CHAPTER 2

Your First GTK+ Applications

In Chapter 1, you were given an overview of the things available to you in the GTK+ libraries as a graphical application developer. In this chapter, you'll learn how to write your own GTK+ applications.

While we will begin with simple examples, there are many important concepts presented in this chapter. We will cover the topics that every other GTK+ application you write will rely on. Therefore, as with any chapter, make sure you understand the concepts presented to you in the next few pages before continuing on.

In this chapter, you will learn the following:

- The basic function calls required by all GTK+ applications
- How to compile GTK+ code with GCC
- The object-oriented nature of the GTK+ widget system
- What role signals, callbacks, and events play in your applications
- How to alter textual styles with the Pango Text Markup Language
- Various other useful functions provided for the widgets presented in this chapter
- How to use the `GtkButton` widget to make a clickable `GtkLabel`
- How to get and set properties of objects using `GObject` methods

Hello World

Every programming book I have read in my lifetime has begun with a “Hello World” example application. I do not want to be the one to break with tradition.

Before we get to the example, you should know that all of the source code for every example is downloadable from this book's web site, found at `www.gtkbook.com`. You can compile each example with the method presented in a later section of this chapter or follow the instructions found in the base folder of the package.

Listing 2-1 is the first and most simple GTK+ application in this book. It initializes GTK+, creates a window, displays it to the user, and waits for the program to be terminated. It is very basic, but it shows the essential code that every GTK+ application you create must have!

■Note The application in Listing 2-1 does not provide a way for you to terminate it. If you click the X in the corner of the window, the window will close, but the application will remain running. Therefore, you will have to press Ctrl+C in your terminal window to force the application to exit!

Listing 2-1. *Greeting the World (helloworld.c)*

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

int main (int argc,
          char *argv[])
{
  GtkWidget *window;

  /* Initialize GTK+ and all of its supporting libraries. */
  gtk_init (&argc, &argv);

  /* Create a new window, give it a title and display it to the user. */
  window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  gtk_window_set_title (GTK_WINDOW (window), "Hello World");
  gtk_widget_show (window);

  /* Hand control over to the main loop. */
  gtk_main ();
  return 0;
}
```

The `<gtk/gtk.h>` file includes all of the widgets, variables, functions, and structures available in GTK+ as well as header files from other libraries that GTK+ depends on, such as `<glib/glib.h>` and `<gdk/gdk.h>`. In most of your applications, `<gtk/gtk.h>` will be the only GTK+ header file you will need to include for GTK+ development, although some more advanced applications may require further inclusions.

Listing 2-1 is one of the simplest applications that you can create with GTK+. It produces a top-level `GtkWindow` widget with a default width and height of 200 pixels. There is no way of exiting the application except to kill it in the terminal where it was launched. You will learn how to use signals to exit the application when necessary in the next example.

This example is rather simple, but it shows the bare essentials you will need for every GTK+ application you create. The first step in understanding the "Hello World" application is to look at the content of the `main()` function.

Initializing GTK+

Initializing the GTK+ libraries is extremely simple for most applications. By calling `gtk_init()`, all initialization work is automatically performed for you.

It begins by setting up the GTK+ environment, including obtaining the GDK display and preparing the GLib main event loop and basic signal handling. If `gtk_init()` does more than you need, you may create your own, small initialization function that calls fewer of the functions, such as gdk_init() and g_main_loop_new(), although this is not necessary for most applications.

One of the great benefits of using open source libraries is the ability to read the code yourself to see how things are done. You can easily view the GTK+ source code to figure out everything that is called by gtk_init() and choose what needs to be performed by your application. However, you should use gtk_init() for now until you learn more about how each of the libraries are used and how they interrelate.

You will also notice that we passed the standard main() argument parameters argc and argv to gtk_init(). The GTK+ initialization function parses through all of the arguments and strips out any it recognizes. Any parameters it uses will be removed from the list, so you should do any argument parsing of your own after calling gtk_init(). This means that a standard list of parameters can be passed and parsed by all GTK+ applications without any extra work performed by you, the developer.

It is important to call gtk_init() before any other function calls to the GTK+ libraries. Otherwise, your application will not function properly and will likely crash.

The gtk_init() function will terminate your application if it is unable to initialize the GUI or has any other significant problems that cannot be resolved. If you would like your application to fall back on a text interface when GUI initialization fails, you need to use gtk_init_check().

