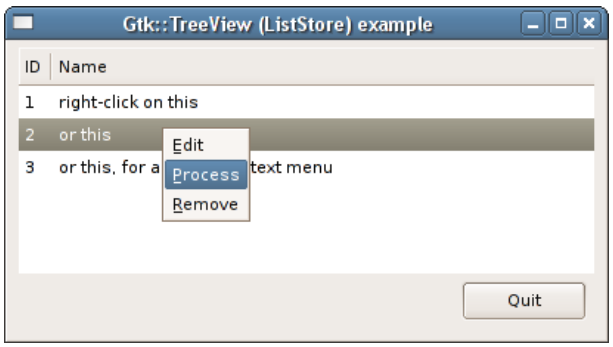
```
#include "examplewindow.h"
#include <gtkmm/main.h>
```

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

  ExampleWindow window;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

9.8.5. Popup Context Menu

This example is much like the `ListStore` example, but derives a custom `TreeView` in order to override the `button_press_event`, and also to encapsulate the tree model code in our derived class. See the TreeView Popup Context Menu section.

Figure 9-7 **TreeView - Popup Context Menu**



Source Code

File: `treeview_withpopup.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLE_TREEVIEW_WITHPOPUP_H
#define GTKMM_EXAMPLE_TREEVIEW_WITHPOPUP_H

#include <gtkmm.h>

class TreeView_WithPopup : public Gtk::TreeView
{
public:
  TreeView_WithPopup();
  virtual ~TreeView_WithPopup();

protected:
  // Override Signal handler:
  // Alternatively, use signal_button_press_event().connect_notify()
  virtual bool on_button_press_event(GdkEventButton *ev);

  //Signal handler for popup menu items:
  void on_menu_file_popup_generic();


  //Tree model columns:
  class ModelColumns : public Gtk::TreeModel::ColumnRecord
  {
  public:

    ModelColumns()
    { add(m_col_id); add(m_col_name); }

    Gtk::TreeModelColumn<unsigned int> m_col_id;
    Gtk::TreeModelColumn<Glib::ustring> m_col_name;
  };

  ModelColumns m_Columns;

  //The Tree model:
  Glib::RefPtr<Gtk::ListStore> m_refTreeModel;

  Gtk::Menu m_Menu_Popup;
};

#endif //GTKMM_EXAMPLE_TREEVIEW_WITHPOPUP_H
```

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <gtkmm.h>
#include "treeview_withpopup.h"

class ExampleWindow : public Gtk::Window
{
public:
  ExampleWindow();
  virtual ~ExampleWindow();

protected:
  //Signal handlers:
  void on_button_quit();



  //Child widgets:
  Gtk::Box m_VBox;

  Gtk::ScrolledWindow m_ScrolledWindow;
  TreeView_WithPopup m_TreeView;

  Gtk::ButtonBox m_ButtonBox;
  Gtk::Button m_Button_Quit;
};

#endif //GTKMM_EXAMPLEWINDOW_H
```

File: `treeview_withpopup.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "treeview_withpopup.h"
#include <iostream>

TreeView_WithPopup::TreeView_WithPopup()
{
  //Create the Tree model:
  m_refTreeModel = Gtk::ListStore::create(m_Columns);
  set_model(m_refTreeModel);

  //Fill the TreeView's model
  Gtk::TreeModel::Row row = *(m_refTreeModel->append());
  row[m_Columns.m_col_id] = 1;
  row[m_Columns.m_col_name] = "right-click on this";

  row = *(m_refTreeModel->append());
  row[m_Columns.m_col_id] = 2;
  row[m_Columns.m_col_name] = "or this";

  row = *(m_refTreeModel->append());
  row[m_Columns.m_col_id] = 3;
  row[m_Columns.m_col_name] = "or this, for a popup context menu";

  //Add the TreeView's view columns:
  append_column("ID", m_Columns.m_col_id);
  append_column("Name", m_Columns.m_col_name);

  //Fill popup menu:
  Gtk::MenuItem* item = Gtk::manage(new Gtk::MenuItem("_Edit", true));
  item->signal_activate().connect(
    sigc::mem_fun(*this, &TreeView_WithPopup::on_menu_file_popup_generic) );
  m_Menu_Popup.append(*item);

  item = Gtk::manage(new Gtk::MenuItem("_Process", true));
  item->signal_activate().connect(
    sigc::mem_fun(*this, &TreeView_WithPopup::on_menu_file_popup_generic) );
  m_Menu_Popup.append(*item);

  item = Gtk::manage(new Gtk::MenuItem("_Remove", true));
  item->signal_activate().connect(
    sigc::mem_fun(*this, &TreeView_WithPopup::on_menu_file_popup_generic) );
  m_Menu_Popup.append(*item);

  m_Menu_Popup.accelerate(*this);
  m_Menu_Popup.show_all(); //Show all menu items when the menu pops up

#ifndef GLIBMM_DEFAULT_SIGNAL_HANDLERS_ENABLED
  signal_button_press_event()
    .connect(sigc::mem_fun(*this, &TreeView_WithPopup::on_button_press_event), false);
#endif
}
```

```
TreeView_WithPopup::~TreeView_WithPopup()
{
}

bool TreeView_WithPopup::on_button_press_event(GdkEventButton* event)
{
  bool return_value = false;

  //Call base class, to allow normal handling,
  //such as allowing the row to be selected by the right-click:
  return_value = TreeView::on_button_press_event(event);

  //Then do our custom stuff:
  if( (event->type == GDK_BUTTON_PRESS) && (event->button == 3) )
  {
    m_Menu_Popup.popup(event->button, event->time);
  }

  return return_value;
}

void TreeView_WithPopup::on_menu_file_popup_generic()
{
  std::cout << "A popup menu item was selected." << std::endl;

  Glib::RefPtr<Gtk::TreeView::Selection> refSelection = get_selection();
  if(refSelection)
  {
    Gtk::TreeModel::iterator iter = refSelection->get_selected();
    if(iter)
    {
      int id = (*iter)[m_Columns.m_col_id];
      std::cout << "  Selected ID=" << id << std::endl;
    }
  }
}
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include <iostream>
#include "examplewindow.h"

ExampleWindow::ExampleWindow()
: m_VBox(Gtk::ORIENTATION_VERTICAL),
  m_Button_Quit("Quit")
{
  set_title("Gtk::TreeView (ListStore) example");
  set_border_width(5);
  set_default_size(400, 200);

  add(m_VBox);

  //Add the TreeView, inside a ScrolledWindow, with the button underneath:
  m_ScrolledWindow.add(m_TreeView);

  //Only show the scrollbars when they are necessary:
  m_ScrolledWindow.set_policy(Gtk::POLICY_AUTOMATIC, Gtk::POLICY_AUTOMATIC);

  m_VBox.pack_start(m_ScrolledWindow);
  m_VBox.pack_start(m_ButtonBox, Gtk::PACK_SHRINK);

  m_ButtonBox.pack_start(m_Button_Quit, Gtk::PACK_SHRINK);
  m_ButtonBox.set_border_width(5);
  m_ButtonBox.set_layout(Gtk::BUTTONBOX_END);
  m_Button_Quit.signal_clicked().connect( sigc::mem_fun(*this,
              &ExampleWindow::on_button_quit) );

  show_all_children();
}

ExampleWindow::~ExampleWindow()
{
}

void ExampleWindow::on_button_quit()
{
  hide();
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

10. Combo Boxes

The `ComboBox` widgets offers a list (or tree) of choices in a dropdown menu. If appropriate, it can show extra information about each item, such as text, a picture, a checkbox, or a progress bar. The `ComboBox` widget usually restricts the user to the available choices, but it can optionally have an `Entry`, allowing the user to enter arbitrary text if the none of the available choices are suitable.

The list is provided via a `TreeModel`, and columns from this model are added to the ComboBox's view with the `ComboBox::pack_start()` method. This provides flexibility and compile-time type-safety, but the `ComboBoxText` class provides a simpler text-based specialization in case that flexibility is not required.

Reference

10.1. The model

The model for a ComboBox can be defined and filled exactly as for a `TreeView`. For instance, you might derive a ComboBox class with one integer and one text columns, like so:

```
ModelColumns()
{ add(m_col_id); add(m_col_name); }

  Gtk::TreeModelColumn<int> m_col_id;
  Gtk::TreeModelColumn<Glib::ustring> m_col_name;
};

ModelColumns m_columns;
```

After appending rows to this model, you should provide the model to the `ComboBox` with the `set_model()` method. Then use the `pack_start()` or `pack_end()` methods to specify what methods will be displayed in the ComboBox. As with the TreeView you may either use the default cell renderer by passing the `TreeModelColumn` to the pack methods, or you may instantiate a specific `CellRenderer` and specify a particular mapping with either `add_attribute()` or `set_cell_data_func()`. Note that these methods are in the `CellLayout` base class.

10.2. The chosen item

To discover what item, if any, the user has chosen from the ComboBox, call `ComboBox::get_active()`. This returns a `TreeModel::iterator` that you can dereference to a `Row` in order to read the values in your columns. For instance, you might read an integer ID value from the model, even though you have chosen only to show the human-readable description in the Combo. For instance:

```
Gtk::TreeModel::iterator iter = m_Combo.get_active();
```

```
if(iter)
{
  Gtk::TreeModel::Row row = *iter;

  //Get the data for the selected row, using our knowledge
  //of the tree model:
  int id = row[m_Columns.m_col_id];
  set_something_id_chosen(id); //Your own function.
}
else
  set_nothing_chosen(); //Your own function.
```

10.3. Responding to changes

You might need to react to every change of selection in the ComboBox, for instance to update other widgets. To do so, you should handle the "changed" signal. For instance:

```
m_combo.signal_changed().connect( sigc::mem_fun(*this,
      &ExampleWindow::on_combo_changed) );
```

10.4. Full Example

Figure 10-1 **ComboBox**

Source Code

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <gtkmm/window.h>
#include <gtkmm/comboboxtext.h>
#include <gtkmm/liststore.h>

class ExampleWindow : public Gtk::Window
{
public:
  ExampleWindow();
  virtual ~ExampleWindow();

protected:
  //Signal handlers:
  void on_combo_changed();

  //Tree model columns:
  class ModelColumns : public Gtk::TreeModel::ColumnRecord
  {
  public:

    ModelColumns()
    { add(m_col_id); add(m_col_name); }

    Gtk::TreeModelColumn<int> m_col_id;
    Gtk::TreeModelColumn<Glib::ustring> m_col_name;
  };

  ModelColumns m_Columns;

  //Child widgets:
  Gtk::ComboBox m_Combo;
  Glib::RefPtr<Gtk::ListStore> m_refTreeModel;
};

#endif //GTKMM_EXAMPLEWINDOW_H
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/stock.h>
```

```
#include <iostream>

ExampleWindow::ExampleWindow()
{
  set_title("ComboBox example");

  //Create the Tree model:
  //m_refTreeModel = Gtk::TreeStore::create(m_Columns);
  m_refTreeModel = Gtk::ListStore::create(m_Columns);
  m_Combo.set_model(m_refTreeModel);

  //Fill the ComboBox's Tree Model:
  Gtk::TreeModel::Row row = *(m_refTreeModel->append());
  row[m_Columns.m_col_id] = 1;
  row[m_Columns.m_col_name] = "Billy Bob";
  /*
  Gtk::TreeModel::Row childrow = *(m_refTreeModel->append(row.children()));
  childrow[m_Columns.m_col_id] = 11;
  childrow[m_Columns.m_col_name] = "Billy Bob Junior";

  childrow = *(m_refTreeModel->append(row.children()));
  childrow[m_Columns.m_col_id] = 12;
  childrow[m_Columns.m_col_name] = "Sue Bob";
  */

  row = *(m_refTreeModel->append());
  row[m_Columns.m_col_id] = 2;
  row[m_Columns.m_col_name] = "Joey Jojo";


  row = *(m_refTreeModel->append());
  row[m_Columns.m_col_id] = 3;
  row[m_Columns.m_col_name] = "Rob McRoberts";

  /*
  childrow = *(m_refTreeModel->append(row.children()));
  childrow[m_Columns.m_col_id] = 31;
  childrow[m_Columns.m_col_name] = "Xavier McRoberts";
  */

  //Add the model columns to the Combo (which is a kind of view),
  //rendering them in the default way:
  m_Combo.pack_start(m_Columns.m_col_id);
  m_Combo.pack_start(m_Columns.m_col_name);

  //Add the ComboBox to the window.
  add(m_Combo);

  //Connect signal handler:
  m_Combo.signal_changed().connect( sigc::mem_fun(*this, &ExampleWindow::on_combo_changed) );

  show_all_children();
}

ExampleWindow::~ExampleWindow()
{
}

void ExampleWindow::on_combo_changed()
{
  Gtk::TreeModel::iterator iter = m_Combo.get_active();
  if(iter)
  {
    Gtk::TreeModel::Row row = *iter;
    if(row)
    {
      //Get the data for the selected row, using our knowledge of the tree
      //model:
      int id = row[m_Columns.m_col_id];
      Glib::ustring name = row[m_Columns.m_col_name];

      std::cout << " ID=" << id << ", name=" << name << std::endl;
    }
  }
  else
    std::cout << "invalid iter" << std::endl;
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

int main(int argc, char *argv[])
```

```
{
  Gtk::Main kit(argc, argv);

  ExampleWindow window;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

10.5. Simple Text Example

Figure 10-2 **ComboBox**

Source Code

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <gtkmm/window.h>
#include <gtkmm/comboboxtext.h>

class ExampleWindow : public Gtk::Window
{
public:
  ExampleWindow();
  virtual ~ExampleWindow();

protected:
  //Signal handlers:
  void on_combo_changed();

  //Child widgets:
  Gtk::ComboBoxText m_Combo;
};

#endif //GTKMM_EXAMPLEWINDOW_H
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/stock.h>
#include <iostream>

ExampleWindow::ExampleWindow()
{
  set_title("ComboBoxText example");

  //Fill the combo:
  m_Combo.append("something");
  m_Combo.append("something else");
  m_Combo.append("something or other");

  add(m_Combo);

  //Connect signal handler:
  m_Combo.signal_changed().connect(sigc::mem_fun(*this,
              &ExampleWindow::on_combo_changed) );

  show_all_children();
}

ExampleWindow::~ExampleWindow()
{
}

void ExampleWindow::on_combo_changed()
{
  Glib::ustring text = m_Combo.get_active_text();
  if(!(text.empty()))
    std::cout << "Combo changed: " << text << std::endl;
}
```

File: main.cc (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

10.6. ComboBox with an Entry

A ComboBox may contain an Entry widget for entering of arbitrary text, by specifying true for the constructor's has_entry parameter.

10.6.1. The text column

So that the Entry can interact with the drop-down list of choices, you must specify which of your model columns is the text column, with set_text_column(). For instance:

```
m_combo.set_text_column(m_columns.m_col_name);
```

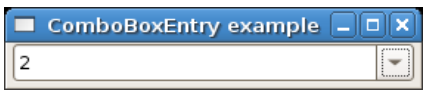
When you select a choice from the drop-down menu, the value from this column will be placed in the Entry.

10.6.2. The entry

Because the user may enter arbitrary text, an active model row isn't enough to tell us what text the user has inputted. Therefore, you should retrieve the Entry widget with the ComboBoxEntry::get_entry() method and call get_text() on that.

10.6.3. Full Example

Figure 10-3 **ComboBoxEntry**



Source Code

File: examplewindow.h (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <gtkmm/window.h>
#include <gtkmm/combobox.h>
#include <gtkmm/liststore.h>

class ExampleWindow : public Gtk::Window
{
public:
  ExampleWindow();
  virtual ~ExampleWindow();
```

```
protected:
  //Signal handlers:
  void on_combo_changed();

  //Tree model columns:
  class ModelColumns : public Gtk::TreeModel::ColumnRecord
  {
  public:

    ModelColumns()
    { add(m_col_id); add(m_col_name); }

    Gtk::TreeModelColumn<Glib::ustring> m_col_id; //The data to choose - this must be text.
    Gtk::TreeModelColumn<Glib::ustring> m_col_name;
  };

  ModelColumns m_Columns;

  //Child widgets:
  Gtk::ComboBox m_Combo;
  Glib::RefPtr<Gtk::ListStore> m_refTreeModel;
};

#endif //GTKMM_EXAMPLEWINDOW_H
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/stock.h>
#include <iostream>

ExampleWindow::ExampleWindow()
: m_Combo(true /* has_entry */)
{
  set_title("ComboBox example");

  //Create the Tree model:
  //m_refTreeModel = Gtk::TreeStore::create(m_Columns);
  m_refTreeModel = Gtk::ListStore::create(m_Columns);
  m_Combo.set_model(m_refTreeModel);

  //Fill the ComboBox's Tree Model:
  Gtk::TreeModel::Row row = *(m_refTreeModel->append());
  row[m_Columns.m_col_id] = "1";
  row[m_Columns.m_col_name] = "Billy Bob";
  /*
  Gtk::TreeModel::Row childrow = *(m_refTreeModel->append(row.children()));
  childrow[m_Columns.m_col_id] = 11;
  childrow[m_Columns.m_col_name] = "Billy Bob Junior";

  childrow = *(m_refTreeModel->append(row.children()));
  childrow[m_Columns.m_col_id] = 12;
  childrow[m_Columns.m_col_name] = "Sue Bob";
  */

  row = *(m_refTreeModel->append());
  row[m_Columns.m_col_id] = "2";
  row[m_Columns.m_col_name] = "Joey Jojo";


  row = *(m_refTreeModel->append());
  row[m_Columns.m_col_id] = "3";
  row[m_Columns.m_col_name] = "Rob McRoberts";

  /*
  childrow = *(m_refTreeModel->append(row.children()));
  childrow[m_Columns.m_col_id] = 31;
  childrow[m_Columns.m_col_name] = "Xavier McRoberts";
  */

  //Add the model columns to the Combo (which is a kind of view),
  //rendering them in the default way:
  //This is automatically rendered when we use set_entry_text_column().
  //m_Combo.pack_start(m_Columns.m_col_id);
  m_Combo.pack_start(m_Columns.m_col_name);

  m_Combo.set_entry_text_column(m_Columns.m_col_id);

  //Add the ComboBox to the window.
  add(m_Combo);

  //Connect signal handler:
  m_Combo.signal_changed().connect(sigc::mem_fun(*this,
              &ExampleWindow::on_combo_changed) );
```

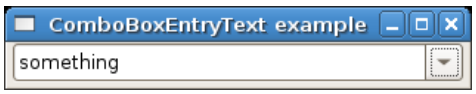
```
  show_all_children();
}

ExampleWindow::~ExampleWindow()
{
}

void ExampleWindow::on_combo_changed()
{
  Gtk::Entry* entry = m_Combo.get_entry();
  //Note: to get changes only when the entry has been completed,
  //instead of on every key press, connect to Entry::signal_changed()
  //instead of ComboBoxEntry::signal_changed.

  if(entry)
  {
    std::cout << " ID=" << entry->get_text() << std::endl;
  }
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

10.6.4. Simple Text Example

Figure 10-4 **ComboBoxEntryText**



Source Code

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <gtkmm/window.h>
#include <gtkmm/comboboxtext.h>

class ExampleWindow : public Gtk::Window
{
public:
  ExampleWindow();
  virtual ~ExampleWindow();

protected:
  //Signal handlers:
  void on_combo_changed();

  //Child widgets:
  Gtk::ComboBoxText m_Combo;
};

#endif //GTKMM_EXAMPLEWINDOW_H
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/stock.h>
#include <iostream>
```

```
ExampleWindow::ExampleWindow()
: m_Combo(true /* has_entry */)
{
  set_title("ComboBoxText example");

  //Fill the combo:
  m_Combo.append("something");
  m_Combo.append("something else");
  m_Combo.append("something or other");

  add(m_Combo);

  //Connect signal handler:
  m_Combo.signal_changed().connect(sigc::mem_fun(*this,
              &ExampleWindow::on_combo_changed) );

  m_Combo.property_has_frame() = false;
  show_all_children();
}

ExampleWindow::~ExampleWindow()
{
}

void ExampleWindow::on_combo_changed()
{
  Glib::ustring text = m_Combo.get_active_text();
  if(!(text.empty()))
    std::cout << "Combo changed: " << text << std::endl;
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

11. TextView

The `TextView` widget can be used to display and edit large amounts of formatted text. Like the `TreeView`, it has a model/view design. In this case the `TextBuffer` is the model.

11.1. The Buffer

`Gtk::TextBuffer` is a model containing the data for the `Gtk::TextView`, like the `Gtk::TreeModel` used by `Gtk::TreeView`. This allows two or more `Gtk::TextView`s to share the same `TextBuffer`, and allows those TextBuffers to be displayed slightly differently. Or you could maintain several `Gtk::TextBuffer`s and choose to display each one at different times in the same `Gtk::TextView` widget.

The `TextView` creates its own default `TextBuffer`, which you can access via the `get_buffer()` method.

Reference

11.1.1. Iterators

11.1.2. Tags and Formatting

11.1.2.1. Tags

To specify that some text in the buffer should have specific formatting, you must define a tag to hold that formatting information, and then apply that tag to the region of text. For instance, to define the tag and its properties:

```
Glib::RefPtr<Gtk::TextBuffer::Tag> refTagMatch =
    Gtk::TextBuffer::Tag::create();
refTagMatch->property_background() = "orange";
```

You can specify a name for the `Tag` when using the `create()` method, but it is not necessary.

The `Tag` class has many other properties.

Reference

11.1.2.2. TagTable

Each `Gtk::TextBuffer` uses a `Gtk::TextBuffer::TagTable`, which contains the `Tag`s for that buffer. 2 or more `TextBuffer`s may share the same `TagTable`. When you create `Tag`s you should add them to the `TagTable`. For instance:

```
Glib::RefPtr<Gtk::TextBuffer::TagTable> refTagTable =
    Gtk::TextBuffer::TagTable::create();
refTagTable->add(refTagMatch);
//Hopefully a future version of gtkmm will have a set_tag_table() method,
//for use after creation of the buffer.
Glib::RefPtr<Gtk::TextBuffer> refBuffer =
    Gtk::TextBuffer::create(refTagTable);
```

You can also use `get_tag_table()` to get, and maybe modify, the `TextBuffer`'s default `TagTable` instead of creating one explicitly.

Reference

11.1.2.3. Applying Tags

If you have created a `Tag` and added it to the `TagTable`, you may apply that tag to part of the `TextBuffer` so that some of the text is displayed with that formatting. You define the start and end of the range of text by specifying `Gtk::TextBuffer::iterator`s. For instance:

```
refBuffer->apply_tag(refTagMatch, iterRangeStart, iterRangeStop);
```

Or you could specify the tag when first inserting the text:
refBuffer->insert_with_tag(iter, "Some text", refTagMatch);

You can apply more than one `Tag` to the same text, by using `apply_tag()` more than once, or by using `insert_with_tags()`. The `Tag`s might specify different values for the same properties, but you can resolve these conflicts by using `Tag::set_priority()`.

11.1.3. Marks

`TextBuffer` iterators are generally invalidated when the text changes, but you can use a `Gtk::TextBuffer::Mark` to remember a position in these situations. For instance,

```
Glib::RefPtr<Gtk::TextBuffer::Mark> refMark =
```

```
    refBuffer->create_mark(iter);
```

You can then use the `get_iter()` method later to create an iterator for the `Mark`'s new position.

There are two built-in `Mark`s - `insert` and `select_bound`, which you can access with `TextBuffer`'s `get_insert()` and `get_selection_bound()` methods.

Reference

11.1.4. The View

As mentioned above, each `TextView` has a `TextBuffer`, and one or more `TextView` can share the same `TextBuffer`.

Like the `TreeView`, you should probably put your `TextView` inside a `ScrolledWindow` to allow the user to see and move around the whole text area with scrollbars.

Reference

11.1.4.1. Default formatting

`TextView` has various methods which allow you to change the presentation of the buffer for this particular view. Some of these may be overridden by the `Gtk::TextTag`s in the buffer, if they specify the same things. For instance, `set_left_margin()`, `set_right_margin()`, `set_indent()`, etc.

11.1.4.2. Scrolling

`Gtk::TextView` has various `scroll_to_*()` methods. These allow you to ensure that a particular part of the text buffer is visible. For instance, your application's Find feature might use `Gtk::TextView::scroll_to_iter()` to show the found text.

11.2. Widgets and ChildAnchors

You can embed widgets, such as `Gtk::Button`s, in the text. Each such child widget needs a `ChildAnchor`. ChildAnchors are associated with `iterators`. For instance, to create a child anchor at a particular position, use `Gtk::TextBuffer::create_child_anchor()`:

```
Glib::RefPtr<Gtk::TextChildAnchor> refAnchor =
    refBuffer->create_child_anchor(iter);
```

Then, to add a widget at that position, use `Gtk::TextView::add_child_at_anchor()`:

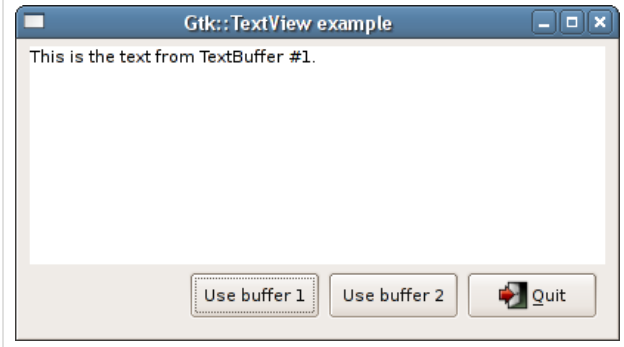
```
m_TextView.add_child_at_anchor(m_Button, refAnchor);
```

Reference

11.3. Examples

11.3.1. Simple Example

Figure 11-1 **TextView**



Source Code

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <gtkmm.h>

class ExampleWindow : public Gtk::Window
{
public:
  ExampleWindow();
  virtual ~ExampleWindow();

protected:

  void fill_buffers();

  //Signal handlers:
  void on_button_quit();
  void on_button_buffer1();
  void on_button_buffer2();

  //Child widgets:
  Gtk::Box m_VBox;

  Gtk::ScrolledWindow m_ScrolledWindow;
  Gtk::TextView m_TextView;

  Glib::RefPtr<Gtk::TextBuffer> m_refTextBuffer1, m_refTextBuffer2;

  Gtk::ButtonBox m_ButtonBox;
  Gtk::Button m_Button_Quit, m_Button_Buffer1, m_Button_Buffer2;
};

#endif //GTKMM_EXAMPLEWINDOW_H
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"

ExampleWindow::ExampleWindow()
: m_VBox(Gtk::ORIENTATION_VERTICAL),
  m_Button_Quit(Gtk::Stock::QUIT),
  m_Button_Buffer1("Use buffer 1"),
  m_Button_Buffer2("Use buffer 2")
{
  set_title("Gtk::TextView example");
  set_border_width(5);
  set_default_size(400, 200);


  add(m_VBox);

  //Add the TreeView, inside a ScrolledWindow, with the button underneath:
  m_ScrolledWindow.add(m_TextView);

  //Only show the scrollbars when they are necessary:
  m_ScrolledWindow.set_policy(Gtk::POLICY_AUTOMATIC, Gtk::POLICY_AUTOMATIC);

  m_VBox.pack_start(m_ScrolledWindow);

  //Add buttons:
```

```
  m_VBox.pack_start(m_ButtonBox, Gtk::PACK_SHRINK);

  m_ButtonBox.pack_start(m_Button_Buffer1, Gtk::PACK_SHRINK);
  m_ButtonBox.pack_start(m_Button_Buffer2, Gtk::PACK_SHRINK);
  m_ButtonBox.pack_start(m_Button_Quit, Gtk::PACK_SHRINK);
  m_ButtonBox.set_border_width(5);
  m_ButtonBox.set_spacing(5);
  m_ButtonBox.set_layout(Gtk::BUTTONBOX_END);

  //Connect signals:
  m_Button_Quit.signal_clicked().connect(sigc::mem_fun(*this,
              &ExampleWindow::on_button_quit) );
  m_Button_Buffer1.signal_clicked().connect(sigc::mem_fun(*this,
              &ExampleWindow::on_button_buffer1) );
  m_Button_Buffer2.signal_clicked().connect(sigc::mem_fun(*this,
              &ExampleWindow::on_button_buffer2) );

  fill_buffers();
  on_button_buffer1();

  show_all_children();
}

void ExampleWindow::fill_buffers()
{
  m_refTextBuffer1 = Gtk::TextBuffer::create();
  m_refTextBuffer1->set_text("This is the text from TextBuffer #1.");

  m_refTextBuffer2 = Gtk::TextBuffer::create();
  m_refTextBuffer2->set_text(
          "This is some alternative text, from TextBuffer #2.");

}

ExampleWindow::~ExampleWindow()
{
}

void ExampleWindow::on_button_quit()
{
  hide();
}

void ExampleWindow::on_button_buffer1()
{
  m_TextView.set_buffer(m_refTextBuffer1);
}

void ExampleWindow::on_button_buffer2()
{
  m_TextView.set_buffer(m_refTextBuffer2);
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

12. Menus and Toolbars

There are specific APIs for Menus and toolbars, but you should usually deal with them together, using the `UIManager` to define `Action`s which you can then arrange in menu and toolbars. In this way you can handle activation of the action instead of responding to the menu and toolbar items separately. And you can enable or disable both the menu and toolbar item via the action.

This involves the use of the `Gtk::ActionGroup`, `Gtk::Action`, and `UIManager` classes, all of which should be instantiated via their `create()` methods, which return `RefPtr`s.

12.1. Actions

First create the `Action`s and add them to an `ActionGroup`, with `ActionGroup::add()`.

The arguments to `Action::create()` specify the action's name and how it will appear in menus and toolbars. Use stock items where possible so that you don't need to specify the label, accelerator, icon, and tooltips, and so you can use pre-existing translations.

You can also specify a signal handler when calling `ActionGroup::add()`. This signal handler will be called when the action is activated via either a menu item or a toolbar button.

Note that you must specify actions for sub menus as well as menu items.

For instance:

```
m_refActionGroup = Gtk::ActionGroup::create();

m_refActionGroup->add( Gtk::Action::create("MenuFile", "_File") );
m_refActionGroup->add( Gtk::Action::create("New", Gtk::Stock::NEW),
  sigc::mem_fun(*this, &ExampleWindow::on_action_file_new) );
m_refActionGroup->add( Gtk::Action::create("ExportData", "Export Data"),
  sigc::mem_fun(*this, &ExampleWindow::on_action_file_open) );
m_refActionGroup->add( Gtk::Action::create("Quit", Gtk::Stock::QUIT),
  sigc::mem_fun(*this, &ExampleWindow::on_action_file_quit) );
```

Note that this is where we specify the names of the actions as they will be seen by users in menus and toolbars. Therefore, this is where you should make strings translatable, by putting them inside the _() macro. When we use the Gtk::Stock items, of course, translations are automatically available.

12.2. UIManager

Next you should create a `UIManager` and add the `ActionGroup` to the `UIManager` with `insert_action_group()` At this point is also a good idea to tell the parent window to respond to the specified keyboard shortcuts, by using `add_accel_group()`.

For instance,

```
Glib::RefPtr<Gtk::UIManager> m_refUIManager =
    Gtk::UIManager::create();
m_refUIManager->insert_action_group(m_refActionGroup);
add_accel_group(m_refUIManager->get_accel_group());
```

Then, you can define the actual visible layout of the menus and toolbars, and add the UI layout to the `UIManager`. This "ui string" uses an XML format, in which you should mention the names of the actions that you have already created. For instance:

```
Glib::ustring ui_info =
    "<ui>"
    "  <menubar name='MenuBar'>"
    "    <menu action='MenuFile'>"
    "      <menuitem action='New'/>"
    "      <menuitem action='Open'/>"
    "      <separator/>"
    "      <menuitem action='Quit'/>"
    "    </menu>"
    "    <menu action='MenuEdit'>"
    "      <menuitem action='Cut'/>"
    "      <menuitem action='Copy'/>"
    "      <menuitem action='Paste'/>"
    "    </menu>"
    "  </menubar>"
    "  <toolbar  name='ToolBar'>"
    "    <toolitem action='Open'/>"
    "    <toolitem action='Quit'/>"
```

```
    "  </toolbar>"
    "</ui>";

m_refUIManager->add_ui_from_string(ui_info);
```

Remember that these names are just the identifiers that we used when creating the actions. They are not the text that the user will see in the menus and toolbars. We provided those human-readable names when we created the actions.

To instantiate a `Gtk::MenuBar` or `Gtk::Toolbar` which you can actually show, you should use the `UIManager::get_widget()` method, and then add the widget to a container. For instance:

```
Gtk::Widget* pMenubar = m_refUIManager->get_widget("/MenuBar");
pBox->add(*pMenuBar, Gtk::PACK_SHRINK);
```

12.3. Popup Menus

`Menus` are normally just added to a window, but they can also be displayed temporarily as the result of a mouse button click. For instance, a context menu might be displayed when the user clicks their right mouse button.

The UI layout for a popup menu should use the `popup` node. For instance:

```
Glib::ustring ui_info =
    "<ui>"
    "  <popup name='PopupMenu'>"
    "    <menuitem action='ContextEdit'/>"
    "    <menuitem action='ContextProcess'/>"
    "    <menuitem action='ContextRemove'/>"
    "  </popup>"
    "</ui>";

m_refUIManager->add_ui_from_string(ui_info);
```

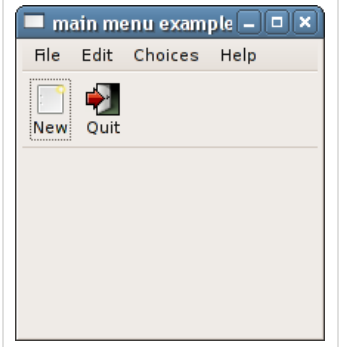
To show the popup menu, use `Gtk::Menu`'s `popup()` method, providing the button identifier and the time of activation, as provided by the `button_press_event` signal, which you will need to handle anyway. For instance:

```
bool ExampleWindow::on_button_press_event(GdkEventButton* event)
{
  if( (event->type == GDK_BUTTON_PRESS) &&
      (event->button == 3) )
  {
    m_Menu_Popup->popup(event->button, event->time);
    return true; //It has been handled.
  }
  else
    return false;
}
```

12.4. Examples

12.4.1. Main Menu example

Figure 12-1 **Main Menu**



Source Code

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <gtkmm.h>

class ExampleWindow : public Gtk::Window
{
public:
  ExampleWindow();
  virtual ~ExampleWindow();

protected:
  //Signal handlers:
  void on_menu_file_new_generic();
  void on_menu_file_quit();
  void on_menu_others();

  void on_menu_choices_one();
  void on_menu_choices_two();

  //Child widgets:
  Gtk::Box m_Box;

  Glib::RefPtr<Gtk::UIManager> m_refUIManager;
  Glib::RefPtr<Gtk::ActionGroup> m_refActionGroup;
  Glib::RefPtr<Gtk::RadioAction> m_refChoiceOne, m_refChoiceTwo;
};

#endif //GTKMM_EXAMPLEWINDOW_H
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/stock.h>
#include <iostream>

ExampleWindow::ExampleWindow()
: m_Box(Gtk::ORIENTATION_VERTICAL)
{
  set_title("main menu example");
  set_default_size(200, 200);

  add(m_Box); // put a MenuBar at the top of the box and other stuff below it.

  //Create actions for menus and toolbars:
  m_refActionGroup = Gtk::ActionGroup::create();

  //File|New sub menu:
  m_refActionGroup->add(Gtk::Action::create("FileNewStandard",
              Gtk::Stock::NEW, "_New", "Create a new file"),
          sigc::mem_fun(*this, &ExampleWindow::on_menu_file_new_generic));

  m_refActionGroup->add(Gtk::Action::create("FileNewFoo",
              Gtk::Stock::NEW, "New Foo", "Create a new foo"),
          sigc::mem_fun(*this, &ExampleWindow::on_menu_file_new_generic));

  m_refActionGroup->add(Gtk::Action::create("FileNewGoo",
              Gtk::Stock::NEW, "_New Goo", "Create a new goo"),
          sigc::mem_fun(*this, &ExampleWindow::on_menu_file_new_generic));
```

```
  //File menu:
  m_refActionGroup->add(Gtk::Action::create("FileMenu", "File"));
  //Sub-menu.
  m_refActionGroup->add(Gtk::Action::create("FileNew", Gtk::Stock::NEW));
  m_refActionGroup->add(Gtk::Action::create("FileQuit", Gtk::Stock::QUIT),
          sigc::mem_fun(*this, &ExampleWindow::on_menu_file_quit));

  //Edit menu:
  m_refActionGroup->add(Gtk::Action::create("EditMenu", "Edit"));
  m_refActionGroup->add(Gtk::Action::create("EditCopy", Gtk::Stock::COPY),
          sigc::mem_fun(*this, &ExampleWindow::on_menu_others));
  m_refActionGroup->add(Gtk::Action::create("EditPaste", Gtk::Stock::PASTE),
          sigc::mem_fun(*this, &ExampleWindow::on_menu_others));
  m_refActionGroup->add(Gtk::Action::create("EditSomething", "Something"),
          Gtk::AccelKey("<control><alt>S"),
          sigc::mem_fun(*this, &ExampleWindow::on_menu_others));


  //Choices menu, to demonstrate Radio items
  m_refActionGroup->add( Gtk::Action::create("ChoicesMenu", "Choices") );
  Gtk::RadioAction::Group group_userlevel;
  m_refChoiceOne = Gtk::RadioAction::create(group_userlevel, "ChoiceOne", "One");
  m_refActionGroup->add(m_refChoiceOne,
          sigc::mem_fun(*this, &ExampleWindow::on_menu_choices_one) );
  m_refChoiceTwo = Gtk::RadioAction::create(group_userlevel, "ChoiceTwo", "Two");
  m_refActionGroup->add(m_refChoiceTwo,
          sigc::mem_fun(*this, &ExampleWindow::on_menu_choices_two) );

  //Help menu:
  m_refActionGroup->add( Gtk::Action::create("HelpMenu", "Help") );
  m_refActionGroup->add( Gtk::Action::create("HelpAbout", Gtk::Stock::HELP),
          sigc::mem_fun(*this, &ExampleWindow::on_menu_others) );

  m_refUIManager = Gtk::UIManager::create();
  m_refUIManager->insert_action_group(m_refActionGroup);

  add_accel_group(m_refUIManager->get_accel_group());

  //Layout the actions in a menubar and toolbar:
  Glib::ustring ui_info =
        "<ui>"
        "  <menubar name='MenuBar'>"
        "    <menu action='FileMenu'>"
        "      <menu action='FileNew'>"
        "        <menuitem action='FileNewStandard'/>"
        "        <menuitem action='FileNewFoo'/>"
        "        <menuitem action='FileNewGoo'/>"
        "      </menu>"
        "      <separator/>"
        "      <menuitem action='FileQuit'/>"
        "    </menu>"
        "    <menu action='EditMenu'>"
        "      <menuitem action='EditCopy'/>"
        "      <menuitem action='EditPaste'/>"
        "      <menuitem action='EditSomething'/>"
        "    </menu>"
        "    <menu action='ChoicesMenu'>"
        "      <menuitem action='ChoiceOne'/>"
        "      <menuitem action='ChoiceTwo'/>"
        "    </menu>"
        "    <menu action='HelpMenu'>"
        "      <menuitem action='HelpAbout'/>"
        "    </menu>"
        "  </menubar>"
        "  <toolbar  name='ToolBar'>"
        "    <toolitem action='FileNewStandard'/>"
        "    <toolitem action='FileQuit'/>"
        "  </toolbar>"
        "</ui>";

  try
  {
    m_refUIManager->add_ui_from_string(ui_info);
  }
  catch(const Glib::Error& ex)
  {
    std::cerr << "building menus failed: " <<  ex.what();
  }

  //Get the menubar and toolbar widgets, and add them to a container widget:
  Gtk::Widget* pMenubar = m_refUIManager->get_widget("/MenuBar");
  if(pMenubar)
    m_Box.pack_start(*pMenubar, Gtk::PACK_SHRINK);

  Gtk::Widget* pToolbar = m_refUIManager->get_widget("/ToolBar") ;
  if(pToolbar)
    m_Box.pack_start(*pToolbar, Gtk::PACK_SHRINK);
```