```
gboolean gtk_init_check (int *argc,
                         char ***argv);
```

If the initialization fails, FALSE is returned. Otherwise, gtk_init_check() will return TRUE. You should only use this function if you have a textual interface to fall back on!

Widget Hierarchy

I consider widget hierarchy one of the most important topics of discussion when learning GTK+. While it is not difficult to understand, without it, widgets would not be possible as they exist today.

To understand this topic, we will look at gtk_window_new(), the function used to create a new GtkWindow object. You will notice in the following line that, while we want to create a new GtkWindow, gtk_window_new() returns a pointer to a GtkWidget. This is because every widget in GTK+ is actually a GtkWidget itself.

```
GtkWidget* gtk_window_new (GtkWindowType type);
```

Widgets in GTK+ use the GObject hierarchy system, which allows you to derive new widgets from those that already exist. Child widgets inherit properties, functions, and signals from their parent, their grandparent, and so on, because they are actually implementations of their ancestors themselves.

Widget hierarchy in GTK+ is a singly inherited system, which means that each child can have only one direct parent. This creates a simple linear relationship that every widget implements. You will learn how to derive your own child widgets in Chapter 11. Until then, we will use widget hierarchy to take advantage of inherited methods and properties.

In Figure 2-2, a simple outline of the widget hierarchy of the GtkWindow class is illustrated.

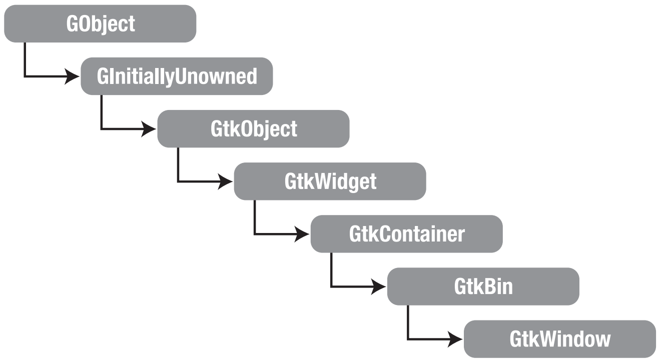

Figure 2-1. *The widget hierarchy of GtkWindow*

Figure 2-1 may look daunting at first, but let's look at each class type one at a time to make things easier to understand:

- `GObject` is the fundamental type providing common attributes for all libraries based on it including GTK+ and Pango. It allows objects derived from it to be constructed, destroyed, referenced, and unreferenced. It also provides the signal system and object property functions. You can cast an object as a `GObject` with `G_OBJECT()`. If you try to cast an object with `G_OBJECT()` that is not a `GObject` or derived from it, GLib will throw a critical error, and the cast will fail. This will occur with any other GTK+ casting function.
- `GInitiallyUnowned` should never be accessed by the programmer, since all of its members are private. It exists so that references can be floating. A floating reference is one that is not owned by anyone.
- `GtkObject` is the base class for all GTK+ objects. It was replaced as the absolute base class of all objects in GTK+ 2.0, but `GtkObject` was kept for backward compatibility of nonwidget classes like `GtkAdjustment`. You can cast an object as a `GtkObject` with `GTK_OBJECT()`.
- `GtkWidget` is an abstract base class for all GTK+ widgets. It introduces style properties and standard functions that are needed by all widgets. The standard practice is to store all widgets as a `GtkWidget`, which can be seen in Listing 2-1. Therefore, you will rarely need to use `GTK_WIDGET()` to cast an object.
- `GtkContainer` is an abstract class that is used to contain one or more widgets. It is an extremely important structure, since you could not add any other widgets to a window without it. Therefore, the whole of Chapter 3 is dedicated to widgets derived from this class. You can cast an object as a `GtkContainer` with `GTK_CONTAINER()`.
- `GtkBin` is another abstract class that allows a widget to contain only one child. It allows multiple widgets to have this functionality without the need for reproduction of code. You can cast an object as a `GtkBin` with `GTK_BIN()`.
- `GtkWindow` is the standard window object you saw in Listing 2-1. You can use `GTK_WINDOW()` to cast an object.

Every widget in this book will use a similar widget hierarchy. It is useful to have the API documentation handy, so you can reference the hierarchy of the widgets you are using. The API documentation is available at `www.gtk.org/api`, if you did not install it along with the libraries.

For now, it is enough to know how to cast objects and what the basic abstract types are used for. In Chapter 11, you will learn how to create your own widgets. At that point, we will delve further into the workings of the GObject system.

GTK+ Windows

The code in Listing 2-1 creates a `GtkWindow` object that is set to the default width and height of 200 pixels. This default size was chosen because a window with a width and height of 0 pixels cannot be resized. You should note that the title bar and window border are included in the total size, so the working area of the window is smaller than 200 pixels by 200 pixels.

We passed `GTK_WINDOW_TOPLEVEL` to `gtk_window_new()`. This tells GTK+ to create a new top-level window. Top-level windows use window manager decorations, have a border frame, and allow themselves to be placed by the window manager. This means that you do *not* have absolute control over your window position and should not assume that you do.