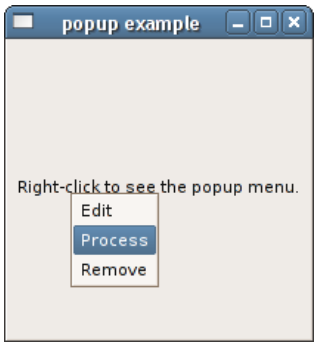
```
  show_all_children();
}

ExampleWindow::~ExampleWindow()
{
}

void ExampleWindow::on_menu_file_quit()
{
  hide(); //Closes the main window to stop the Gtk::Main::run().
}

void ExampleWindow::on_menu_file_new_generic()
{
   std::cout << "A File|New menu item was selected." << std::endl;
}

void ExampleWindow::on_menu_others()
{
  std::cout << "A menu item was selected." << std::endl;
}

void ExampleWindow::on_menu_choices_one()
{
  Glib::ustring message;
  if(m_refChoiceOne->get_active())
    message = "Choice 1 was selected.";
  else
    message = "Choice 1 was deselected";

  std::cout << message << std::endl;
}

void ExampleWindow::on_menu_choices_two()
{
  Glib::ustring message;
  if(m_refChoiceTwo->get_active())
    message = "Choice 2 was selected.";
  else
    message = "Choice 2 was deselected";

  std::cout << message << std::endl;
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

12.4.2. Popup Menu example

Figure 12-2 **Popup Menu**



Source Code

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <gtkmm.h>

class ExampleWindow : public Gtk::Window
{
public:
  ExampleWindow();
  virtual ~ExampleWindow();

protected:
  //Signal handlers:
  virtual bool on_button_press_event(GdkEventButton* event);
  void on_menu_file_popup_generic();

  //Child widgets:
  Gtk::Box m_Box;
  Gtk::EventBox m_EventBox;
  Gtk::Label m_Label;

  Glib::RefPtr<Gtk::UIManager> m_refUIManager;
  Glib::RefPtr<Gtk::ActionGroup> m_refActionGroup;

  Gtk::Menu* m_pMenuPopup;
};

#endif //GTKMM_EXAMPLEWINDOW_H
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/stock.h>
#include <iostream>

ExampleWindow::ExampleWindow()
: m_Box(Gtk::ORIENTATION_VERTICAL),
  m_Label("Right-click to see the popup menu."),
  m_pMenuPopup(0)
  /* m_Image(Gtk::Stock::DIALOG_QUESTION, Gtk::ICON_SIZE_MENU) */
{
  set_title("popup example");
  set_default_size(200, 200);

  add(m_Box);

  //Add an event box that can catch button_press events:
  m_Box.pack_start(m_EventBox);
  m_EventBox.signal_button_press_event().connect(sigc::mem_fun(*this,
              &ExampleWindow::on_button_press_event) );

  m_EventBox.add(m_Label);

  //Create actions:

  //Fill menu:

  m_refActionGroup = Gtk::ActionGroup::create();

  //File|New sub menu:
  //These menu actions would normally already exist for a main menu, because a
  //context menu should not normally contain menu items that are only available
  //via a context menu.
  m_refActionGroup->add(Gtk::Action::create("ContextMenu", "Context Menu"));

  m_refActionGroup->add(Gtk::Action::create("ContextEdit", "Edit"),
          sigc::mem_fun(*this, &ExampleWindow::on_menu_file_popup_generic));

  m_refActionGroup->add(Gtk::Action::create("ContextProcess", "Process"),
          Gtk::AccelKey("<control>P"),
          sigc::mem_fun(*this, &ExampleWindow::on_menu_file_popup_generic));

  m_refActionGroup->add(Gtk::Action::create("ContextRemove", "Remove"),
          sigc::mem_fun(*this, &ExampleWindow::on_menu_file_popup_generic));

  //TODO:
  /*
    //Add a ImageMenuElem:
    menulist.push_back( Gtk::Menu_Helpers::ImageMenuElem("_Something", m_Image,
```

```
      sigc::mem_fun(*this, &ExampleWindow::on_menu_file_popup_generic) ) ) ;
  */

  m_refUIManager = Gtk::UIManager::create();
  m_refUIManager->insert_action_group(m_refActionGroup);

  add_accel_group(m_refUIManager->get_accel_group());

  //Layout the actions in a menubar and toolbar:
  Glib::ustring ui_info =
        "<ui>"
        "  <popup name='PopupMenu'>"
        "    <menuitem action='ContextEdit'/>"
        "    <menuitem action='ContextProcess'/>"
        "    <menuitem action='ContextRemove'/>"
        "  </popup>"
        "</ui>";

  try
  {
    m_refUIManager->add_ui_from_string(ui_info);
  }
  catch(const Glib::Error& ex)
  {
    std::cerr << "building menus failed: " <<  ex.what();
  }

  //Get the menu:
  m_pMenuPopup = dynamic_cast<Gtk::Menu*>(
          m_refUIManager->get_widget("/PopupMenu"));
  if(!m_pMenuPopup)
    g_warning("menu not found");

  show_all_children();
}

ExampleWindow::~ExampleWindow()
{
}

void ExampleWindow::on_menu_file_popup_generic()
{
   std::cout << "A popup menu item was selected." << std::endl;
}

bool ExampleWindow::on_button_press_event(GdkEventButton* event)
{
  if( (event->type == GDK_BUTTON_PRESS) && (event->button == 3) )
  {
    if(m_pMenuPopup)
      m_pMenuPopup->popup(event->button, event->time);

    return true; //It has been handled.
  }
  else
    return false;
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

13. ToolPalette

A `ToolPalette` is similar to a `Toolbar` but can contain a grid of items, categorized into groups. The user may hide or expand each group. As in a toolbar, the items may be displayed as only icons, as only text, or as icons with text.

The `ToolPalette`'s items might be dragged or simply activated. For instance, the user

might drag objects to a canvas to create new items there. Or the user might click an item to activate a certain brush size in a drawing application.

`ToolItemGroup`s should be added to the tool pallete via the base class's `Gtk::Container::add()` method, for instance like so:

```
Gtk::ToolItemGroup* group_brushes =
  Gtk::manage(new Gtk::ToolItemGroup("Brushes"));
m_ToolPalette.add(*group_brushes);
```

`Gtk::ToolItem`s can then be added to the group. For instance, like so:

```
Gtk::ToolButton* button = Gtk::manage(new Gtk::ToolButton(icon, "Big"));
button->set_tooltip_text("Big Brush);
group_brushes->insert(*button);
```

You might then handle the `ToolButton`'s `clicked` signal. Alternatively, you could allow the item to be dragged to another widget, by calling `Gtk::ToolPalette::add_drag_dest()` and then using `Gtk::ToolPalette::get_drag_item()` in the other widget's `drag_data_received` signal handler.

ToolPalette Reference

ToolItemGroup Reference

ToolItem Reference

13.1. Drag and Drop

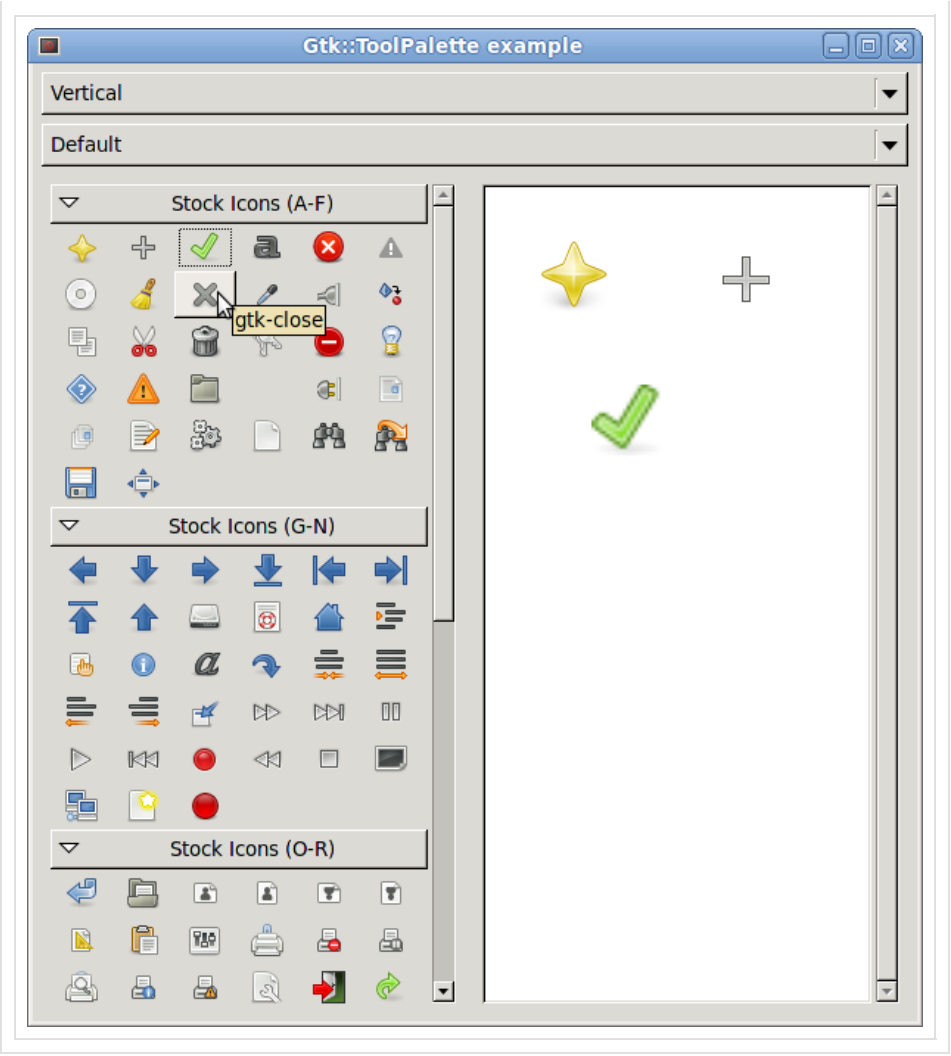
Call `add_drag_dest()` to allow items or groups to be dragged from the tool palette to a particular destination widget. You can then use `get_drag_item()` to discover which ToolItem or ToolItemGroup is being dragged. You can use `dynamic_cast` to discover whether is it an item or a group. For instance, you might use this in your `drag_data_received` signal handler, to add a dropped item, or to show a suitable icon while dragging.

See the Drag and Drop chapter for general advice about Drag and Drop with gtkmm.

13.2. ToolPalette Example

This example adds a `ToolPalette` and a `DrawingArea` to a window and allows the user to drag icons from the tool palette to the drawing area. The tool palette contains several groups of items. The combo boxes allow the user to change the style and orientation of the tool palette.

Figure 13-1 **ToolPalette**



Source Code

File: `canvas.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLE_CANVAS_H
#define GTKMM_EXAMPLE_CANVAS_H

#include <gtkmm.h>

// This little canvas class is only here
// because gtkmm does not have a canvas class yet.
// Applications should probably use GooCanvas::Canvas (goocanvasmm) instead.
class Canvas : public Gtk::DrawingArea
{
public:
  Canvas();
  virtual ~Canvas();

private:

  class CanvasItem
  {
  public:
    CanvasItem(Gtk::Widget* canvas, Gtk::ToolButton* button, double x, double y)
    {
      const Gtk::StockID stock_id(button->get_stock_id());
      this->pixbuf = canvas->render_icon_pixbuf(stock_id, Gtk::ICON_SIZE_DIALOG);
      this->x = x;
      this->y = y;
    }

    Glib::RefPtr<Gdk::Pixbuf> pixbuf;
    double x, y;
  };

  void item_draw(const CanvasItem *item,
```

```
    const Cairo::RefPtr<Cairo::Context>& cr,
    bool preview);

  virtual bool on_draw(const Cairo::RefPtr<Cairo::Context>& cr);
  virtual void on_drag_data_received(const Glib::RefPtr<Gdk::DragContext>& context,
    int x, int y, const Gtk::SelectionData& selection_data, guint info, guint time);
  virtual bool on_drag_motion(const Glib::RefPtr<Gdk::DragContext>& context, int x, int y, guint time);
  virtual bool on_drag_drop(const Glib::RefPtr<Gdk::DragContext>& context, int x, int y, guint time);
  virtual void on_drag_leave(const Glib::RefPtr<Gdk::DragContext>& context, guint time);

  bool m_drag_data_requested_for_drop; //So we know what to do in on_drag_data_received().
  CanvasItem* m_drop_item;

  typedef std::vector<CanvasItem*> type_vec_items;
  type_vec_items m_canvas_items;
};

#endif //GTKMM_EXAMPLE_CANVAS_H
```

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <gtkmm.h>
#include "canvas.h"

class ExampleWindow : public Gtk::Window
{
public:
  ExampleWindow();
  virtual ~ExampleWindow();

private:

  void load_stock_items();
  void load_toggle_items();
  void load_special_items();

  //Signal handlers:
  void on_combo_orientation_changed();
  void on_combo_style_changed();

  //Tree model columns:
  class ModelColumnsOrientation : public Gtk::TreeModel::ColumnRecord
  {
  public:

    ModelColumnsOrientation()
    { add(m_col_value); add(m_col_name); }

    Gtk::TreeModelColumn<Gtk::Orientation> m_col_value;
    Gtk::TreeModelColumn<Glib::ustring> m_col_name;
  };

  ModelColumnsOrientation m_ColumnsOrientation;

  //Tree model columns:
  class ModelColumnsStyle : public Gtk::TreeModel::ColumnRecord
  {
  public:

    ModelColumnsStyle()
    { add(m_col_value); add(m_col_name); }

    Gtk::TreeModelColumn<int> m_col_value; //We use int to also allow -1
    Gtk::TreeModelColumn<Glib::ustring> m_col_name;
  };

  ModelColumnsStyle m_ColumnsStyle;


  //Child widgets:
  Gtk::Box m_VBox;
  Gtk::Box m_HBox;
  Gtk::ComboBox m_ComboOrientation;
  Glib::RefPtr<Gtk::ListStore> m_refTreeModelOrientation;
  Gtk::ComboBox m_ComboStyle;
  Glib::RefPtr<Gtk::ListStore> m_refTreeModelStyle;
  Gtk::ToolPalette m_ToolPalette;
  Gtk::ScrolledWindow m_ScrolledWindowPalette;
  Gtk::ScrolledWindow m_ScrolledWindowCanvas;
  Canvas m_Canvas;
};
```

```
#endif //GTKMM_EXAMPLEWINDOW_H
```

File: examplewindow.cc (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"

static bool sort_predicate(const Gtk::StockID& a, const Gtk::StockID& b)
{
  return a.get_string() < b.get_string();
}


void ExampleWindow::load_stock_items()
{
  Gtk::ToolItemGroup* group_af =
    Gtk::manage(new Gtk::ToolItemGroup("Stock Icons (A-F)"));
  m_ToolPalette.add(*group_af);
  Gtk::ToolItemGroup* group_gn =
    Gtk::manage(new Gtk::ToolItemGroup("Stock Icons (G-N)"));
  m_ToolPalette.add(*group_gn);
  Gtk::ToolItemGroup* group_or =
    Gtk::manage(new Gtk::ToolItemGroup("Stock Icons (O-R)"));
  m_ToolPalette.add(*group_or);
  Gtk::ToolItemGroup* group_sz =
    Gtk::manage(new Gtk::ToolItemGroup("Stock Icons (S-Z)"));
  m_ToolPalette.add(*group_sz);

  // Obtain the IDs of all stock items:
  typedef std::vector<Gtk::StockID> type_vecIDs;
  type_vecIDs vecIDs = Gtk::Stock::get_ids();

  std::sort(vecIDs.begin(), vecIDs.end(), &sort_predicate);

  Gtk::ToolItemGroup* group = 0;

  // Iterate through them, populating the ListStore as appropriate:
  for(type_vecIDs::const_iterator iterIDs = vecIDs.begin(); iterIDs != vecIDs.end(); ++iterIDs)
  {
    const Gtk::StockID& stockid = *iterIDs;
    const Glib::ustring str = stockid.get_string();
    if(str.size() < 4)
      continue;

    switch(str[4])
    {
      case 'a':
        group = group_af;
        break;
      case 'g':
        group = group_gn;
        break;
      case 'o':
        group = group_or;
        break;
      case 's':
        group = group_sz;
        break;
      default:
        //Use the previous group
        //(They are sorted.)
        break;
    }

    if(!group)
      continue;

    Gtk::ToolButton* button = Gtk::manage(new Gtk::ToolButton(stockid));
    button->set_tooltip_text(str);
    button->set_is_important();
    group->insert(*button);

    Gtk::StockItem stockitem;
    if(!Gtk::StockItem::lookup(stockid, stockitem) ||
      stockitem.get_label().empty())
        button->set_label(str);
  }
}


void ExampleWindow::load_toggle_items()
{
  Gtk::ToolItemGroup* group =
    Gtk::manage(new Gtk::ToolItemGroup("Radio Item"));
  m_ToolPalette.add(*group);
```

```
  Gtk::RadioToolButton::Group radio_group;

  for(int i = 1; i <= 10; ++i)
  {
    const Glib::ustring label = Glib::ustring::compose("#%1", i);
    Gtk::RadioToolButton* button = Gtk::manage(new Gtk::RadioToolButton());
    button->set_group(radio_group);
    button->set_label(label);

    group->insert(*button);
  }
}


static Gtk::ToolItem* create_entry_item(const Glib::ustring& text)
{
  Gtk::Entry* entry = Gtk::manage(new Gtk::Entry());
  entry->set_text(text);
  entry->set_width_chars(5);

  Gtk::ToolItem* item = Gtk::manage(new Gtk::ToolItem());
  item->add(*entry);

  return item;
}

void ExampleWindow::load_special_items()
{
  Gtk::ToolItemGroup* group = Gtk::manage(new Gtk::ToolItemGroup());

  Gtk::Button *label_button = Gtk::manage(new Gtk::Button("Advanced Features"));
  label_button->show();
  group->set_label_widget(*label_button);
  m_ToolPalette.add(*group);

  Gtk::ToolItem* item = create_entry_item ("homogeneous=false");
  group->insert(*item);
  //TODO: Add Gtk::Container::set_child_property().
  gtk_container_child_set (GTK_CONTAINER (group->gobj()), GTK_WIDGET (item->gobj()),
                           "homogeneous", FALSE, NULL);

  item = create_entry_item ("homogeneous=FALSE, expand=TRUE");
  group->insert(*item);
  gtk_container_child_set (GTK_CONTAINER (group->gobj()), GTK_WIDGET (item->gobj()),
                           "homogeneous", FALSE, "expand", TRUE,
                           NULL);

  item = create_entry_item ("homogeneous=FALSE, expand=TRUE, fill=FALSE");
  group->insert(*item);
  gtk_container_child_set (GTK_CONTAINER (group->gobj()), GTK_WIDGET (item->gobj()),
                           "homogeneous", FALSE, "expand", TRUE,
                           "fill", FALSE, NULL);

  item = create_entry_item ("homogeneous=FALSE, expand=TRUE, new-row=TRUE");
  group->insert(*item);
  gtk_container_child_set (GTK_CONTAINER (group->gobj()), GTK_WIDGET (item->gobj()),
                           "homogeneous", FALSE, "expand", TRUE,
                           "new-row", TRUE, NULL);

  item = Gtk::manage(new Gtk::ToolButton(Gtk::Stock::GO_UP));
  item->set_tooltip_text("Show on vertical palettes only");
  group->insert(*item);
  item->set_visible_horizontal(false);

  item = Gtk::manage(new Gtk::ToolButton(Gtk::Stock::GO_FORWARD));
  item->set_tooltip_text("Show on horizontal palettes only");
  group->insert(*item);
  item->set_visible_vertical(false);

  item = Gtk::manage(new Gtk::ToolButton(Gtk::Stock::FULLSCREEN));
  item->set_tooltip_text("Expanded this item");
  group->insert(*item);
  gtk_container_child_set (GTK_CONTAINER (group->gobj()), GTK_WIDGET (item->gobj()),
                           "homogeneous", FALSE,
                           "expand", TRUE,
                           NULL);

  item = Gtk::manage(new Gtk::ToolButton(Gtk::Stock::HELP));
  item->set_tooltip_text("A regular item");
  group->insert(*item);
}

ExampleWindow::ExampleWindow()
: m_VBox(Gtk::ORIENTATION_VERTICAL, 6),
  m_HBox(Gtk::ORIENTATION_HORIZONTAL, 6)
{
  set_title("Gtk::ToolPalette example");
  set_size_request(600, 600);
```

```
  set_border_width(6);

  add(m_VBox);

  //The Orientation ComboBox:
  m_refTreeModelOrientation = Gtk::ListStore::create(m_ColumnsOrientation);
  Gtk::TreeModel::Row row = *(m_refTreeModelOrientation->append());
  row[m_ColumnsOrientation.m_col_value] = Gtk::ORIENTATION_HORIZONTAL;
  row[m_ColumnsOrientation.m_col_name] = "Horizontal";\
  row = *(m_refTreeModelOrientation->append());
  row[m_ColumnsOrientation.m_col_value] = Gtk::ORIENTATION_VERTICAL;
  row[m_ColumnsOrientation.m_col_name] = "Vertical";
  m_ComboOrientation.set_model(m_refTreeModelOrientation);
  m_VBox.pack_start(m_ComboOrientation, Gtk::PACK_SHRINK);
  m_ComboOrientation.pack_start(m_ColumnsOrientation.m_col_name);
  m_ComboOrientation.signal_changed().connect(
    sigc::mem_fun(*this, &ExampleWindow::on_combo_orientation_changed) );
  m_ComboOrientation.set_active(row);

  //The Style ComboBox:
  m_refTreeModelStyle = Gtk::ListStore::create(m_ColumnsStyle);
  row = *(m_refTreeModelStyle->append());
  row[m_ColumnsStyle.m_col_value] = Gtk::TOOLBAR_TEXT;
  row[m_ColumnsStyle.m_col_name] = "Text";\
  row = *(m_refTreeModelStyle->append());
  row[m_ColumnsStyle.m_col_value] = Gtk::TOOLBAR_BOTH;
  row[m_ColumnsStyle.m_col_name] = "Both";
  row = *(m_refTreeModelStyle->append());
  row[m_ColumnsStyle.m_col_value] = Gtk::TOOLBAR_BOTH_HORIZ;
  row[m_ColumnsStyle.m_col_name] = "Both: Horizontal";
  row = *(m_refTreeModelStyle->append());
  row[m_ColumnsStyle.m_col_value] = Gtk::TOOLBAR_ICONS;
  row[m_ColumnsStyle.m_col_name] = "Icons";
  row = *(m_refTreeModelStyle->append());
  row[m_ColumnsStyle.m_col_value] = -1; // A custom meaning for this demo.
  row[m_ColumnsStyle.m_col_name] = "Default";
  m_ComboStyle.set_model(m_refTreeModelStyle);
  m_VBox.pack_start(m_ComboStyle, Gtk::PACK_SHRINK);
  m_ComboStyle.pack_start(m_ColumnsStyle.m_col_name);
  m_ComboStyle.signal_changed().connect(
    sigc::mem_fun(*this, &ExampleWindow::on_combo_style_changed) );
  m_ComboStyle.set_active(row);

  //Add and fill the ToolPalette:
  load_stock_items();
  load_toggle_items();
  load_special_items();

  m_VBox.pack_start(m_HBox, Gtk::PACK_EXPAND_WIDGET);

  m_ScrolledWindowPalette.set_policy(Gtk::POLICY_NEVER, Gtk::POLICY_AUTOMATIC);
  m_ScrolledWindowPalette.set_border_width(6);
  m_ScrolledWindowPalette.add(m_ToolPalette);
  m_HBox.pack_start(m_ScrolledWindowPalette);

  on_combo_orientation_changed();

  m_ScrolledWindowCanvas.set_policy(Gtk::POLICY_AUTOMATIC, Gtk::POLICY_ALWAYS);
  m_ScrolledWindowCanvas.set_border_width(6);
  m_ScrolledWindowCanvas.add(m_Canvas);
  m_ScrolledWindowCanvas.set_size_request(200, -1);
  m_HBox.pack_start(m_ScrolledWindowCanvas);

  m_ToolPalette.add_drag_dest(m_Canvas,
    Gtk::DEST_DEFAULT_HIGHLIGHT, Gtk::TOOL_PALETTE_DRAG_ITEMS, Gdk::ACTION_COPY);

  show_all_children();
}

ExampleWindow::~ExampleWindow()
{
}

void ExampleWindow::on_combo_orientation_changed()
{
  Gtk::TreeModel::iterator iter = m_ComboOrientation.get_active();
  if(!iter)
    return;

  Gtk::TreeModel::Row row = *iter;
  const Gtk::Orientation value = row[m_ColumnsOrientation.m_col_value];

  m_ToolPalette.set_orientation(value);

  if(value == Gtk::ORIENTATION_HORIZONTAL)
    m_ScrolledWindowPalette.set_policy(Gtk::POLICY_AUTOMATIC, Gtk::POLICY_NEVER);
  else
    m_ScrolledWindowPalette.set_policy(Gtk::POLICY_NEVER, Gtk::POLICY_AUTOMATIC);
```

```
}

void ExampleWindow::on_combo_style_changed()
{
  Gtk::TreeModel::iterator iter = m_ComboStyle.get_active();
  if(!iter)
    return;

  Gtk::TreeModel::Row row = *iter;
  const int value = row[m_ColumnsStyle.m_col_value];

  if(value == -1)
    m_ToolPalette.unset_style();
  else
    m_ToolPalette.set_style((Gtk::ToolbarStyle)value);
}
```

File: `canvas.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "canvas.h"
#include <iostream>

Canvas::Canvas()
: m_drag_data_requested_for_drop(false),
  m_drop_item()
{
  set_app_paintable();
}

Canvas::~Canvas()
{
  while(!m_canvas_items.empty())
  {
    type_vec_items::iterator iter = m_canvas_items.begin();
    CanvasItem* item = *iter;
    delete item;
    m_canvas_items.erase(iter);
  }

  if(m_drop_item)
    delete m_drop_item;
}

void Canvas::item_draw(const CanvasItem *item,
  const Cairo::RefPtr<Cairo::Context>& cr,
  bool preview)
{
  if(!item || !item->pixbuf)
    return;

  const double cx = item->pixbuf->get_width();
  const double cy = item->pixbuf->get_height();

  Gdk::Cairo::set_source_pixbuf(cr,
    item->pixbuf,
    item->x - cx * 0.5, item->y - cy * 0.5);

  if(preview)
    cr->paint_with_alpha(0.6);
  else
    cr->paint();
}

bool Canvas::on_draw(const Cairo::RefPtr<Cairo::Context>& cr)
{
  cr->set_source_rgb(1.0, 1.0, 1.0);
  const Gtk::Allocation allocation = get_allocation();
  cr->rectangle(0, 0, allocation.get_width(), allocation.get_height());
  cr->fill();

  for(type_vec_items::iterator iter = m_canvas_items.begin();
    iter != m_canvas_items.end(); ++iter )
  {
    item_draw(*iter, cr, false);
  }

  if(m_drop_item)
    item_draw (m_drop_item, cr, true);

  return true;
}


bool Canvas::on_drag_motion(const Glib::RefPtr<Gdk::DragContext>& context,
```

```
  int x, int y, guint time)
{
  m_drag_data_requested_for_drop = false; //It's for drag-motion instead.

  if(m_drop_item)
  {
    // We already have a drop indicator so just update its position.

    m_drop_item->x = x;
    m_drop_item->y = y;

    queue_draw();
    context->drag_status(Gdk::ACTION_COPY, time);
  }
  else
  {
    // Request DnD data for creating a drop indicator.
    // This will cause on_drag_data_received() to be called.
    const Glib::ustring target = drag_dest_find_target(context);

    if (target.empty())
      return false;

    drag_get_data(context, target, time);
  }

  Gtk::DrawingArea::on_drag_motion(context, x, y, time);
  return true;
}


void Canvas::on_drag_data_received(const Glib::RefPtr<Gdk::DragContext>& context, int x, int y, const Gtk::SelectionData& selectio
{
  // Find the tool button which is the source of this DnD operation.
  Gtk::Widget* widget = drag_get_source_widget(context);

  Gtk::ToolPalette* drag_palette = dynamic_cast<Gtk::ToolPalette*>(widget);
  while(widget && !drag_palette)
  {
    widget = widget->get_parent();
    drag_palette = dynamic_cast<Gtk::ToolPalette*>(widget);
  }

  Gtk::ToolItem* drag_item = 0;
  if(drag_palette)
    drag_item = drag_palette->get_drag_item(selection_data);

  // Create a drop indicator when a tool button was found:
  Gtk::ToolButton* button = dynamic_cast<Gtk::ToolButton*>(drag_item);
  if(!button)
    return;

  if(m_drop_item)
  {
    delete m_drop_item;
    m_drop_item = 0;
  }

  CanvasItem* item = new CanvasItem(this, button, x, y);

  if(m_drag_data_requested_for_drop)
  {
    m_canvas_items.push_back(item);

    // Signal that the item was accepted and then redraw.
    context->drag_finish(true /* success */, false /* del */, time);
  }
  else
  {
    m_drop_item = item;

    // We are getting this data due to a request in drag_motion,
    // rather than due to a request in drag_drop, so we are just
    // supposed to call gdk_drag_status (), not actually paste in
    // the data.
    context->drag_status(Gdk::ACTION_COPY, time);
  }

  queue_draw();

  Gtk::DrawingArea::on_drag_data_received(context, x, y, selection_data, info, time);
}


bool Canvas::on_drag_drop(const Glib::RefPtr<Gdk::DragContext>& context, int /* x */, int /* y */, guint time)
{
  // Request DnD data for creating a dopped item.
  // This will cause on_drag_data_received() to be called.
```

```
  const Glib::ustring target = drag_dest_find_target(context);

  if (target.empty())
    return false;

  m_drag_data_requested_for_drop = true;
  drag_get_data(context, target, time);

  return true;
}

void Canvas::on_drag_leave(const Glib::RefPtr<Gdk::DragContext>& context, guint time)
{
  //This signal is emitted to clean up the item used for drag-motion,
  //either when the cursor moves out of the widget or when we drop.

  if(!m_drop_item)
    return;

  delete m_drop_item;
  m_drop_item = 0;

  queue_draw();

  Gtk::DrawingArea::on_drag_leave(context, time);
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

14. Adjustments

gtkmm has various widgets that can be visually adjusted using the mouse or the keyboard, such as the `Range` widgets (described in the Range Widgets section). There are also a few widgets that display some adjustable part of a larger area, such as the `Viewport` widget. These widgets have `Gtk::Adjustment` objects that express this common part of their API.

So that applications can react to changes, for instance when a user moves a scrollbar, `Gtk::Adjustment` has a `changed` signal. You can then use the `get_changed()` method to discover the new value.

14.1. Creating an Adjustment

The `Gtk::Adjustment` constructor is as follows:

```
Gtk::Adjustment(float value,
                float lower,
                float upper,
                float step_increment = 1,
                float page_increment = 10,
                float page_size = 0);
```

The `value` argument is the initial value of the adjustment, usually corresponding to the topmost or leftmost position of an adjustable widget. The `lower` and `upper` arguments specifies the possible range of values which the adjustment can hold. The

step_increment argument specifies the smaller of the two increments by which the user can change the value, while the page_increment is the larger one. The page_size argument usually corresponds somehow to the visible area of a panning widget. The upper argument is used to represent the bottom most or right most coordinate in a panning widget's child.

14.2. Using Adjustments the Easy Way

The adjustable widgets can be roughly divided into those which use and require specific units for these values, and those which treat them as arbitrary numbers.

The group which treats the values as arbitrary numbers includes the Range widgets (Scrollbars and Scales, the Progressbar widget, and the SpinButton widget). These widgets are typically "adjusted" directly by the user with the mouse or keyboard. They will treat the lower and upper values of an adjustment as a range within which the user can manipulate the adjustment's value. By default, they will only modify the value of an adjustment.

The other group includes the Viewport widget and the ScrolledWindow widget. All of these widgets use pixel values for their adjustments. These are also typically adjusted indirectly using scrollbars. While all widgets which use adjustments can either create their own adjustments or use ones you supply, you'll generally want to let this particular category of widgets create its own adjustments.

If you share an adjustment object between a Scrollbar and a TextView widget, manipulating the scrollbar will automagically adjust the TextView widget. You can set it up like this:

```
// creates its own adjustments
Gtk::TextView textview;
// uses the newly-created adjustment for the scrollbar as well
Gtk::VScrollbar vscrollbar (*(textview.get_vadjustment()));
```

14.3. Adjustment Internals

OK, you say, that's nice, but what if I want to create my own handlers to respond when the user adjusts a Range widget or a SpinButton. To access the value of a Gtk::Adjustment, you can use the get_value() and set_value() methods:

As mentioned earlier, Gtk::Adjustment can emit signals. This is, of course, how updates happen automatically when you share an Adjustment object between a Scrollbar and another adjustable widget; all adjustable widgets connect signal handlers to their adjustment's value_changed signal, as can your program.

So, for example, if you have a Scale widget, and you want to change the rotation of a picture whenever its value changes, you would create a signal handler like this:

```
void cb_rotate_picture (Gtk::Widget *picture)
{
  picture->set_rotation (adj->value);
...
```

and connect it to the scale widget's adjustment like this:

```
adj.value_changed.connect(sigc::bind<Widget*>(sigc::mem_fun(*this,
    &cb_rotate_picture), picture));
```

What if a widget reconfigures the upper or lower fields of its Adjustment, such as when a user adds more text to a text widget? In this case, it emits the changed signal.

Range widgets typically connect a handler to this signal, which changes their appearance to reflect the change - for example, the size of the slider in a scrollbar will grow or shrink in inverse proportion to the difference between the lower and upper values of its Adjustment.

You probably won't ever need to attach a handler to this signal, unless you're writing a new type of range widget.

```
adjustment->changed();
```

15. Widgets Without X-Windows

Some Widgets do not have an associated X-Window, so they therefore do not receive X events. This means that the signals described in the X event signals section will not be emitted. If you want to capture events for these widgets you can use a special container called `Gtk::EventBox`, which is described in the EventBox section.

Here is a list of some of these Widgets:

```
Gtk::Alignment
Gtk::Arrow
Gtk::AspectFrame
Gtk::Bin
Gtk::Box
Gtk::Button
Gtk::CheckButton
Gtk::Fixed
Gtk::Frame
Gtk::Grid
Gtk::Image
Gtk::Label
Gtk::MenuItem
Gtk::Notebook
Gtk::Paned
Gtk::RadioButton
Gtk::Range
Gtk::ScrolledWindow
Gtk::Separator
Gtk::Table
Gtk::Toolbar
```

These widgets are mainly used for decoration or layout, so you won't often need to capture events on them. They are intended to have no X-Window in order to improve performance.

15.1. EventBox

Some *gtkmm* widgets don't have associated X windows; they draw on their parents' windows. Because of this, they cannot receive events. Also, if they are incorrectly sized, they don't clip, so you can get messy overwriting etc. To receive events on one of these widgets, you can it inside an `EventBox` widget and then call `Gtk::Widget::set_events()` on the EventBox before showing it.

Although the name `EventBox` emphasises the event-handling method, the widget can also be used for clipping (and more; see the example below).

The constructor for `Gtk::EventBox` is:

```
Gtk::EventBox();
```

A child widget can be added to the `EventBox` using:

```
event_box.add(child_widget);
```

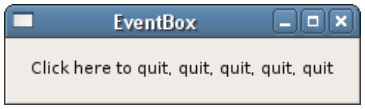
Reference

15.1.1. Example

The following example demonstrates both uses of an `EventBox` - a label is created that is clipped to a small box, and set up so that a mouse-click on the label causes the

program to exit. Resizing the window reveals varying amounts of the label.

Figure 15-1 **EventBox**



Source Code

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <gtkmm.h>

class ExampleWindow : public Gtk::Window
{
public:
  ExampleWindow();
  virtual ~ExampleWindow();

protected:
  //Signal handlers:
  virtual bool on_eventbox_button_press(GdkEventButton* event);

  //Child widgets:
  Gtk::EventBox m_EventBox;
  Gtk::Label m_Label;
};

#endif //GTKMM_EXAMPLEWINDOW_H
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"

ExampleWindow::ExampleWindow()
: m_Label("Click here to quit, quit, quit, quit, quit")
{
  set_title ("EventBox");
  set_border_width(10);

  add(m_EventBox);

  m_EventBox.add(m_Label);

  //Clip the label short:
  m_Label.set_size_request(110, 20);

  //And bind an action to it:
  m_EventBox.set_events(Gdk::BUTTON_PRESS_MASK);
  m_EventBox.signal_button_press_event().connect(
    sigc::mem_fun(*this, &ExampleWindow::on_eventbox_button_press) );

  m_EventBox.set_tooltip_text("Click me!");

  show_all_children();
}

ExampleWindow::~ExampleWindow()
{
}

bool ExampleWindow::on_eventbox_button_press(GdkEventButton*)
{
  hide();
  return true;
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

int main(int argc, char *argv[])
```

```
{
  Gtk::Main kit(argc, argv);

  ExampleWindow window;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

16. Dialogs

Dialogs are used as secondary windows, to provide specific information or to ask questions. `Gtk::Dialog` windows contain a few pre-packed widgets to ensure consistency, and a `run()` method which blocks until the user dismisses the dialog.

There are several derived `Dialog` classes which you might find useful. `Gtk::MessageDialog` is used for most simple notifications. But at other times you might need to derive your own dialog class to provide more complex functionality.

To pack widgets into a custom dialog, you should pack them into the `Gtk::Box`, available via `get_content_area()`. To just add a `Button` to the bottom of the `Dialog`, you could use the `add_button()` method.

The `run()` method returns an `int`. This may be a value from the `Gtk::ResponseType` if the user closed the button by clicking a standard button, or it could be the custom response value that you specified when using `add_button()`.

Reference

16.1. MessageDialog

`MessageDialog` is a convenience class, used to create simple, standard message dialogs, with a message, an icon, and buttons for user response. You can specify the type of message and the text in the constructor, as well as specifying standard buttons via the `Gtk::ButtonsType` enum.

Reference

16.1.1. Example

Figure 16-1 **MessageDialog**

Source Code

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <gtkmm.h>
```

```
class ExampleWindow : public Gtk::Window
{
public:
  ExampleWindow();
  virtual ~ExampleWindow();

protected:
  //Signal handlers:
  void on_button_info_clicked();
  void on_button_question_clicked();

  //Child widgets:
  Gtk::ButtonBox m_ButtonBox;
  Gtk::Button m_Button_Info, m_Button_Question;
};

#endif //GTKMM_EXAMPLEWINDOW_H
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/messagedialog.h>
#include <iostream>


ExampleWindow::ExampleWindow()
: m_ButtonBox(Gtk::ORIENTATION_VERTICAL),
  m_Button_Info("Show Info MessageDialog"),
  m_Button_Question("Show Question MessageDialog")
{
  set_title("Gtk::MessageDialog example");

  add(m_ButtonBox);

  m_ButtonBox.pack_start(m_Button_Info);
  m_Button_Info.signal_clicked().connect(sigc::mem_fun(*this,
              &ExampleWindow::on_button_info_clicked) );

  m_ButtonBox.pack_start(m_Button_Question);
  m_Button_Question.signal_clicked().connect(sigc::mem_fun(*this,
              &ExampleWindow::on_button_question_clicked) );

  show_all_children();
}

ExampleWindow::~ExampleWindow()
{
}

void ExampleWindow::on_button_info_clicked()
{
  Gtk::MessageDialog dialog(*this, "This is an INFO MessageDialog");
  dialog.set_secondary_text(
          "And this is the secondary text that explains things.");

  dialog.run();
}

void ExampleWindow::on_button_question_clicked()
{
  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();

  //Handle the response:
  switch(result)
  {
    case(Gtk::RESPONSE_OK):
    {
      std::cout << "OK clicked." << std::endl;
      break;
    }
    case(Gtk::RESPONSE_CANCEL):
    {
      std::cout << "Cancel clicked." << std::endl;
      break;
    }
    default:
    {
      std::cout << "Unexpected button clicked." << std::endl;
```

```
      break;
    }
  }
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

16.2. FileChooserDialog

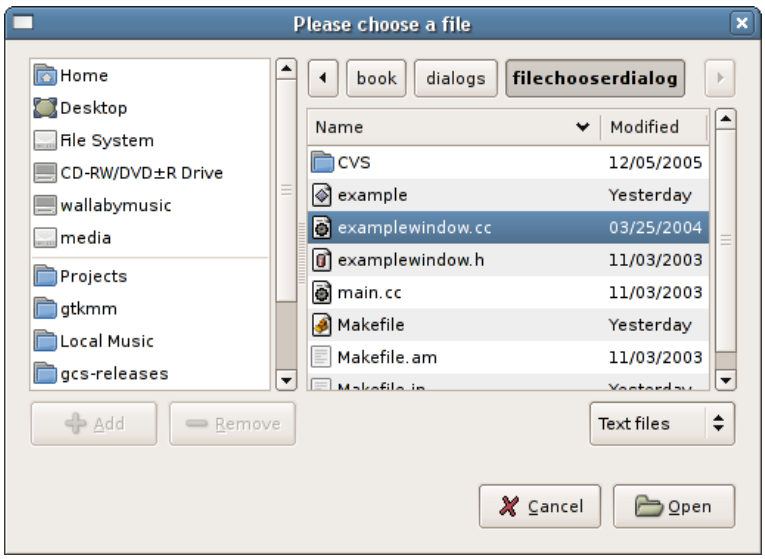
The `FileChooserDialog` is suitable for use with "Open" or "Save" menu items.

Most of the useful member methods for this class are actually in the `Gtk::FileChooser` base class.

Reference

16.2.1. Example

Figure 16-2 **FileChooser**



Source Code

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <gtkmm.h>

class ExampleWindow : public Gtk::Window
{
public:
  ExampleWindow();
```

```
  virtual ~ExampleWindow();

protected:
  //Signal handlers:
  void on_button_file_clicked();
  void on_button_folder_clicked();

  //Child widgets:
  Gtk::ButtonBox m_ButtonBox;
  Gtk::Button m_Button_File, m_Button_Folder;
};

#endif //GTKMM_EXAMPLEWINDOW_H
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <iostream>


ExampleWindow::ExampleWindow()
: m_ButtonBox(Gtk::ORIENTATION_VERTICAL),
  m_Button_File("Choose File"),
  m_Button_Folder("Choose Folder")
{
  set_title("Gtk::FileSelection example");

  add(m_ButtonBox);

  m_ButtonBox.pack_start(m_Button_File);
  m_Button_File.signal_clicked().connect(sigc::mem_fun(*this,
              &ExampleWindow::on_button_file_clicked) );

  m_ButtonBox.pack_start(m_Button_Folder);
  m_Button_Folder.signal_clicked().connect(sigc::mem_fun(*this,
              &ExampleWindow::on_button_folder_clicked) );

  show_all_children();
}

ExampleWindow::~ExampleWindow()
{
}

void ExampleWindow::on_button_folder_clicked()
{
  Gtk::FileChooserDialog dialog("Please choose a folder",
          Gtk::FILE_CHOOSER_ACTION_SELECT_FOLDER);
  dialog.set_transient_for(*this);

  //Add response buttons the the dialog:
  dialog.add_button(Gtk::Stock::CANCEL, Gtk::RESPONSE_CANCEL);
  dialog.add_button("Select", Gtk::RESPONSE_OK);

  int result = dialog.run();

  //Handle the response:
  switch(result)
  {
    case(Gtk::RESPONSE_OK):
    {
      std::cout << "Select clicked." << std::endl;
      std::cout << "Folder selected: " << dialog.get_filename()
          << std::endl;
      break;
    }
    case(Gtk::RESPONSE_CANCEL):
    {
      std::cout << "Cancel clicked." << std::endl;
      break;
    }
    default:
    {
      std::cout << "Unexpected button clicked." << std::endl;
      break;
    }
  }
}

void ExampleWindow::on_button_file_clicked()
{
  Gtk::FileChooserDialog dialog("Please choose a file",
          Gtk::FILE_CHOOSER_ACTION_OPEN);
  dialog.set_transient_for(*this);
```

```
  //Add response buttons the the dialog:
  dialog.add_button(Gtk::Stock::CANCEL, Gtk::RESPONSE_CANCEL);
  dialog.add_button(Gtk::Stock::OPEN, Gtk::RESPONSE_OK);

  //Add filters, so that only certain file types can be selected:

  Glib::RefPtr<Gtk::FileFilter> filter_text = Gtk::FileFilter::create();
  filter_text->set_name("Text files");
  filter_text->add_mime_type("text/plain");
  dialog.add_filter(filter_text);

  Glib::RefPtr<Gtk::FileFilter> filter_cpp = Gtk::FileFilter::create();
  filter_cpp->set_name("C/C++ files");
  filter_cpp->add_mime_type("text/x-c");
  filter_cpp->add_mime_type("text/x-c++");
  filter_cpp->add_mime_type("text/x-c-header");
  dialog.add_filter(filter_cpp);

  Glib::RefPtr<Gtk::FileFilter> filter_any = Gtk::FileFilter::create();
  filter_any->set_name("Any files");
  filter_any->add_pattern("*");
  dialog.add_filter(filter_any);

  //Show the dialog and wait for a user response:
  int result = dialog.run();

  //Handle the response:
  switch(result)
  {
    case(Gtk::RESPONSE_OK):
    {
      std::cout << "Open clicked." << std::endl;

      //Notice that this is a std::string, not a Glib::ustring.
      std::string filename = dialog.get_filename();
      std::cout << "File selected: " <<  filename << std::endl;
      break;
    }
    case(Gtk::RESPONSE_CANCEL):
    {
      std::cout << "Cancel clicked." << std::endl;
      break;
    }
    default:
    {
      std::cout << "Unexpected button clicked." << std::endl;
      break;
    }
  }
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

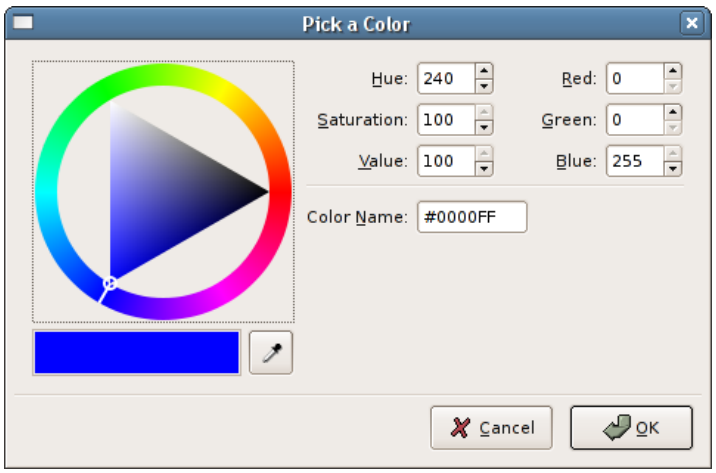
16.3. ColorSelectionDialog

The `ColorSelectionDialog` allows the user to choose a color. The `ColorButton` opens a color selection dialog when it is clicked.

Reference

16.3.1. Example

Figure 16-3 **ColorSelectionDialog**



Source Code

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <gtkmm.h>

class ExampleWindow : public Gtk::Window
{
public:
  ExampleWindow();
  virtual ~ExampleWindow();

protected:
  //Signal handlers:
  void on_color_button_color_set();
  void on_button_dialog_clicked();

  //Child widgets:
  Gtk::Box m_VBox;
  Gtk::ColorButton m_ColorButton;
  Gtk::Button m_Button_Dialog;
  Gtk::DrawingArea m_DrawingArea; //To show the color.

  Gdk::RGBA m_Color;
};

#endif //GTKMM_EXAMPLEWINDOW_H
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <iostream>

ExampleWindow::ExampleWindow()
: m_VBox(Gtk::ORIENTATION_VERTICAL, 5),
  m_Button_Dialog("Choose Color")
{
  set_title("Gtk::ColorSelectionDialog example");
  set_default_size(200, 200);

  add(m_VBox);

  m_VBox.pack_start(m_ColorButton, Gtk::PACK_SHRINK);
  m_ColorButton.signal_color_set().connect(sigc::mem_fun(*this,
    &ExampleWindow::on_color_button_color_set) );

  m_VBox.pack_start(m_Button_Dialog, Gtk::PACK_SHRINK);
  m_Button_Dialog.signal_clicked().connect(sigc::mem_fun(*this,
    &ExampleWindow::on_button_dialog_clicked) );

  //Set start color:
  m_Color.set_red(0.0);
  m_Color.set_green(0.0);
  m_Color.set_blue(1.0);
```

```
  m_Color.set_alpha(1.0); //opaque
  m_ColorButton.set_rgba(m_Color);

  m_DrawingArea.override_background_color(m_Color);

  m_VBox.pack_start(m_DrawingArea);

  show_all_children();
}

ExampleWindow::~ExampleWindow()
{
}

void ExampleWindow::on_color_button_color_set()
{
  //Store the chosen color, and show it:
  m_Color = m_ColorButton.get_rgba();
  m_DrawingArea.override_background_color(m_Color);
}

void ExampleWindow::on_button_dialog_clicked()
{
  Gtk::ColorSelectionDialog dialog("Please choose a color");
  dialog.set_transient_for(*this);

  //Get the previously selected color:
  dialog.get_color_selection()->set_current_rgba(m_Color);

  int result = dialog.run();

  //Handle the response:
  switch(result)
  {
    case Gtk::RESPONSE_OK:
    {
      //Store the chosen color, and show it:
      m_Color = dialog.get_color_selection()->get_current_rgba();
      m_ColorButton.set_rgba(m_Color);
      m_DrawingArea.override_background_color(m_Color);
      break;
    }
    case Gtk::RESPONSE_CANCEL:
    {
      std::cout << "Cancel clicked." << std::endl;
      break;
    }
    default:
    {
      std::cout << "Unexpected button clicked: " << result << std::endl;
      break;
    }
  }
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

16.4. FontChooserDialog

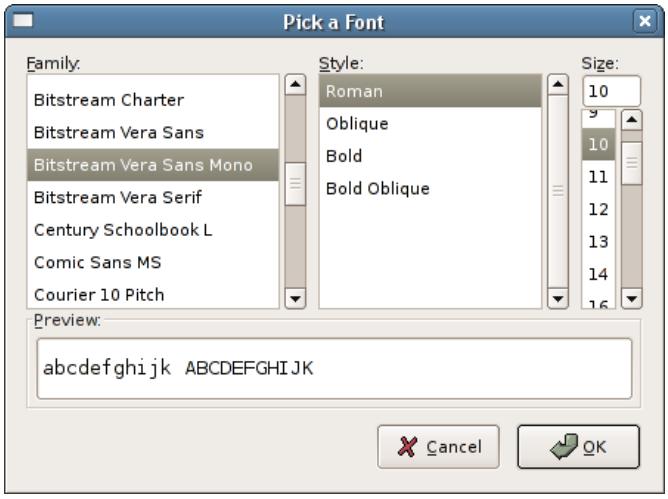
The `FontChooserDialog` allows the user to choose a font. The `FontButton` opens a font chooser dialog when it is clicked.

Reference

16.4.1. Example

16.4.1. Example

Figure 16-4 **FontChooserDialog**



Source Code

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <gtkmm.h>

class ExampleWindow : public Gtk::Window
{
public:
  ExampleWindow();
  virtual ~ExampleWindow();

protected:
  //Signal handlers:
  void on_font_button_font_set();
  void on_button_dialog_clicked();

  //Child widgets:
  Gtk::ButtonBox m_ButtonBox;
  Gtk::FontButton m_FontButton;
  Gtk::Button m_Button_Dialog;
};

#endif //GTKMM_EXAMPLEWINDOW_H
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <iostream>

ExampleWindow::ExampleWindow()
: m_ButtonBox(Gtk::ORIENTATION_VERTICAL),
  m_FontButton("Sans 10"),
  m_Button_Dialog("Choose Font")
{
  set_title("Gtk::FontChooserDialog example");

  add(m_ButtonBox);

  m_ButtonBox.pack_start(m_FontButton);
  m_FontButton.set_use_font(true);
  m_FontButton.set_use_size(true);
  m_FontButton.signal_font_set().connect(sigc::mem_fun(*this,
    &ExampleWindow::on_font_button_font_set) );

  m_ButtonBox.pack_start(m_Button_Dialog);
  m_Button_Dialog.signal_clicked().connect(sigc::mem_fun(*this,
    &ExampleWindow::on_button_dialog_clicked) );
  show_all_children();
}
```

```
ExampleWindow::~ExampleWindow()
{
}

void ExampleWindow::on_font_button_font_set()
{
  Glib::ustring font_name = m_FontButton.get_font_name();
  std::cout << "Font chosen: " << font_name << std::endl;
}

void ExampleWindow::on_button_dialog_clicked()
{
  Gtk::FontChooserDialog dialog("Please choose a font", *this);

  //Get the previously selected font name from the FontButton:
  dialog.set_font(m_FontButton.get_font_name());

  int result = dialog.run();

  //Handle the response:
  switch(result)
  {
    case Gtk::RESPONSE_OK:
    {
      Glib::ustring font_name = dialog.get_font();
      std::cout << "Font chosen: " << font_name << std::endl;
      m_FontButton.set_font_name(font_name);
      break;
    }
    case Gtk::RESPONSE_CANCEL:
    {
      std::cout << "Cancel clicked." << std::endl;
      break;
    }
    default:
    {
      std::cout << "Unexpected button clicked: " << result << std::endl;
      break;
    }
  }
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

17. The Drawing Area Widget

The `DrawingArea` widget is a blank window that gives you the freedom to create any graphic you desire. Along with that freedom comes the responsibility to handle draw signals on the widget. When a widget is first shown, or when it is covered and then uncovered again it needs to redraw itself. Most widgets have code to do this, but the DrawingArea does not, allowing you to write your own draw signal handler to determine how the contents of the widget will be drawn. This is most often done by overriding the virtual `on_draw()` member function.

GTK+ uses the Cairo drawing API. With gtkmm, you may use the cairomm C++ API for cairo.

You can draw very sophisticated shapes using Cairo, but the methods to do so are quite basic. Cairo provides methods for drawing straight lines, curved lines, and arcs (including circles). These basic shapes can be combined to create more complex shapes and paths which can be filled with solid colors, gradients, patterns, and other things. In addition, Cairo can perform complex transformations, do compositing of images, and render antialiased text.

Cairo and Pango

Although Cairo can render text, it's not meant to be a replacement for Pango. Pango is a better choice if you need to perform more advanced text rendering such as wrapping or ellipsizing text. Drawing text with Cairo should only be done if the text is part of a graphic.

In this section of the tutorial, we'll cover the basic Cairo drawing model, describe each of the basic drawing elements in some detail (with examples), and then present a simple application that uses Cairo to draw a custom clock widget.

17.1. The Cairo Drawing Model

The basic concept of drawing in Cairo involves defining 'invisible' paths and then stroking or filling them to make them visible.

To do any drawing in *gtkmm* with Cairo, you must first create a `Cairo::Context` object. This class holds all of the graphics state parameters that describe how drawing is to be done. This includes information such as line width, color, the surface to draw to, and many other things. This allows the actual drawing functions to take fewer arguments to simplify the interface. In *gtkmm*, a `Cairo::Context` is created by calling the `Gdk::Window::create_cairo_context()` function. Since Cairo contexts are reference-counted objects, this function returns a `Cairo::RefPtr<Cairo::Context>` object.

The following example shows how to set up a Cairo context with a foreground color of red and a width of 2. Any drawing functions that use this context will use these settings.

```
Gtk::DrawingArea myArea;
Cairo::RefPtr<Cairo::Context> myContext = myArea.get_window()->create_cairo_context();
myContext->set_source_rgb(1.0, 0.0, 0.0);
myContext->set_line_width(2.0);
```

Each `Cairo::Context` is associated with a particular `Gdk::Window`, so the first line of the above example creates a `Gtk::DrawingArea` widget and the second line uses its associated `Gdk::Window` to create a `Cairo::Context` object. The final two lines change the graphics state of the context.

There are a number of graphics state variables that can be set for a Cairo context. The most common context attributes are color (using `set_source_rgb()` or `set_source_rgba()` for translucent colors), line width (using `set_line_width()`), line dash pattern (using `set_dash()`), line cap style (using `set_line_cap()`), and line join style (using `set_line_join()`), and font styles (using `set_font_size()`, `set_font_face()` and others). There are many other settings as well, such as transformation matrices, fill rules, whether to perform antialiasing, and others. For further information, see the cairomm API documentation.

The current state of a `Cairo::Context` can be saved to an internal stack of saved states and later be restored to the state it was in when you saved it. To do this, use the `save()` method and the `restore()` method. This can be useful if you need to temporarily change the line width and color (or any other graphics setting) in order to draw something and then return to the previous settings. In this situation, you could call `Cairo::Context::save()`, change the graphics settings, draw the lines, and then call `Cairo::Context::restore()` to restore the original graphics state. Multiple calls to `save()` and `restore()` can be nested; each call to `restore()` restores the state from the matching paired `save()`.

It is good practice to put all modifications to the graphics state between `save()`/`restore()` function calls. For example, if you have a function that takes a `Cairo::Context` reference as an argument, you might implement it as follows:

```
void doSomething(const Cairo::RefPtr<Cairo::Context>& context, int x)
{
    context->save();
    // change graphics state
    // peform drawing operations
    context->restore();
}
```

The virtual `on_draw()` method provides a Cairo context that you shall use for drawing in the `Gtk::DrawingArea` widget. It is not necessary to save and restore this Cairo context in `on_draw()`.

17.2. Drawing Straight Lines

Now that we understand the basics of the Cairo graphics library, we're almost ready to start drawing. We'll start with the simplest of drawing elements: the straight line. But first you need to know a little bit about Cairo's coordinate system. The origin of the Cairo coordinate system is located in the upper-left corner of the window with positive x values to the right and positive y values going down.

> Since the Cairo graphics library was written with support for multiple output targets (the X window system, PNG images, OpenGL, etc), there is a distinction between user-space and device-space coordinates. The mapping between these two coordinate systems defaults to one-to-one so that integer values map roughly to pixels on the screen, but this setting can be adjusted if desired. Sometimes it may be useful to scale the coordinates so that the full width and height of a window both range from 0 to 1 (the 'unit square') or some other mapping that works for your application. This can be done with the `Cairo::Context::scale()` function.

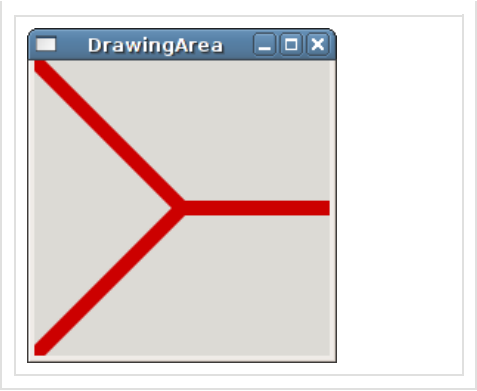
17.2.1. Example

In this example, we'll construct a small but fully functional *gtkmm* program and draw some lines into the window. The lines are drawn by creating a path and then stroking it. A path is created using the functions `Cairo::Context::move_to()` and `Cairo::Context::line_to()`. The function `move_to()` is similar to the act of lifting your pen off of the paper and placing it somewhere else -- no line is drawn between the point you were at and the point you moved to. To draw a line between two points, use the `line_to()` function.

After you've finished creating your path, you still haven't drawn anything visible yet. To make the path visible, you must use the function `stroke()` which will stroke the current path with the line width and style specified in your `Cairo::Context` object. After stroking, the current path will be cleared so that you can start on your next path.

> Many Cairo drawing functions have a `_preserve()` variant. Normally drawing functions such as `clip()`, `fill()`, or `stroke()` will clear the current path. If you use the `_preserve()` variant, the current path will be retained so that you can use the same path with the next drawing function.

Figure 17-1 **Drawing Area - Lines**



Source Code

File: `myarea.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLE_MYAREA_H
#define GTKMM_EXAMPLE_MYAREA_H

#include <gtkmm/drawingarea.h>

class MyArea : public Gtk::DrawingArea
{
public:
  MyArea();
  virtual ~MyArea();

protected:
  //Override default signal handler:
  virtual bool on_draw(const Cairo::RefPtr<Cairo::Context>& cr);
};

#endif // GTKMM_EXAMPLE_MYAREA_H
```

File: `myarea.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "myarea.h"
#include <cairomm/context.h>

MyArea::MyArea()
{
}

MyArea::~MyArea()
{
}

bool MyArea::on_draw(const Cairo::RefPtr<Cairo::Context>& cr)
{
  Gtk::Allocation allocation = get_allocation();
  const int width = allocation.get_width();
  const int height = allocation.get_height();

  // coordinates for the center of the window
  int xc, yc;
  xc = width / 2;
  yc = height / 2;

  cr->set_line_width(10.0);

  // draw red lines out from the center of the window
  cr->set_source_rgb(0.8, 0.0, 0.0);
  cr->move_to(0, 0);
  cr->line_to(xc, yc);
  cr->line_to(0, height);
  cr->move_to(xc, yc);
  cr->line_to(width, yc);
  cr->stroke();

  return true;
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "myarea.h"
#include <gtkmm/main.h>
#include <gtkmm/window.h>

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

   Gtk::Window win;
   win.set_title("DrawingArea");

   MyArea area;
   win.add(area);
   area.show();

   Gtk::Main::run(win);

   return 0;
}
```

This program contains a single class, `MyArea`, which is a subclass of `Gtk::DrawingArea` and contains an `on_draw()` member function. This function is called whenever the image in the drawing area needs to be redrawn. It is passed a `Cairo::RefPtr` pointer to a `Cairo::Context` that we use for the drawing. The actual drawing code sets the color we want to use for drawing by using `set_source_rgb()` which takes arguments defining the Red, Green, and Blue components of the desired color (valid values are between 0 and 1). After setting the color, we created a new path using the functions `move_to()` and `line_to()`, and then stroked this path with `stroke()`.

Drawing with relative coordinates

In the example above we drew everything using absolute coordinates. You can also draw using relative coordinates. For a straight line, this is done with the function `Cairo::Context::rel_line_to()`.

17.2.2. Line styles

In addition to drawing basic straight lines, there are a number of things that you can customize about a line. You've already seen examples of setting a line's color and width, but there are others as well.

If you've drawn a series of lines that form a path, you may want them to join together in a certain way. Cairo offers three different ways to join lines together: Miter, Bevel, and Round. These are show below:

Figure 17-2 **Different join types in Cairo**

The line join style is set using the function `Cairo::Context::set_line_join()`.

Line ends can have different styles as well. The default style is for the line to start and stop exactly at the destination points of the line. This is called a Butt cap. The other options are Round (uses a round ending, with the center of the circle at the end point) or Square (uses a squared ending, with the center of the square at the end point). This setting is set using the function `Cairo::Context::set_line_cap()`.

There are other things you can customize as well, including creating dashed lines and other things. For more information, see the Cairo API documentation.

17.3. Drawing Curved Lines

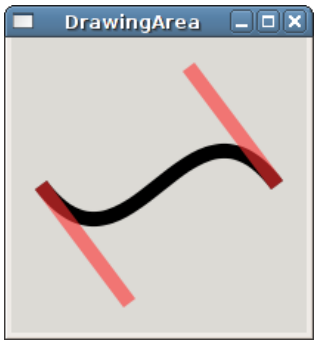
In addition to drawing straight lines Cairo allows you to easily draw curved lines (technically a cubic Bézier spline) using the `Cairo::Context::curve_to()` and `Cairo::Context::rel_curve_to()` functions. These functions take coordinates for a

destination point as well as coordinates for two 'control' points. This is best explained using an example, so let's dive in.

17.3.1. Example

This simple application draws a curve with Cairo and displays the control points for each end of the curve.

Figure 17-3 **Drawing Area - Lines**



Source Code

File: `myarea.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLE_MYAREA_H
#define GTKMM_EXAMPLE_MYAREA_H

#include <gtkmm/drawingarea.h>

class MyArea : public Gtk::DrawingArea
{
public:
  MyArea();
  virtual ~MyArea();

protected:
  //Override default signal handler:
  virtual bool on_draw(const Cairo::RefPtr<Cairo::Context>& cr);
};

#endif // GTKMM_EXAMPLE_MYAREA_H
```

File: `myarea.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "myarea.h"
#include <cairomm/context.h>

MyArea::MyArea()
{
}

MyArea::~MyArea()
{
}

bool MyArea::on_draw(const Cairo::RefPtr<Cairo::Context>& cr)
{
  Gtk::Allocation allocation = get_allocation();
  const int width = allocation.get_width();
  const int height = allocation.get_height();

  double x0=0.1, y0=0.5, // start point
         x1=0.4, y1=0.9,  // control point #1
         x2=0.6, y2=0.1,  // control point #2
         x3=0.9, y3=0.5;  // end point

  // scale to unit square (0 to 1 width and height)
  cr->scale(width, height);
```

```
  cr->set_line_width(0.05);
  // draw curve
  cr->move_to(x0, y0);
  cr->curve_to(x1, y1, x2, y2, x3, y3);
  cr->stroke();
  // show control points
  cr->set_source_rgba(1, 0.2, 0.2, 0.6);
  cr->move_to(x0, y0);
  cr->line_to (x1, y1);
  cr->move_to(x2, y2);
  cr->line_to (x3, y3);
  cr->stroke();

  return true;
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "myarea.h"
#include <gtkmm/main.h>
#include <gtkmm/window.h>

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

   Gtk::Window win;
   win.set_title("DrawingArea");

   MyArea area;
   win.add(area);
   area.show();

   Gtk::Main::run(win);

   return 0;
}
```

The only difference between this example and the straight line example is in the `on_draw()` function, but there are a few new concepts and functions introduced here, so let's examine them briefly.

We make a call to `Cairo::Context::scale()`, passing in the width and height of the drawing area. This scales the user-space coordinate system such that the width and height of the widget are both equal to 1.0 'units'. There's no particular reason to scale the coordinate system in this case, but sometimes it can make drawing operations easier.

The call to `Cairo::Context::curve_to()` should be fairly self-explanatory. The first pair of coordinates define the control point for the beginning of the curve. The second set of coordinates define the control point for the end of the curve, and the last set of coordinates define the destination point. To make the concept of control points a bit easier to visualize, a line has been draw from each control point to the end-point on the curve that it is associated with. Note that these control point lines are both translucent. This is achieved with a variant of `set_source_rgb()` called `set_source_rgba()`. This function takes a fourth argument specifying the alpha value of the color (valid values are between 0 and 1).

17.4. Drawing Arcs and Circles

With Cairo, the same function is used to draw arcs, circles, or ellipses: `Cairo::Context::arc()`. This function takes five arguments. The first two are the coordinates of the center point of the arc, the third argument is the radius of the arc, and the final two arguments define the start and end angle of the arc. All angles are defined in radians, so drawing a circle is the same as drawing an arc from 0 to 2 * M_PI radians. An angle of 0 is in the direction of the positive X axis (in user-space). An angle of M_PI/2 radians (90 degrees) is in the direction of the positive Y axis (in user-space). Angles increase in the direction from the positive X axis toward the positive Y axis. So with the default transformation matrix, angles increase in a clockwise direction. (Remember that the positive Y axis points downwards.)

To draw an ellipse, you can scale the current transformation matrix by different amounts in the X and Y directions. For example, to draw an ellipse with center at `x`, `y` and size `width`, `height`:

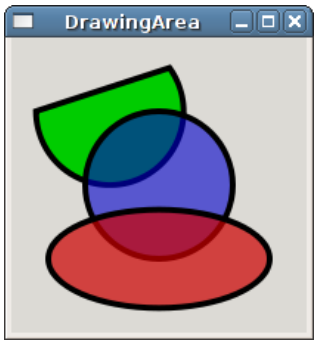
```
context->save();
context->translate(x, y);
context->scale(width / 2.0, height / 2.0);
context->arc(0.0, 0.0, 1.0, 0.0, 2 * M_PI);
context->restore();
```

17.4.1. Example

17.4.1. Example

Here's an example of a simple program that draws an arc, a circle and an ellipse into a drawing area.

Figure 17-4 **Drawing Area - Arcs**



Source Code

File: `myarea.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLE_MYAREA_H
#define GTKMM_EXAMPLE_MYAREA_H

#include <gtkmm/drawingarea.h>

class MyArea : public Gtk::DrawingArea
{
public:
  MyArea();
  virtual ~MyArea();

protected:
  //Override default signal handler:
  virtual bool on_draw(const Cairo::RefPtr<Cairo::Context>& cr);
};

#endif // GTKMM_EXAMPLE_MYAREA_H
```

File: `myarea.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "myarea.h"
#include <cairomm/context.h>
#include <cmath>

MyArea::MyArea()
{
}

MyArea::~MyArea()
{
}

bool MyArea::on_draw(const Cairo::RefPtr<Cairo::Context>& cr)
{
  // This is where we draw on the window
  Gtk::Allocation allocation = get_allocation();
  const int width = allocation.get_width();
  const int height = allocation.get_height();
  const int lesser = MIN(width, height);
```

```
  // coordinates for the center of the window
  int xc, yc;
  xc = width / 2;
  yc = height / 2;

  cr->set_line_width(lesser * 0.02);  // outline thickness changes
                                      // with window size

  // first draw a simple unclosed arc
  cr->save();
  cr->arc(width / 3.0, height / 4.0, lesser / 4.0, -(M_PI / 5.0), M_PI);
  cr->close_path();   // line back to start point
  cr->set_source_rgb(0.0, 0.8, 0.0);
  cr->fill_preserve();
  cr->restore();  // back to opaque black
  cr->stroke();   // outline it

  // now draw a circle
  cr->save();
  cr->arc(xc, yc, lesser / 4.0, 0.0, 2.0 * M_PI); // full circle
  cr->set_source_rgba(0.0, 0.0, 0.8, 0.6);    // partially translucent
  cr->fill_preserve();
  cr->restore();  // back to opaque black
  cr->stroke();

  // and finally an ellipse
  double ex, ey, ew, eh;
  // center of ellipse
  ex = xc;
  ey = 3.0 * height / 4.0;
  // ellipse dimensions
  ew = 3.0 * width / 4.0;
  eh = height / 3.0;

  cr->save();

  cr->translate(ex, ey);  // make (ex, ey) == (0, 0)
  cr->scale(ew / 2.0, eh / 2.0);  // for width: ew / 2.0 == 1.0
                                  // for height: eh / 2.0 == 1.0

  cr->arc(0.0, 0.0, 1.0, 0.0, 2 * M_PI);  // 'circle' centered at (0, 0)
                                          // with 'radius' of 1.0

  cr->set_source_rgba(0.8, 0.0, 0.0, 0.7);
  cr->fill_preserve();
  cr->restore();  // back to opaque black
  cr->stroke();

  return true;
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "myarea.h"
#include <gtkmm/main.h>
#include <gtkmm/window.h>

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

   Gtk::Window win;
   win.set_title("DrawingArea");

   MyArea area;
   win.add(area);
   area.show();

   Gtk::Main::run(win);

   return 0;
}
```

There are a couple of things to note about this example code. Again, the only real difference between this example and the previous ones is the `on_draw()` function, so we'll limit our focus to that function. In addition, the first part of the function is nearly identical to the previous examples, so we'll skip that portion.

Note that in this case, we've expressed nearly everything in terms of the height and width of the window, including the width of the lines. Because of this, when you resize the window, everything scales with the window. Also note that there are three drawing sections in the function and each is wrapped with a `save()`/`restore()` pair so

that we're back at a known state after each drawing.

The section for drawing an arc introduces one new function, `close_path()`. This function will in effect draw a straight line from the current point back to the first point in the path. There is a significant difference between calling `close_path()` and manually drawing a line back to the starting point, however. If you use `close_path()`, the lines will be nicely joined together. If you use `line_to()` instead, the lines will end at the same point, but Cairo won't do any special joining.

> **Drawing counter-clockwise**
> The function `Cairo::Context::arc_negative()` is exactly the same as `Cairo::Context::arc()` but the angles go the opposite direction.

17.5. Drawing Text

17.5.1. Drawing Text with Pango

Text is drawn via Pango Layouts. The easiest way to create a `Pango::Layout` is to use `Gtk::Widget::create_pango_layout()`. Once created, the layout can be manipulated in various ways, including changing the text, font, etc. Finally, the layout can be rendered using the `Pango::Layout::show_in_cairo_context()` method.

The Printing chapter contains an example of drawing text.

17.6. Drawing Images

There is a method for drawing from a `Gdk::Pixbuf` to a `Cairo::Context`. A `Gdk::Pixbuf` buffer is a useful wrapper around a collection of pixels, which can be read from files, and manipulated in various ways.

Probably the most common way of creating `Gdk::Pixbuf`s is to use `Gdk::Pixbuf::create_from_file()`, which can read an image file, such as a png file into a pixbuf ready for rendering.

The `Gdk::Pixbuf` can be rendered by setting it as the source pattern of the Cairo context with `Gdk::Cairo::set_source_pixbuf()`. Then draw the image with either `Cairo::Context::paint()` (to draw the whole image), or `Cairo::Context::rectangle()` and `Cairo::Context::fill()` (to fill the specified rectangle). `set_source_pixbuf()` is not a member of `Cairo::Context`. It takes a `Cairo::Context` as its first parameter.

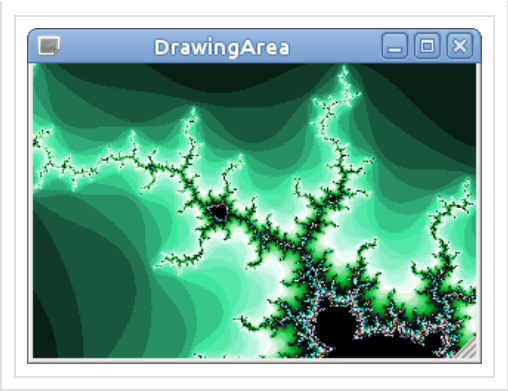
Here is a small bit of code to tie it all together: (Note that usually you wouldn't load the image every time in the draw signal handler! It's just shown here to keep it all together.)

```
bool MyArea::on_draw(const Cairo::RefPtr<Cairo::Context>& cr)
{
  Glib::RefPtr<Gdk::Pixbuf> image = Gdk::Pixbuf::create_from_file("myimage.png");
  // Draw the image at 110, 90, except for the outermost 10 pixels.
  Gdk::Cairo::set_source_pixbuf(cr, image, 100, 80);
  cr->rectangle(110, 90, image->get_width()-20, image->get_height()-20);
  cr->fill();
  return true;
}
```


17.6.1. Example

Here is an example of a simple program that draws an image.

Figure 17-5 **Drawing Area - Image**



Source Code

File: `myarea.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLE_MYAREA_H
#define GTKMM_EXAMPLE_MYAREA_H

#include <gtkmm/drawingarea.h>
#include <gdkmm/pixbuf.h>

class MyArea : public Gtk::DrawingArea
{
public:
  MyArea();
  virtual ~MyArea();

protected:
  //Override default signal handler:
  virtual bool on_draw(const Cairo::RefPtr<Cairo::Context>& cr);

  Glib::RefPtr<Gdk::Pixbuf> m_image;
};

#endif // GTKMM_EXAMPLE_MYAREA_H
```

File: `myarea.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "myarea.h"
#include <cairomm/context.h>
#include <gdkmm/general.h> // set_source_pixbuf()
#include <glibmm/fileutils.h>
#include <iostream>

MyArea::MyArea()
{
  try
  {
    // The fractal image has been created by the XaoS program.
    // http://xaos.sourceforge.net
    m_image = Gdk::Pixbuf::create_from_file("fractal_image.png");
  }
  catch(const Glib::FileError& ex)
  {
    std::cerr << "FileError: " << ex.what() << std::endl;
  }
  catch(const Gdk::PixbufError& ex)
  {
    std::cerr << "PixbufError: " << ex.what() << std::endl;
  }

  // Show at least a quarter of the image.
  if (m_image)
    set_size_request(m_image->get_width()/2, m_image->get_height()/2);
}

MyArea::~MyArea()
{
}

bool MyArea::on_draw(const Cairo::RefPtr<Cairo::Context>& cr)
{
  if (!m_image)
    return false;
```

```
  Gtk::Allocation allocation = get_allocation();
  const int width = allocation.get_width();
  const int height = allocation.get_height();

  // Draw the image in the middle of the drawing area, or (if the image is
  // larger than the drawing area) draw the middle part of the image.
  Gdk::Cairo::set_source_pixbuf(cr, m_image,
    (width - m_image->get_width())/2, (height - m_image->get_height())/2);
  cr->paint();

  return true;
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "myarea.h"
#include <gtkmm/main.h>
#include <gtkmm/window.h>

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

  Gtk::Window win;
  win.set_title("DrawingArea");
  win.set_default_size(300, 200);

  MyArea area;
  win.add(area);
  area.show();

  Gtk::Main::run(win);

  return 0;
}
```

17.7. Example Application: Creating a Clock with Cairo

Now that we've covered the basics of drawing with Cairo, let's try to put it all together and create a simple application that actually does something. The following example uses Cairo to create a custom `Clock` widget. The clock has a second hand, a minute hand, and an hour hand, and updates itself every second.

Source Code

File: `clock.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLE_CLOCK_H
#define GTKMM_EXAMPLE_CLOCK_H

#include <gtkmm/drawingarea.h>

class Clock : public Gtk::DrawingArea
{
public:
  Clock();
  virtual ~Clock();

protected:
  //Override default signal handler:
  virtual bool on_draw(const Cairo::RefPtr<Cairo::Context>& cr);

  bool on_timeout();
```

```
  double m_radius;
  double m_line_width;

};

#endif // GTKMM_EXAMPLE_CLOCK_H
```

File: `clock.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include <ctime>
#include <cmath>
#include <cairomm/context.h>
#include <glibmm/main.h>
#include "clock.h"

Clock::Clock()
: m_radius(0.42), m_line_width(0.05)
{
  Glib::signal_timeout().connect( sigc::mem_fun(*this, &Clock::on_timeout), 1000 );

  #ifndef GLIBMM_DEFAULT_SIGNAL_HANDLERS_ENABLED
  //Connect the signal handler if it isn't already a virtual method override:
  signal_draw().connect(sigc::mem_fun(*this, &Clock::on_draw), false);
  #endif //GLIBMM_DEFAULT_SIGNAL_HANDLERS_ENABLED
}

Clock::~Clock()
{
}

bool Clock::on_draw(const Cairo::RefPtr<Cairo::Context>& cr)
{
  Gtk::Allocation allocation = get_allocation();
  const int width = allocation.get_width();
  const int height = allocation.get_height();

  // scale to unit square and translate (0, 0) to be (0.5, 0.5), i.e.
  // the center of the window
  cr->scale(width, height);
  cr->translate(0.5, 0.5);
  cr->set_line_width(m_line_width);

  cr->save();
  cr->set_source_rgba(0.337, 0.612, 0.117, 0.9);   // green
  cr->paint();
  cr->restore();
  cr->arc(0, 0, m_radius, 0, 2 * M_PI);
  cr->save();
  cr->set_source_rgba(1.0, 1.0, 1.0, 0.8);
  cr->fill_preserve();
  cr->restore();
  cr->stroke_preserve();
  cr->clip();

  //clock ticks
  for (int i = 0; i < 12; i++)
  {
    double inset = 0.05;

    cr->save();
    cr->set_line_cap(Cairo::LINE_CAP_ROUND);

    if(i % 3 != 0)
    {
      inset *= 0.8;
      cr->set_line_width(0.03);
    }

    cr->move_to(
      (m_radius - inset) * cos (i * M_PI / 6),
      (m_radius - inset) * sin (i * M_PI / 6));
    cr->line_to (
      m_radius * cos (i * M_PI / 6),
      m_radius * sin (i * M_PI / 6));
    cr->stroke();
    cr->restore(); /* stack-pen-size */
  }

  // store the current time
  time_t rawtime;
  time(&rawtime);
  struct tm * timeinfo = localtime (&rawtime);

  // compute the angles of the indicators of our clock
```

```
  double minutes = timeinfo->tm_min * M_PI / 30;
  double hours = timeinfo->tm_hour * M_PI / 6;
  double seconds= timeinfo->tm_sec * M_PI / 30;

  cr->save();
  cr->set_line_cap(Cairo::LINE_CAP_ROUND);

  // draw the seconds hand
  cr->save();
  cr->set_line_width(m_line_width / 3);
  cr->set_source_rgba(0.7, 0.7, 0.7, 0.8); // gray
  cr->move_to(0, 0);
  cr->line_to(sin(seconds) * (m_radius * 0.9),
    -cos(seconds) * (m_radius * 0.9));
  cr->stroke();
  cr->restore();

  // draw the minutes hand
  cr->set_source_rgba(0.117, 0.337, 0.612, 0.9);   // blue
  cr->move_to(0, 0);
  cr->line_to(sin(minutes + seconds / 60) * (m_radius * 0.8),
    -cos(minutes + seconds / 60) * (m_radius * 0.8));
  cr->stroke();

  // draw the hours hand
  cr->set_source_rgba(0.337, 0.612, 0.117, 0.9);   // green
  cr->move_to(0, 0);
  cr->line_to(sin(hours + minutes / 12.0) * (m_radius * 0.5),
    -cos(hours + minutes / 12.0) * (m_radius * 0.5));
  cr->stroke();
  cr->restore();

  // draw a little dot in the middle
  cr->arc(0, 0, m_line_width / 3.0, 0, 2 * M_PI);
  cr->fill();

  return true;
}


bool Clock::on_timeout()
{
    // force our program to redraw the entire clock.
    Glib::RefPtr<Gdk::Window> win = get_window();
    if (win)
    {
        Gdk::Rectangle r(0, 0, get_allocation().get_width(),
                get_allocation().get_height());
        win->invalidate_rect(r, false);
    }
    return true;
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "clock.h"
#include <gtkmm/main.h>
#include <gtkmm/window.h>

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

   Gtk::Window win;
   win.set_title("Cairomm Clock");

   Clock c;
   win.add(c);
   c.show();

   Gtk::Main::run(win);

   return 0;
}
```

As before, almost all of the interesting stuff is done in the draw signal handler `on_draw()`. Before we dig into the draw signal handler, notice that the constructor for the `Clock` widget connects a handler function `on_timeout()` to a timer with a timeout period of 1000 milliseconds (1 second). This means that `on_timeout()` will get called once per second. The sole responsibility of this function is to invalidate the window so that *gtkmm* will be forced to redraw it.