```
GtkWidget *window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
```

It is important to make the distinction between what GTK+ controls and what the window manager controls. You are able to make recommendations and requests for the size and placement of top-level widgets. However, the window manager has ultimate control of these features.

Conversely, you can use `GTK_WINDOW_POPUP` to create a pop-up window, although its name is somewhat misleading in GTK+. Pop-up windows are used for things that are not normally thought of as windows, such as tooltips and menus.

Pop-up windows are ignored by the window manager, and therefore, they have no decorations or border frame. There is no way to minimize or maximize a pop-up window, because the window manager does not know about them. Resize grips are not shown, and default key bindings will not work.

`GTK_WINDOW_TOPLEVEL` and `GTK_WINDOW_POPUP` are the only two elements available in the `GtkWindowType` enumeration. In most cases, you will want to use `GTK_WINDOW_TOPLEVEL`, unless there is a compelling reason not to.

Note You should not use `GTK_WINDOW_POPUP` if you only want window manager decorations turned off for the window. Instead, use `gtk_window_set_decorated (GtkWindow *window, gboolean show)` to turn off window decorations.

The following function requests the title bar and taskbar to display "Hello World!" as the title of the window. Since `gtk_window_set_title()` requires a `GtkWindow` object as it's the first parameter, we must cast our window using the `GTK_WINDOW()` function.

```
void gtk_window_set_title (GtkWindow *window,
                           const gchar *title);
```

The second parameter of `gtk_window_set_title()` is the title that will be displayed by the window. It uses GLib's implementation of `char`, which is called `gchar`. When you see a parameter listed as `gchar*`, it will also accept `const char*`, because `gchar*` is defined as a `typedef` of the standard C string object.

The last function of interest in this section is `gtk_widget_show()`, which tells GTK+ to set the specified widget as visible. The widget may not be immediately shown when you call `gtk_widget_show()`, because GTK+ queues the widget until all preprocessing is complete before it is drawn onto the screen.

It is important to note that `gtk_widget_show()` will only show the widget it is called on. If the widget has children that are not already set as visible, they will not be drawn on the screen. Furthermore, if the widget's parent is not visible, it will not be drawn on the screen. Instead, it will be queued until its parent is set as visible as well.

In addition to showing a widget, it is also possible to use `gtk_widget_hide()` to hide a widget from the user's view.