Now let's take a look at the code that performs the actual drawing. The first section

of on_draw() should be pretty familiar by now. This example again scales the coordinate system to be a unit square so that it's easier to draw the clock as a percentage of window size so that it will automatically scale when the window size is adjusted. Furthermore, the coordinate system is scaled over and down so that the (0, 0) coordinate is in the very center of the window.

The function Cairo::Context::paint() is used here to set the background color of the window. This function takes no arguments and fills the current surface (or the clipped portion of the surface) with the source color currently active. After setting the background color of the window, we draw a circle for the clock outline, fill it with white, and then stroke the outline in black. Notice that both of these actions use the _preserve variant to preserve the current path, and then this same path is clipped to make sure than our next lines don't go outside the outline of the clock.

After drawing the outline, we go around the clock and draw ticks for every hour, with a larger tick at 12, 3, 6, and 9. Now we're finally ready to implement the time-keeping functionality of the clock, which simply involves getting the current values for hours, minutes and seconds, and drawing the hands at the correct angles.

18. Drag and Drop

Gtk::Widget has several methods and signals which are prefixed with "drag_". These are used for Drag and Drop.

18.1. Sources and Destinations

Things are dragged from sources to be dropped on destinations. Each source and destination has infomation about the data formats that it can send or receive, provided by Gtk::TargetEntry items. A drop destination will only accept a dragged item if they both share a compatible Gtk::TargetEntry item. Appropriate signals will then be emitted, telling the signal handlers which Gtk::TargetEntry was used.

Gtk::TargetEntry objects contain this information:

- target: A name, such as "STRING"
- info: An identifier which will be sent to your signals to tell you which TargetEntry was used.
- flags: Used only for drag and drop, this specifies whether the data may be dragged to other widgets and applications, or only to the same ones.

18.2. Methods

Widgets can be identified as sources or destinations using these Gtk::Widget methods:

```
void drag_source_set(const ArrayHandle_TargetEntry& targets,
      GdkModifierType start_button_mask, GdkDragAction actions);
```

- targets is a container of Gtk::TargetEntry (std::list<Gtk::TargetEntry> or std::vector<Gtk::TargetEntry>, for instance) elements.
- start_button_mask is an ORed combination of values, which specify which modifier key or mouse button must be pressed to start the drag.
- actions is an ORed combination of values, which specified which Drag and Drop operations will be possible from this source - for instance, copy, move, or link. The user can choose between the actions by using modifier keys, such as **Shift** to change from copy to move, and this will be shown by a different cursor.

```
void drag_dest_set(const ArrayHandle_TargetEntry& targets,
    GtkDestDefaults flags, GdkDragAction actions);
```

- `flags` is an ORed combination of values which indicates how the widget will respond visually to Drag and Drop items.
- `actions` indicates the Drag and Drop actions which this destination can receive - see the description above.

18.3. Signals

When a drop destination has accepted a dragged item, certain signals will be emitted, depending on what action has been selected. For instance, the user might have held down the **Shift** key to specify a `move` rather than a `copy`. Remember that the user can only select the actions which you have specified in your calls to `drag_dest_set()` and `drag_source_set()`.

18.3.1. Copy

The source widget will emit these signals, in this order:

- `drag_begin`: Provides DragContext.
- `drag_motion`: Provides DragContext and coordinates. You can call the drag_status() method of the DragContext to indicate which target will be accepted.
- `drag_get`: Provides `info` about the dragged data format, and a `GtkSelectionData` structure, in which you should put the requested data.
- `drag_drop`: Provides DragContext and coordinates.
- `drag_end`: Provides DragContext.

The destination widget will emit this signal, after the source destination has emitted the `drag_get` signal:

- `drag_data_received`: Provides `info` about the dragged data format, and a `GtkSelectionData` structure which contains the dropped data. You should call the `drag_finish()` method of the `DragContext` to indicate whether the operation was successful.

18.3.2. Move

During a `move`, the source widget will also emit this signal:

- `drag_delete`: Gives the source the opportunity to delete the original data if that's appropriate.

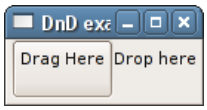
18.4. DragContext

The drag and drop signals provide a DragContext, which contains some information about the drag and drop operation and can be used to influence the process. For instance, you can discover the source widget, or change the drag and drop icon, by using the `set_icon()` methods. More importantly, you should call the `drag_finish()` method from your `drag_data_received` signal handler to indicate whether the drop was successful.

18.5. Example

Here is a very simple example, demonstrating a drag and drop `Copy` operation:

Figure 18-1 **Drag and Drop**



Source Code

File: `dndwindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLE_DNDWINDOW_H
#define GTKMM_EXAMPLE_DNDWINDOW_H

#include <gtkmm/box.h>
#include <gtkmm/label.h>
#include <gtkmm/window.h>
#include <gtkmm/button.h>

class DnDWindow : public Gtk::Window
{

public:
  DnDWindow();
  virtual ~DnDWindow();

protected:
  //Signal handlers:
  void on_button_drag_data_get(
          const Glib::RefPtr<Gdk::DragContext>& context,
          Gtk::SelectionData& selection_data, guint info, guint time);
  void on_label_drop_drag_data_received(
          const Glib::RefPtr<Gdk::DragContext>& context, int x, int y,
          const Gtk::SelectionData& selection_data, guint info, guint time);

  //Member widgets:
  Gtk::Box m_HBox;
  Gtk::Button m_Button_Drag;
  Gtk::Label m_Label_Drop;
};

#endif // GTKMM_EXAMPLE_DNDWINDOW_H
```

File: `dndwindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "dndwindow.h"
#include <iostream>

DnDWindow::DnDWindow()
: m_Button_Drag("Drag Here\n"),
  m_Label_Drop("Drop here\n")
{
  set_title("DnD example");

  add(m_HBox);

  //Targets:
  std::vector<Gtk::TargetEntry> listTargets;
  listTargets.push_back( Gtk::TargetEntry("STRING") );
  listTargets.push_back( Gtk::TargetEntry("text/plain") );

  //Drag site:

  //Make m_Button_Drag a DnD drag source:
  m_Button_Drag.drag_source_set(listTargets);

  //Connect signals:
  m_Button_Drag.signal_drag_data_get().connect(sigc::mem_fun(*this,
              &DnDWindow::on_button_drag_data_get));

  m_HBox.pack_start(m_Button_Drag);

  //Drop site:

  //Make m_Label_Drop a DnD drop destination:
  m_Label_Drop.drag_dest_set(listTargets);

  //Connect signals:
  m_Label_Drop.signal_drag_data_received().connect(sigc::mem_fun(*this,
              &DnDWindow::on_label_drop_drag_data_received) );

  m_HBox.pack_start(m_Label_Drop);
```

```
  show_all();
}

DnDWindow::~DnDWindow()
{
}

void DnDWindow::on_button_drag_data_get(
        const Glib::RefPtr<Gdk::DragContext>&,
        Gtk::SelectionData& selection_data, guint, guint)
{
  selection_data.set(selection_data.get_target(), 8 /* 8 bits format */,
          (const guchar*)"I'm Data!",
          9 /* the length of I'm Data! in bytes */);
}

void DnDWindow::on_label_drop_drag_data_received(
        const Glib::RefPtr<Gdk::DragContext>& context, int, int,
        const Gtk::SelectionData& selection_data, guint, guint time)
{
  const int length = selection_data.get_length();
  if((length >= 0) && (selection_data.get_format() == 8))
  {
    std::cout << "Received \"" << selection_data.get_data_as_string()
        << "\" in label " << std::endl;
  }

  context->drag_finish(false, false, time);
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "dndwindow.h"
#include <gtkmm/main.h>

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

  DnDWindow dndWindow;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(dndWindow);

  return 0;
}
```

There is a more complex example in examples/dnd.

19. The Clipboard

Simple text copy-paste functionality is provided for free by widgets such as Gtk::Entry and Gtk::TextView, but you might need special code to deal with your own data formats. For instance, a drawing program would need special code to allow copy and paste within a view, or between documents.

`Gtk::Clipboard` is a singleton. You can get the one and only instance with `Gtk::Clipboard::get()`.

So your application doesn't need to wait for clipboard operations, particularly between the time when the user chooses Copy and then later chooses Paste, most `Gtk::Clipboard` methods take `sigc::slot`s which specify callback methods. When `Gtk::Clipboard` is ready, it will call these methods, either providing the requested data, or asking for data.

Reference

19.1. Targets

Different applications contain different types of data, and they might make that data

available in a variety of formats. *gtkmm* calls these data types `target`s.

For instance, gedit can supply and receive the `"UTF8_STRING"` target, so you can paste data into gedit from any application that supplies that target. Or two different image editing applications might supply and receive a variety of image formats as targets. As long as one application can receive one of the targets that the other supplies then you will be able to copy data from one to the other.

A target can be in a variety of binary formats. This chapter, and the examples, assume that the data is 8-bit text. This would allow us to use an XML format for the clipboard data. However this would probably not be appropriate for binary data such as images. `Gtk::Clipboard` provides overloads that allow you to specify the format in more detail if necessary.

The Drag and Drop API uses the same mechanism. You should probably use the same data targets and formats for both Clipboard and Drag and Drap operations.

19.2. Copy

When the user asks to copy some data, you should tell the `Clipboard` what targets are available, and provide the callback methods that it can use to get the data. At this point you should store a copy of the data, to be provided when the clipboard calls your callback method in repsonse to a paste.

For instance,

```
Glib::RefPtr<Gtk::Clipboard> refClipboard = Gtk::Clipboard::get();

//Targets:
std::list<Gtk::TargetEntry> listTargets;
listTargets.push_back( Gtk::TargetEntry("example_custom_target") );
listTargets.push_back( Gtk::TargetEntry("UTF8_STRING") );

refClipboard->set( listTargets,
    sigc::mem_fun(*this, &ExampleWindow::on_clipboard_get),
    sigc::mem_fun(*this, &ExampleWindow::on_clipboard_clear) );
```

Your callback will then provide the store data when the user chooses to paste the data. For instance:

```
void ExampleWindow::on_clipboard_get(
    Gtk::SelectionData& selection_data, guint info)
{
  const Glib::ustring target = selection_data.get_target();

  if(target == "example_custom_target")
    selection_data.set("example_custom_target", m_ClipboardStore);
}
```

The `ideal` example below can supply more than one clipboard target.

The clear callback allows you to free the memory used by your stored data when the clipboard replaces its data with something else.

19.3. Paste

When the user asks to paste data from the `Clipboard`, you should request a specific format and provide a callback method which will be called with the actual data. For instance:

```
refClipboard->request_contents("example_custom_target",
    sigc::mem_fun(*this, &ExampleWindow::on_clipboard_received) );
```

Here is an example callback method:

```
void ExampleWindow::on_clipboard_received(
    const Gtk::SelectionData& selection_data)
{
  Glib::ustring clipboard_data = selection_data.get_data_as_string();
```

```
  //Do something with the pasted data.
}
```


19.3.1. Discovering the available targets

To find out what targets are currently available on the `Clipboard` for pasting, call the `request_targets()` method, specifying a method to be called with the information. For instance:

```
refClipboard->request_targets( sigc::mem_fun(*this,
    &ExampleWindow::on_clipboard_received_targets) );
```

In your callback, compare the list of available targets with those that your application supports for pasting. You could enable or disable a Paste menu item, depending on whether pasting is currently possible. For instance:

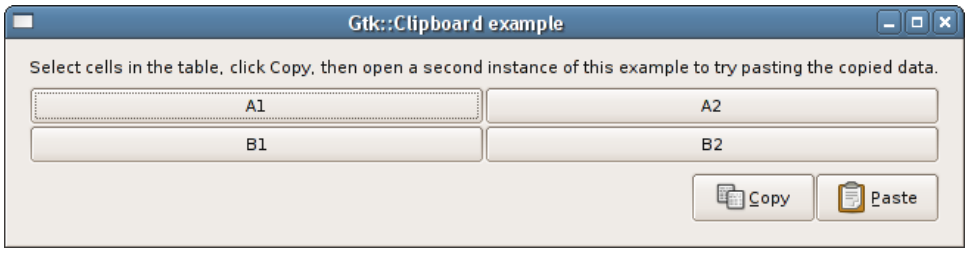
```
void ExampleWindow::on_clipboard_received_targets(
  const Glib::StringArrayHandle& targets_array)
{
  // Get the list of available clipboard targets:
  std::list<std::string> targets = targets_array;

  const bool bPasteIsPossible =
    std::find(targets.begin(), targets.end(),
      example_target_custom) != targets.end();

  // Enable/Disable the Paste button appropriately:
  m_Button_Paste.set_sensitive(bPasteIsPossible);
}
```

19.4. Examples

19.4.1. Simple

This example allows copy and pasting of application-specific data, using the standard text target. Although this is simple, it's not ideal because it does not identify the `Clipboard` data as being of a particular type.

Figure 19-1 **Clipboard - Simple**



Source Code

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <gtkmm.h>

class ExampleWindow : public Gtk::Window
{
```

```
public:
  ExampleWindow();
  virtual ~ExampleWindow();

protected:
  //Signal handlers:
  void on_button_copy();
  void on_button_paste();
  void on_clipboard_text_received(const Glib::ustring& text);

  //Child widgets:
  Gtk::Box m_VBox;

  Gtk::Label m_Label;

  Gtk::Table m_Table;
  Gtk::ToggleButton m_ButtonA1, m_ButtonA2, m_ButtonB1, m_ButtonB2;

  Gtk::ButtonBox m_ButtonBox;
  Gtk::Button m_Button_Copy, m_Button_Paste;
};

#endif //GTKMM_EXAMPLEWINDOW_H
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"

ExampleWindow::ExampleWindow()
: m_VBox(Gtk::ORIENTATION_VERTICAL),
  m_Label("Select cells in the table, click Copy, then open a second "
        "instance of this example to try pasting the copied data."),
  m_Table(2, 2, true),
  m_ButtonA1("A1"), m_ButtonA2("A2"), m_ButtonB1("B1"), m_ButtonB2("B2"),
  m_Button_Copy(Gtk::Stock::COPY), m_Button_Paste(Gtk::Stock::PASTE)
{
  set_title("Gtk::Clipboard example");
  set_border_width(12);

  add(m_VBox);

  m_VBox.pack_start(m_Label, Gtk::PACK_SHRINK);

  //Fill Table:
  m_VBox.pack_start(m_Table);
  m_Table.attach(m_ButtonA1, 0, 1, 0, 1);
  m_Table.attach(m_ButtonA2, 1, 2, 0, 1);
  m_Table.attach(m_ButtonB1, 0, 1, 1, 2);
  m_Table.attach(m_ButtonB2, 1, 2, 1, 2);

  //Add ButtonBox to bottom:
  m_VBox.pack_start(m_ButtonBox, Gtk::PACK_SHRINK);
  m_VBox.set_spacing(6);

  //Fill ButtonBox:
  m_ButtonBox.set_layout(Gtk::BUTTONBOX_END);
  m_ButtonBox.pack_start(m_Button_Copy, Gtk::PACK_SHRINK);
  m_Button_Copy.signal_clicked().connect(sigc::mem_fun(*this,
              &ExampleWindow::on_button_copy) );
  m_ButtonBox.pack_start(m_Button_Paste, Gtk::PACK_SHRINK);
  m_Button_Paste.signal_clicked().connect(sigc::mem_fun(*this,
              &ExampleWindow::on_button_paste) );

  show_all_children();
}

ExampleWindow::~ExampleWindow()
{
}

void ExampleWindow::on_button_copy()
{
  //Build a string representation of the stuff to be copied:
  //Ideally you would use XML, with an XML parser here:
  Glib::ustring strData;
  strData += m_ButtonA1.get_active() ? "1" : "0";
  strData += m_ButtonA2.get_active() ? "1" : "0";
  strData += m_ButtonB1.get_active() ? "1" : "0";
  strData += m_ButtonB2.get_active() ? "1" : "0";

  Glib::RefPtr<Gtk::Clipboard> refClipboard = Gtk::Clipboard::get();
  refClipboard->set_text(strData);
}

void ExampleWindow::on_button_paste()
```

```
{
  //Tell the clipboard to call our method when it is ready:
  Glib::RefPtr<Gtk::Clipboard> refClipboard = Gtk::Clipboard::get();
  refClipboard->request_text(sigc::mem_fun(*this,
              &ExampleWindow::on_clipboard_text_received) );
}

void ExampleWindow::on_clipboard_text_received(const Glib::ustring& text)
{
  //See comment in on_button_copy() about this silly clipboard format.
  if(text.size() >= 4)
  {
    m_ButtonA1.set_active( text[0] == '1' );
    m_ButtonA2.set_active( text[1] == '1' );
    m_ButtonB1.set_active( text[2] == '1' );
    m_ButtonB2.set_active( text[3] == '1' );
  }
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

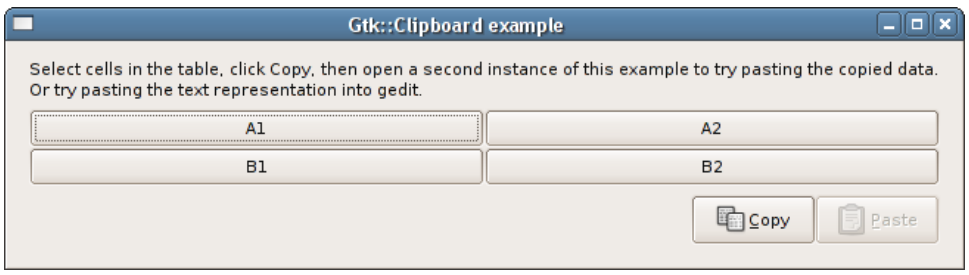
```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

19.4.2. Ideal

This is like the simple example, but it

1. Defines a custom clipboard target, though the format of that target is still text.
2. It supports pasting of 2 targets - both the custom one and a text one that creates an arbitrary text representation of the custom data.
3. It uses `request_targets()` and the `owner_change` signal and disables the Paste button if it can't use anything on the clipboard.

Figure 19-2 **Clipboard - Ideal**



Source Code

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <gtkmm.h>

class ExampleWindow : public Gtk::Window
{
public:
  ExampleWindow();
  virtual ~ExampleWindow();
```

```
protected:
  //Signal handlers:
  void on_button_copy();
  void on_button_paste();

  void on_clipboard_owner_change(GdkEventOwnerChange* event);
  void on_clipboard_get(Gtk::SelectionData& selection_data, guint info);
  void on_clipboard_clear();

  void on_clipboard_received(const Gtk::SelectionData& selection_data);
  void on_clipboard_received_targets(const Glib::StringArrayHandle& targets_array);

  virtual void update_paste_status(); //Disable the paste button if there is nothing to paste.

  //Child widgets:
  Gtk::Box m_VBox;

  Gtk::Label m_Label;

  Gtk::Table m_Table;
  Gtk::ToggleButton m_ButtonA1, m_ButtonA2, m_ButtonB1, m_ButtonB2;

  Gtk::ButtonBox m_ButtonBox;
  Gtk::Button m_Button_Copy, m_Button_Paste;

  Glib::ustring m_ClipboardStore; //Keep copied stuff here, until it is pasted. This could be a big complex data structure.
};

#endif //GTKMM_EXAMPLEWINDOW_H
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <algorithm>

namespace
{

//These should usually be MIME types.
const char example_target_custom[] = "gtkmmclipboardexample";
const char example_target_text[]   = "UTF8_STRING";

} // anonymous namespace


ExampleWindow::ExampleWindow()
: m_VBox(Gtk::ORIENTATION_VERTICAL),
  m_Label("Select cells in the table, click Copy, then open a second instance "
          "of this example to try pasting the copied data.\nOr try pasting the "
          "text representation into gedit."),
  m_Table(2, 2, true),
  m_ButtonA1("A1"), m_ButtonA2("A2"), m_ButtonB1("B1"), m_ButtonB2("B2"),
  m_Button_Copy(Gtk::Stock::COPY), m_Button_Paste(Gtk::Stock::PASTE)
{
  set_title("Gtk::Clipboard example");
  set_border_width(12);

  add(m_VBox);

  m_VBox.pack_start(m_Label, Gtk::PACK_SHRINK);

  //Fill Table:
  m_VBox.pack_start(m_Table);
  m_Table.attach(m_ButtonA1, 0, 1, 0, 1);
  m_Table.attach(m_ButtonA2, 1, 2, 0, 1);
  m_Table.attach(m_ButtonB1, 0, 1, 1, 2);
  m_Table.attach(m_ButtonB2, 1, 2, 1, 2);

  //Add ButtonBox to bottom:
  m_VBox.pack_start(m_ButtonBox, Gtk::PACK_SHRINK);
  m_VBox.set_spacing(6);

  //Fill ButtonBox:
  m_ButtonBox.set_layout(Gtk::BUTTONBOX_END);
  m_ButtonBox.pack_start(m_Button_Copy, Gtk::PACK_SHRINK);
  m_Button_Copy.signal_clicked().connect(sigc::mem_fun(*this,
              &ExampleWindow::on_button_copy) );
  m_ButtonBox.pack_start(m_Button_Paste, Gtk::PACK_SHRINK);
  m_Button_Paste.signal_clicked().connect(sigc::mem_fun(*this,
              &ExampleWindow::on_button_paste) );

  //Connect a signal handler that will be called when the contents of
  //the clipboard change.
  Gtk::Clipboard::get()->signal_owner_change().connect(sigc::mem_fun(*this,
              &ExampleWindow::on_clipboard_owner_change) );
```

```
  show_all_children();

  update_paste_status();
}

ExampleWindow::~ExampleWindow()
{
}

void ExampleWindow::on_button_copy()
{
  //Build a string representation of the stuff to be copied:
  //Ideally you would use XML, with an XML parser here:
  Glib::ustring strData;
  strData += m_ButtonA1.get_active() ? "1" : "0";
  strData += m_ButtonA2.get_active() ? "1" : "0";
  strData += m_ButtonB1.get_active() ? "1" : "0";
  strData += m_ButtonB2.get_active() ? "1" : "0";

  Glib::RefPtr<Gtk::Clipboard> refClipboard = Gtk::Clipboard::get();

  //Targets:
  std::vector<Gtk::TargetEntry> listTargets;

  listTargets.push_back( Gtk::TargetEntry(example_target_custom) );
  listTargets.push_back( Gtk::TargetEntry(example_target_text) );

  refClipboard->set(listTargets, sigc::mem_fun(*this,
              &ExampleWindow::on_clipboard_get), sigc::mem_fun(*this,
                  &ExampleWindow::on_clipboard_clear) );

  //Store the copied data until it is pasted:
  //(Must be done after the call to refClipboard->set, because that call
  //may trigger a call to on_clipboard_clear.)
  m_ClipboardStore = strData;

  update_paste_status();
}

void ExampleWindow::on_button_paste()
{
  //Tell the clipboard to call our method when it is ready:
  Glib::RefPtr<Gtk::Clipboard> refClipboard = Gtk::Clipboard::get();

  refClipboard->request_contents(example_target_custom,
    sigc::mem_fun(*this, &ExampleWindow::on_clipboard_received) );

  update_paste_status();
}

void ExampleWindow::on_clipboard_owner_change(GdkEventOwnerChange*)
{
  update_paste_status();
}

void ExampleWindow::on_clipboard_get(Gtk::SelectionData& selection_data, guint)
{
  //info is meant to indicate the target, but it seems to be always 0,
  //so we use the selection_data's target instead.

  const std::string target = selection_data.get_target();

  if(target == example_target_custom)
  {
    // This set() override uses an 8-bit text format for the data.
    selection_data.set(example_target_custom, m_ClipboardStore);
  }
  else if(target == example_target_text)
  {
    //Build some arbitrary text representation of the data,
    //so that people see something when they paste into a text editor:
    Glib::ustring text_representation;

    text_representation += m_ButtonA1.get_active() ? "A1, " : "";
    text_representation += m_ButtonA2.get_active() ? "A2, " : "";
    text_representation += m_ButtonB1.get_active() ? "B1, " : "";
    text_representation += m_ButtonB2.get_active() ? "B2, " : "";

    selection_data.set_text(text_representation);
  }
  else
  {
    g_warning("ExampleWindow::on_clipboard_get(): "
            "Unexpected clipboard target format.");
  }
}
```

```
void ExampleWindow::on_clipboard_clear()
{
  //This isn't really necessary. I guess it might save memory.
  m_ClipboardStore.clear();
}

void ExampleWindow::on_clipboard_received(
        const Gtk::SelectionData& selection_data)
{
  const std::string target = selection_data.get_target();

  //It should always be this, because that' what we asked for when calling
  //request_contents().
  if(target == example_target_custom)
  {
    Glib::ustring clipboard_data = selection_data.get_data_as_string();

    //See comment in on_button_copy() about this silly clipboard format.
    if(clipboard_data.size() >= 4)
    {
      m_ButtonA1.set_active( clipboard_data[0] == '1' );
      m_ButtonA2.set_active( clipboard_data[1] == '1' );
      m_ButtonB1.set_active( clipboard_data[2] == '1' );
      m_ButtonB2.set_active( clipboard_data[3] == '1' );
    }
  }
}

void ExampleWindow::update_paste_status()
{
  //Disable the paste button if there is nothing to paste.

  Glib::RefPtr<Gtk::Clipboard> refClipboard = Gtk::Clipboard::get();

  //Discover what targets are available:
  refClipboard->request_targets(sigc::mem_fun(*this,
              &ExampleWindow::on_clipboard_received_targets) );
}

void ExampleWindow::on_clipboard_received_targets(
  const Glib::StringArrayHandle& targets_array)
{
  // Get the list of available clipboard targets:
  std::vector<std::string> targets = targets_array;

  const bool bPasteIsPossible =
    std::find(targets.begin(), targets.end(),
      example_target_custom) != targets.end();

  // Enable/Disable the Paste button appropriately:
  m_Button_Paste.set_sensitive(bPasteIsPossible);
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

20. Printing

At the application development level, *gtkmm*'s printing API provides dialogs that are consistent across applications and allows us of Cairo's common drawing API, with Pango-driven text rendering. In the implementation of this common API, platform-specific backends and printer-specific drivers are used.

20.1. PrintOperation

The primary object is `Gtk::PrintOperation`, allocated for each print operation. To handle page drawing connect to its signals, or inherit from it and override the default virtual signal handlers. `PrintOperation` automatically handles all the settings affecting the print loop.

20.1.1. Signals

The `PrintOperation::run()` method starts the print loop, during which various signals are emitted:

- `begin_print`: You must handle this signal, because this is where you create and set up a `Pango::Layout` using the provided `Gtk::PrintContext`, and break up your printing output into pages.
- `paginate`: Pagination is potentially slow so if you need to monitor it you can call the `PrintOperation::set_show_progress()` method and handle this signal.
- For each page that needs to be rendered, the following signals are emitted:
  - `request_page_setup`: Provides a `PrintContext`, page number and `Gtk::PageSetup`. Handle this signal if you need to modify page setup on a per-page basis.
  - `draw_page`: You must handle this signal, which provides a `PrintContext` and a page number. The `PrintContext` should be used to create a `Cairo::Context` into which the provided page should be drawn. To render text, iterate over the `Pango::Layout` you created in the `begin_print` handler.
- `end_print`: A handler for it is a safe place to free any resources related to a `PrintOperation`. If you have your custom class that inherits from `PrintOperation`, it is naturally simpler to do it in the destructor.
- `done`: This signal is emitted when printing is finished, meaning when the print data is spooled. Note that the provided `Gtk::PrintOperationResult` may indicate that an error occurred. In any case you probably want to notify the user about the final status.
- `status_changed`: Emitted whenever a print job's status changes, until it is finished. Call the `PrintOperation::set_track_print_status()` method to monitor the job status after spooling. To see the status, use `get_status()` or `get_status_string()`.

Reference

20.2. Page setup

The `PrintOperation` class has a method called `set_default_page_setup()` which selects the default paper size, orientation and margins. To show a page setup dialog from your application, use the `Gtk::run_page_setup_dialog()` method, which returns a `Gtk::PageSetup` object with the chosen settings. Use this object to update a `PrintOperation` and to access the selected `Gtk::PaperSize`, `Gtk::PageOrientation` and printer-specific margins.

You should save the chosen `Gtk::PageSetup` so you can use it again if the page setup dialog is shown again.

For instance,

```
//Within a class that inherits from Gtk::Window and keeps m_refPageSetup and m_refSettings as members...
```

```
Glib::RefPtr<Gtk::PageSetup> new_page_setup = Gtk::run_page_setup_dialog(*this, m_refPageSetup, m_refSettings);
m_refPageSetup = new_page_setup;
```

Reference

The Cairo coordinate system, in the `draw_page` handler, is automatically rotated to the current page orientation. It is normally within the printer margins, but you can change that via the `PrintOperation::set_use_full_page()` method. The default measurement unit is device pixels. To select other units, use the `PrintOperation::set_unit()` method.

20.3. Rendering text

Text rendering is done using Pango. The `Pango::Layout` object for printing should be created by calling the `PrintContext::create_pango_layout()` method. The `PrintContext` object also provides the page metrics, via `get_width()` and `get_height()`. The number of pages can be set with `PrintOperation::set_n_pages()`. To actually render the Pango text in `on_draw_page`, get a `Cairo::Context` with `PrintContext::get_cairo_context()` and show the `Pango::LayoutLine`s that appear within the requested page number.

See an example of exactly how this can be done.

20.4. Asynchronous operations

By default, `PrintOperation::run()` returns when a print operation is completed. If you need to run a non-blocking print operation, call `PrintOperation::set_allow_async()`. Note that `set_allow_async()` is not supported on all platforms, however the `done` signal will still be emitted.

`run()` may return `PRINT_OPERATION_RESULT_IN_PROGRESS`. To track status and handle the result or error you need to implement signal handlers for the `done` and `status_changed` signals:

For instance,

```
// in class ExampleWindow's method...
Glib::RefPtr<PrintOperation> op = PrintOperation::create();
// ...set up op...
op->signal_done().connect(sigc::bind(sigc::mem_fun(*this, &ExampleWindow::on_printoperation_done), op));
// run the op
```

Second, check for an error and connect to the `status_changed` signal. For instance:

```
void ExampleWindow::on_printoperation_done(Gtk::PrintOperationResult result, const Glib::RefPtr<PrintOperation>& op)
{
  if (result == Gtk::PRINT_OPERATION_RESULT_ERROR)
    //notify user
  else if (result == Gtk::PRINT_OPERATION_RESULT_APPLY)
    //Update PrintSettings with the ones used in this PrintOperation

  if (! op->is_finished())
    op->signal_status_changed().connect(sigc::bind(sigc::mem_fun(*this, &ExampleWindow::on_printoperation_status_changed), op));
}
```

Finally, check the status. For instance,

```
void ExampleWindow::on_printoperation_status_changed(const Glib::RefPtr<PrintFormOperation>& op)
{
  if (op->is_finished())
    //the print job is finished
  else
    //get the status with get_status() or get_status_string()

  //update UI
}
```

20.5. Export to PDF

The 'Print to file' option is available in the print dialog, without the need for extra implementation. However, it is sometimes useful to generate a pdf file directly from code. For instance,

```
Glib::RefPtr<Gtk::PrintOperation> op = Gtk::PrintOperation::create();
// ...set up op...
op->set_export_filename("test.pdf");
Gtk::PrintOperationResult res = op->run(Gtk::PRINT_OPERATION_ACTION_EXPORT);
```

20.6. Extending the print dialog

You may add a custom tab to the print dialog:

- Set the title of the tab via `PrintOperation::set_custom_tab_label()`, create a new widget and return it from the `create_custom_widget` signal handler. You'll probably want this to be a container widget, packed with some others.
- Get the data from the widgets in the `custom_widget_apply` signal handler.

Although the `custom_widget_apply` signal provides the widget you previously created, to simplify things you can keep the widgets you expect to contain some user input as class members. For example, let's say you have a `Gtk::Entry` called `m_Entry` as a member of your `CustomPrintOperation` class:

```
Gtk::Widget* CustomPrintOperation::on_create_custom_widget()
{
  set_custom_tab_label("My custom tab");

  Gtk::HBox* hbox = new Gtk::HBox(false, 8);
  hbox->set_border_width(6);

  Gtk::Label* label = Gtk::manage(new Gtk::Label("Enter some text: "));
  hbox->pack_start(*label, false, false);
  label->show();

  hbox->pack_start(m_Entry, false, false);
  m_Entry.show();

  return hbox;
}

void CustomPrintOperation::on_custom_widget_apply(Gtk::Widget* /* widget */)
{
  Glib::ustring user_input = m_Entry.get_text();
  //...
}
```

The example in examples/book/printing/advanced demonstrates this.

20.7. Preview

The native GTK+ print dialog has a preview button, but you may also start a preview directly from an application:

```
// in a class that inherits from Gtk::Window...
Glib::RefPtr<PrintOperation> op = PrintOperation::create();
// ...set up op...
op->run(Gtk::PRINT_OPERATION_ACTION_PREVIEW, *this);
```

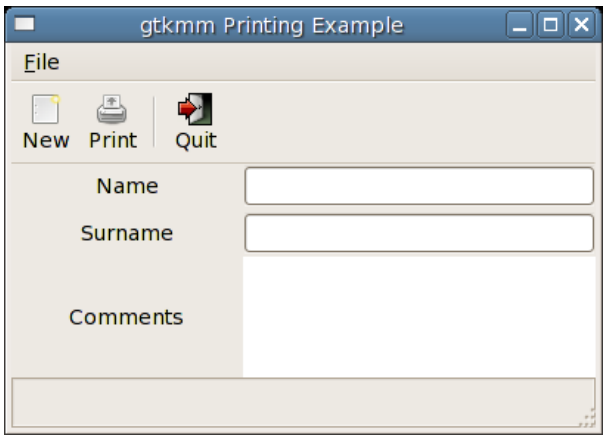
On Unix, the default preview handler uses an external viewer program. On Windows, the native preview dialog will be shown. If necessary you may override this behaviour and provide a custom preview dialog. See the example located in /examples /book/printing/advanced.

20.8. Example

20.8.1. Simple

The following example demonstrates how to print some input from a user interface. It shows how to implement `on_begin_print` and `on_draw_page`, as well as how to track print status and update the print settings.

Figure 20-1 **Printing - Simple**



Source Code

File: `printformoperation.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_PRINT_FORM_OPERATION_H
#define GTKMM_PRINT_FORM_OPERATION_H

#include <pangomm.h>
#include <gtkmm.h>
#include <vector>

//We derive our own class from PrintOperation,
//so we can put the actual print implementation here.
class PrintFormOperation : public Gtk::PrintOperation
{
 public:
  static Glib::RefPtr<PrintFormOperation> create();
  virtual ~PrintFormOperation();

  void set_name(const Glib::ustring& name) { m_Name = name; }
  void set_comments(const Glib::ustring& comments) { m_Comments = comments; }

 protected:
  PrintFormOperation();

  //PrintOperation default signal handler overrides:
  virtual void on_begin_print(const Glib::RefPtr<Gtk::PrintContext>& context);
  virtual void on_draw_page(const Glib::RefPtr<Gtk::PrintContext>& context, int page_nr);

  Glib::ustring m_Name;
  Glib::ustring m_Comments;
  Glib::RefPtr<Pango::Layout> m_refLayout;
  std::vector<int> m_PageBreaks; // line numbers where a page break occurs
};

#endif // GTKMM_PRINT_FORM_OPERATION_H
```

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <memory>
#include <vector>

#include <pangomm.h>
#include <gtkmm.h>

class PrintFormOperation;

class ExampleWindow : public Gtk::Window
{
public:
  ExampleWindow();
  virtual ~ExampleWindow();

protected:
```

```
  void build_main_menu();

  void print_or_preview(Gtk::PrintOperationAction print_action);

  //PrintOperation signal handlers.
  //We handle these so can get necessary information to update the UI or print settings.
  //Our derived PrintOperation class also overrides some default signal handlers.
  void on_printoperation_status_changed(const Glib::RefPtr<PrintFormOperation>& operation);

  void on_printoperation_done(Gtk::PrintOperationResult result, const Glib::RefPtr<PrintFormOperation>& operation);

  //Action signal handlers:
  void on_menu_file_new();
  void on_menu_file_page_setup();
  void on_menu_file_print_preview();
  void on_menu_file_print();
  void on_menu_file_quit();

  //Printing-related objects:
  Glib::RefPtr<Gtk::PageSetup> m_refPageSetup;
  Glib::RefPtr<Gtk::PrintSettings> m_refSettings;


  //Child widgets:
  Gtk::Box m_VBox;
  Gtk::Table m_Table;

  Gtk::Label m_NameLabel;
  Gtk::Entry m_NameEntry;

  Gtk::Label m_SurnameLabel;
  Gtk::Entry m_SurnameEntry;

  Gtk::Label m_CommentsLabel;
  Gtk::ScrolledWindow m_ScrolledWindow;
  Gtk::TextView m_TextView;

  Glib::RefPtr<Gtk::TextBuffer> m_refTextBuffer;

  unsigned m_ContextId;
  Gtk::Statusbar m_Statusbar;

  Glib::RefPtr<Gtk::UIManager> m_refUIManager;
  Glib::RefPtr<Gtk::ActionGroup> m_refActionGroup;
};

#endif //GTKMM_EXAMPLEWINDOW_H
```

File: `printformoperation.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "printformoperation.h"

PrintFormOperation::PrintFormOperation()
{
}

PrintFormOperation::~PrintFormOperation()
{
}

Glib::RefPtr<PrintFormOperation> PrintFormOperation::create()
{
  return Glib::RefPtr<PrintFormOperation>(new PrintFormOperation());
}

void PrintFormOperation::on_begin_print(
        const Glib::RefPtr<Gtk::PrintContext>& print_context)
{
  //Create and set up a Pango layout for PrintData based on the passed
  //PrintContext: We then use this to calculate the number of pages needed, and
  //the lines that are on each page.
  m_refLayout = print_context->create_pango_layout();

  Pango::FontDescription font_desc("sans 12");
  m_refLayout->set_font_description(font_desc);

  const double width = print_context->get_width();
  const double height = print_context->get_height();

  m_refLayout->set_width(static_cast<int>(width * Pango::SCALE));

  //Set and mark up the text to print:
  Glib::ustring marked_up_form_text;
  marked_up_form_text += "<b>Name</b>: " + m_Name + "\n\n";
  marked_up_form_text += "<b>Comments</b>: " + m_Comments;
```

```
  m_refLayout->set_markup(marked_up_form_text);

  //Set the number of pages to print by determining the line numbers
  //where page breaks occur:
  const int line_count = m_refLayout->get_line_count();

  Glib::RefPtr<Pango::LayoutLine> layout_line;
  double page_height = 0;

  for (int line = 0; line < line_count; ++line)
  {
    Pango::Rectangle ink_rect, logical_rect;

    layout_line = m_refLayout->get_line(line);
    layout_line->get_extents(ink_rect, logical_rect);

    const double line_height = logical_rect.get_height() / 1024.0;

    if (page_height + line_height > height)
    {
      m_PageBreaks.push_back(line);
      page_height = 0;
    }

    page_height += line_height;
  }

  set_n_pages(m_PageBreaks.size() + 1);
}

void PrintFormOperation::on_draw_page(
        const Glib::RefPtr<Gtk::PrintContext>& print_context, int page_nr)
{
  //Decide which lines we need to print in order to print the specified page:
  int start_page_line = 0;
  int end_page_line = 0;

  if(page_nr == 0)
  {
    start_page_line = 0;
  }
  else
  {
    start_page_line = m_PageBreaks[page_nr - 1];
  }

  if(page_nr < static_cast<int>(m_PageBreaks.size()))
  {
    end_page_line = m_PageBreaks[page_nr];
  }
  else
  {
    end_page_line = m_refLayout->get_line_count();
  }

  //Get a Cairo Context, which is used as a drawing board:
  Cairo::RefPtr<Cairo::Context> cairo_ctx = print_context->get_cairo_context();

  //We'll use black letters:
  cairo_ctx->set_source_rgb(0, 0, 0);

  //Render Pango LayoutLines over the Cairo context:
  Pango::LayoutIter iter = m_refLayout->get_iter();

  double start_pos = 0;
  int line_index = 0;

  do
  {
    if(line_index >= start_page_line)
    {
      Glib::RefPtr<Pango::LayoutLine> layout_line = iter.get_line();
      Pango::Rectangle logical_rect = iter.get_line_logical_extents();
      int baseline = iter.get_baseline();

      if (line_index == start_page_line)
      {
        start_pos = logical_rect.get_y() / 1024.0;
      }

      cairo_ctx->move_to(logical_rect.get_x() / 1024.0,
        baseline / 1024.0 - start_pos);

      layout_line->show_in_cairo_context(cairo_ctx);
    }

    line_index++;
```

```
  }
  while(line_index < end_page_line && iter.next_line());
}
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include "printformoperation.h"

#include <iostream>

#include <pangomm.h>

const Glib::ustring app_title = "gtkmm Printing Example";

ExampleWindow::ExampleWindow()
: m_VBox(Gtk::ORIENTATION_VERTICAL),
  m_Table(3, 2),
  m_NameLabel("Name"),
  m_SurnameLabel("Surname"),
  m_CommentsLabel("Comments")
{
  m_refPageSetup = Gtk::PageSetup::create();
  m_refSettings = Gtk::PrintSettings::create();

  m_ContextId = m_Statusbar.get_context_id(app_title);

  set_title(app_title);
  set_default_size(400, 300);

  add(m_VBox);

  build_main_menu();

  m_VBox.pack_start(m_Table);

  //Arrange the widgets inside the table:
  m_Table.attach(m_NameLabel, 0, 1, 0, 1);
  m_Table.attach(m_NameEntry, 1, 2, 0, 1);

  m_Table.attach(m_SurnameLabel, 0, 1, 1, 2, Gtk::SHRINK);
  m_Table.attach(m_SurnameEntry, 1, 2, 1, 2);

  //Add the TreeView, inside a ScrolledWindow:
  m_ScrolledWindow.add(m_TextView);

  //Only show the scrollbars when they are necessary:
  m_ScrolledWindow.set_policy(Gtk::POLICY_AUTOMATIC, Gtk::POLICY_AUTOMATIC);

  m_Table.attach(m_CommentsLabel, 0, 1, 2, 3, Gtk::SHRINK);
  m_Table.attach(m_ScrolledWindow, 1, 2, 2, 3);

  m_refTextBuffer = Gtk::TextBuffer::create();
  m_TextView.set_buffer(m_refTextBuffer);

  m_VBox.pack_start(m_Statusbar);

  show_all_children();
}

ExampleWindow::~ExampleWindow()
{
}

void ExampleWindow::build_main_menu()
{
  //Create actions for menus and toolbars:
  m_refActionGroup = Gtk::ActionGroup::create();

  //File menu:
  m_refActionGroup->add(
    Gtk::Action::create("FileMenu", "_File"));

  m_refActionGroup->add(
    Gtk::Action::create("New", Gtk::Stock::NEW),
    sigc::mem_fun(*this, &ExampleWindow::on_menu_file_new));

  m_refActionGroup->add(
    Gtk::Action::create("PageSetup", "Page _Setup"),
    sigc::mem_fun(*this, &ExampleWindow::on_menu_file_page_setup));

  m_refActionGroup->add(
    Gtk::Action::create("PrintPreview", "Print Preview"),
    sigc::mem_fun(*this, &ExampleWindow::on_menu_file_print_preview));
```

```
  m_refActionGroup->add(
    Gtk::Action::create("Print", Gtk::Stock::PRINT),
    sigc::mem_fun(*this, &ExampleWindow::on_menu_file_print));

  m_refActionGroup->add(
    Gtk::Action::create("Quit", Gtk::Stock::QUIT),
    sigc::mem_fun(*this, &ExampleWindow::on_menu_file_quit));

  m_refUIManager = Gtk::UIManager::create();
  m_refUIManager->insert_action_group(m_refActionGroup);

  add_accel_group(m_refUIManager->get_accel_group());

  //Layout the actions in a menubar and toolbar:

  Glib::ustring ui_info =
        "<ui>"
        "  <menubar name='MenuBar'>"
        "    <menu action='FileMenu'>"
        "      <menuitem action='New'/>"
        "      <menuitem action='PageSetup'/>"
        "      <menuitem action='PrintPreview'/>"
        "      <menuitem action='Print'/>"
        "      <separator/>"
        "      <menuitem action='Quit'/>"
        "    </menu>"
        "  </menubar>"
        "  <toolbar  name='ToolBar'>"
        "    <toolitem action='New'/>"
        "    <toolitem action='Print'/>"
        "      <separator/>"
        "    <toolitem action='Quit'/>"
        "  </toolbar>"
        "</ui>";

  try
  {
    m_refUIManager->add_ui_from_string(ui_info);
  }
  catch(const Glib::Error& ex)
  {
    std::cerr << "building menus failed: " << ex.what();
  }

  //Get the menubar and toolbar widgets, and add them to a container widget:
  Gtk::Widget* pMenubar = m_refUIManager->get_widget("/MenuBar");
  if(pMenubar)
    m_VBox.pack_start(*pMenubar, Gtk::PACK_SHRINK);

  Gtk::Widget* pToolbar = m_refUIManager->get_widget("/ToolBar") ;
  if(pToolbar)
    m_VBox.pack_start(*pToolbar, Gtk::PACK_SHRINK);
}

void ExampleWindow::on_printoperation_status_changed(
        const Glib::RefPtr<PrintFormOperation>& operation)
{
  Glib::ustring status_msg;

  if (operation->is_finished())
  {
    status_msg = "Print job completed.";
  }
  else
  {
    //You could also use get_status().
    status_msg = operation->get_status_string();
  }

  m_Statusbar.push(status_msg, m_ContextId);
}

void ExampleWindow::on_printoperation_done(Gtk::PrintOperationResult result,
        const Glib::RefPtr<PrintFormOperation>& operation)
{
  //Printing is "done" when the print data is spooled.

  if (result == Gtk::PRINT_OPERATION_RESULT_ERROR)
  {
    Gtk::MessageDialog err_dialog(*this, "Error printing form", false,
            Gtk::MESSAGE_ERROR, Gtk::BUTTONS_OK, true);
    err_dialog.run();
  }
  else if (result == Gtk::PRINT_OPERATION_RESULT_APPLY)
  {
    //Update PrintSettings with the ones used in this PrintOperation:
    m_refSettings = operation->get_print_settings();
```

```
  }

  if (! operation->is_finished())
  {
    //We will connect to the status-changed signal to track status
    //and update a status bar. In addition, you can, for example,
    //keep a list of active print operations, or provide a progress dialog.
    operation->signal_status_changed().connect(sigc::bind(sigc::mem_fun(*this,
                  &ExampleWindow::on_printoperation_status_changed),
                operation));
  }
}

void ExampleWindow::print_or_preview(Gtk::PrintOperationAction print_action)
{
  //Create a new PrintOperation with our PageSetup and PrintSettings:
  //(We use our derived PrintOperation class)
  Glib::RefPtr<PrintFormOperation> print = PrintFormOperation::create();

  print->set_name(m_NameEntry.get_text() + " " + m_SurnameEntry.get_text());
  print->set_comments(m_refTextBuffer->get_text(false /*Don't include hidden*/));

  print->set_track_print_status();
  print->set_default_page_setup(m_refPageSetup);
  print->set_print_settings(m_refSettings);

  print->signal_done().connect(sigc::bind(sigc::mem_fun(*this,
                  &ExampleWindow::on_printoperation_done), print));
  try
  {
    print->run(print_action /* print or preview */, *this);
  }
  catch (const Gtk::PrintError& ex)
  {
    //See documentation for exact Gtk::PrintError error codes.
    std::cerr << "An error occurred while trying to run a print operation:"
        << ex.what() << std::endl;
  }
}

void ExampleWindow::on_menu_file_new()
{
  //Clear entries and textview:
  m_NameEntry.set_text("");
  m_SurnameEntry.set_text("");
  m_refTextBuffer->set_text("");
  m_TextView.set_buffer(m_refTextBuffer);
}

void ExampleWindow::on_menu_file_page_setup()
{
  //Show the page setup dialog, asking it to start with the existing settings:
  Glib::RefPtr<Gtk::PageSetup> new_page_setup =
      Gtk::run_page_setup_dialog(*this, m_refPageSetup, m_refSettings);

  //Save the chosen page setup dialog for use when printing, previewing, or
  //showing the page setup dialog again:
  m_refPageSetup = new_page_setup;
}

void ExampleWindow::on_menu_file_print_preview()
{
  print_or_preview(Gtk::PRINT_OPERATION_ACTION_PREVIEW);
}

void ExampleWindow::on_menu_file_print()
{
  print_or_preview(Gtk::PRINT_OPERATION_ACTION_PRINT_DIALOG);
}

void ExampleWindow::on_menu_file_quit()
{
  hide();
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;
```

```
  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

21. Recently Used Documents

gtkmm provides an easy way to manage recently used documents. The classes involved in implementing this functionality are `RecentManager`, `RecentChooserDialog`, `RecentChooserMenu`, `RecentChooserWidget`, `RecentAction`, and `RecentFilter`.

Each item in the list of recently used files is identified by its URI, and can have associated metadata. The metadata can be used to specify how the file should be displayed, a description of the file, its mime type, which application registered it, whether it's private to the registering application, and several other things.

21.1. RecentManager

`RecentManager` acts as a database of recently used files. You use this class to register new files, remove files from the list, or look up recently used files. There is one list of recently used files per user.

You can create a new `RecentManager`, but you'll most likely just want to use the default one. You can get a reference to the default `RecentManager` with `get_default()`.

`RecentManager` is the model of a model-view pattern, where the view is a class that implements the `RecentChooser` interface.

21.1.1. Adding Items to the List of Recent Files

To add a new file to the list of recent documents, in the simplest case, you only need to provide the URI. For example:

```
Glib::RefPtr<Gtk::RecentManager> recent_manager = Gtk::RecentManager::get_default();
recent_manager->add_item(uri);
```

If you want to register a file with metadata, you can pass a `RecentManager::Data` parameter to `add_item()`. The metadata that can be set on a particular file item is as follows:

- `app_exec`: The command line to be used to launch this resource. This string may contain the "f" and "u" escape characters which will be expanded to the resource file path and URI respectively
- `app_name`: The name of the application that registered the resource
- `description`: A short description of the resource as a UTF-8 encoded string
- `display_name`: The name of the resource to be used for display as a UTF-8 encoded string
- `groups`: A list of groups associated with this item. Groups are essentially arbitrary strings associated with a particular resource. They can be thought of as 'categories' (such as "email", "graphics", etc) or tags for the resource.
- `is_private`: Whether this resource should be visible only to applications that have registered it or not
- `mime_type`: The MIME type of the resource

In addition to adding items to the list, you can also look up items from the list and modify or remove items.

21.1.2. Looking up Items in the List of Recent Files

To look up recently used files, `RecentManager` provides several functions. To look up a specific item by its URI, you can use the `lookup_item()` function, which will return a `RecentInfo` class. If the specified URI did not exist in the list of recent files, `lookup_item()` throws a `RecentManagerError` exception. For example:

```
Glib::RefPtr<Gtk::RecentInfo> info;
try
{
  info = recent_manager->lookup_item(uri);
}
catch(const Gtk::RecentManagerError& ex)
{
  std::cerr << "RecentManagerError: " << ex.what() << std::endl;
}
if (info)
{
  // item was found
}
```

A `RecentInfo` object is essentially an object containing all of the metadata about a single recently-used file. You can use this object to look up any of the properties listed above.

If you don't want to look for a specific URI, but instead want to get a list of all recently used items, `RecentManager` provides the `get_items()` function. The return value of this function is a `std::vector` of all recently used files. The following code demonstrates how you might get a list of recently used files:

```
std::vector< Glib::RefPtr<Gtk::RecentInfo> > info_list = recent_manager->get_items();
```

The maximum age of items in the recently used files list can be set with `Gtk::Settings::property_gtk_recent_files_max_age()`. Default value: 30 days.

21.1.3. Modifying the List of Recent Files

There may be times when you need to modify the list of recent files. For instance, if a file is moved or renamed, you may need to update the file's location in the recent files list so that it doesn't point to an incorrect location. You can update an item's location by using `move_item()`.

In addition to changing a file's URI, you can also remove items from the list, either one at a time or by clearing them all at once. The former is accomplished with `remove_item()`, the latter with `purge_items()`.

> The functions `move_item()`, `remove_item()` and `purge_items()` have no effect on the actual files that are referred to by the URIs, they only modify the list of recent files.

21.2. RecentChooser

`RecentChooser` is an interface that can be implemented by widgets displaying the list of recently used files. *gtkmm* provides four built-in implementations for choosing recent files: `RecentChooserWidget`, `RecentChooserDialog`, `RecentChooserMenu`, and `RecentAction`.

`RecentChooserWidget` is a simple widget for displaying a list of recently used files. `RecentChooserWidget` is the basic building block for `RecentChooserDialog`, but you can embed it into your user interface if you want to.

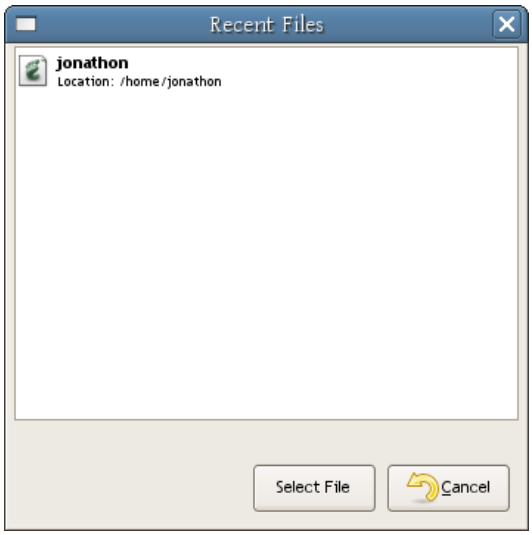
`RecentChooserMenu` and `RecentAction` allow you to list recently used files as a menu.

21.2.1. Simple RecentChooserDialog example

Shown below is a simple example of how to use the `RecentChooserDialog` and the `RecentAction` classes in a program. This simple program has a menubar with a **Recent Files Dialog** menu item. When you select this menu item, a dialog pops up showing the list of recently used files.

> If this is the first time you're using a program that uses the Recent Files framework, the dialog may be empty at first. Otherwise it should show the list of recently used documents registered by other applications.

After selecting the **Recent Files Dialog** menu item, you should see something similar to the following window.



Source Code

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <gtkmm.h>

class ExampleWindow : public Gtk::Window
{
public:
  ExampleWindow();
  virtual ~ExampleWindow();

protected:
  //Signal handlers:
  void on_menu_file_recent_files_item();
  void on_menu_file_recent_files_dialog();
  void on_menu_file_quit();
  void on_menu_file_new();

  //Child widgets:
  Gtk::Box m_Box;

  Glib::RefPtr<Gtk::UIManager> m_refUIManager;
  Glib::RefPtr<Gtk::ActionGroup> m_refActionGroup;
  Glib::RefPtr<Gtk::RecentAction> m_refRecentAction;

  Glib::RefPtr<Gtk::RecentManager> m_refRecentManager;
};

#endif //GTKMM_EXAMPLEWINDOW_H
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/stock.h>
#include <iostream>
```

```
ExampleWindow::ExampleWindow()
: m_Box(Gtk::ORIENTATION_VERTICAL),
  m_refRecentManager(Gtk::RecentManager::get_default())
{
  set_title("recent files example");
  set_default_size(200, 200);

  //We can put a MenuBar at the top of the box and other stuff below it.
  add(m_Box);

  //Create actions for menus and toolbars:
  m_refActionGroup = Gtk::ActionGroup::create();

  //File menu:
  m_refActionGroup->add( Gtk::Action::create("FileMenu", "_File") );
  m_refActionGroup->add( Gtk::Action::create("FileNew", Gtk::Stock::NEW),
    sigc::mem_fun(*this, &ExampleWindow::on_menu_file_new));

  //A recent-files submenu:
  m_refRecentAction = Gtk::RecentAction::create("FileRecentFiles", "_Recent Files");
  m_refActionGroup->add(m_refRecentAction);
  //Connect to RecentChooser's item_activated signal
  //instead of Action's activate signal:
  m_refRecentAction->signal_item_activated().connect(
    sigc::mem_fun(*this, &ExampleWindow::on_menu_file_recent_files_item) );

  //A menu item to open the recent-files dialog:
  m_refActionGroup->add( Gtk::Action::create("FileRecentDialog", "Recent Files _Dialog"),
    sigc::mem_fun(*this, &ExampleWindow::on_menu_file_recent_files_dialog) );

  m_refActionGroup->add( Gtk::Action::create("FileQuit", Gtk::Stock::QUIT),
    sigc::mem_fun(*this, &ExampleWindow::on_menu_file_quit) );

  m_refUIManager = Gtk::UIManager::create();
  m_refUIManager->insert_action_group(m_refActionGroup);

  add_accel_group(m_refUIManager->get_accel_group());

  //Layout the actions in a menubar and toolbar:
  Glib::ustring ui_info =
        "<ui>"
        "  <menubar name='MenuBar'>"
        "    <menu action='FileMenu'>"
        "      <menuitem action='FileNew'/>"
        "      <menuitem action='FileRecentFiles'/>"
        "      <menuitem action='FileRecentDialog'/>"
        "      <separator/>"
        "      <menuitem action='FileQuit'/>"
        "    </menu>"
        "  </menubar>"
        "  <toolbar  name='ToolBar'>"
        "    <toolitem action='FileNew'/>"
        "    <toolitem action='FileQuit'/>"
        "  </toolbar>"
        "</ui>";

  try
  {
    m_refUIManager->add_ui_from_string(ui_info);
  }
  catch(const Glib::Error& ex)
  {
    std::cerr << "building menus failed: " <<  ex.what();
  }

  //Get the menubar and toolbar widgets, and add them to a container widget:
  Gtk::Widget* pMenubar = m_refUIManager->get_widget("/MenuBar");
  if(pMenubar)
    m_Box.pack_start(*pMenubar, Gtk::PACK_SHRINK);

  Gtk::Widget* pToolbar = m_refUIManager->get_widget("/ToolBar");
  if(pToolbar)
    m_Box.pack_start(*pToolbar, Gtk::PACK_SHRINK);

  show_all_children();
}

ExampleWindow::~ExampleWindow()
{
}

void ExampleWindow::on_menu_file_new()
{
  std::cout << " New File" << std::endl;
}

void ExampleWindow::on_menu_file_quit()
```

```
{
  hide(); //Closes the main window to stop the Gtk::Main::run().
}

void ExampleWindow::on_menu_file_recent_files_item()
{
  std::cout << "URI selected = " << m_refRecentAction->get_current_uri() << std::endl;
}

void ExampleWindow::on_menu_file_recent_files_dialog()
{
  Gtk::RecentChooserDialog dialog(*this, "Recent Files", m_refRecentManager);
  dialog.add_button("Select File", Gtk::RESPONSE_OK);
  dialog.add_button(Gtk::Stock::CANCEL, Gtk::RESPONSE_CANCEL);

  const int response = dialog.run();
  dialog.hide();
  if(response == Gtk::RESPONSE_OK)
  {
    std::cout << "URI selected = " << dialog.get_current_uri() << std::endl;
  }
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

The constructor for `ExampleWindow` creates the menu using `UIManager` (see Chapter 12 — Menus and Toolbars for more information). It then adds the menu and the toolbar to the window.

21.2.2. Filtering Recent Files

For any of the `RecentChooser` classes, if you don't wish to display all of the items in the list of recent files, you can filter the list to show only those that you want. You can filter the list with the help of the `RecentFilter` class. This class allows you to filter recent files by their name (`add_pattern()`), their mime type (`add_mime_type()`), the application that registered them (`add_application()`), or by a custom filter function (`add_custom()`). It also provides the ability to filter based on how long ago the file was modified and which groups it belongs to.

After you've created and set up the filter to match only the items you want, you can apply a filter to a chooser widget with the `RecentChooser::add_filter()` function.

22. Plugs and Sockets

22.1. Overview

From time to time, it may be useful to be able to embed a widget from another application within your application. *gtkmm* allows you to do this with the `Gtk::Socket` and `Gtk::Plug` classes. It is not anticipated that very many applications will need this functionality, but in the rare case that you need to display a widget that is running in a completely different process, these classes can be very helpful.

The communication between a `Socket` and a `Plug` follows the XEmbed protocol. This protocol has also been implemented in other toolkits (e.g. Qt), which allows the same level of integration when embedding a Qt widget in GTK+ or vice versa.

The way that Sockets and Plugs work together is through their window ids. Both a Socket and a Plug have IDs that can be retrieved with their get_id() member functions. The use of these IDs will be explained below in Section 22.1.3 — Connecting Plugs and Sockets.

22.1.1. Sockets

A Socket is a special kind of container widget that provides the ability to embed widgets from one process into another process in a way that is transparent to the user.

22.1.2. Plugs

A Plug is a special kind of Window that can be plugged into a Socket. Besides the normal properties and methods of Gtk::Window, a Plug provides a constructor that takes the ID of a Socket, which will automatically embed the Plug into the Socket that matches that ID.

Since a Plug is just a special type of Gtk::Window class, you can add containers or widgets to it like you would to any other window.

22.1.3. Connecting Plugs and Sockets

After a Socket or Plug object is realized, you can obtain its ID with its get_id() function. This ID can then be shared with other processes so that other processes know how to connect to eachother.

There are two basic strategies that can be used:

- Create a Socket object in one process and pass the ID of that Socket to another process so that it can create a Plug object by specifying the given Socket ID in its constructor. There is no way to assign a Plug to a particular Socket after creation, so you must pass the Socket ID to the Plug's constructor.
- Create a Plug independantly from any particular Socket and pass the ID of the Plug to other processes that need to use it. The ID of the Plug can be associated with a particular Socket object using the Socket::add_id() function. This is the approach used in the example below.

22.2. Plugs and Sockets Example

The following is a simple example of using sockets and plugs. The method of communication between processes is deliberately kept very simple: The Plug writes its ID out to a text file named plug.id and the process with the socket reads the ID from this files. In a real program, you may want to use a more sophisticated method of inter-process communication.

Source Code

File: socket.cc (For use with gtkmm 3, not gtkmm 2)

```
#include <iostream>
#include <fstream>
#include <gtkmm.h>
#include <gtkmm/socket.h>

using namespace std;

const char* id_filename = "plug.id";

void plug_added()
{
  cout << "A plug was added" << endl;
}

bool plug_removed()
{
```

```
  cout << "A Plug was removed" << endl;
  return true;
}

class MySocketWindow : public Gtk::Window
{
  public:
    MySocketWindow()
    {
      ifstream infile(id_filename);
      if (infile)
      {
        Gtk::Socket* socket = Gtk::manage(new Gtk::Socket());
        add(*socket);
        socket->signal_plug_added().connect(sigc::ptr_fun(plug_added));
        socket->signal_plug_removed().connect(sigc::ptr_fun(plug_removed));
        ::Window plug_id = 0;
        infile >> plug_id;
        infile.close();
        socket->add_id(plug_id);
      }
      else
      {
        Gtk::Label* label = Gtk::manage(
            new Gtk::Label(
              "Plug id file not found.\n Make sure plug is running."));
        add(*label);
        set_size_request(150, 50);
      }
      show_all();
    }
};

int main(int argc, char** argv)
{
  Gtk::Main app(argc, argv);
  MySocketWindow win;
  app.run(win);
  return 0;
}
```

File: `plug.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include <iostream>
#include <fstream>
#include <gtkmm.h>
#include <gtkmm/plug.h>
#include <glib/gstdio.h>

using namespace std;

const char* id_filename = "plug.id";

void on_embed()
{
  cout << "I've been embedded." << endl;
}

class MyPlug : public Gtk::Plug
{
  public:
    MyPlug() :
      m_label("I am the plug")
  {
    set_size_request(150, 100);
    add(m_label);
    signal_embedded().connect(sigc::ptr_fun(on_embed));
    show_all();
  }

  private:
    Gtk::Label m_label;
};


int main(int argc, char** argv)
{
  Gtk::Main app(argc, argv);
  MyPlug plug;

  ofstream out(id_filename);
  out << plug.get_id();
  out.close();
  cout << "The window ID is: " << plug.get_id() << endl;
```

```
  app.run(plug);

  // remove the ID file when the program exits
  g_remove(id_filename);
  return 0;
}
```

This example creates two executable programs: `socket` and `plug`. The idea is that `socket` has an application window that will embed a widget from the `plug` program. The way this example is designed, `plug` must be running first before starting `socket`. To see the example in action, execute the following commands in order from within the example directory:

Start the `plug` program and send it to the background (or just use a different terminal).

```
$ ./plug &
```

After which you should see something like the following:

```
The window ID is: 69206019
```

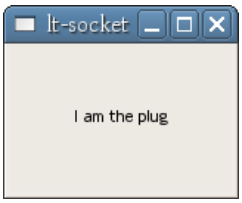
Then start the `socket` program:

```
$ ./socket
```

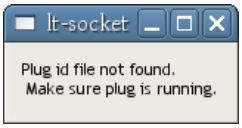
After starting `socket`, you should see the following output in the terminal:

```
I've been embedded.
A plug was added
```

The first line of output is from `plug`, after it has been notified that it has been embedded inside of a `Socket`. The second line was emitted by `socket` in response to its `plug_added` signal. If everything was done as described above, the `socket` window should look roughly like the following:



If for some reason the `Socket` couldn't attach the `Plug`, the window would look something like this:



23. Keyboard Events

X events differ in some ways from other signals. These differences are described in the X Event signals section in the appendix. Here we will use keyboard events to show how X events can be used in a program.

23.1. Overview

Whenever you press or release a key, an event is emitted. You can connect a signal handler to handle such events.

To receive the keyboard events, you must first call the `Gtk::Widget::add_events()` function with a bit mask of the events you're interested in. The event signal handler will receive an argument that depends on the type of event. For keyboard events it's a `GdkEventKey*`. As discribed in the appendix, the event signal handler returns a `bool` value, to indicate that the signal is fully handled (`true`) or allow event propagation (`false`).

To determine which key was pressed or released, you read the value of `GdkEventKey::keyval` and compare it with a constant in the `<gdk/gdkkeysyms.h>` header file. The states of modifier keys (shift, ctrl, etc.) are available as bit-flags in `GdkEventKey::state`.

Here's a simple example:

```
bool on_key_press_or_release_event(GdkEventKey* event)
{
  if (event->type == GDK_KEY_PRESS &&
    event->keyval == GDK_KEY_1 &&
    (event->state & (GDK_SHIFT_MASK | GDK_CONTROL_MASK | GDK_MOD1_MASK)) == GDK_MOD1_MASK)
  {
    handle_alt_1_press(); // GDK_MOD1_MASK is normally the Alt key
    return true;
  }
  return false;
}

Gtk::Entry m_entry; // in a class definition

// in the class constructor
m_entry.signal_key_press_event().connect( sigc::ptr_fun(&on_key_press_or_release_event) );
m_entry.signal_key_release_event().connect( sigc::ptr_fun(&on_key_press_or_release_event) );
m_entry.add_events(Gdk::KEY_PRESS_MASK | Gdk::KEY_RELEASE_MASK);
```

23.1.1. Example

23.1.1. Example

In this example there are three keyboard shortcuts: **Alt+1** selects the first radio button, **Alt+2** selects the second one, and the **Esc** key hides (closes) the window.

Figure 23-1 **Keyboard Events - Simple**

Source Code

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <gtkmm.h>


class ExampleWindow : public Gtk::Window
{
public:

  ExampleWindow();
  virtual ~ExampleWindow();

private:
  bool on_key_press_event(GdkEventKey *event);

  Gtk::Grid m_container;
  Gtk::RadioButton m_first;
  Gtk::RadioButton m_second;
};


#endif //GTKMM_EXAMPLEWINDOW_H
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"

ExampleWindow::ExampleWindow()
{
  set_title("Keyboard Events");
  set_border_width(10);
  add(m_container);

  // Radio buttons:
  m_first.set_label("First");
  m_second.set_label("Second");

  Gtk::RadioButton::Group group = m_first.get_group();
  m_second.set_group(group);
  m_first.set_active();

  // Main Container:
  m_container.add(m_first);
  m_container.add(m_second);

  // Events
  add_events(Gdk::KEY_PRESS_MASK);
  signal_key_press_event().connect(sigc::mem_fun(*this,
    &ExampleWindow::on_key_press_event));

  show_all_children();
}

bool ExampleWindow::on_key_press_event(GdkEventKey *event)
{
  //GDK_MOD1_MASK -> the 'alt' key(mask)
  //GDK_KEY_1 -> the '1' key
  //GDK_KEY_2 -> the '2' key

  //select the first radio button, when we press alt + 1
  if((event->keyval == GDK_KEY_1) &&
    (event->state &(GDK_SHIFT_MASK | GDK_CONTROL_MASK | GDK_MOD1_MASK)) == GDK_MOD1_MASK)
  {
    m_first.set_active();
  }
  else if((event->keyval == GDK_KEY_2) &&
    (event->state & (GDK_SHIFT_MASK | GDK_CONTROL_MASK | GDK_MOD1_MASK)) == GDK_MOD1_MASK)
  {
    //and the second radio button, when we press alt + 2
    m_second.set_active();
  }
  else if(event->keyval == GDK_KEY_Escape)
  {
    //close the window, when the 'esc' key is pressed
    hide();
  }

  //returning true, cancels the propagation of the event
  return true;
}


ExampleWindow::~ExampleWindow()
{
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;

  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

23.2. Event Propagation

Event propagation means that, when an event is emitted on a particular widget, it can be passed to its parent widget (and that widget can pass it to its parent, and so on) and, if the parent has an event handler, that handler will be called.

The event will propagate until it reaches the top-level widget, or until you stop the propagation by returning `true` from an event handler.

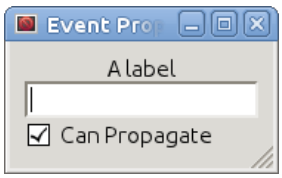
Notice, that after canceling an event, no other function will be called (even if it is from the same widget).

23.2.1. Example

In this example there are three event handlers, one in the `Gtk::Entry`, one in the `Gtk::Grid` and one in the `Gtk::Window`.

When you write in the entry, a key release event will be emitted first in the `Entry` and, depending on whether we let it propagate, it can reach the `Grid`'s and the `Window`'s event handlers. If it propagates, the text you're writing will appear in the `Label` above the `Entry`.

Figure 23-2 **Keyboard Events - Event Propagation**



Source Code

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EVENT_PROPAGATION_H
#define GTKMM_EVENT_PROPAGATION_H

#include <gtkmm.h>
#include <iostream>

class ExampleWindow : public Gtk::Window
{
public:

  ExampleWindow();
  virtual ~ExampleWindow();


private:

  bool entryKeyRelease(GdkEventKey *event);
  bool gridKeyRelease(GdkEventKey *event);
  bool windowKeyRelease(GdkEventKey *event);

  Gtk::Grid m_container;

  Gtk::Label m_label;
  Gtk::Entry m_entry;
  Gtk::CheckButton m_checkbutton_can_propagate;
};


#endif //GTKMM_EVENT_PROPAGATION_H
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"

ExampleWindow::ExampleWindow()
{
  add(m_container);
  set_title("Event Propagation");
```

```
  set_border_width(10);

  m_label.set_label("A label");
  m_checkbutton_can_propagate.set_label("Can Propagate");
  m_checkbutton_can_propagate.set_active();

  // Main Container
  m_container.set_orientation(Gtk::ORIENTATION_VERTICAL);
  m_container.add(m_label);
  m_container.add(m_entry);
  m_container.add(m_checkbutton_can_propagate);

  // Events
  add_events(Gdk::KEY_RELEASE_MASK);

  m_entry.signal_key_release_event().connect(
    sigc::mem_fun(*this, &ExampleWindow::entryKeyRelease));

  m_container.signal_key_release_event().connect(
    sigc::mem_fun(*this, &ExampleWindow::gridKeyRelease));

  signal_key_release_event().connect(
    sigc::mem_fun(*this, &ExampleWindow::windowKeyRelease));

  show_all_children();
}

//By changing the return value we allow, or don't allow, the event to propagate to other elements.
bool ExampleWindow::entryKeyRelease(GdkEventKey* /* event */ )
{
  std::cout << "Entry" << std::endl;

  if(m_checkbutton_can_propagate.get_active())
  {
    return false;
  }

  return true;
}

bool ExampleWindow::gridKeyRelease(GdkEventKey* /* event */ )
{
  std::cout << "Grid" << std::endl;

  //Let it propagate:
  return false;
}

// This will set the entry's text in the label, every time a key is pressed.
bool ExampleWindow::windowKeyRelease(GdkEventKey* /* event */ )
{
  std::cout << "Window" << std::endl;

  //checking if the entry is on focus, otherwise the label would get changed by pressing keys
  //on the window (when the entry is not on focus), even if canPropagate wasn't active
  if(m_entry.has_focus())
  {
    m_label.set_text(m_entry.get_text());
  }

  return true;
}

ExampleWindow::~ExampleWindow()
{
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;

  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

24. Timeouts, I/O and Idle Functions

24.1. Timeouts

You may be wondering how to make *gtkmm* do useful work while it's idling along (well, sleeping actually) in `Gtk::Main::run()`. Happily, you have several options. Using the following methods you can create a timeout method that will be called every few milliseconds.

```
sigc::connection Glib::SignalTimeout::connect(const sigc::slot<bool>& slot, unsigned int interval, int priority = Glib::PRIORITY_DI
```

The first argument is a `slot` you wish to have called when the timeout occurs. The second argument is the number of milliseconds between calls to that method. You receive a `sigc::connection` object that can be used to deactivate the connection using its `disconnect()` method:

```
my_connection.disconnect();
```

Another way of destroying the connection is your signal handler. It has to be of the type `sigc::slot<bool>`. As you see from the definition your signal handler has to return a value of the type `bool`. A definition of a sample method might look like this:

```
bool MyCallback() { std::cout << "Hello World!\n" << std::endl; return true; }
```

You can stop the timeout method by returning `false` from your signal handler. Therefore, if you want your method to be called repeatedly, it should return `true`.

Here's an example of this technique:

Source Code

File: `timerexample.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLE_TIMEREXAMPLE_H
#define GTKMM_EXAMPLE_TIMEREXAMPLE_H

#include <gtkmm.h>
#include <iostream>
#include <map>

class TimerExample : public Gtk::Window
{
public:
  TimerExample();

protected:
  // signal handlers
  void on_button_add_timer();
  void on_button_delete_timer();
  void on_button_quit();

  // This is the callback function the timeout will call
  bool on_timeout(int timer_number);

  // Member data:

  Gtk::Box m_Box;
  Gtk::Button m_ButtonAddTimer, m_ButtonDeleteTimer, m_ButtonQuit;

  // Keep track of the timers being added:
  int m_timer_number;

  // These two constants are initialized in the constructor's member initializer:
  const int count_value;
  const int timeout_value;
```

```
  // STL map for storing our connections
  std::map<int, sigc::connection> m_timers;

  // STL map for storing our timer values.
  // Each timer counts back from COUNT_VALUE to 0 and is removed when it reaches 0
  std::map<int, int> m_counters;
};

#endif // GTKMM_EXAMPLE_TIMEREXAMPLE_H
```

File: `timerexample.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "timerexample.h"

TimerExample::TimerExample() :
  m_Box(Gtk::ORIENTATION_HORIZONTAL, 10),
    // use Gtk::Stock wherever possible for buttons, etc.
  m_ButtonAddTimer(Gtk::Stock::ADD),
  m_ButtonDeleteTimer(Gtk::Stock::REMOVE),
  m_ButtonQuit(Gtk::Stock::QUIT),
  m_timer_number(0), // start numbering the timers at 0
  count_value(5), // each timer will count down 5 times before disconnecting
  timeout_value(1500) // 1500 ms = 1.5 seconds
{
  set_border_width(10);

  add(m_Box);
  m_Box.pack_start(m_ButtonAddTimer);
  m_Box.pack_start(m_ButtonDeleteTimer);
  m_Box.pack_start(m_ButtonQuit);

  // Connect the three buttons:
  m_ButtonQuit.signal_clicked().connect(sigc::mem_fun(*this,
              &TimerExample::on_button_quit));
  m_ButtonAddTimer.signal_clicked().connect(sigc::mem_fun(*this,
              &TimerExample::on_button_add_timer));
  m_ButtonDeleteTimer.signal_clicked().connect(sigc::mem_fun(*this,
              &TimerExample::on_button_delete_timer));

  show_all_children();
}

void TimerExample::on_button_quit()
{
  hide();
}

void TimerExample::on_button_add_timer()
{
  // Creation of a new object prevents long lines and shows us a little
  // how slots work.  We have 0 parameters and bool as a return value
  // after calling sigc::bind.
  sigc::slot<bool> my_slot = sigc::bind(sigc::mem_fun(*this,
              &TimerExample::on_timeout), m_timer_number);

  // This is where we connect the slot to the Glib::signal_timeout()
  sigc::connection conn = Glib::signal_timeout().connect(my_slot,
          timeout_value);

  // Remember the connection:
  m_timers[m_timer_number] = conn;

  // Initialize timer count:
  m_counters[m_timer_number] = count_value + 1;

  // Print some info to the console for the user:
  std::cout << "added timeout " << m_timer_number++ << std::endl;
}

void TimerExample::on_button_delete_timer()
{
  // any timers?
  if(m_timers.empty())
  {
    // no timers left
    std::cout << "Sorry, there are no timers left." << std::endl;
  }
  else
  {
    // get the number of the first timer
    int timer_number = m_timers.begin()->first;

    // Give some info to the user:
    std::cout << "manually disconnecting timer " << timer_number
        << std::endl;
```

```
    // Remove the entry in the counter values
    m_counters.erase(timer_number);

    // Diconnect the signal handler:
    m_timers[timer_number].disconnect();

    // Forget the connection:
    m_timers.erase(timer_number);
  }
}

bool TimerExample::on_timeout(int timer_number)
{
  // Print the timer:
  std::cout << "This is timer " << timer_number;

  // decrement and check counter value
  if (--m_counters[timer_number] == 0)
  {
    std::cout << " being disconnected" <<  std::endl;

    // delete the counter entry in the STL MAP
    m_counters.erase(timer_number);

    // delete the connection entry in the STL MAP
    m_timers.erase(timer_number);

    // Note that we do not have to explicitly call disconnect() on the
    // connection since Gtk::Main does this for us when we return false.
    return false;
  }

  // Print the timer value
  std::cout << " - " << m_counters[timer_number] << "/"
      << count_value << std::endl;

 // Keep going (do not disconnect yet):
  return true;
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "timerexample.h"
#include <gtkmm/main.h>

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

  TimerExample example;
  Gtk::Main::run(example);

  return 0;
}
```

24.2. Monitoring I/O

A nifty feature of Glib (one of the libraries underlying *gtkmm*) is the ability to have it check for data on a file descriptor for you. This is especially useful for networking applications. The following method is used to do this:

```
sigc::connection Glib::SignalInput::connect(const sigc::slot<bool,Glib::IOCondition>& slot,
                                    int fd, Glib::IOCondition condition,
                                    int priority = Glib::PRIORITY_DEFAULT);
```

The first argument is a slot you wish to have called when then the specified event (see argument 3) occurs on the file descriptor you specify using argument two. Argument three may be one or more (using `|`) of:

- Glib::IO_IN - Call your method when there is data ready for reading on your file descriptor.
- Glib::IO_OUT - Call your method when the file descriptor is ready for writing.
- Glib::IO_PRI - Call your method when the file descriptor has urgent data to be read.

- Glib::IO_ERR - Call your method when an error has occurred on the file descriptor.
- Glib::IO_HUP - Call your method when hung up (the connection has been broken usually for pipes and sockets).