```
void gtk_widget_hide (GtkWidget *widget);
```

This will hide all child widgets from view, but you should be careful. This function only sets the specified widget as hidden. If you show the widget at a later time, its children will be visible as well, since they were never marked as hidden. This will become an important distinction to make when you learn how to show and hide multiple widgets at once.

The Main Loop Function

After all initialization is complete and necessary signals are connected in a GTK+ application, there will come a time when you want to let the GTK+ main loop take control and start processing events. To do this, you will call `gtk_main()`, which will continue to run until you call `gtk_main_quit()` or the application terminates. This should be the last GTK+ function called in `main()`.

After you call `gtk_main()`, it is not possible to regain control of the program until a callback function is initialized. In GTK+, signals and callback functions are triggered by user actions such as button clicks, asynchronous input-output events, programmable timeouts, and others. We will start exploring signals, events, and callback functions in the next example.

■ **Note** It is also possible to create functions that are called at a specified interval of time; these are referred to as timeouts. Another type of callback function, referred to as an idle function, is called when the operating system is not busy processing other tasks. Both of these features are a part of GLib and will be explored in detail in Chapter 6.

Other than those few situations, control of the application is managed by signals, timeout functions, and various other callback functions once `gtk_main()` is called. Later in this chapter, you will see how to use signals and callbacks in your own applications.

Using GCC and pkg-config to Compile

Now that you understand how Listing 2-1 works, it is time to compile the code into an executable. To do this, you run the following command from a terminal:

```
gcc -Wall -g helloworld.c -o helloworld `pkg-config --cflags gtk+-2.0` \
    `pkg-config --libs gtk+-2.0`
```

This command can be used for all of the examples in this book except those in Chapter 10, which will require libglade as well. I decided to use the GCC compiler, because it is the standard C compiler on Linux, but most C and C++ compilers will work. To use another compiler, you will need to reference its documentation.

The previous compile command is parsed with multiple provided options. The `-Wall` option enables all types of compiler warnings. While this may not always be desirable, it can help you detect simple programming errors as you begin programming with GTK+. Debugging is enabled with `-g`, so that you will be able to use your compiled application with GDB or your debugger of choice.

The next set of commands, `helloworld.c -o helloworld`, compiles the specified file and outputs it to an executable file named `helloworld`. One or many source files may be specified for compilation by GCC.

■ **Caution** The single, slanted quotation mark used in the compile command is a backquote, which is found on the key in the top-left corner of most keyboards. This tells your terminal that the command between the quotes should be run and replaced by the output before the rest of the line is executed.

In addition to the GCC compiler, you need to use the pkg-config application, which returns a list of specified libraries or paths.

The first instance, `pkg-config --cflags gtk+-2.0`, returns directory names to the compiler's include path. This will make sure that the GTK+ header files are available to the compiler. Try running `pkg-config --cflags gtk+-2.0` in your terminal to see an example of what is being output to the compiler.

The second call, `pkg-config --libs gtk+-2.0`, appends options to the command line used by the linker including library directory path extensions and a list of libraries needed for linking to the executable. The libraries that are returned in a standard Linux environment follow:

- GTK+ (`-lgtk`): Graphical widgets
- GDK (`-lgdk`): The standard graphics rendering library
- GdkPixbuf (`-lgdk_pixbuf`): Client-side image manipulation
- Pango (`-lpango`): Font rendering and output
- GObject (`-lgobject`): Object-oriented type system
- GModule (`-lgmodule`): Dynamically loading libraries
- GLib (`-lglib`): Data types and utility functions
- Xlib (`-lX11`): X Window System protocol library
- Xext (`-lXext`): X extensions library routines
- GNU math library (`-lm`): The GNU library from which GTK+ uses many routines

As you can see, pkg-config provides a convenient way for you to avoid hard-coding a long list of includes and libraries manually every time you compile a GTK+ application.

Listing 2-1 is one of the simplest applications that you can create with GTK+. It produces a top-level `GtkWindow` widget with a default width and height of 200 pixels, as displayed in Figure 2-2.

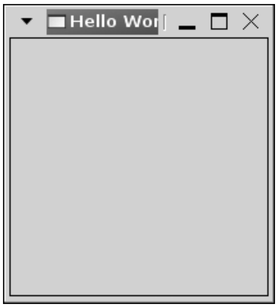

Figure 2-2. *The Hello World window at the default size*

Even though the window includes the standard X on the right side of the title bar, you'll notice that clicking that X will only cause the window to disappear. The application continues to wait for events, and control will not be returned to the launching terminal until you press Ctrl+C. You will learn how to implement a shutdown callback with signals in the next example.

Extending "Hello World"

Every GTK+ application you write requires the function calls shown in Listing 2-1, but the example on its own is clearly not exceptionally useful. Now that you understand how to get started, it is time for us to say "hello" to the world in a more useful manner.

Listing 2-2 expands upon our "Hello World" application in two ways. First, it connects call-back functions to window signals, so the application can terminate itself. Secondly, this example introduces the `GtkContainer` structure, which allows a widget to contain one or more other widgets.

Listing 2-2. *Greeting the World Again (helloworld2.c)*

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

static void destroy (GtkWidget*, gpointer);
static gboolean delete_event (GtkWidget*, GdkEvent*, gpointer);

int main (int argc,
          char *argv[])
{
  GtkWidget *window, *label;

  gtk_init (&argc, &argv);

  window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  gtk_window_set_title (GTK_WINDOW (window), "Hello World!");
  gtk_container_set_border_width (GTK_CONTAINER (window), 10);
  gtk_widget_set_size_request (window, 200, 100);

  /* Connect the main window to the destroy and delete-event signals. */
  g_signal_connect (G_OBJECT (window), "destroy",
                    G_CALLBACK (destroy), NULL);
  g_signal_connect (G_OBJECT (window), "delete_event",
                    G_CALLBACK (delete_event), NULL);

  /* Create a new GtkLabel widget that is selectable. */
  label = gtk_label_new ("Hello World");
  gtk_label_set_selectable (GTK_LABEL (label), TRUE);

  /* Add the label as a child widget of the window. */
  gtk_container_add (GTK_CONTAINER (window), label);
  gtk_widget_show_all (window);

  gtk_main ();
  return 0;
}
```

```
/* Stop the GTK+ main loop function when the window is destroyed. */
static void
destroy (GtkWidget *window,
         gpointer data)
{
  gtk_main_quit ();
}

/* Return FALSE to destroy the widget. By returning TRUE, you can cancel
 * a delete-event. This can be used to confirm quitting the application. */
static gboolean
delete_event (GtkWidget *window,
              GdkEvent *event,
              gpointer data)
{
   return FALSE;
}
```

In Figure 2-3, you can see a screenshot of Listing 2-2 in action. It shows the `GtkLabel` contained by a `GtkWindow`. Let us now take a look at the new features presented by this example.

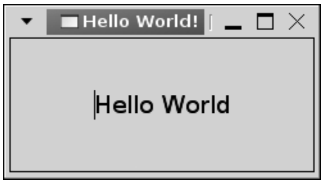

Figure 2-3. *The extended Hello World window*

The GtkLabel Widget

In Listing 2-2, a new type of widget called `GtkLabel` was created. As the name implies, `GtkLabel` widgets are normally used to label other widgets. However, they can also be used for such things as creating large blocks of noneditable, formatted, or wrapped text.

You can create a new label widget by calling `gtk_label_new()`. Passing `NULL` to `gtk_label_new()` is equivalent to passing an empty string. This will cause the label to be displayed without any text.

```
GtkWidget* gtk_label_new (const gchar *str);
```

It is not possible for users to edit a normal `GtkLabel` with the keyboard or mouse (without some extra work by the programmer, that is), but by using `gtk_label_set_selectable()`, the user will be able to select and copy the text. The widget will also be able to accept cursor focus, so you can use the Tab key to move between the label and other widgets.

```
void gtk_label_set_selectable (GtkLabel *label,
                               gboolean selectable);
```

The ability to select labels is turned off by default, because this feature should only be used when there is a need for the user to retain the information. For example, error messages should be set as selectable, so they can easily be copied into other applications such as a web browser.