The return value is a `sigc::connection` that may be used to stop monitoring this file descriptor using its `disconnect()` method. The `slot` signal handler should be declared as follows:

```
bool input_callback(Glib::IOCondition condition);
```

where `condition` is as specified above. As usual the slot is created with `sigc::mem_fun()` (for a member method of an object.), or `sigc::ptr_fun()` (for a function).

A little example follows. To use the example just execute it from a terminal; it doesn't create a window. It will create a pipe named `testfifo` in the current directory. Then start another shell and execute `echo "Hello" > testfifo`. The example will print each line you enter until you execute `echo "Q" > testfifo`.

Source Code

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include <build/config.h>
#include <gtkmm/main.h>
#include <glibmm/main.h>
#include <glibmm/iochannel.h>
#include <fcntl.h>
#include <iostream>

#include <unistd.h> //The SUN Forte compiler puts F_OK here.

//The SUN Forte compiler needs these for mkfifo:
#include <sys/types.h>
#include <sys/stat.h>

int read_fd;
Glib::RefPtr<Glib::IOChannel> iochannel;

/*
  send to the fifo with:
  echo "Hello" > testfifo

  quit the program with:
  echo "Q" > testfifo
*/

// this will be our signal handler for read operations
// it will print out the message sent to the fifo
// and quit the program if the message was 'Q'.
bool MyCallback(Glib::IOCondition io_condition)
{

  if ((io_condition & Glib::IO_IN) == 0) {
    std::cerr << "Invalid fifo response" << std::endl;
  }
  else {
   Glib::ustring buf;

   iochannel->read_line(buf);
   std::cout << buf;
   if (buf == "Q\n")
       Gtk::Main::quit ();

  }
  return true;
}


int main(int argc, char *argv[])
{
  // the usual Gtk::Main object
  Gtk::Main app(argc, argv);

  if (access("testfifo", F_OK) == -1) {
    // fifo doesn't exit - create it
    #ifdef HAVE_MKFIFO
    if (mkfifo("testfifo", 0666) != 0) {
      std::cerr << "error creating fifo" << std::endl;
```

```
      return -1;
    }
    #else
      std::cerr << "error creating fifo: This platform does not have mkfifo()"
          << std::endl;
    #endif //HAVE_MKFIFO
  }

  read_fd = open("testfifo", O_RDONLY);
  if (read_fd == -1)
  {
    std::cerr << "error opening fifo" << std::endl;
    return -1;
  }

  // connect the signal handler
  Glib::signal_io().connect(sigc::ptr_fun(MyCallback), read_fd, Glib::IO_IN);

  // Creates a iochannel from the file descriptor
  iochannel = Glib::IOChannel::create_from_fd(read_fd);

  // and last but not least - run the application main loop
  app.run();

  // now remove the temporary fifo
  if(unlink("testfifo"))
    std::cerr << "error removing fifo" << std::endl;

  return 0;
}
```

24.3. Idle Functions

If you want to specify a method that gets called when nothing else is happening, use the following:

```
sigc::connection  Glib::SignalIdle::connect(const sigc::slot<bool>& slot, int priority = Glib::PRIORITY_DEFAULT_IDLE);
```

This causes *gtkmm* to call the specified method whenever nothing else is happening. You can add a priority (lower numbers are higher priorities). There are two ways to remove the signal handler: calling `disconnect()` on the `sigc::connection` object, or returning `false` in the signal handler, which should be declared as follows:

```
bool idleFunc();
```

Since this is very similar to the methods above this explanation should be sufficient to understand what's going on. However, here's a little example:

Source Code

File: `idleexample.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLE_IDLEEXAMPLE_H
#define GTKMM_EXAMPLE_IDLEEXAMPLE_H

#include <gtkmm.h>
#include <iostream>

class IdleExample : public Gtk::Window
{
public:
  IdleExample();

protected:
  // Signal Handlers:
  bool on_timer();
  bool on_idle();
  void on_button_clicked();

  // Member data:
  Gtk::Box m_Box;
  Gtk::Button m_ButtonQuit;
  Gtk::ProgressBar m_ProgressBar_c;
  Gtk::ProgressBar m_ProgressBar_d;
};
```

```
#endif // GTKMM_EXAMPLE_IDLEEXAMPLE_H
```

File: `idleexample.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "idleexample.h"

IdleExample::IdleExample() :
  m_Box(Gtk::ORIENTATION_VERTICAL, 5),
  m_ButtonQuit(Gtk::Stock::QUIT)
{
  set_border_width(5);

  // Put buttons into container

  // Adding a few widgets:
  add(m_Box);
  m_Box.pack_start( *Gtk::manage(new Gtk::Label("Formatting Windows drive C:")));
  m_Box.pack_start( *Gtk::manage(new Gtk::Label("100 MB")) );
  m_Box.pack_start(m_ProgressBar_c);

  m_Box.pack_start( *Gtk::manage(new Gtk::Label("")) );

  m_Box.pack_start( *Gtk::manage(new Gtk::Label("Formatting Windows drive D:")));
  m_Box.pack_start( *Gtk::manage(new Gtk::Label("5000 MB")) );
  m_Box.pack_start(m_ProgressBar_d);

  Gtk::Box* hbox = Gtk::manage( new Gtk::Box(Gtk::ORIENTATION_HORIZONTAL,10));
  m_Box.pack_start(*hbox);
  hbox->pack_start(m_ButtonQuit, Gtk::PACK_EXPAND_PADDING);

  // Connect the signal handlers:
  m_ButtonQuit.signal_clicked().connect( sigc::mem_fun(*this,
              &IdleExample::on_button_clicked) );

  // formatting drive c in timeout signal handler - called once every 50ms
  Glib::signal_timeout().connect( sigc::mem_fun(*this, &IdleExample::on_timer),
          50 );

  // formatting drive d in idle signal handler - called as quickly as possible
  Glib::signal_idle().connect( sigc::mem_fun(*this, &IdleExample::on_idle) );

  show_all_children();
}


void IdleExample::on_button_clicked()
{
  hide();
}

// this timer callback function is executed once every 50ms (set in connection
// above).  Use timeouts when speed is not critical. (ie periodically updating
// something).
bool IdleExample::on_timer()
{
  double value = m_ProgressBar_c.get_fraction();

  // Update progressbar 1/500th each time:
  m_ProgressBar_c.set_fraction(value + 0.002);

  return value < 0.99;  // return false when done
}


// This idle callback function is executed as often as possible, hence it is
// ideal for processing intensive tasks.
bool IdleExample::on_idle()
{
  double value = m_ProgressBar_d.get_fraction();

  // Update progressbar 1/5000th each time:
  m_ProgressBar_d.set_fraction(value + 0.0002);

  return value < 0.99;  // return false when done
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "idleexample.h"
#include <gtkmm/main.h>

int main (int argc, char *argv[])
{
```

```
  Gtk::Main app(argc, argv);

  IdleExample example;
  Gtk::Main::run(example);

  return 0;
}
```

This example points out the difference of idle and timeout methods a little. If you need methods that are called periodically, and speed is not very important, then you want timeout methods. If you want methods that are called as often as possible (like calculating a fractal in background), then use idle methods.

Try executing the example and increasing the system load. The upper progress bar will increase steadily; the lower one will slow down.

25. Memory management

25.1. Widgets

25.1.1. Normal C++ memory management

gtkmm allows the programmer to control the lifetime (that is, the construction and destruction) of any widget in the same manner as any other C++ object. This flexibility allows you to use `new` and `delete` to create and destroy objects dynamically or to use regular class members (that are destroyed automatically when the class is destroyed) or to use local instances (that are destroyed when the instance goes out of scope). This flexibility is not present in some C++ GUI toolkits, which restrict the programmer to only a subset of C++'s memory management features.

Here are some examples of normal C++ memory management:

25.1.1.1. Class Scope widgets

If a programmer does not need dynamic memory allocation, automatic widgets in class scope may be used. One advantage of automatic widgets in class scope is that memory management is grouped in one place. The programmer does not risk memory leaks from failing to `delete` a widget.

The primary disadvantages of using class scope widgets are revealing the class implementation rather than the class interface in the class header. Class scope widgets also require Automatic widgets in class scope suffer the same disadvantages as any other class scope automatic variable.

```
#include <gtkmm/button.h>
class Foo
{
private:
  Gtk::Button theButton;
  // will be destroyed when the Foo object is destroyed
};
```

25.1.1.2. Function scope widgets

If a programmer does not need a class scope widget, a function scope widget may also be used. The advantages to function scope over class scope are the increased

data hiding and reduced dependencies.

```
{
  Gtk::Button aButton;
  aButton.show();
  ...
  kit.run();
}
```

25.1.1.3. Dynamic allocation with new and delete

Although, in most cases, the programmer will prefer to allow containers to automatically destroy their children using `manage()` (see below), the programmer is not required to use `manage()`. The traditional `new` and `delete` operators may also be used.

```
Gtk::Button* pButton = new Gtk::Button("Test");

// do something useful with pButton

delete pButton;
```

Here, the programmer deletes pButton to prevent a memory leak.

25.1.2. Managed Widgets

Alternatively, you can let a widget's container control when the widget is destroyed. In most cases, you want a widget to last only as long as the container it is in. To delegate the management of a widget's lifetime to its container, first create it with `manage()` and pack it into its container with `add()`. Now, the widget will be destroyed whenever its container is destroyed.

25.1.2.1. Dynamic allocation with manage() and add()

gtkmm provides the `manage()` function and `add()` methods to create and destroy widgets. Every widget except a top-level window must be added or packed into a container in order to be displayed. The `manage()` function marks a packed widget so that when the widget is added to a container, the container becomes responsible for deleting the widget.

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

Now, when objects of type `MyWidget` are destroyed, the button will also be deleted. It is no longer necessary to delete pButton to free the button's memory; its deletion has been delegated to the `MyWidget` object.

gtkmm also provides the `set_manage()` method for all widgets. This can be used to generate the same result as `manage()`, but is more tedious:

foo.add((w=new Gtk::Label("Hello"), w->set_manage(), &w));

is the same as

foo.add(manage(new Gtk::Label("Hello")));

Of course, a top level container will not be added to another container. The programmer is responsible for destroying the top level container using one of the traditional C++ techniques. For instance, your top-level Window might just be an instance in your `main()` function..

25.2. Shared resources

Some objects, such as `Gdk::Pixbuf`s and `Pango::Font`s, are obtained from a shared store. Therefore you cannot instantiate your own instances. These classes typically inherit

from `Glib::Object`. Rather than requiring you to reference and unreference these objects, *gtkmm* uses the `Glib::RefPtr<>` smartpointer. Cairomm has its own smartpointer, `Cairo::RefPtr<>`.

Objects such as `Gdk::Pixbuf` can only be instantiated with a `create()` function. For instance,

```
Glib::RefPtr<Gdk::Pixbuf> pixbuf = Gdk::Pixbuf::create_from_file(filename);
```

You have no way of getting a bare `Gdk::Pixbuf`. In the example, `pixbuf` is a smart pointer, so you can do this, much like a normal pointer:

```
int width = 0;
if(pixbuf)
{
  width = pixbuf->get_width();
}
```

When `pixbuf` goes out of scope an `unref()` will happen in the background and you don't need to worry about it anymore. There's no `new` so there's no `delete`.

If you copy a `RefPtr`, for instance

```
Glib::RefPtr<Gdk::Pixbuf> pixbuf2 = pixbuf;
```

, or if you pass it as a method argument or a return type, then `RefPtr` will do any necessary referencing to ensure that the instance will not be destroyed until the last `RefPtr` has gone out of scope.

See the appendix for detailed information about RefPtr.

If you wish to learn more about smartpointers, you might look in these books:

- Bjarne Stroustrup, "The C++ Programming Language" - section 14.4.2
- Nicolai M. Josuttis, "The C++ Standard Library" - section 4.2

26. Glade and Gtk::Builder

Although you can use C++ code to instantiate and arrange widgets, this can soon become tedious and repetitive. And it requires a recompilation to show changes. The *Glade* application allows you to layout widgets on screen and then save an XML description of the arrangement. Your application can then use the *Gtk::Builder* API to load that XML file at runtime and obtain a pointer to specifically named widget instances.

This has the following advantages:

1. Less C++ code is required.
2. UI changes can be seen more quickly, so UIs are able to improve.
3. Designers without programming skills can create and edit UIs.

You still need C++ code to deal with User Interface changes triggered by user actions, but using *Gtk::Builder* for the widget layout allows you to focus on implementing that functionality.

26.1. Loading the .glade file

`Gtk::Builder` must be used via a `Glib::RefPtr`. Like all such classes, you need to use a `create()` method to instantiate it. For instance,

```
Glib::RefPtr<Gtk::Builder> builder = Gtk::Builder::create_from_file("basic.glade");
```

This will instantiate the windows defined in the .glade file, though they will not be shown immediately unless you have specified that via the **Properties** window in *Glade*.

To instantiate just one window, or just one of the child widgets, you can specify the name of a widget as the second parameter. For instance,

```
Glib::RefPtr<Gtk::Builder> builder = Gtk::Builder::create_from_file("basic.glade", "treeview_products");
```

26.2. Accessing widgets

To access a widget, for instance to `show()` a dialog, use the `get_widget()` method, providing the widget's name. This name should be specified in the *Glade* Properties window. If the widget could not be found, or is of the wrong type, then the pointer will be set to 0.

```
Gtk::Dialog* pDialog = 0;
builder->get_widget("DialogBasic", pDialog);
```

Gtk::Builder checks for a null pointer, and checks that the widget is of the expected type, and will show warnings on the command line about these.

Remember that you are not instantiating a widget with `get_widget()`, you are just obtaining a pointer to one that already exists. You will always receive a pointer to the same instance when you call `get_widget()` on the same `Gtk::Builder`, with the same widget name. The widgets are instantiated during `Gtk::Builder::create_from_file()`.

`get_widget()` returns child widgets that are `manage()`ed (see the Memory Management chapter), so they will be deleted when their parent container is deleted. So, if you get only a child widget from *Gtk::Builder*, instead of a whole window, then you must either put it in a `Container` or delete it. `Windows` (such as `Dialogs`) cannot be managed because they have no parent container, so you must delete them at some point.

26.2.1. Example

This simple example shows how to load a *Glade* file at runtime and access the widgets with *Gtk::Builder*.

Source Code

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include <gtkmm.h>
#include <iostream>

Gtk::Dialog* pDialog = 0;

static
void on_button_clicked()
{
  if(pDialog)
    pDialog->hide(); //hide() will cause main::run() to end.
}

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

  //Load the GtkBuilder file and instantiate its widgets:
  Glib::RefPtr<Gtk::Builder> refBuilder = Gtk::Builder::create();
  try
  {
    refBuilder->add_from_file("basic.glade");
  }
  catch(const Glib::FileError& ex)
  {
```

```
    std::cerr << "FileError: " << ex.what() << std::endl;
    return 1;
  }
  catch(const Glib::MarkupError& ex)
  {
    std::cerr << "MarkupError: " << ex.what() << std::endl;
    return 1;
  }
  catch(const Gtk::BuilderError& ex)
  {
    std::cerr << "BuilderError: " << ex.what() << std::endl;
    return 1;
  }

  //Get the GtkBuilder-instantiated Dialog:
  refBuilder->get_widget("DialogBasic", pDialog);
  if(pDialog)
  {
    //Get the GtkBuilder-instantiated Button, and connect a signal handler:
    Gtk::Button* pButton = 0;
    refBuilder->get_widget("quit_button", pButton);
    if(pButton)
    {
      pButton->signal_clicked().connect( sigc::ptr_fun(on_button_clicked) );
    }

    kit.run(*pDialog);
  }

  delete pDialog;

  return 0;
}
```

26.3. Using derived widgets

You can use *Glade* to layout your own custom widgets derived from *gtkmm* widget classes. This keeps your code organized and encapsulated. Of course you won't see the exact appearance and properties of your derived widget in *Glade*, but you can specify its location and child widgets and the properties of its *gtkmm* base class.

Use `Gtk::Builder::get_widget_derived()` like so:

```
DerivedDialog* pDialog = 0;
builder->get_widget_derived("DialogBasic", pDialog);
```

Your derived class must have a constructor that takes a pointer to the underlying C type, and the `Gtk::Builder` instance. All relevant classes of *gtkmm* typedef their underlying C type as `BaseObjectType` (`Gtk::Dialog` typedefs `BaseObjectType` as `GtkDialog`, for instance).

You must call the base class's constructor in the initialization list, providing the C pointer. For instance,

```
DerivedDialog::DerivedDialog(BaseObjectType* cobject, const Glib::RefPtr<Gtk::Builder>& builder)
: Gtk::Dialog(cobject)
{
}
```

You could then encapsulate the manipulation of the child widgets in the constructor of the derived class, maybe using `get_widget()` or `get_widget_derived()` again. For instance,

```
DerivedDialog::DerivedDialog(BaseObjectType* cobject, const Glib::RefPtr<Gtk::Builder>& builder)
: Gtk::Dialog(cobject),
  m_builder(builder),
  m_pButton(0)
{
  //Get the Glade-instantiated Button, and connect a signal handler:
  m_builder->get_widget("quit_button", m_pButton);
  if(m_pButton)
  {
    m_pButton->signal_clicked().connect( sigc::mem_fun(*this, &DerivedDialog::on_button_quit) );
  }
}
```

26.3.1. Example

This example shows how to load a *Glade* file at runtime and access the widgets via a derived class.

Source Code

File: `deriveddialog.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLE_DERIVED_DIALOG_H
#define GTKMM_EXAMPLE_DERIVED_DIALOG_H

#include <gtkmm.h>


class DerivedDialog : public Gtk::Dialog
{
public:
  DerivedDialog(BaseObjectType* cobject, const Glib::RefPtr<Gtk::Builder>& refGlade);
  virtual ~DerivedDialog();

protected:
  //Signal handlers:
  void on_button_quit();

  Glib::RefPtr<Gtk::Builder> m_refGlade;
  Gtk::Button* m_pButton;
};

#endif //GTKMM_EXAMPLE_DERIVED_WINDOW_H
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "deriveddialog.h"
#include <iostream>

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

  //Load the Glade file and instiate its widgets:
  Glib::RefPtr<Gtk::Builder> refBuilder = Gtk::Builder::create();
  try
  {
    refBuilder->add_from_file("derived.glade");
  }
  catch(const Glib::FileError& ex)
  {
    std::cerr << "FileError: " << ex.what() << std::endl;
    return 1;
  }
  catch(const Glib::MarkupError& ex)
  {
    std::cerr << "MarkupError: " << ex.what() << std::endl;
    return 1;
  }
  catch(const Gtk::BuilderError& ex)
  {
    std::cerr << "BuilderError: " << ex.what() << std::endl;
    return 1;
  }

  //Get the GtkBuilder-instantiated dialog::
  DerivedDialog* pDialog = 0;
  refBuilder->get_widget_derived("DialogDerived", pDialog);
  if(pDialog)
  {
    //Start:
    kit.run(*pDialog);
  }

  delete pDialog;

  return 0;
}
```

File: `deriveddialog.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "derivedialog.h"

DerivedDialog::DerivedDialog(BaseObjectType* cobject, const Glib::RefPtr<Gtk::Builder>& refGlade)
: Gtk::Dialog(cobject),
  m_refGlade(refGlade),
  m_pButton(0)
{
  //Get the Glade-instantiated Button, and connect a signal handler:
  m_refGlade->get_widget("quit_button", m_pButton);
  if(m_pButton)
  {
    m_pButton->signal_clicked().connect( sigc::mem_fun(*this, &DerivedDialog::on_button_quit) );
  }
}

DerivedDialog::~DerivedDialog()
{
}

void DerivedDialog::on_button_quit()
{
  hide(); //hide() will cause main::run() to end.
}
```

27. Internationalization and Localization

gtkmm applications can easily support multiple languages, including non-European languages such as Chinese and right-to-left languages such as Arabic. An appropriately-written and translated *gtkmm* application will use the appropriate language at runtime based on the user's environment.

You might not anticipate the need to support additional languages, but you can never rule it out. And it's easier to develop the application properly in the first place rather than retrofitting later.

The process of writing source code that allows for translation is called `internationalization`, often abbreviated to `i18n`. The `Localization` process, sometimes abbreviated as `l10n`, provides translated text for other languages, based on that source code.

The main activity in the internationalization process is finding strings seen by users and marking them for translation. You do not need to do it all at once - if you set up the necessary project infrastructure correctly then your application will work normally regardless of how many strings you've covered.

String literals should be typed in the source code in English, but surrounded by a macro. The *gettext* (or intltool) utility can then extract the marked strings for tramslation, and substitute the translated text at runtime.

27.1. Preparing your project

> In the instructions below we will assume that you will not be using *gettext* directly, but *intltool*, which was written specifically for `GNOME`. *intltool* uses `gettext()`, which extracts strings from source code, but *intltool* can also combine strings from other files, for example from desktop menu details, and GUI resource files such as *Glade* files, into standard *gettext* `.pot/.po` files.
>
> We also assume that you are using autotools (e.g. *automake* and *autoconf*) to build your project, and that you are using http://svn.gnome.org/viewcvs/gnome-common/trunk/autogen.sh?view=markup, which, among other things, takes care of some *intltool* initialization.

Create a sub-directory named `po` in your project's root directory. This directory will

eventually contain all of your translations. Within it, create a file named `LINGUAS` and a file named `POTFILES.in`. It is common practice to also create a `ChangeLog` file in the `po` directory so that translators can keep track of translation changes.

`LINGUAS` contains an alphabetically sorted list of codes identifying the languages for which your program is translated (comment lines starting with a `#` are ignored). Each language code listed in the `LINGUAS` file must have a corresponding `.po` file. So, if your program has German and Japanese translations, your `LINGUAS` file would look like this:

```
# keep this file sorted alphabetically, one language code per line
de
ja
```

(In addition, you'd have the files `ja.po` and `de.po` in your `po` directory which contain the German and Japanese translations, respectively.)

`POTFILES.in` is a list of paths to all files which contain strings marked up for translation, starting from the project root directory. So for example, if your project sources were located in a subdirectory named `src`, and you had two files that contained strings that should be translated, your `POTFILES.in` file might look like this:

```
src/main.cc
src/other.cc
```

If you are using *gettext* directly, you can only mark strings for translation if they are in source code file. However, if you use *intltool*, you can mark strings for translation in a variety of other file formats, including *Glade* UI files, xml, .desktop files and several more. So, if you have designed some of the application UI in *Glade* then also add your `.glade` files to the list in `POTFILES.in`.

Now that there is a place to put your translations, you need to initialize *intltool* and *gettext*. Add the following code to your `configure.ac`, substituting 'programname' with the name of your program:

```
IT_PROG_INTLTOOL([0.35.0])

GETTEXT_PACKAGE=programname
AC_SUBST(GETTEXT_PACKAGE)
AC_DEFINE_UNQUOTED([GETTEXT_PACKAGE], ["$GETTEXT_PACKAGE"],
                   [The domain to use with gettext])
AM_GLIB_GNU_GETTEXT

PROGRAMNAME_LOCALEDIR=[${datadir}/locale]
AC_SUBST(PROGRAMNAME_LOCALEDIR)
```

This `PROGRAMNAME_LOCALEDIR` variable will be used later in the `Makefile.am` file, to define a macro that will be used when you initialize *gettext* in your source code.

In the top-level Makefile.am:

- Add `po` to the `SUBDIRS` variable. Without this, your translations won't get built and installed when you build the program
- Define `INTLTOOL_FILES` as:

  ```
  INTLTOOL_FILES = intltool-extract.in \
                   intltool-merge.in \
                   intltool-update.in
  ```

- Add `INTLTOOL_FILES` to the `EXTRA_DIST` list of files. This ensures that when you do a `make dist`, these commands will be included in the source tarball.
- Update your `DISTCLEANFILES`:

```
DISTCLEANFILES = ... intltool-extract \
                 intltool-merge \
                 intltool-update \
                 po/.intltool-merge-cache
```

In your `src/Makefile.am`, update your `AM_CPPFLAGS` to add the following preprocessor macro definition:

```
AM_CPPFLAGS = ... -DPROGRAMNAME_LOCALEDIR=\"${PROGRAMNAME_LOCALEDIR}\"
```

This macro will be used when you initialize `gettext` in your source code.

27.2. Marking strings for translation

String literals should be typed in the source code in English, but they should be surrounded by a call to the `gettext()` function. These strings will be extracted for translation and the translations may be used at runtime instead of the original English strings.

The *GNU gettext* package allows you to mark strings in source code, extract those strings for translation, and use the translated strings in your application.

However, *Glib* defines `gettext()` support macros which are shorter wrappers in an easy-to-use form. To use these macros, include `<glibmm/i18n.h>`, and then, for example, substitute:

```
display_message("Getting ready for i18n.");
```

with:

```
display_message(_("Getting ready for i18n."));
```

For reference, it is possible to generate a file which contains all strings which appear in your code, even if they are not marked for translation, together with file name and line number references. To generate such a file named `my-strings`, execute the following command, within the source code directory:

```
xgettext -a -o my-strings --omit-header *.cc *.h
```

Finally, to let you program use the translation for the current locale, add this code to the beginning of your `main.cc` file, to initialize gettext.

```
bindtextdomain(GETTEXT_PACKAGE, PROGRAMNAME_LOCALEDIR);
bind_textdomain_codeset(GETTEXT_PACKAGE, "UTF-8");
textdomain(GETTEXT_PACKAGE);
```


27.2.1. How gettext works

intltool / *xgettext* script extracts the strings and puts them in a `mypackage.pot` file. The translators of your application create their translations by first copying this `.pot` file to a `localename.po` file. A locale identifies a language and an encoding for that language, including date and numerical formats. Later, when the text in your source code has changed, the `msmerge` script is used to update the `localename.po` files from the regenerated `.pot` file.

At install time, the `.po` files are converted to a binary format (with the extension `.mo`) and placed in a system-wide directory for locale files, for example `/usr/share/locale/`.

When the application runs, the *gettext* library checks the system-wide directory to

see if there is a `.mo` file for the user's locale environment (you can set the locale with, for instance, "export LANG=de_DE.UTF-8" from a bash console). Later, when the program reaches a `gettext` call, it looks for a translation of a particular string. If none is found, the original string is used.

27.2.2. Testing and adding translations

To convince yourself that you've done well, you may wish to add a translation for a new locale. In order to do that, go to the `po` subdirectory of your project and execute the following command:

```
intltool-update --pot
```

That will create a file named `programname.pot`. Now copy that file to `languagecode.po`, such as `de.po` or `hu.po`. Also add that language code to `LINGUAS`. The `.po` file contains a header and a list of English strings, with space for the translated strings to be entered. Make sure you set the encoding of the `.po` file (specified in the header, but also as content) to `UTF-8`.

> It's possible that certain strings will be marked as `fuzzy` in the `.po` file. These translations will not substitute the original string. To make them appear, simply remove the `fuzzy` tag.

27.2.3. Resources

More information about what lies behind the internationalization and localization process is presented and demonstrated in:

- Internationalizing GNOME applications
- Intltool README
- How to use GNOME CVS as a Translator
- gettext manual
- `gtkmm_hello` example package
- `gnomemm_hello` example package

27.3. Expecting UTF8

A properly internationalized application will not make assumptions about the number of bytes in a character. That means that you shouldn't use pointer arithmetic to step through the characters in a string, and it means you shouldn't use `std::string` or standard C functions such as `strlen()` because they make the same assumption.

However, you probably already avoid bare char* arrays and pointer arithmetic by using `std::string`, so you just need to start using `Glib::ustring` instead. See the Basics chapter about `Glib::ustring`.

27.3.1. Glib::ustring and std::iostreams

Unfortunately, the integration with the standard iostreams is not completely foolproof. *gtkmm* converts `Glib::ustring`s to a locale-specific encoding (which usually is not UTF-8) if you output them to an `ostream` with `operator<<`. Likewise, retrieving `Glib::ustrings` from `istream` with `operator>>` causes a conversion in the opposite direction. But this scheme breaks down if you go through a `std::string`, e.g. by inputting text from a stream to a `std::string` and then implicitly converting it to a `Glib::ustring`. If the string contained non-ASCII characters and the current locale is not UTF-8 encoded, the result is a corrupted `Glib::ustring`. You can work around this with a manual conversion. For instance, to retrieve the `std::string` from a `ostringstream`:

```
std::ostringstream output;
```

```
output.imbue(std::locale("")); // use the user's locale for this stream
output << percentage << " % done";
label->set_text(Glib::locale_to_utf8(output.str()));
```

27.4. Pitfalls

There are a few common mistakes that you would discover eventually yourself. But this section might help you to avoid them.

27.4.1. Same strings, different semantics

Sometimes two english strings are identical but have different meanings in different contexts, so they would probably not be identical when translated. Since the English strings are used as look-up keys, this causes problems.

In these cases, you should add extra characters to the strings. For instance, use `"jumps[noun]"` and `"jumps[verb]"` instead of just `"jumps"`) and strip them again outside the `gettext` call. If you add extra characters you should also add a comment for the translators before the `gettext` call. Such comments will be shown in the `.po` files. For instance:

```
// note to translators: don't translate the "[noun]" part - it is
// just here to distinguish the string from another "jumps" string
text = strip(gettext("jumps[noun]"), "[noun]");
```

27.4.2. Composition of strings

C programmers use `sprintf()` to compose and concatenate strings. C++ favours streams, but unfortunately, this approach makes translation difficult, because each fragment of text is translated separately, without allowing the translators to rearrange them according to the grammar of the language.

For instance, this code would be problematic:

```
std::cout << _("Current amount: ") << amount
          << _(" Future: ") << future << std::endl;

label.set_text(_("Really delete ") + filename + _(" now?"));
```

So you should either avoid this situation or revert to the C-style `sprintf()`. One possible solution is the compose library which supports syntax such as:

```
label.set_text(compose(_("Really delete %1 now?"), filename));
```

27.4.3. Assuming the displayed size of strings

You never know how much space a string will take on screen when translated. It might very possibly be twice the size of the original English string. Luckily, most *gtkmm* widgets will expand at runtime to the required size.

27.4.4. Unusual words

You should avoid cryptic abbreviations, slang, or jargon. They are usually difficult to translate, and are often difficult for even native speakers to understand. For instance, prefer "application" to "app"

27.4.5. Using non-ASCII characters in strings

Currently, *gettext* does not support non-ASCII characters (i.e. any characters with a code above 127) in source code. For instance, you cannot use the copyright sign (©).

To work around this, you could write a comment in the source code just before the string, telling the translators to use the special character if it is available in their languages. For english, you could then make an American English `en_US.po` translation which used that special charactger.

27.5. Getting help with translations

If your program is free software, there is a whole `GNOME` subproject devoted to helping you make translations, the `GNOME` Translation Project.

The way it works is that you contact the gnome-i18n mailing list to find out how the translators can access your `po/` subdirectory, and to add your project to the big status tables.

Then you make sure you update the file `POTFILES.in` in the `po/` subdirectory (`intltool-update -M` can help with this) so that the translators always access updated `myprogram.pot` files, and simply freeze the strings at least a couple of days before you make a new release, announcing it on gnome-i18n. Depending on the number of strings your program contains and how popular it is, the translations will then start to tick in as `languagename.po` files.

Note that most language teams only consist of 1-3 persons, so if your program contains a lot of strings, it might last a while before anyone has the time to look at it. Also, most translators do not want to waste their time (translating is a very time-consuming task) so if they do not assess your project as being really serious (in the sense that it is polished and being maintained) they may decide to spend their time on some other project.

28. Custom Widgets

gtkmm makes it very easy to derive new widgets by inheriting from an existing widget class, either by deriving from a container and adding child widgets, or by deriving from a single-item widget, and changing its behaviour. But you might occasionally find that no suitable starting point already exists. In this case, you can implement a widget from scratch.

28.1. Custom Containers

When deriving from `Gtk::Container`, you should override the following virtual methods:

- `get_request_mode_vfunc()`: Return what `Gtk::SizeRequestMode` is preferred by the container.
- `get_preferred_width_vfunc()`: Calculate the minimum and natural width of the container.
- `get_preferred_height_vfunc()`: Calculate the minimum and natural height of the container.
- `get_preferred_width_for_height_vfunc()`: Calculate the minimum and natural width of the container, if it would be given the specified height.
- `get_preferred_height_for_width_vfunc()`: Calculate the minimum and natural height of the container, if it would be given the specified width.
- `on_size_allocate()`: Position the child widgets, given the height and width that the container has actually been given.
- `forall_vfunc()`: Call the same callback for each of the children.
- `on_add()`: Add a child widget to the container.
- `on_remove()`: Remove a child widget from the container.

- `child_type_vfunc()`: Return what type of child can be added.

The `get_request_mode_vfunc()`, `get_preferred_width_vfunc()`, `get_preferred_height_vfunc()`, `get_preferred_width_for_height_vfunc()`, `get_preferred_height_for_width_vfunc()`, and `on_size_allocate()` virtual methods control the layout of the child widgets. For instance, if your container has 2 child widgets, with one below the other, your `get_request_mode_vfunc()` might request height-for-width layout. Then your `get_preferred_width_vfunc()` might report the maximum of the widths of the child widgets, and `get_preferred_height_for_width_vfunc()` might report the sum of their heights. If you want padding between the child widgets then you would add that to the width and height too. Your widget's container will use this result to ensure that your widget gets enough space, and not less. By examining each widget's parent, and its parent, this logic will eventually decide the size of the top-level window.

You are not guaranteed to get the `Gtk::SizeRequestMode` that you request. Therefore all four of the `get_preferred_xxx_vfunc()` methods must return sensible values.

`on_size_allocate()` receives the actual height and width that the parent container has decided to give to your widget. This might be more than the minimum, or even more than the natural size, for instance if the top-level window has been expanded. You might choose to ignore the extra space and leave a blank area, or you might choose to expand your child widgets to fill the space, or you might choose to expand the padding between your widgets. It's your container, so you decide. Don't forget to call `set_allocation()` inside your `on_size_allocate()` implementation to actually use the allocated space that has been offered by the parent container.

Unless your container is a top-level window that derives from `Gtk::Window`, you should probably also call `Gtk::Widget::set_has_window(false)` in your constructor. This means that your container does not create its own `Gdk::Window`, but uses its parent's window. (Note the difference between `Gtk::Window` and `Gdk::Window`.) If your container does need its own `Gdk::Window`, and does not derive from `Gtk::Window`, you must also override the `on_realize()` method as described in the Custom Widgets section. And unless your container draws directly onto the underlying `Gdk::Window`, you should probably call `set_redraw_on_allocate(false)` to improve performance.

By overriding `forall_vfunc()` you can allow applications to operate on all of the container's child widgets. For instance, `show_all_children()` uses this to find all the child widgets and show them.

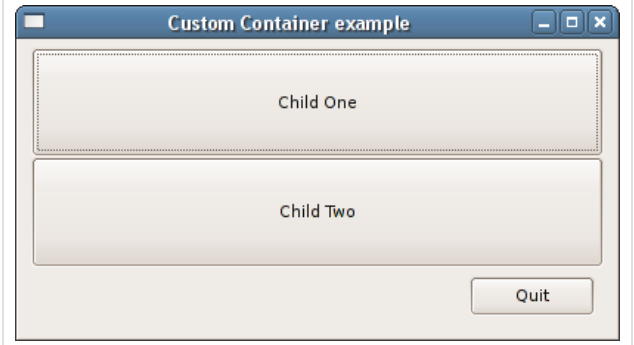
Although your container might have its own method to set the child widgets, you should still provide an implementation for the virtual `on_add()` and `on_remove()` methods from the base class, so that the add() and remove() methods will do something appropriate if they are called.

Your implementation of the `child_type_vfunc()` method should report the type of widget that may be added to your container, if it is not yet full. This is usually `Gtk::Widget::get_type()` to indicate that the container may contain any class derived from `Gtk::Widget`. If the container may not contain any more widgets, then this method should return `G_TYPE_NONE`.

28.1.1. Example

This example implements a container with two child widgets, one above the other. Of course, in this case it would be far simpler just to use a `Gtk::VBox`.