The text in a `GtkLabel` does not have to remain in the same state as the text string you specified during creation. You can easily change it with `gtk_label_set_text()`. Any text currently contained by the label will be overwritten as well as any mnemonics.

```
void gtk_label_set_text (GtkLabel *label,
                         const gchar *str);
```

■ **Note** A mnemonic is a combination of keys that, when pressed by the user, will perform some type of action. It is possible to add a mnemonic to a `GtkLabel` that will activate a designated widget when pressed.

The string currently being displayed by the label can be retrieved with `gtk_label_get_text()`. The returned string will not include any markup or mnemonic information. The label also uses it internally, so you should never modify the returned string!

The last `GtkLabel` method you should know about is `gtk_label_set_markup()`, which allows you to define custom styles for the displayed text. There are a number of tags provided by the Pango Text Markup Language, which can be found in Appendix C in the back of this book.

```
void gtk_label_set_markup (GtkLabel *label,
                           const gchar *str);
```

The Pango Text Markup Language provides two types of style methods. You can use the `<span>` tag with some attributes such as the font type, size, weight, foreground color, and others. It also provides various other tags such as `<b>`, `<tt>`, and `<i>`, which make the enclosed text bold, monospace, or italic.

Container Widgets and Layout

Recall from the first example in this chapter that the `GtkWindow` structure is derived indirectly from `GtkContainer`. This indicates that `GtkWindow` is a `GtkContainer` and inherits all of the `GtkContainer` functions, signals, and properties.

By using `gtk_container_add()`, you can add a widget as the child of the container. It follows that the container is now the widget's parent. The language popularly used to describe this container and contained relationship is "parent and child," where the parent is the containing widget, and the child is contained in the parent.

```
void gtk_container_add (GtkContainer *container,
                        GtkWidget *child);
```

This language unfortunately often causes confusion, because GTK+ is object oriented in every sense. Because of this, when using and talking about GTK+, one must be aware of the context in which "parent" and "child" is used. They are used to talk about both container widget relationships and about widget derivation relationships.

The purpose of the `GtkContainer` class is to allow a parent widget to contain one or more children. `GtkWindow` is derived from a type of container called `GtkBin`. `GtkBin` allows the parent to contain only one child. Windows, as containers, are therefore limited to directly containing a single child. Fortunately, that single child may be a more complex container widget itself, which, in turn, may contain more than one child widget.

It is important to notice that our window is no longer the default 200 by 200 pixels in size and that the square aspect ratio is not retained. This is because GTK+ uses, primarily, an automatic and dynamically sized layout system. This dynamic sizing is the reason behind the existence of container objects. The sizing system will be discussed in more detail in the next chapter, which covers container widgets.

Because our window is a `GtkContainer`, we can also use the function `gtk_container_set_border_width()` to place a 10-pixel border around the inside edge of the window. The border is set on all four sides of the child widget.

```
void gtk_container_set_border_width (GtkContainer *container,
                                     guint border_width);
```

Without adding the border, the layout manager would allow the window to shrink to the default size of the `GtkLabel` widget. In Listing 2-1, the window is set to a width of 200 pixels and a height of 100 pixels. With this size, there will be more than a 10-pixel border around the label on most systems. The border will prevent the user from resizing the window to a smaller size than allocated by the widget and the border.

We then call `gtk_widget_show_all()` on the window. This function recursively draws the window, its children, its children's children and so on. Without this function, you would have to call `gtk_widget_show()` on every single child widget. Instead, by using `gtk_widget_show_all()`, GTK+ does all of the work for you by showing each widget until they are all visible on the screen.

```
void gtk_widget_show_all (GtkWidget *widget);
```

Like the nonrecursive `gtk_widget_show()`, if you call this function on a widget whose parent is not set as visible, it will not be shown. The widget will be queued until its parent is set as visible.