Figure 28-1 **Custom Container**



Source Code

File: `mycontainer.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_CUSTOM_CONTAINER_MYCONTAINER_H
#define GTKMM_CUSTOM_CONTAINER_MYCONTAINER_H

#include <gtkmm/container.h>

class MyContainer : public Gtk::Container
{
public:
  MyContainer();
  virtual ~MyContainer();

  void set_child_widgets(Gtk::Widget& child_one, Gtk::Widget& child_two);

protected:

  //Overrides:
  virtual Gtk::SizeRequestMode get_request_mode_vfunc() const;
  virtual void get_preferred_width_vfunc(int& minimum_width, int& natural_width) const;
  virtual void get_preferred_height_for_width_vfunc(int width, int& minimum_height, int& natural_height) const;
  virtual void get_preferred_height_vfunc(int& minimum_height, int& natural_height) const;
  virtual void get_preferred_width_for_height_vfunc(int height, int& minimum_width, int& natural_width) const;
  virtual void on_size_allocate(Gtk::Allocation& allocation);

  virtual void forall_vfunc(gboolean include_internals, GtkCallback callback, gpointer callback_data);

  virtual void on_add(Gtk::Widget* child);
  virtual void on_remove(Gtk::Widget* child);
  virtual GType child_type_vfunc() const;

  Gtk::Widget* m_child_one;
  Gtk::Widget* m_child_two;
};

#endif //GTKMM_CUSTOM_CONTAINER_MYCONTAINER_H
```

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <gtkmm.h>
#include "mycontainer.h"

class ExampleWindow : public Gtk::Window
{
public:
  ExampleWindow();
  virtual ~ExampleWindow();

protected:
  //Signal handlers:
  void on_button_quit();

  //Child widgets:
  Gtk::Box m_VBox;
  MyContainer m_MyContainer;
  Gtk::Button m_Button_One;
  Gtk::Label m_Label_Two;
  Gtk::ButtonBox m_ButtonBox;
  Gtk::Button m_Button_Quit;
```

```
};

#endif //GTKMM_EXAMPLEWINDOW_H
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include <iostream>
#include "examplewindow.h"

ExampleWindow::ExampleWindow()
: m_VBox(Gtk::ORIENTATION_VERTICAL),
  m_Button_One("Child One"),
  m_Label_Two("Child 2"),
  m_Button_Quit("Quit")
{
  set_title("Custom Container example");
  set_border_width(6);
  set_default_size(400, 200);

  add(m_VBox);

  //Add the child widgets to the custom container:
  m_MyContainer.set_child_widgets(m_Button_One, m_Label_Two);

  m_Label_Two.set_alignment(1.0, 0.5);

  m_VBox.pack_start(m_MyContainer, Gtk::PACK_EXPAND_WIDGET);
  m_VBox.pack_start(m_ButtonBox, Gtk::PACK_SHRINK);

  m_ButtonBox.pack_start(m_Button_Quit, Gtk::PACK_SHRINK);
  m_ButtonBox.set_border_width(6);
  m_ButtonBox.set_layout(Gtk::BUTTONBOX_END);
  m_Button_Quit.signal_clicked().connect( sigc::mem_fun(*this,
              &ExampleWindow::on_button_quit) );

  show_all_children();
}

ExampleWindow::~ExampleWindow()
{
}

void ExampleWindow::on_button_quit()
{
  hide();
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

File: `mycontainer.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include <iostream>
#include <algorithm> // std::max
#include "mycontainer.h"

MyContainer::MyContainer()
: m_child_one(0), m_child_two(0)
{
  set_has_window(false);
  set_redraw_on_allocate(false);
}

MyContainer::~MyContainer()
{
}
```

```
void MyContainer::set_child_widgets(Gtk::Widget& child_one,
        Gtk::Widget& child_two)
{
  m_child_one = &child_one;
  m_child_two = &child_two;

  m_child_one->set_parent(*this);
  m_child_two->set_parent(*this);
}

//This example container is a simplified VBox with at most two children.
Gtk::SizeRequestMode MyContainer::get_request_mode_vfunc() const
{
  return Gtk::SIZE_REQUEST_HEIGHT_FOR_WIDTH;
}

//Discover the total amount of minimum space and natural space needed by
//this container and its children.
void MyContainer::get_preferred_width_vfunc(int& minimum_width, int& natural_width) const
{
  int child_minimum_width[2] = {0, 0};
  int child_natural_width[2] = {0, 0};

  if(m_child_one && m_child_one->get_visible())
    m_child_one->get_preferred_width(child_minimum_width[0], child_natural_width[0]);

  if(m_child_two && m_child_two->get_visible())
    m_child_two->get_preferred_width(child_minimum_width[1], child_natural_width[1]);

  //Request a width equal to the width of the widest visible child.
  minimum_width = std::max(child_minimum_width[0], child_minimum_width[1]);
  natural_width = std::max(child_natural_width[0], child_natural_width[1]);
}

void MyContainer::get_preferred_height_for_width_vfunc(int width,
   int& minimum_height, int& natural_height) const
{
  int child_minimum_height[2] = {0, 0};
  int child_natural_height[2] = {0, 0};
  int nvis_children = 0;

  if(m_child_one && m_child_one->get_visible())
  {
    ++nvis_children;
    m_child_one->get_preferred_height_for_width(width, child_minimum_height[0],
                                                child_natural_height[0]);
  }

  if(m_child_two && m_child_two->get_visible())
  {
    ++nvis_children;
    m_child_two->get_preferred_height_for_width(width, child_minimum_height[1],
                                                child_natural_height[1]);
  }

  //The allocated height will be divided equally among the visible children.
  //Request a height equal to the number of visible children times the height
  //of the highest child.
  minimum_height = nvis_children * std::max(child_minimum_height[0],
                                            child_minimum_height[1]);
  natural_height = nvis_children * std::max(child_natural_height[0],
                                            child_natural_height[1]);
}

void MyContainer::get_preferred_height_vfunc(int& minimum_height, int& natural_height) const
{
  int child_minimum_height[2] = {0, 0};
  int child_natural_height[2] = {0, 0};
  int nvis_children = 0;

  if(m_child_one && m_child_one->get_visible())
  {
    ++nvis_children;
    m_child_one->get_preferred_height(child_minimum_height[0], child_natural_height[0]);
  }

  if(m_child_two && m_child_two->get_visible())
  {
    ++nvis_children;
    m_child_two->get_preferred_height(child_minimum_height[1], child_natural_height[1]);
  }

  //The allocated height will be divided equally among the visible children.
  //Request a height equal to the number of visible children times the height
  //of the highest child.
  minimum_height = nvis_children * std::max(child_minimum_height[0],
                                            child_minimum_height[1]);
```

```
  natural_height = nvis_children * std::max(child_natural_height[0],
                                            child_natural_height[1]);
}

void MyContainer::get_preferred_width_for_height_vfunc(int height,
   int& minimum_width, int& natural_width) const
{
  int child_minimum_width[2] = {0, 0};
  int child_natural_width[2] = {0, 0};
  int nvis_children = 0;

  //Get number of visible children.
  if(m_child_one && m_child_one->get_visible())
    ++nvis_children;
  if(m_child_two && m_child_two->get_visible())
    ++nvis_children;

  if(nvis_children > 0)
  {
    //Divide the height equally among the visible children.
    const int height_per_child = height / nvis_children;

    if(m_child_one && m_child_one->get_visible())
      m_child_one->get_preferred_width_for_height(height_per_child,
                  child_minimum_width[0], child_natural_width[0]);

    if(m_child_two && m_child_two->get_visible())
      m_child_two->get_preferred_width_for_height(height_per_child,
                  child_minimum_width[1], child_natural_width[1]);
  }

  //Request a width equal to the width of the widest child.
  minimum_width = std::max(child_minimum_width[0], child_minimum_width[1]);
  natural_width = std::max(child_natural_width[0], child_natural_width[1]);
}

void MyContainer::on_size_allocate(Gtk::Allocation& allocation)
{
  //Do something with the space that we have actually been given:
  //(We will not be given heights or widths less than we have requested, though
  //we might get more.)

  //Use the offered allocation for this container:
  set_allocation(allocation);

  //Get number of visible children.
  int nvis_children = 0;
  if(m_child_one && m_child_one->get_visible())
    ++nvis_children;
  if(m_child_two && m_child_two->get_visible())
    ++nvis_children;

  if(nvis_children <= 0)
    return;

  //Assign space to the children:
  Gtk::Allocation child_allocation_one;
  Gtk::Allocation child_allocation_two;

  //Place the first child at the top-left:
  child_allocation_one.set_x( allocation.get_x() );
  child_allocation_one.set_y( allocation.get_y() );

  //Make it take up the full width available:
  child_allocation_one.set_width( allocation.get_width() );

  if(m_child_one && m_child_one->get_visible())
  {
    //Divide the height equally among the visible children.
    child_allocation_one.set_height( allocation.get_height() / nvis_children);
    m_child_one->size_allocate(child_allocation_one);
  }
  else
    child_allocation_one.set_height(0);

  //Place the second child below the first child:
  child_allocation_two.set_x( allocation.get_x() );
  child_allocation_two.set_y( allocation.get_y() +
          child_allocation_one.get_height());

  //Make it take up the full width available:
  child_allocation_two.set_width( allocation.get_width() );

  //Make it take up the remaining height:
  child_allocation_two.set_height( allocation.get_height() -
          child_allocation_one.get_height());

  if(m_child_two && m_child_two->get_visible())
```

```
    m_child_two->size_allocate(child_allocation_two);
}

void MyContainer::forall_vfunc(gboolean, GtkCallback callback, gpointer callback_data)
{
  if(m_child_one)
    callback(m_child_one->gobj(), callback_data);

  if(m_child_two)
    callback(m_child_two->gobj(), callback_data);
}

void MyContainer::on_add(Gtk::Widget* child)
{
  if(!m_child_one)
  {
    m_child_one = child;
    m_child_one->set_parent(*this);
  }
  else if(!m_child_two)
  {
    m_child_two = child;
    m_child_two->set_parent(*this);
  }
}

void MyContainer::on_remove(Gtk::Widget* child)
{
  if(child)
  {
    const bool visible = child->get_visible();
    bool found = false;

    if(child == m_child_one)
    {
      m_child_one = 0;
      found = true;
    }
    else if(child == m_child_two)
    {
      m_child_two = 0;
      found = true;
    }

    if(found)
    {
      child->unparent();

      if(visible)
        queue_resize();
    }
  }
}

GType MyContainer::child_type_vfunc() const
{
  //If there is still space for one widget, then report the type of widget that
  //may be added.
  if(!m_child_one || !m_child_two)
    return Gtk::Widget::get_type();
  else
  {
    //No more widgets may be added.
    return G_TYPE_NONE;
  }
}
```

28.2. Custom Widgets

By deriving directly from `Gtk::Widget` you can do all the drawing for your widget directly, instead of just arranging child widgets. For instance, a `Gtk::Label` draws the text of the label, but does not do this by using other widgets.

When deriving from `Gtk::Widget`, you should override the following virtual methods. The methods marked (optional) need not be overridden in all custom widgets. The base class's methods may be appropriate.

- `get_request_mode_vfunc()`: (optional) Return what `Gtk::SizeRequestMode` is preferred by the widget.
- `get_preferred_width_vfunc()`: Calculate the minimum and natural width of the widget.

- `get_preferred_height_vfunc()`: Calculate the minimum and natural height of the widget.
- `get_preferred_width_for_height_vfunc()`: Calculate the minimum and natural width of the widget, if it would be given the specified height.
- `get_preferred_height_for_width_vfunc()`: Calculate the minimum and natural height of the widget, if it would be given the specified width.
- `on_size_allocate()`: Position the widget, given the height and width that it has actually been given.
- `on_realize()`: Associate a `Gdk::Window` with the widget.
- `on_unrealize()`: (optional) Break the association with the `Gdk::Window`.
- `on_map()`: (optional)
- `on_unmap()`: (optional)
- `on_draw()`: Draw on the supplied `Cairo::Context`.

The first 6 methods in the previous table are also overridden in custom containers. They are briefly described in the Custom Containers section.

Most custom widgets need their own `Gdk::Window` to draw on. Then you can call `Gtk::Widget::set_has_window(true)` in your constructor. (This is the default value.) If you do not call `set_has_window(false)`, you must override `on_realize()` and call `Gtk::Widget::set_realized()` and `Gtk::Widget::set_window()` from there.

28.2.1. Example

This example implements a widget which draws a Penrose triangle.

Figure 28-2 **Custom Widget**

Source Code

File: `mywidget.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_CUSTOM_WIDGET_MYWIDGET_H
#define GTKMM_CUSTOM_WIDGET_MYWIDGET_H

#include <gtkmm/widget.h>
#include <gtkmm/cssprovider.h>

class MyWidget : public Gtk::Widget
{
public:
  MyWidget();
  virtual ~MyWidget();

protected:

  //Overrides:
  virtual Gtk::SizeRequestMode get_request_mode_vfunc() const;
  virtual void get_preferred_width_vfunc(int& minimum_width, int& natural_width) const;
  virtual void get_preferred_height_for_width_vfunc(int width, int& minimum_height, int& natural_height) const;
  virtual void get_preferred_height_vfunc(int& minimum_height, int& natural_height) const;
  virtual void get_preferred_width_for_height_vfunc(int height, int& minimum_width, int& natural_width) const;
  virtual void on_size_allocate(Gtk::Allocation& allocation);
  virtual void on_map();
  virtual void on_unmap();
  virtual void on_realize();
  virtual void on_unrealize();
  virtual bool on_draw(const Cairo::RefPtr<Cairo::Context>& cr);

  Glib::RefPtr<Gdk::Window> m_refGdkWindow;
  Glib::RefPtr<Gtk::CssProvider> m_refStyleProvider;

  int m_scale;
};
```

```
#endif //GTKMM_CUSTOM_WIDGET_MYWIDGET_H
```

File: `examplewindow.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLEWINDOW_H
#define GTKMM_EXAMPLEWINDOW_H

#include <gtkmm.h>
#include "mywidget.h"

class ExampleWindow : public Gtk::Window
{
public:
  ExampleWindow();
  virtual ~ExampleWindow();

protected:
  //Signal handlers:
  void on_button_quit();

  //Child widgets:
  Gtk::Box m_VBox;
  MyWidget m_MyWidget;
  Gtk::ButtonBox m_ButtonBox;
  Gtk::Button m_Button_Quit;
};

#endif //GTKMM_EXAMPLEWINDOW_H
```

File: `mywidget.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "mywidget.h"
#include <gdkmm/general.h>  // for cairo helper functions
#include <iostream>
//#include <gtk/gtkwidget.h> //For GTK_IS_WIDGET()
#include <cstring>


MyWidget::MyWidget() :
  //The GType name will actually be gtkmm__CustomObject_mywidget
  Glib::ObjectBase("mywidget"),
  Gtk::Widget(),
  m_scale(1000)
{
  set_has_window(true);

  //This shows the GType name, which must be used in the CSS file.
  std::cout << "GType name: " << G_OBJECT_TYPE_NAME(gobj()) << std::endl;

  //This shows that the GType still derives from GtkWidget:
  //std::cout << "Gtype is a GtkWidget?:" << GTK_IS_WIDGET(gobj()) << std::endl;

  //Install a style so that an aspect of this widget may be themed via a CSS
  //style sheet file:
  gtk_widget_class_install_style_property(GTK_WIDGET_CLASS(
              G_OBJECT_GET_CLASS(gobj())),
      g_param_spec_int("example_scale",
        "Scale of Example Drawing",
        "The scale to use when drawing. This is just a silly example.",
        G_MININT,
        G_MAXINT,
        500,
        G_PARAM_READABLE) );

  m_refStyleProvider = Gtk::CssProvider::create();
  Glib::RefPtr<Gtk::StyleContext> refStyleContext = get_style_context();
  refStyleContext->add_provider(m_refStyleProvider,
    GTK_STYLE_PROVIDER_PRIORITY_APPLICATION);

  try
  {
    m_refStyleProvider->load_from_path("custom_gtk.css");
  }
  catch(const Glib::Error& ex)
  {
    std::cerr << "Gtk::CssProvider::load_from_path() failed: " << ex.what() << std::endl;
  }
}

MyWidget::~MyWidget()
{
}
```

```
Gtk::SizeRequestMode MyWidget::get_request_mode_vfunc() const
{
  //Accept the default value supplied by the base class.
  return Gtk::Widget::get_request_mode_vfunc();
}

//Discover the total amount of minimum space and natural space needed by
//this widget.
//Let's make this simple example widget always need minimum 60 by 50 and
//natural 100 by 70.
void MyWidget::get_preferred_width_vfunc(int& minimum_width, int& natural_width) const
{
  minimum_width = 60;
  natural_width = 100;
}

void MyWidget::get_preferred_height_for_width_vfunc(int /* width */,
   int& minimum_height, int& natural_height) const
{
  minimum_height = 50;
  natural_height = 70;
}

void MyWidget::get_preferred_height_vfunc(int& minimum_height, int& natural_height) const
{
  minimum_height = 50;
  natural_height = 70;
}

void MyWidget::get_preferred_width_for_height_vfunc(int /* height */,
   int& minimum_width, int& natural_width) const
{
  minimum_width = 60;
  natural_width = 100;
}

void MyWidget::on_size_allocate(Gtk::Allocation& allocation)
{
  //Do something with the space that we have actually been given:
  //(We will not be given heights or widths less than we have requested, though
  //we might get more)

  //Use the offered allocation for this container:
  set_allocation(allocation);

  if(m_refGdkWindow)
  {
    m_refGdkWindow->move_resize( allocation.get_x(), allocation.get_y(),
            allocation.get_width(), allocation.get_height() );
  }
}

void MyWidget::on_map()
{
  //Call base class:
  Gtk::Widget::on_map();
}

void MyWidget::on_unmap()
{
  //Call base class:
  Gtk::Widget::on_unmap();
}

void MyWidget::on_realize()
{
  //Do not call base class Gtk::Widget::on_realize().
  //It's intended only for widgets that set_has_window(false).

  set_realized();

  //Get the themed style from the CSS file:
  get_style_property("example_scale", m_scale);
  std::cout << "m_scale (example_scale from the theme/css-file) is: "
      << m_scale << std::endl;

  if(!m_refGdkWindow)
  {
    //Create the GdkWindow:

    GdkWindowAttr attributes;
    memset(&attributes, 0, sizeof(attributes));

    Gtk::Allocation allocation = get_allocation();

    //Set initial position and size of the Gdk::Window:
    attributes.x = allocation.get_x();
```

```
    attributes.y = allocation.get_y();
    attributes.width = allocation.get_width();
    attributes.height = allocation.get_height();

    attributes.event_mask = get_events () | Gdk::EXPOSURE_MASK;
    attributes.window_type = GDK_WINDOW_CHILD;
    attributes.wclass = GDK_INPUT_OUTPUT;

    m_refGdkWindow = Gdk::Window::create(get_parent_window(), &attributes,
            GDK_WA_X | GDK_WA_Y);
    set_window(m_refGdkWindow);

    //set colors
    override_background_color(Gdk::RGBA("red"));
    override_color(Gdk::RGBA("blue"));

    //make the widget receive expose events
    m_refGdkWindow->set_user_data(gobj());
  }
}

void MyWidget::on_unrealize()
{
  m_refGdkWindow.reset();

  //Call base class:
  Gtk::Widget::on_unrealize();
}

bool MyWidget::on_draw(const Cairo::RefPtr<Cairo::Context>& cr)
{
  const double scale_x = (double)get_allocation().get_width() / m_scale;
  const double scale_y = (double)get_allocation().get_height() / m_scale;

  // paint the background
  Gdk::Cairo::set_source_rgba(cr, get_style_context()->get_background_color());
  cr->paint();

  // draw the foreground
  Gdk::Cairo::set_source_rgba(cr, get_style_context()->get_color());
  cr->move_to(155.*scale_x, 165.*scale_y);
  cr->line_to(155.*scale_x, 838.*scale_y);
  cr->line_to(265.*scale_x, 900.*scale_y);
  cr->line_to(849.*scale_x, 564.*scale_y);
  cr->line_to(849.*scale_x, 438.*scale_y);
  cr->line_to(265.*scale_x, 100.*scale_y);
  cr->line_to(155.*scale_x, 165.*scale_y);
  cr->move_to(265.*scale_x, 100.*scale_y);
  cr->line_to(265.*scale_x, 652.*scale_y);
  cr->line_to(526.*scale_x, 502.*scale_y);
  cr->move_to(369.*scale_x, 411.*scale_y);
  cr->line_to(633.*scale_x, 564.*scale_y);
  cr->move_to(369.*scale_x, 286.*scale_y);
  cr->line_to(369.*scale_x, 592.*scale_y);
  cr->move_to(369.*scale_x, 286.*scale_y);
  cr->line_to(849.*scale_x, 564.*scale_y);
  cr->move_to(633.*scale_x, 564.*scale_y);
  cr->line_to(155.*scale_x, 838.*scale_y);
  cr->stroke();

  return true;
}
```

File: `examplewindow.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"

ExampleWindow::ExampleWindow()
: m_VBox(Gtk::ORIENTATION_VERTICAL),
  m_Button_Quit("Quit")
{
  set_title("Custom Widget example");
  set_border_width(6);
  set_default_size(400, 200);

  add(m_VBox);
  m_VBox.pack_start(m_MyWidget, Gtk::PACK_EXPAND_WIDGET);
  m_MyWidget.show();

  m_VBox.pack_start(m_ButtonBox, Gtk::PACK_SHRINK);

  m_ButtonBox.pack_start(m_Button_Quit, Gtk::PACK_SHRINK);
  m_ButtonBox.set_border_width(6);
  m_ButtonBox.set_layout(Gtk::BUTTONBOX_END);
  m_Button_Quit.signal_clicked().connect( sigc::mem_fun(*this, &ExampleWindow::on_button_quit) );
```

```
  show_all_children();
}

ExampleWindow::~ExampleWindow()
{
}

void ExampleWindow::on_button_quit()
{
  hide();
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "examplewindow.h"
#include <gtkmm/main.h>

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

  ExampleWindow window;
  //Shows the window and returns when it is closed.
  Gtk::Main::run(window);

  return 0;
}
```

29. Recommended Techniques

This section is simply a gathering of wisdom, general style guidelines and hints for creating *gtkmm* applications.

Use GNU *autoconf* and *automake*! They are your friends :) *Automake* examines C files, determines how they depend on each other, and generates a `Makefile` so the files can be compiled in the correct order. *Autoconf* permits automatic configuration of software installation, handling a large number of system quirks to increase portability..

Subclass Widgets to better organize your code. You should probably subclass your main `Window` at least. Then you can make your child Widgets and signal handlers members of that class.

Create your own signals instead of passing pointers around. Objects can communicate with each other via signals and signal handlers. This is much simpler than objects holding pointers to each other and calling each other's methods. *gtkmm*'s classes uses special versions of `sigc::signal`, but you should use normal `sigc::signal`s, as described in the *libsigc++* documentation.

29.1. Application Lifetime

Most applications will have only one `Window`, or only one main window. These applications can use the `Gtk::Main::run(Gtk::Window&)` overload. It shows the window and returns when the window has been hidden. This might happen when the user closes the window, or when your code decides to `hide()` the window. You can prevent the user from closing the window (for instance, if there are unsaved changes) by overriding `Gtk::Window::on_delete_event()`.

Most of our examples use this technique.

29.2. Using a *gtkmm* widget

Our examples all tend to have the same structure. They follow these steps for using a `Widget`:

1. Declare a variable of the type of `Widget` you wish to use, generally as member variable of a derived container class. You could also declare a pointer to the

widget type, and then create it with `new` in your code. Even when using the widget via a pointer, it's still probably best to make that pointer a member variable of a container class so that you can access it later.

2. Set the attributes of the widget. If the widget has no default constructor, then you will need to initialize the widget in the initalizer list of your container class's constructor.

3. Connect any signals you wish to use to the appropriate handlers.

4. Pack the widget into a container using the appropriate call, e.g. `Gtk::Container::add()` or `pack_start()`.

5. Call `show()` to display the widget.

`Gtk::Widget::show()` lets *gtkmm* know that we have finished setting the attributes of the widget, and that it is ready to be displayed. You can use `Gtk::Widget::hide()` to make it disappear again. The order in which you show the widgets is not important, but we do suggest that you show the top-level window last; this way, the whole window will appear with its contents already drawn. Otherwise, the user will first see a blank window, into which the widgets will be gradually drawn.

30. Contributing

This document, like so much other great software out there, was created for free by volunteers. If you are at all knowledgeable about any aspect of *gtkmm* that does not already have documentation, please consider contributing to this document.

Ideally, we would like you to provide a patch to the `docs/tutorial/C/gtkmm-tutorial-in.xml` file. This file is currently in the `gtkmm-documentation` module in GNOME git.

If you do decide to contribute, please post your contribution to the *gtkmm* mailing list at <gtkmm-list@gnome.org>. Also, be aware that the entirety of this document is free, and any addition you provide must also be free. That is, people must be able to use any portion of your examples in their programs, and copies of this document (including your contribution) may be distributed freely.

A. The RefPtr smartpointer

`Glib::RefPtr` is a smartpointer. Specifically, it is a reference-counting smartpointer. You might be familiar with `std::auto_ptr<>`, which is also a smartpointer, but `Glib::RefPtr<>` is much simpler, and more useful. We expect a future version of the C++ Standard Library to contain a reference-counting shared smartpointer, and a future version of *gtkmm* might possibly use that instead.

Reference

A smartpointer acts much like a normal pointer. Here are a few examples.

A.1. Copying

You can copy `RefPtr`s, just like normal pointers. But unlike normal pointers, you don't need to worry about deleting the underlying instance.

```
Glib::RefPtr<Gdk::Pixbuf> refPixbuf = Gdk::Pixbuf::create_from_file(filename);
Glib::RefPtr<Gdk::Pixbuf> refPixbuf2 = refPixbuf;
```

Of course this means that you can store `RefPtr`s in standard containers, such as `std::vector` or `std::list`.

```
std::list< Glib::RefPtr<Gdk::Pixbuf> > listPixbufs;
Glib::RefPtr<Gdk::Pixbuf> refPixbuf = Gdk::Pixbuf::create_from_file(filename);
listPixbufs.push_back(refPixbuf);
```

A.2. Dereferencing

You can dereference a smartpointer with the -> operator, to call the methods of the underlying instance, just like a normal pointer.

```
Glib::RefPtr<Gdk::Pixbuf> refPixbuf = Gdk::Pixbuf::create_from_file(filename);
int width = refPixbuf->get_width();
```

But unlike most smartpointers, you can't use the * operator to access the underlying instance.

```
Glib::RefPtr<Gdk::Pixbuf> refPixbuf = Gdk::Pixbuf::create_from_file(filename);
Gdk::Pixbuf& underlying = *refPixbuf; //Syntax error - will not compile.
```

A.3. Casting

You can cast `RefPtrs` to base types, just like normal pointers.

```
Glib::RefPtr<Gtk::TreeStore> refStore = Gtk::TreeStore::create(columns);
Glib::RefPtr<Gtk::TreeModel> refModel = refStore;
```

This means that any method which takes a `const Glib::RefPtr<BaseType>` argument can also take a `const Glib::RefPtr<DerivedType>`. The cast is implicit, just as it would be for a normal pointer.

You can also cast to a derived type, but the syntax is a little different than with a normal pointer.

```
Glib::RefPtr<Gtk::TreeStore> refStore =
Glib::RefPtr<Gtk::TreeStore>::cast_dynamic(refModel);
Glib::RefPtr<Gtk::TreeStore> refStore2 =
Glib::RefPtr<Gtk::TreeStore>::cast_static(refModel);
```

A.4. Checking for null

Just like normal pointers, you can check whether a `RefPtr` points to anything.

```
Glib::RefPtr<Gtk::TreeModel> refModel = m_TreeView.get_model();
if(refModel)
{
  int cols_count = refModel->get_n_columns();
  ...
}
```

But unlike normal pointers, `RefPtrs` are automatically initialized to null so you don't need to remember to do that yourself.

A.5. Constness

The use of the `const` keyword in C++ is not always clear. You might not realise that `const Something*` declares a pointer to a `const Something`, The pointer can be changed, but not the `Something` that it points to.

Therefore, the `RefPtr` equivalent of `Something*` for a method parameter is `const Glib::RefPtr<Something>&`, and the equivalent of `const Something*` is `const Glib::RefPtr<const Something>&`.

The `const ... &` around both is just for efficiency, like using `const std::string&` instead of `std::string` for a method parameter to avoid unnecessary copying.

B. Signals

B.1. Connecting signal handlers

gtkmm widget classes have signal accessor methods, such as `Gtk::Button::signal_clicked()`, which allow you to connect your signal handler. Thanks to the flexibility of *libsigc++*, the callback library used by *gtkmm*, the signal handler can be almost any kind of function, but you will probably want to use a class method. Among *GTK+* C coders, these signal handlers are often named callbacks.

Here's an example of a signal handler being connected to a signal:

```
#include <gtkmm/button.h>

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

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

There's rather a lot to think about in this (non-functional) code. First let's identify the parties involved:

- The signal handler is `on_button_clicked()`.
- We're hooking it up to the `Gtk::Button` object called `button`.
- When the Button emits its `clicked` signal, `on_button_clicked()` will be called.

Now let's look at the connection again:

```
    ...
    button.signal_clicked().connect(sigc::ptr_fun(&on_button_clicked));
    ...
```

Note that we don't pass a pointer to `on_button_clicked()` directly to the signal's `connect()` method. Instead, we call `sigc::ptr_fun()`, and pass the result to `connect()`.

`sigc::ptr_fun()` generates a `sigc::slot`. A slot is an object which looks and feels like a function, but is actually an object. These are also known as function objects, or functors. `sigc::ptr_fun()` generates a slot for a standalone function or static method. `sigc::mem_fun()` generates a slot for a member method of a particular instance.

Here's a slightly larger example of slots in action:

```
void on_button_clicked();

class some_class
{
    void on_button_clicked();
};

some_class some_object;

main()
{
    Gtk::Button button;
    button.signal_clicked().connect( sigc::ptr_fun(&on_button_clicked) );
    button.signal_clicked().connect( sigc::mem_fun(some_object, &some_class::on_button_clicked) );
```

```
}
```

The first call to `connect()` is just like the one we saw last time; nothing new here.

The next is more interesting. `sigc::mem_fun()` is called with two arguments. The first argument is `some_object`, which is the object that our new slot will be pointing at. The second argument is a pointer to one of its methods. This particular version of `sigc::mem_fun()` creates a slot which will, when "called", call the pointed-to method of the specified object, in this case `some_object.on_button_clicked()`.

Another thing to note about this example is that we made the call to `connect()` twice for the same signal object. This is perfectly fine - when the button is clicked, both signal handlers will be called.

We just told you that the button's `clicked` signal is expecting to call a method with no arguments. All signals have requirements like this - you can't hook a function with two arguments to a signal expecting none (unless you use an adapter, such as `sigc::bind()`, of course). Therefore, it's important to know what type of signal handler you'll be expected to connect to a given signal.

B.2. Writing signal handlers

To find out what type of signal handler you can connect to a signal, you can look it up in the reference documentation or the header file. Here's an example of a signal declaration you might see in the *gtkmm* headers:

```
Glib::SignalProxy1<bool, Gtk::DirectionType> signal_focus()
```

Other than the signal's name (`focus`), two things are important to note here: the number following the word `SignalProxy` at the beginning (1, in this case), and the types in the list (`bool` and `Gtk::DirectionType`). The number indicates how many arguments the signal handler should have; the first type, `bool`, is the type that the signal handler should return; and the next type, `Gtk::DirectionType`, is the type of this signal's first, and only, argument. By looking at the reference documentation, you can see the names of the arguments too.

The same principles apply for signals which have more arguments. Here's one with three (taken from `<gtkmm/editable.h>`):

```
Glib::SignalProxy3<void, const Glib::ustring&, int, int*> signal_insert_text()
```

It follows the same form. The number 3 at the end of the type's name indicates that our signal handler will need three arguments. The first type in the type list is `void`, so that should be our signal handler's return type. The following three types are the argument types, in order. Our signal handler's prototype could look like this:

```
void on_insert_text(const Glib::ustring& text, int length, int* position);
```

B.3. Disconnecting signal handlers

Let's take another look at a Signal's `connect` method:

```
sigc::signal<void,int>::iterator signal<void,int>::connect( const sigc::slot<void,int>& );
```

Notice that the return value is of type `sigc::signal<void,int>::iterator`. This can be implicitely converted into a `sigc::connection` which in turn can be used to control the connection. By keeping a connection object you can disconnect its associated signal handler using the method `sigc::connection::disconnect()`.

B.4. Overriding default signal handlers

So far we've told you to perform actions in response to button-presses and the like by

handling signals. That's certainly a good way to do things, but it's not the only way.

Instead of laboriously connecting signal handlers to signals, you can simply make a new class which inherits from a widget - say, a Button - and then override the default signal handler, such as Button::on_clicked(). This can be a lot simpler than hooking up signal handlers for everything.

Subclassing isn't always the best way to accomplish things. It is only useful when you want the widget to handle its own signal by itself. If you want some other class to handle the signal then you'll need to connect a separate handler. This is even more true if you want several objects to handle the same signal, or if you want one signal handler to respond to the same signal from different objects.

gtkmm classes are designed with overriding in mind; they contain virtual member methods specifically intended to be overridden.

Let's look at an example of overriding:

```
#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();
}
```

Here we define a new class called `OverriddenButton`, which inherits from `Gtk::Button`. The only thing we change is the `on_clicked()` method, which is called whenever `Gtk::Button` emits the `clicked` signal. This method prints "Hello World" to `stdout`, and then calls the original, overridden method, to let `Gtk::Button` do what it would have done had we not overridden.

You don't always need to call the parent's method; there are times when you might not want to. Note that we called the parent method *after* writing "Hello World", but we could have called it before. In this simple example, it hardly matters much, but there are times when it will. With signals, it's not quite so easy to change details like this, and you can do something here which you can't do at all with connected signal handlers: you can call the parent method in the *middle* of your custom code.

B.5. Binding extra arguments

If you use one signal handler to catch the same signal from several widgets, you might like that signal handler to receive some extra information. For instance, you might want to know which button was clicked. You can do this with `sigc::bind()`. Here's some code from the helloworld2 example, which you will encounter later.

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

This says that we want the signal to send an extra `Glib::ustring` argument to the signal handler, and that the value of that argument should be "button 1". Of course we will need to add that extra argument to the declaration of our signal handler:

```
virtual void on_button_clicked(Glib::ustring data);
```

Of course, a normal "clicked" signal handler would have no arguments.

`sigc::bind()` is not commonly used, but you might find it helpful sometimes. If you are familiar with *GTK+* programming then you have probably noticed that this is similar to the extra `gpointer data` arguments which all GTK+ callbacks have. This is generally overused in *GTK+* to pass information that should be stored as member data in a derived widget, but widget derivation is very difficult in C. We have far less need of this hack in *gtkmm*.

B.6. X Event signals

The `Widget` class has some special signals which correspond to the underlying X-Windows events. These are suffixed by `_event`; for instance, `Widget::signal_button_pressed_event()`.

You might occasionally find it useful to handle X events when there's something you can't accomplish with normal signals. `Gtk::Button`, for example, does not send mouse-pointer coordinates with its `clicked` signal, but you could handle `button_pressed_event` if you needed this information. X events are also often used to handle key-presses.

These signals behave slightly differently. The value returned from the signal handler indicates whether it has fully "handled" the event. If the value is `false` then *gtkmm* will pass the event on to the next signal handler. If the value is `true` then no other signal handlers will need to be called.

Handling an X event doesn't affect the Widget's other signals. If you handle `button_pressed_event` for `Gtk::Button`, you'll still be able to get the `clicked` signal. They are emitted at (nearly) the same time.

Note also that not all widgets receive all X events by default. To receive additional X events, you can use `Gtk::Widget::set_events()` before showing the widget, or `Gtk::Widget::add_events()` after showing the widget. However, some widgets must first be placed inside an `EventBox` widget. See the Widgets Without X-Windows chapter.

Here's a simple example:

```
bool on_button_press(GdkEventButton* event);
Gtk::Button button("label");
button.signal_button_press_event().connect( sigc::ptr_fun(&on_button_press) );
```

When the mouse is over the button and a mouse button is pressed, `on_button_press()` will be called.

`GdkEventButton` is a structure containing the event's parameters, such as the coordinates of the mouse pointer at the time the button was pressed. There are several different types of `GdkEvent` structures for the various events.

B.6.1. Signal Handler sequence

By default, your signal handlers are called after any previously-connected signal handlers. However, this can be a problem with the X Event signals. For instance, the existing signal handlers, or the default signal handler, might return `true` to stop other signal handlers from being called. To specify that your signal handler should be called before the other signal handlers, so that it will always be called, you can specify `false` for the optional `after` parameter. For instance,

```
button.signal_button_press_event().connect( sigc::ptr_fun(&on_mywindow_button_press), false );
```

The event is delivered first to the widget the event occurred in. If all signal handlers in that widget return `false` (indicating that the event has not been handled), then the signal will be propagated to the parent widget and emitted there. This continues all the way up to the top-level widget if no one handles the event.

C. Creating your own signals

Now that you've seen signals and signal handlers in *gtkmm*, you might like to use the same technique to allow interaction between your own classes. That's actually very simple by using the *libsigc++* library directly.

This isn't purely a *gtkmm* or GUI issue. *gtkmm* uses *libsigc++* to implement its proxy wrappers for the *GTK+* signal system, but for new, non-GTK+ signals, you can create pure C++ signals, using the `sigc::signal<>` template.

For instance, to create a signal that sends 2 parameters, a `bool` and an `int`, just declare a `sigc::signal`, like so:

```
sigc::signal<void, bool, int> signal_something;
```

You could just declare that signal as a public member variable, but some people find that distasteful and prefer to make it available via an accessor method, like so:

```
class Server
{
public:
  //signal accessor:
  typedef sigc::signal<void, bool, int> type_signal_something;
  type_signal_something signal_something();

protected:
  type_signal_something m_signal_something;
};

Server::type_signal_something Server::signal_something()
{
  return m_signal_something;
}
```

You can then connect to the signal using the same syntax used when connecting to *gtkmm* signals. For instance,

```
server.signal_something().connect(
  sigc::mem_fun(client, &Client::on_server_something) );
```

C.1. Example

C.1. Example

This is a full working example that defines and uses custom signal.

Source Code

File: `client.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLE_CLIENT_H
#define GTKMM_EXAMPLE_CLIENT_H

#include <sigc++/sigc++.h>

//Client must inherit from sigc::trackable.
//because libsigc++ needs to keep track of the lifetime of signal handlers.
class Client : public sigc::trackable
{
public:
  Client();
  virtual ~Client();

  //Signal handler:
  void on_server_something(bool a, int b);
};

#endif //GTKMM_EXAMPLE_CLIENT_H
```

File: `server.h` (For use with gtkmm 3, not gtkmm 2)

```
#ifndef GTKMM_EXAMPLE_SERVER_H
#define GTKMM_EXAMPLE_SERVER_H

#include <sigc++/sigc++.h>

class Server
{
public:
  Server();
  virtual ~Server();
```

```
  void do_something();

  //signal accessor:
  typedef sigc::signal<void, bool, int> type_signal_something;
  type_signal_something signal_something();

protected:
  type_signal_something m_signal_something;
};

#endif //GTKMM_EXAMPLE_SERVER_H
```

File: `server.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "server.h"
#include <iostream>

Server::Server()
{
}

Server::~Server()
{
}

Server::type_signal_something Server::signal_something()
{
  return m_signal_something;
}

void Server::do_something()
{
  m_signal_something.emit(false, 5);
}
```

File: `main.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "server.h"
#include "client.h"
#include <iostream>

int main(int, char**)
{
  Server server;
  Client client;

  //Connect a Server signal to the signal handler in Client.
  server.signal_something().connect(sigc::mem_fun(client,
              &Client::on_server_something) );

  std::cout << "Before Server::do_something()" << std::endl;

  //Tell the server to do something that will eventually cause it to emit the
  //"something" signal.
  server.do_something();    // Client::on_server_something() will run before
                            // Server::do_something() has completed.

  std::cout << "After Server::do_something()" << std::endl;

  return 0;
}
```

File: `client.cc` (For use with gtkmm 3, not gtkmm 2)

```
#include "client.h"
#include <iostream>

Client::Client()
{
}

Client::~Client()
{
}

void Client::on_server_something(bool a, int b)
{
  std::cout << "Client::on_server_something() called with these parameters: "
```

```
      << a << ", " << b << std::endl;
}
```

D. Comparison with other signalling systems

(An aside: *GTK+* calls this scheme "signalling"; the sharp-eyed reader with GUI toolkit experience will note that this same design is often seen under the name of "broadcaster-listener" (e.g., in Metrowerks' PowerPlant framework for the Macintosh). It works in much the same way: one sets up `broadcasters`, and then connects `listeners` to them; the broadcaster keeps a list of the objects listening to it, and when someone gives the broadcaster a message, it calls all of its objects in its list with the message. In *gtkmm*, signal objects play the role of broadcasters, and slots play the role of listeners - sort of. More on this later.)

gtkmm signal handlers are strongly-typed, whereas *GTK+* C code allows you to connect a callback with the wrong number and type of arguments, leading to a segfault at runtime. And, unlike *Qt*, *gtkmm* achieves this without modifying the C++ language.

Re. Overriding signal handlers: You can do this in the straight-C world of GTK+ too; that's what GTK's object system is for. But in GTK+, you have to go through some complicated procedures to get object-oriented features like inheritance and overloading. In C++, it's simple, since those features are supported in the language itself; you can let the compiler do the dirty work.

This is one of the places where the beauty of C++ really comes out. One wouldn't think of subclassing a GTK+ widget simply to override its action method; it's just too much trouble. In GTK+, you almost always use signals to get things done, unless you're writing a new widget. But because overriding methods is so easy in C++, it's entirely practical - and sensible - to subclass a button for that purpose.

E. *gtkmm* and Win32

One of the major advantages of *gtkmm* is that it is crossplatform. *gtkmm* programs written on other platforms such as GNU/Linux can generally be transferred to Windows (and vice versa) with few modifications to the source.

gtkmm currently works with the MingW/GCC3.4 compiler and Microsoft Visual C++ 2005 or later (including the freely available express editions) on the Windows platform. There is an installer available for gtkmm on Microsoft Windows. Refer to http://live.gnome.org/gtkmm/MSWindows for instructions how to use it.

E.1. Building *gtkmm* on Win32

Please see http://live.gnome.org/gtkmm/MSWindows/BuildingGtkmm for instructions on how to build gtkmm on Windws.

F. Working with gtkmm's Source Code

If you are interested in helping out with the development of *gtkmm*, or fixing a bug in *gtkmm*, you'll probably need to build the development version of *gtkmm*. However, you should not install a development version over your stable version. Instead, you should install it alongside your existing *gtkmm* installation, in a separate path.

The easiest way to do this is using jhbuild. *jhbuild* is a program that makes building GNOME software much easier by calculating dependencies and building things in the correct order. This section will give a brief explanation of how to set up *jhbuild* to build and install *gtkmm* from the source repository (git). For up-to-date information on *jhbuild*, please refer to the jhbuild manual. If you need assistance using *jhbuild*, you should ask for help on the gnome-love mailing list.

> Note that to build *gtkmm* from git, you'll often need to build all of its dependencies from git as well. *jhbuild* makes this easier than it would normally be, but it will take quite a while to build and install them all. You

will probably encounter build problems, though these will usually be corrected quickly if you report them.

F.1. Setting up jhbuild

To set up *jhbuild*, follow the basic installation instructions from the jhbuild manual. After you have installed *jhbuild*, you should copy the sample *jhbuild* configuration file into your home directory by executing the following command from the *jhbuild* directory:

```
$ cp examples/sample.jhbuildrc ~/.jhbuildrc
```

The *gtkmm* module is defined in the `gnome-suites-core-deps-3.x.modules` moduleset, so edit your `.jhbuildrc` file and set your moduleset setting to the latest version e.g. like so:

```
moduleset = 'gnome-suites-core-deps-3.4'
```

After setting the correct moduleset, you need to tell *jhbuild* which module or modules to build. To build *gtkmm* and all of its dependencies, set `modules` like so:

```
modules = [ 'gtkmm' ]
```

You can build several modules by setting the `modules` variable to a meta-package, e.g. `meta-gnome-core`, or listing more than one module name. The `modules` variable specifies which modules that will be built when you don't explicitly specify anything on the command line. You can always build a different moduleset later by specifying it on the commandline (e.g. `jhbuild build gtkmm`).

Setting a prefix

By default, *jhbuild*'s configuration is configured to install all software built with *jhbuild* under the `/opt/gnome` prefix. You can choose a different prefix, but it is recommended that you keep this prefix different from other software that you've installed (don't set it to `/usr`!) If you've followed the jhbuild instructions then this prefix belongs to your user, so you don't need to run jhbuild as `root`.