GTK+ also provides `gtk_widget_hide_all()`, which will set the specified widget and all of its children as hidden. Because contained widgets are invisible when their container is hidden, it will appear that `gtk_widget_hide()`, when called on the containing object, does the same thing as `gtk_widget_hide_all()`, because both will hide the container and all of its children. However, there is an important difference. Calling `gtk_widget_hide()` sets the visible property to `FALSE` on only one widget, while `gtk_widget_hide_all()` changes that property on the passed widget and recursively on all contained widgets.

```
void gtk_widget_hide_all (GtkWidget *widget);
```

The `gtk_widget_show()` and `gtk_widget_show_all()` set of functions have the same relationship. So, if you use `gtk_widget_hide_all()` but call `gtk_widget_show()` on the same widget, all of its children will remain invisible.

Container widgets and managing the application layout will be covered in more detail in the next chapter. Since you have enough information to understand the `GtkContainer` in Listing 2-2, we will continue on to signals and callback functions.

Signals and Callbacks

GTK+ is a system that relies on signals and callback functions. A signal is a notification to your application that the user has performed some action. You can tell GTK+ to run a function when the signal is emitted. These are named callback functions.

■ **Caution** GTK+ signals are not the same thing as POSIX signals! Signals in GTK+ are propagated by events from the X Window System. Each provides separate methods, and these two signal types should not be used interchangeably.

After you initialize your user interface, control is given to the `gtk_main()` function, which sleeps until a signal is emitted. At this point, control is passed to other functions called callback functions.

You, as the programmer, connect signals to their callback functions before calling `gtk_main()`. The callback function will be called when the action has occurred and the signal is emitted or when you have explicitly emitted the signal. You also have the capability of stopping signals from being emitted at all.

■ **Note** It is possible to connect signals at any point within your applications. For example, new signals can be connected within callback functions. However, you should try to initialize mission-critical callbacks before calling gtk_main().

There are many types of signals, and just like functions, they are inherited from parent structures. Many signals are generic to all widgets such as `hide` and `grab-focus` or specific to the widget such as the `GtkRadioButton` signal `group-changed`. In either case, widgets derived from a class can use all of the signals available to all of its ancestors.

Connecting the Signal

The first instance of a signal you have encountered was in Listing 2-2. The `GtkWindow` was connected to the `destroy()` callback function. This function will be called when the `destroy` signal is emitted.

```
g_signal_connect (G_OBJECT (window), "destroy",
                  G_CALLBACK (destroy), NULL);
```

GTK+ emits the `destroy` signal when `gtk_widget_destroy()` is called on the widget or when `FALSE` is returned from a `delete_event()` callback function. If you reference the API documentation, you will see that the `destroy` signal belongs to the `GtkObject` class. This means that every class in GTK+ inherits the signal, and you can be notified of the destruction of any GTK+ structure.

There are four parameters to every `g_signal_connect()` call. The first is the widget that is to be monitored for the signal. Next, you specify the name of the signal you want to track. Each widget has many possible signals, all of which can be found in the API documentation. Remember that widgets are free to use the signals of their ancestors, since each widget is actually an implementation of each of its ancestors. You can use the "Object Hierarchy" section of the API to reference parent classes.

```
gulong g_signal_connect (gpointer object,
                         const gchar *signal_name,
                         GCallback handler,
                         gpointer data);
```

When typing the signal name, the underscore and dash characters are interchangeable. They will be parsed as the same character, so it does not make any difference which one you choose. I will use the underscore character for all of the examples in this book.

The third parameter in `g_signal_connect()` is the callback function that will be called when the signal is emitted, cast with `G_CALLBACK()`. The format of the callback function depends on the function prototype requirements of each specific signal. An example callback function is shown in the next section.

The last parameter in `g_signal_connect()` allows you to send a pointer to the callback function. In Listing 2-2, we passed `NULL`, so the pointer was void, but let us assume for a moment that we wanted to pass the `GtkLabel` to the callback function.