When you downloaded *jhbuild* from the git repository, you got a number of `.modules` files, specifying dependencies between modules. By default *jhbuild* does not use the downloaded versions of these files, but reads the latest versions in the git repository. This is usually what you want. If you don't want it, use the `use_local_modulesets` variable in `.jhbuildrc`.

F.2. Installing and Using the git version of *gtkmm*

Once you've configured *jhbuild* as described above, building *gtkmm* should be relatively straightforward. The first time you run *jhbuild*, you should run the following sequence of commands to ensure that *jhbuild* has the required tools and verify that it is set up correctly:

```
$ jhbuild bootstrap
$ jhbuild sanitycheck
```


F.2.1. Installing *gtkmm* with *jhbuild*

If everything worked correctly, you should be able to build *gtkmm* and all of its

dependencies from git by executing `jhbuild build` (or, if you didn't specify *gtkmm* in the `modules` variable, with the command `jhbuild build gtkmm`).

This command will build and install a series of modules and will probably take quite a long time the first time through. After the first time, however, it should go quite a bit faster since it only needs to rebuild files that changed since the last build. Alternatively, after you've built and installed *gtkmm* the first time, you can rebuild *gtkmm* by itself (without rebuilding all of its dependencies) with the command `jhbuild buildone gtkmm`.

F.2.2. Using the git version of *gtkmm*

After you've installed the git version of *gtkmm*, you're ready to start using and experimenting with it. In order to use the new version of *gtkmm* you've just installed, you need to set some environment variables so that your `configure` script knows where to find the new libraries. Fortunately, *jhbuild* offers an easy solution to this problem. Executing the command `jhbuild shell` will start a new shell with all of the correct environment variables set. Now if you re-configure and build your project just as you usually do, it should link against the newly installed libraries. To return to your previous environment, simply exit the *jhbuild* shell.

Once you've built your software, you'll need to run your program within the jhbuild environment as well. To do this, you can again use the `jhbuild shell` command to start a new shell with the *jhbuild* environment set up. Alternatively, you can execute a one-off command in the *jhbuild* environment using the following command: `jhbuild run command-name`. In this case, the command will be run with the correct environment variables set, but will return to your previous environment after the program exits.

G. Wrapping C Libraries with gmmproc

gtkmm uses the `gmmproc` tool to generate most of its source code, using .defs files that define the APIs of `GObject`-based libraries. So it's quite easy to create additional gtkmm-style wrappers of other glib/GObject-based libraries.

This involves a variety of tools, some of them crufty, but it does at least work, and has been used successfully by several projects.

G.1. The build structure

Generation of the source code for a gtkmm-style wrapper API requires use of tools such as `gmmproc` and `generate_wrap_init.pl`. In theory you could write your own build files to use these appropriately, but a much better option is to make use of the build infrastructure provided by the mm-common module. To get started, it helps a lot to pick an existing binding module as an example to look at.

For instance, let's pretend that we are wrapping a C library called libexample. It provides a `GObject`-based API with types named, for instance, `ExampleThing` and `ExampleStuff`.

G.1.1. Copying the skeleton project

Typically our wrapper library would be called libsomethingmm. We can start by copying the skeleton source tree from the mm-common module.

```
$ git clone git://git.gnome.org/mm-common
```

```
$ cp -a mm-common/skeletonmm libsomethingmm
```

This provides a directory structure for the source .hg and .ccg files and the generated .h and .cc files, with `filelist.am` Automake include files that can specify the various files in use, in terms of generic Automake variables. The directory structure usually looks like this, after we have renamed the directories appropriately:

- `libsomethingmm`: The top-level directory.
  - `libsomething`: Contains the main include file and the pkg-config .pc file.
    - `src`: Contains .hg and .ccg source files.
    - `libsomethingmm`: Contains generated and hand-written .h and .cc files.
      - `private`: Contains generated `*_p.h` files.

As well as renaming the directories, we should rename some of the source files. For instance:

```
$ for f in $(find libsomethingmm -depth -name '*skeleton*'); do \
    d="${f%/*}"; b="${f##*/}"; mv "$f" "$d/${b//skeleton/libsomething}"; \
  done
```

A number of the skeleton files must still be filled in with project-specific content later.

Note that files ending in `.in` will be used to generate files with the same name but without the `.in` suffix, by replacing some variables with actual values during the configure stage.

G.1.2. Modifying build files

Now we edit the files to adapt them to our needs. You might prefer to use a multiple-file search-replace utility for this, such as `regexxer`. Note that nearly all of the files provided with the skeleton source tree contain placeholder text. Thus, the substitutions should be performed globally, and not be limited to the Automake and Autoconf files.

All mentions of `skeleton` should be replaced by the correct name of the C library you are wrapping, such as "something" or "libsomething". In the same manner, all instances of `SKELETON` should be replaced by "SOMETHING" or "LIBSOMETHING", and all occurrences of `Skeleton` changed to "Something".

Likewise, replace all instances of `Joe Hacker` by the name of the intended copyright holder, which is probably you. Do the same for the `joe@example.com` email address.

G.1.2.1. configure.ac

In `configure.ac`,

- The `AC_CONFIG_SRCDIR()` line must mention a file in our source tree. We can edit this later if we don't yet know the names of any of the files that we will create.
- It is common for binding modules to track the version number of the library they are wrapping. So, for instance, if the C library is at version 1.23.4, then the initial version of the binding module would be 1.23.0. However, avoid starting with an even minor version number as that usually indicates a stable release.
- The `AC_CONFIG_HEADERS()` line is used to generate two or more configuration header files. The first header file in the list contains all configuration macros which are set during the configure run. The remaining headers in the list contain only a subset of configuration macros and their corresponding `configh.h.in` file will not be autogenerated. The reason for this separation is that the namespaced configuration headers are installed with your library and define publically visible macros.

- The `AC_SUBST([SOMETHINGMM_MODULES], ['...'])` line may need to be modified to check for the correct dependencies.
- The `AC_CONFIG_FILES()` block must mention the correct directory names, as described above.

G.1.2.2. Makefile.am files

Next we must adapt the various `Makefile.am` files:

- In `skeleton/src/Makefile.am` we must mention the correct values for the generic variables that are used elsewhere in the build system:

  `binding_name`

  The name of the library, such as libsomethingmm.

  `wrap_init_flags`

  Additional command-line flags passed to the `generate_wrap_init.pl` script, such as the C++ namespace and the parent directory prefix of include files.

- In `skeleton/skeletonmm/Makefile.am` we must mention the correct values for the generic variables that are used elsewhere in the build system:

  `lib_LTLIBRARIES`

  This variable must mention the correct library name, and this library name must be used to form the `_SOURCES`, `_LDFLAGS`, and `_LIBADD` variable names. It is permissible to use variables substituted by `configure` like `@SOMETHINGMM_API_VERSION@` as part of the variable names.

  `AM_CPPFLAGS`

  The command line options passed to the C preprocessor.

  `AM_CXXFLAGS`

  The command line options passed to the C++ compiler.

G.1.2.3. Creating .hg and .ccg files

We should now create our first `.hg` and `.ccg` files, to wrap one of the objects in the C library. One pair of example source files already exists: `skeleton.ccg` and `skeleton.hg`. Create copies of these files as necessary.

We must mention all of our `.hg` and `.ccg` files in the `skeleton/src/filelist.am` file, typically in the `files_hg` variable.

Any additional non-generated `.h` and `.cc` source files may be placed in `skeleton/skeletonmm/` and listed in `skeleton/skeletonmm/filelist.am`, typically in the `files_extra_h` and `files_extra_cc` variables.

In the .hg and .ccg files section you can learn about the syntax used in these files.

G.2. Generating the .defs files.

The `.defs` files are text files, in a lisp format, that describe the API of a C library, including its

- objects (GObjects, widgets, interfaces, boxed-types and plain structs)
- functions
- enums
- signals
- properties

- vfuncs

At the moment, we have separate tools for generating different parts of these `.defs`, so we split them up into separate files. For instance, in the `gtk/src` directory of the *gtkmm* sources, you will find these files:

`gtk.defs`
: Includes the other files.

`gtk_methods.defs`
: Objects and functions.

`gtk_enums.defs`
: Enumerations.

`gtk_signals.defs`
: Signals and properties.

`gtk_vfuncs.defs`
: vfuncs (function pointer member fields in structs), written by hand.

G.2.1. Generating the methods .defs

This `.defs` file describes objects and their functions. It is generated by the `h2def.py` script which you can find in glibmm's `tools/defs_gen` directory. For instance,

```
$ ./h2def.py /usr/include/gtk-3.0/gtk/*.h > gtk_methods.defs
```

G.2.2. Generating the enums .defs

This `.defs` file describes enum types and their possible values. It is generated by the `enum.pl` script which you can find in glibmm's `tools` directory. For instance,

```
$ ./enum.pl /usr/include/gtk-3.0/gtk/*.h > gtk_enums.defs
```

G.2.3. Generating the signals and properties .defs

This `.defs` file describes signals and properties. It is generated by the special `extra_defs` utility that is in every wrapping project, such as `gtkmm/tools/extra_defs_gen/`. For instance

```
$ cd tools/extra_defs_gen
$ ./generate_extra_defs > gtk_signals.defs
```

You must edit the source code of your own `generate_extra_defs` tool in order to generate the `.defs` for the GObject C types that you wish to wrap. In the skeleton source tree, the source file is named `codegen/extradefs/generate_extra_defs_skeleton.cc`. If not done so already, the file should be renamed, with the basename of your new binding substituted for the `skeleton` placeholder. The `codegen/Makefile.am` file should also mention the new source filename.

Then edit the `.cc` file to specify the correct types. For instance, your `main()` function might look like this:

```
#include <libsomething.h>

int main(int, char**)
```

```
{
  something_init();

  std::cout << get_defs(EXAMPLE_TYPE_SOMETHING)
            << get_defs(EXAMPLE_TYPE_THING);
  return 0;
}
```

G.2.4. Writing the vfuncs .defs

This `.defs` file describes virtual functions (vfuncs). It must be written by hand. There is no skeleton to start from. You can look at *gtkmm*'s `gtk/src/gtk_vfuncs.defs` file.

G.3. The .hg and .ccg files

The .hg and .ccg source files are very much like .h and .cc C++ source files, but they contain extra macros, such as `_CLASS_GOBJECT()` and `_WRAP_METHOD()`, from which `gmmproc` generates appropriate C++ source code, usually at the same position in the header. Any additional C++ source code will be copied verbatim into the corresponding .h or .cc file.

A .hg file will typically include some headers and then declare a class, using some macros to add API or behaviour to this class. For instance, gtkmm's `button.hg` looks roughly like this:

```
#include <gtkmm/bin.h>
#include <gtkmm/activatable.h>
#include <gtkmm/stockid.h>
_DEFS(gtkmm,gtk)
_PINCLUDE(gtkmm/private/bin_p.h)

namespace Gtk
{

class Button
  : public Bin,
    public Activatable
{
  _CLASS_GTKOBJECT(Button,GtkButton,GTK_BUTTON,Gtk::Bin,GtkBin)
  _IMPLEMENTS_INTERFACE(Activatable)
public:

  _CTOR_DEFAULT
  explicit Button(const Glib::ustring& label, bool mnemonic = false);
  explicit Button(const StockID& stock_id);

  _WRAP_METHOD(void set_label(const Glib::ustring& label), gtk_button_set_label)

  ...

  _WRAP_SIGNAL(void clicked(), "clicked")

  ...

  _WRAP_PROPERTY("label", Glib::ustring)
};

} // namespace Gtk
```

The macros in this example do the following:

`_DEFS()`

Specifies the destination directory for generated sources, and the name of the main .defs file that `gmmproc` should parse.

`_PINCLUDE()`

Tells `gmmproc` to include a header from the generated private/button_p.h file.

`_CLASS_GTKOBJECT()`

Tells `gmmproc` to add some typedefs, constructors, and standard methods to this class, as appropriate when wrapping a widget.

`_IMPLEMENTS_INTERFACE()`

Tells `gmmproc` to add initialization code for the interface.

`_CTOR_DEFAULT`

Add a default constructor.

`_WRAP_METHOD()`, `_WRAP_SIGNAL()`, and `_WRAP_PROPERTY()`

Add methods to wrap parts of the C API.

The .h and .cc files will be generated from the .hg and .ccg files by processing them with `gmmproc` like so, though this happens automatically when using the above build structure:

```
$ cd gtk/src
$ /usr/lib/glibmm-2.4/proc/gmmproc -I ../../tools/m4 --defs . button . ./../gtkmm
```

Notice that we provided `gmmproc` with the path to the .m4 convert files, the path to the .defs file, the name of a .hg file, the source directory, and the destination directory.

You should avoid including the C header from your C++ header, to avoid polluting the global namespace, and to avoid exporting unnecessary public API. But you will need to include the necessary C headers from your .ccg file.

The macros are explained in more detail in the following sections.

G.3.1. m4 Conversions

The macros that you use in the .hg and .ccg files often need to know how to convert a C++ type to a C type, or vice-versa. gmmproc takes this information from an .m4 file in your `tools/m4/` directory. This allows it to call a C function in the implementation of your C++ method, passing the appropriate parameters to that C functon. For instance, this tells gmmproc how to convert a GtkTreeView pointer to a Gtk::TreeView pointer:

```
_CONVERSION(`GtkTreeView*',`TreeView*',`Glib::wrap($3)')
```

`$3` will be replaced by the parameter name when this conversion is used by gmmproc.

Some extra macros make this easier and consistent. Look in gtkmm's .m4 files for examples. For instance:

```
_CONVERSION(`PrintSettings&',`GtkPrintSettings*',__FR2P)
_CONVERSION(`const PrintSettings&',`GtkPrintSettings*',__FCR2P)
_CONVERSION(`const Glib::RefPtr<Printer>&',`GtkPrinter*',__CONVERT_REFPTR_TO_P($3))
```

G.3.2. m4 Initializations

Often when wrapping methods, it is desirable to store the return of the C function in what is called an output parameter. In this case, the C++ method returns `void` but an output parameter in which to store the value of the C function is included in the argument list of the C++ method. gmmproc allows such functionality, but appropriate inizialization macros must be included to tell gmmproc how to initialize the C++ parameter from the return of the C function.

For example, if there was a C function that returned a `GtkWidget*` and for some reason, instead of having the C++ method also return the widget, it was desirable to have the C++ method place the widget in a specified output parameter, an initialization

macro such as the following would be necessary:

```
_INITIALIZATION(`Gtk::Widget&',`GtkWidget*',`$3 = Glib::wrap($4)')
```

`$3` will be replaced by the output parameter name of the C++ method and `$4` will be replaced by the return of the C function when this initialization is used by gmmproc. For convenience, `$1` will also be replaced by the C++ type without the ampersand (&) and `$2` will be replaced by the C type.

G.3.3. Class macros

The class macro declares the class itself and its relationship with the underlying C type. It generates some internal constructors, the member `gobject_`, typedefs, the `gobj()` accessors, type registration, and the `Glib::wrap()` method, among other things.

Other macros, such as `_WRAP_METHOD()` and `_WRAP_SIGNAL()` may only be used after a call to a `_CLASS_*` macro.

G.3.3.1. _CLASS_GOBJECT

This macro declares a wrapper for a type that is derived from `GObject`, but whose wrapper is not derived from `Gtk::Object`.

```
_CLASS_GOBJECT( C++ class, C class, C casting macro, C++ base class, C base class )
```

For instance, from `accelgroup.hg`:

```
_CLASS_GOBJECT(AccelGroup, GtkAccelGroup, GTK_ACCEL_GROUP, Glib::Object, GObject)
```

G.3.3.2. _CLASS_GTKOBJECT

This macro declares a wrapper for a type whose wrapper is derived from `Gtk::Object`, such as a widget or dialog.

```
_CLASS_GTKOBJECT( C++ class, C class, C casting macro, C++ base class, C base class )
```

For instance, from `button.hg`:

```
_CLASS_GTKOBJECT(Button, GtkButton, GTK_BUTTON, Gtk::Bin, GtkBin)
```

You will typically use this macro when the class already derives from Gtk::Object. For instance, you will use it when wrapping a GTK+ Widget, because Gtk::Widget derives from Gtk::Object.

You might also derive non-widget classes from Gtk::Object so they can be used without `Glib::RefPtr`. For isntance, they could then be instantiated with `Gtk::manage()` or on the stack as a member variable. This is convenient, but you should use this only when you are sure that true reference-counting is not needed. We consider it useful for widgets.

G.3.3.3. _CLASS_BOXEDTYPE

This macro declares a wrapper for a non-`GObject` struct, registered with `g_boxed_type_register_static()`.

```
_CLASS_BOXEDTYPE( C++ class, C class, new function, copy function, free function )
```

For instance, for `Gdk::Color`:

```
_CLASS_BOXEDTYPE(Color, GdkColor, NONE, gdk_color_copy, gdk_color_free)
```

G.3.3.4. _CLASS_BOXEDTYPE_STATIC

This macro declares a wrapper for a simple assignable struct such as `GdkRectangle`. It is similar to `_CLASS_BOXEDTYPE`, but the C struct is not allocated dynamically.

```
_CLASS_BOXEDTYPE_STATIC( C++ class, C class )
```

For instance, for `Gdk::Rectangle`:

```
_CLASS_BOXEDTYPE_STATIC(Rectangle, GdkRectangle)
```

G.3.3.5. _CLASS_OPAQUE_COPYABLE

This macro declares a wrapper for an opaque struct that has copy and free functions. The new, copy and free functions will be used to instantiate the default constructor, copy constructor and destructor.

```
_CLASS_OPAQUE_COPYABLE( C++ class, C class, new function, copy function, free function )
```

For instance, from `stockitem.hg`:

```
_CLASS_OPAQUE_COPYABLE(StockItem, GtkStockItem, NONE, gtk_stock_item_copy, gtk_stock_item_free)
```

G.3.3.6. _CLASS_OPAQUE_REFCOUNTED

This macro declares a wrapper for a reference-counted opaque struct. The C++ wrapper cannot be directly instantiated and can only be used with `Glib::RefPtr`.

```
_CLASS_OPAQUE_REFCOUNTED( C++ class, C class, new function, ref function, unref function )
```

For instance, for `Pango::Coverage`:

```
_CLASS_OPAQUE_REFCOUNTED(Coverage, PangoCoverage, pango_coverage_new, pango_coverage_ref, pango_coverage_unref)
```

G.3.3.7. _CLASS_GENERIC

This macro can be used to wrap structs which don't fit into any specialized category.

```
_CLASS_GENERIC( C++ class, C class )
```

For instance, for `Pango::AttrIter`:

```
_CLASS_GENERIC(AttrIter, PangoAttrIterator)
```

G.3.3.8. _CLASS_INTERFACE

This macro declares a wrapper for a type that is derived from `GTypeInterface`.

```
_CLASS_INTERFACE( C++ class, C class, C casting macro, C interface struct, Base C++ class
(optional), Base C class (optional) )
```

For instance, from `celleditable.hg`:

```
_CLASS_INTERFACE(CellEditable, GtkCellEditable, GTK_CELL_EDITABLE, GtkCellEditableIface)
```

Two extra parameters are optional, for the case that the interface derives from another interface, which should be the case when the GInterface has another GInterface as a prerequisitite. For instance, from `loadableicon.hg`:

```
_CLASS_INTERFACE(LoadableIcon, GLoadableIcon, G_LOADABLE_ICON, GLoadableIconIface, Icon, GIcon)
```

G.3.4. Constructor macros

The `_CTOR_DEFAULT()` and `_WRAP_CTOR()` macros add constructors, wrapping the specified `*_new()` C functions. These macros assume that the C object has properties with the same names as the function parameters, as is usually the case, so that it can supply the parameters directly to a `g_object_new()` call. These constructors never actually call the `*_new()` C functions, because gtkmm must actually instantiate derived GTypes, and the `*_new()` C functions are meant only as convenience functions for C programmers.

When using `_CLASS_GOBJECT()`, the constructors should be protected (rather than public) and each constructor should have a corresponding `_WRAP_CREATE()` in the public section. This prevents the class from being instantiated without using a `RefPtr`. For instance:

```
class ActionGroup : public Glib::Object
{
  _CLASS_GOBJECT(ActionGroup, GtkActionGroup, GTK_ACTION_GROUP, Glib::Object, GObject)

protected:
  _WRAP_CTOR(ActionGroup(const Glib::ustring& name = Glib::ustring()), gtk_action_group_new)

public:
  _WRAP_CREATE(const Glib::ustring& name = Glib::ustring())
```


G.3.4.1. _CTOR_DEFAULT

This macro creates a default constructor with no arguments.

G.3.4.2. _WRAP_CTOR

This macro creates a constructor with arguments, equivalent to a `*_new()` C function. It won't actually call the `*_new()` function, but will simply create an equivalent constructor with the same argument types. It takes a C++ constructor signature, and a C function name.

When wrapping constructors, it is possible for gmmproc to generate convenience overloads of the wrapped constructors if the C function has parameters that are optional (ie. the C API allows null for those parameters). For instance, to specify if a parameter is optional, the `_WRAP_CTOR()` macro would look something like the following:

```
_WRAP_CTOR(ToolButton(Widget& icon_widget, const Glib::ustring& label{?}), gtk_tool_button_new)
```

The `{?}` following the name of the `label` parameter means that that parameter is optional. In this case, gmmproc will generate an extra constructor without that parameter.

It is also possible to have the order of the parameters of the constructor different from that of the C function by using gmmproc's C++ to C parameter mapping functionality. Using this functionality, it is possible to map a C++ parameter to a C parameter by specifying the C parameter name. For instance, assuming that the declaration of the `gtk_tool_button_new()` function is the following:

```
GtkToolItem* gtk_tool_button_new(GtkWidget* icon_widget, const gchar* label);
```

The parameters of the wrapped constructor could be reordered using the following:

```
_WRAP_CTOR(ToolButton(const Glib::ustring& label{label}, Widget& icon_widget{icon_widget}), gtk_tool_button_new)
```

The `{param_name}` following each of the names of the parameters tells gmmproc to map those C++ parameters to the C parameters with the given names. Since the C++ parameter names correspond to the C ones, the above could be re-written as:

```
_WRAP_CTOR(ToolButton(const Glib::ustring& label{.}, Widget& icon_widget{.}), gtk_tool_button_new)
```

This same optional parameter syntax and parameter reordering is available for `_WRAP_CREATE()`. Additional `create()` overloads would be generated by gmmproc without the specified optional parameters.

G.3.4.3. Hand-coding constructors

When a constructor must be partly hand written because, for instance, the `*_new()` C function's parameters do not correspond directly to object properties, or because the `*_new()` C function does more than call `g_object_new()`, the `_CONSTRUCT()` macro may be used in the .ccg file to save some work. The `_CONSTRUCT` macro takes a series of property names and values. For instance, from `button.ccg`:

```
Button::Button(const Glib::ustring& label, bool mnemonic)
:
  _CONSTRUCT("label", label.c_str(), "use_underline", gboolean(mnemonic))
{}
```

G.3.5. Method macros

G.3.5.1. _WRAP_METHOD

This macro generates the C++ method to wrap a C function.

```
_WRAP_METHOD( C++ method signature, C function name)
```

For instance, from `entry.hg`:

```
_WRAP_METHOD(void set_text(const Glib::ustring& text), gtk_entry_set_text)
```

The C function (e.g. `gtk_entry_set_text`) is described more fully in the .defs file, and the `convert*.m4` files contain the necessary conversion from the C++ parameter type to the C parameter type. This macro also generates doxygen documentation comments based on the `*_docs.xml` and `*_docs_override.xml` files.

There are some optional extra arguments:

refreturn

Do an extra `reference()` on the return value, in case the C function does not provide a reference.

errthrow

Use the last GError* parameter of the C function to throw an exception.

deprecated

Puts the generated code in #ifdef blocks. Text about the deprecation can be specified as an optional parameter.

constversion

Just call the non-const version of the same function, instead of generating almost duplicate code.

ifdef

Puts the generated code in #ifdef blocks.

As with `_WRAP_CTOR()` it is possible to specify if there are any optional parameters. If that is the case, gmmproc will generate convenience overload methods without those parameters. For example:

```
_WRAP_METHOD(void get_preferred_size(Requisition& minimum_size, Requisition& natural_size{?}) const, gtk_widget_get_preferred_size)
```

Would indicate that the `natural_size` parameter is optional because its name ends with `{?}`. In this case, gmmproc would generate a method overload without that parameter.

Also, as with `_WRAP_CTOR()`, it is possible to reorder the parameters of the C++ method by using gmmproc's C++ to C parameter mapping functionality. Using this functionality, it is possible to map a C++ parameter to a C parameter by specifying the C parameter name. For example, if the `gtk_widget_set_device_events()` declaration is the following:

```
void gtk_widget_set_device_events(GtkWidget* widget, GdkDevice* device, GdkEventMask events);
```

Something like the following would change the order of the parameters in the C++ method:

```
_WRAP_METHOD(void set_device_events(Gdk::EventMask events{events}, const Glib::RefPtr<const Gdk::Device>& device{device}), gtk_widg
```

The `{param_name}` following each of the names of the parameters tells gmmproc to map those C++ parameters to the C parameters with the given names. Since the C++ parameter names correspond to the C ones, the above could be re-written as:

```
_WRAP_METHOD(void set_device_events(Gdk::EventMask events{.}, const Glib::RefPtr<const Gdk::Device>& device{.}), gtk_widget_set_dev
```

With `_WRAP_METHOD()` it is also possible to include an output parameter in the C++ method declaration in which the return of the C function would be placed and to have the C++ method return `void`. To do that, simply include the output parameter declaration in the C++ method declaration appending a `{OUT}` to the output parameter name. For example, if `gtk_widget_get_request_mode()` is declared as the following:

```
GtkSizeRequestMode gtk_widget_get_request_mode(GtkWidget* widget);
```

And having the C++ method set an output parameter is desired instead of returning a `SizeRequestMode`, something like the following could be used:

```
_WRAP_METHOD(void get_request_mode(SizeRequestMode& mode{OUT}) const, gtk_widget_get_request_mode)
```

the `{OUT}` appended to the name of the `mode` output parameter tells gmmproc to place the return of the C function in that output parameter. In this case, however, a necessary initialization macro like the following would also have to be specified:

```
_INITIALIZATION(`SizeRequestMode&',`GtkSizeRequestMode',`$3 = (SizeRequestMode)($4)')
```

Which could also be written as:

```
_INITIALIZATION(`SizeRequestMode&',`GtkSizeRequestMode',`$3 = ($1)($4)')
```

Though it's usually obvious what C++ types should be used in the C++ method, here are some hints:

- Objects used via `RefPtr`: Pass the `RefPtr` as a const reference. For instance, `const Glib::RefPtr<Gtk::Action>& action`.
- Const Objects used via `RefPtr`: If the object should not be changed by the function, then make sure that the object is const, even if the `RefPtr` is already const. For instance, `const Glib::RefPtr<const Gtk::Action>& action`.
- Wrapping `GList*` and `GSList*` parameters: First, you need to discover what objects are contained in the list's data field for each item, usually by reading the documentation for the C function. The list can then be wrapped by a `std::vector` type. For instance, `std::vector< Glib::RefPtr<Action> >`. You may need to define a

Traits type to specify how the C and C++ types should be converted.

- Wrapping `GList*` and `GSList*` return types: You must discover whether the caller should free the list and whether it should release the items in the list, again by reading the documentation of the C function. With this information you can choose the ownership (none, shallow or deep) for the m4 conversion rule, which you should probably put directly into the .hg file because the ownership depends on the function rather than the type. For instance:

```
#m4 _CONVERSION(`GSList*',`std::vector<Widget*>',`Glib::SListHandler<Widget*>::slist_to_vector($3, Glib::OWNERSHIP_SHALLOW)')
```

G.3.5.2. _WRAP_METHOD_DOCS_ONLY

This macro is like `_WRAP_METHOD()`, but it generates only the documentation for a C++ method that wraps a C function. Use this when you must hand-code the method, but you want to use the documentation that would be generated if the method was generated.

```
_WRAP_METHOD_DOCS_ONLY(C function name)
```

For instance, from `container.hg`:

```
_WRAP_METHOD_DOCS_ONLY(gtk_container_remove)
```

G.3.5.3. _IGNORE / _IGNORE_SIGNAL

`gmmproc` will warn you on stdout about functions and signals that you have forgotten to wrap, helping to ensure that you are wrapping the complete API. But if you don't want to wrap some functions or signals, or if you chose to hand-code some methods then you can use the _IGNORE() or _IGNORE_SIGNAL() macro to make `gmmproc` stop complaining.

```
_IGNORE(C function name 1, C function name2, etc)

_IGNORE_SIGNAL(C signal name 1, C signal name2, etc)
```

For instance, from `buttonbox.hg`:

```
_IGNORE(gtk_button_box_set_spacing, gtk_button_box_get_spacing)
```

G.3.5.4. _WRAP_SIGNAL

This macro generates the C++ libsigc++-style signal to wrap a C GObject signal. It actually generates a public accessor method, such as `signal_clicked()`, which returns a proxy object. `gmmproc` uses the .defs file to discover the C parameter types and the .m4 convert files to discover appropriate type conversions.

```
_WRAP_SIGNAL( C++ signal handler signature, C signal name)
```

For instance, from `button.hg`:

```
_WRAP_SIGNAL(void clicked(),"clicked")
```

Signals usually have function pointers in the GTK struct, with a corresponding enum value. and a `g_signal_new()` in the .c file.

There are some optional extra arguments:

no_default_handler

Do not generate an `on_something()` virtual method to allow easy overriding of the default signal handler. Use this when adding a signal with a default signal handler would break the ABI by increasing the size of the class's virtual function table.

custom_default_handler

Generate a declaration of the `on_something()` virtual method in the `.h` file, but do not generate a definition in the `.cc` file. Use this when you must generate the definition by hand.

custom_c_callback

Do not generate a C callback function for the signal. Use this when you must generate the callback function by hand.

refreturn

Do an extra `reference()` on the return value of the `on_something()` virtual method, in case the C function does not provide a reference.

ifdef

Puts the generated code in #ifdef blocks.

G.3.5.5. _WRAP_PROPERTY

This macro generates the C++ method to wrap a C GObject property. You must specify the property name and the wanted C++ type for the property. `gmmproc` uses the .defs file to discover the C type and the .m4 convert files to discover appropriate type conversions.

```
_WRAP_PROPERTY(C property name, C++ type)
```

For instance, from `button.hg`:

```
_WRAP_PROPERTY("label", Glib::ustring)
```

G.3.5.6. _WRAP_VFUNC

This macro generates the C++ method to wrap a virtual C function.

```
_WRAP_VFUNC( C++ method signature, C function name)
```

For instance, from `widget.hg`:

```
_WRAP_VFUNC(SizeRequestMode get_request_mode() const, get_request_mode)
```

The C function (e.g. `get_request_mode`) is described more fully in the `*_vfuncs.defs` file, and the `convert*.m4` files contain the necessary conversion from the C++ parameter type to the C parameter type.

There are some optional extra arguments:

refreturn

Do an extra `reference()` on the return value of the `something_vfunc()` function, in case the virtual C function does not provide a reference.

refreturn_ctype

Do an extra `reference()` on the return value of an overridden `something_vfunc()` function in the C callback function, in case the calling C function expects it to provide a reference.

custom_vfunc

Do not generate a definition of the vfunc in the `.cc` file. Use this when you must generate the vfunc by hand.

custom_vfunc_callback

Do not generate a C callback function for the vfunc. Use this when you must generate the callback function by hand.

ifdef

Puts the generated code in #ifdef blocks.

A rule to which there may be exceptions: If the virtual C function returns a pointer to an object derived from `GObject`, i.e. a reference-counted object, then the virtual C++ function shall return a `Glib::RefPtr<>` object. One of the extra arguments `refreturn` or `refreturn_ctype` is required.

G.3.6. Other macros

G.3.6.1. _IMPLEMENTS_INTERFACE

This macro generates initialization code for the interface.

```
_IMPLEMENTS_INTERFACE(C++ interface name)
```

For instance, from `button.hg`:

```
_IMPLEMENTS_INTERFACE(Activatable)
```

There is one optional extra argument:

ifdef
: Puts the generated code in #ifdef blocks.

G.3.6.2. _WRAP_ENUM

This macro generates a C++ enum to wrap a C enum. You must specify the desired C++ name and the name of the underlying C enum.

For instance, from `widget.hg`:

```
_WRAP_ENUM(WindowType, GdkWindowType)
```

If the enum is not a `GType`, you must pass a third parameter NO_GTYPE. This is the case when there is no `*_get_type()` function for the C enum, but be careful that you don't just need to include an extra header for that function. You should also file a bug against the C API, because all enums should be registered as GTypes.

For example, from `icontheme.hg`:

```
_WRAP_ENUM(IconLookupFlags, GtkIconLookupFlags, NO_GTYPE)
```

G.3.6.3. _WRAP_GERROR

This macro generates a C++ exception class, derived from Glib::Error, with a Code enum and a code() method. You must specify the desired C++ name, the name of the corresponding C enum, and the prefix for the C enum values.

This exception can then be thrown by methods which are generated from _WRAP_METHOD() with the errthrow option.

For instance, from `pixbuf.hg`:

```
_WRAP_GERROR(PixbufError, GdkPixbufError, GDK_PIXBUF_ERROR)
```

G.3.6.4. _MEMBER_GET / _MEMBER_SET

Use these macros if you're wrapping a simple struct or boxed type that provides direct access to its data members, to create getters and setters for the data members.

```
_MEMBER_GET(C++ name, C name, C++ type, C type)

_MEMBER_SET(C++ name, C name, C++ type, C type)
```

For example, in `rectangle.hg`:

```
_MEMBER_GET(x, x, int, int)
```

G.3.6.5. _MEMBER_GET_PTR / _MEMBER_SET_PTR

Use these macros to automatically provide getters and setters for a data member that is a pointer type. For the getter function, it will create two methods, one const and one non-const.

```
_MEMBER_GET_PTR(C++ name, C name, C++ type, C type)

_MEMBER_SET_PTR(C++ name, C name, C++ type, C type)
```

For example, for `Pango::Analysis` in `item.hg`:

```
// _MEMBER_GET_PTR(engine_lang, lang_engine, EngineLang*, PangoEngineLang*)
// It's just a comment. It's difficult to find a real-world example.
```

G.3.6.6. _MEMBER_GET_GOBJECT / _MEMBER_SET_GOBJECT

Use these macros to provide getters and setters for a data member that is a `GObject` type that must be referenced before being returned.

```
_MEMBER_GET_GOBJECT(C++ name, C name, C++ type, C type)

_MEMBER_SET_GOBJECT(C++ name, C name, C++ type, C type)
```

For example, in Pangomm, `layoutline.hg`:

```
_MEMBER_GET_GOBJECT(layout, layout, Pango::Layout, PangoLayout*)
```

G.3.7. Basic Types

Some of the basic types that are used in C APIs have better alternatives in C++. For example, there's no need for a `gboolean` type since C++ has `bool`. The following list shows some commonly-used types in C APIs and what you might convert them to in a C++ wrapper library.

Basic Type equivalents

C type: `gboolean`

C++ type: `bool`

C type: `gint`

C++ type: `int`

C type: `guint`

C++ type: `guint`

C type: `gdouble`

C++ type: `double`

C type: `gunichar`

C++ type: `gunichar`

C type: `gchar*`

C++ type: `Glib::ustring` (or `std::string` for filenames)

G.4. Hand-coded source files

You might want to include additional source files that will not be generated by `gmmproc` from `.hg` and `.ccg` files. You can simply place these in your `libsomething/libsomethingmm` directory and mention them in the `Makefile.am` in the `files_extra_h` and `files_extra_cc` variables.

G.5. Initialization

Your library must be initialized before it can be used, to register the new types that it makes available. Also, the C library that you are wrapping might have its own initialization function that you should call. You can do this in an `init()` function that you can place in hand-coded `init.h` and `init.cc` files. This function should initialize your dependencies (such as the C function, and *gtkmm*) and call your generated `wrap_init()` function. For instance:

```
void init()
{
  Gtk::Main::init_gtkmm_internals(); //Sets up the g type system and the Glib::wrap() table.
  wrap_init(); //Tells the Glib::wrap() table about the libsomethingmm classes.
}
```

The implementation of the `wrap_init()` method in `wrap_init.cc` is generated by `generate_wrap_init.pl`, but the declaration in `wrap_init.h` is hand-coded, so you will need to adjust `wrap_init.h` so that the `init()` function appears in the correct C++ namespace.

G.6. Problems in the C API.

You are likely to encounter some problems in the library that you are wrapping, particularly if it is a new project. Here are some common problems, with solutions.

G.6.1. Unable to predeclare structs

By convention, structs are declared in glib/GTK+-style headers like so:

```
typedef struct _ExampleWidget ExampleWidget;

struct _ExampleWidget
{
  ...
};
```

The extra typedef allows the struct to be used in a header without including its full definition, simply by predeclaring it, by repeating that typedef. This means that you don't have to include the C library's header in your C++ header, thus keeping it out of your public API. `gmmproc` assumes that this technique was used, so you will see compiler errors if that is not the case.

This compiler error might look like this:

```
example-widget.h:56: error: using typedef-name 'ExampleWidget' after 'struct'
../../libexample/libexamplemm/example-widget.h:34: error: 'ExampleWidget' has a previous declaration here
make[4]: *** [example-widget.lo] Error 1
```

or this:

```
example-widget.h:60: error: '_ExampleWidget ExampleWidget' redeclared as different kind of symbol
../../libexample/libexamplemm/example-widget.h:34: error: previous declaration of 'typedef struct _ExampleWidget ExampleWidget'
```

This is easy to correct in the C library, so do send a patch to the relevant maintainer.

G.6.2. Lack of properties

By convention, glib/GTK+-style objects have `*_new()` functions, such as `example_widget_new()` that do nothing more than call `g_object_new()` and return the result. The input parameters are supplied to `g_object_new()` along with the names of the properties for which they are values. For instance,

```
GtkWidget* example_widget_new(int something, const char* thing)
{
        return g_object_new (EXAMPLE_TYPE_WIDGET, "something", something, "thing", thing, NULL);
}
```

This allows language bindings to implement their own equivalents (such as C++ constructors), without using the `*_new()` function. This is often necessary so that they can actually instantiate a derived GType, to add their own hooks for signal handlers and vfuncs.

At the least, the `_new()` function should not use any private API (functions that are only in a .c file). Even when there are no functions, we can sometimes reimplement 2 or 3 lines of code in a `_new()` function as long as those lines of code use API that is available to us.

Another workaround is to add a `*_construct()` function that the C++ constructor can call after instantiating its own type. For instance,

```
GtkWidget* example_widget_new(int something, const char* thing)
{
        ExampleWidget* widget;
        widget = g_object_new (EXAMPLE_TYPE_WIDGET, NULL);
        example_widget_construct(widget, "something", something, "thing", thing);
}

void example_widget_construct(ExampleWidget* widget, int something, const char* thing)
{
        //Do stuff that uses private API:
        widget->priv->thing = thing;
        do_something(something);
}
```

Adding properties, and ensuring that they interact properly with each other, is relatively difficult to correct in the C library, but it is possible, so do file a bug and try to send a patch to the relevant maintainer.

G.7. Documentation

In general, gtkmm-style projects use Doxygen, which reads specially formatted C++ comments and generates HTML documentation. You may write these doxygen comments directly in the header files.

G.7.1. Reusing C documentation

You might wish to reuse documentation that exists for the C library that you are wrapping. GTK-style C libraries typically use gtk-doc and therefore have source code comments formatted for gtk-doc and some extra documentation in .sgml and .xml files. The docextract_to_xml.py script, from glibmm's `tools/defs_gen` directory, can read these files and generate an .xml file that `gmmproc` can use to generate doxygen comments. `gmmproc` will even try to transform the documentation to make it more appropriate for a C++ API.

For instance,

```
./docextract_to_xml.py -s ~/checkout/gnome/gtk+/gtk/ -s ~/checkout/gnome/gtk+/docs/reference/gtk/ > gtk_docs.xml
```

Because this automatic transformation is not always appropriate, you might want to provide hand-written text for a particular method. You can do this by copying the XML node for the function from your `something_docs.xml` file to the `something_docs_override.xml` file and changing the contents.

G.7.2. Documentation build structure

If you copied the skeleton source tree in mm-common and substituted the placeholder text, then you will already have suitable `Makefile.am` and `Doxyfile.in` files. With the mm-common build setup, the list of Doxygen input files is not defined in the Doxygen configuration file, but passed along from `make` to the standard input of `doxygen`. The input file list is defined by the `doc_input` variable in the `Makefile.am` file.