In this instance of `g_signal_connect()`, the `GtkLabel` was cast as a `gpointer`, which will be passed to the callback function. A `gpointer` is simply a type definition of a void pointer. You can recast this in the callback function, but `g_signal_connect()` requires a `gpointer` type.

```
g_signal_connect (G_OBJECT (window), "destroy",
                  G_CALLBACK (destroy),
                  (gpointer) label);
```

The return value for `g_signal_connect()` is the handler identifier of the signal. You can use this with `g_signal_handler_block()`, `g_signal_hander_unblock()`, and `g_signal_handler_disconnect()`. These functions will stop a callback function from being called, re-enable the callback function, and remove the signal handler from memory, respectively. More information can be found in the API documentation.

Callback Functions

Callback functions specified in `g_signal_connect()` will be called when the signal is emitted on the widget to which it was connected. For all signals, with the exception of events, which will be covered in the next section, callback functions are in the following form.

```
static void
callback_function (GtkWidget *widget,
                   ... /* Other Possible Arguments */ ...,
                   gpointer data);
```

You can find an example format of a callback function for each signal in the API documentation, but this is the generic format. The first parameter is the object from `g_signal_connect()`, except it must always be cast as the widget type for which the signal was created. If you need access to a widget type from which the widget was derived, you can use the built-in casting functions.

There are other possible arguments that may appear in the middle as well, although this is not always the case. For these parameters, you need to reference the documentation of the signal you are utilizing.

The last parameter of your callback function corresponds to the last parameter of `g_signal_connect()`. Since the data is passed as a void pointer, you can place the data type you want it cast to as the last parameter of the callback function. Let us assume that you passed a `GtkLabel` to the fourth parameter of `g_signal_connect()`.

```
static void
destroy (GtkWidget *window,
         GtkLabel *label)
```

In this example, we were sure that the object was of the type `GtkLabel`, so we used `GtkLabel` as the last parameter of the callback function. This will avoid having to recast the object from a `gpointer` to the desired data type.

In Chapter 11, you will be covering how to create your own signals when you are taught how to create custom widgets.

Emitting and Stopping Signals

Before we move onto events, there are two interesting functions that you should know about that relate to signals. By using `g_signal_emit_by_name()`, you can emit a signal on an object by using its textual name. You can use the signal identifier to emit a signal as well, but it is much more likely that you will have access to the signal's name. If you have the signal identifier, you can emit the signal with `g_signal_emit()`.

```
void g_signal_emit_by_name (gponter instance,
                            const gchar *signal_name,
                            ...);
```

The last parameters of `g_signal_emit_by_name()` are a list of parameters that should be passed to the signal and the location to store the return value. The return value can safely be ignored if it is a void function.

You can also use `g_signal_stop_emission_by_name()` to stop the current emission of a signal. This allows you to temporarily disable a signal that will be emitting because of some action performed by your code.

```
void g_signal_stop_emission_by_name (gpointer instance,
                                     const gchar *signal_name);
```

Events

Events are special types of signals that are emitted by the X Window System. They are initially emitted by the X Window System and then sent from the window manager to your application to be interpreted by the signal system provided by GLib. For example, the `destroy` signal is emitted on the widget, but the `delete-event` event is first recognized by the underlying `GdkWindow` of the widget and then emitted as a signal of the widget.

The first instance of an event you encountered was delete-event in Listing 2-2. The delete-event signal is emitted when the user tries to close the window. The window can be exited by clicking the close button on the title bar, using the close pop-up menu item in the taskbar, or by any other means provided by the window manager.

Connecting events to a callback function is done in the same manner with g_signal_connect() as with other GTK+ signals. However, your callback function will be set up slightly differently.

```
static gboolean
callback_function (GtkWidget *widget,
                   GdkEvent *event,
                   gpointer data);
```

The first difference in the callback function is the gboolean return value. If TRUE is returned from an event callback, GTK+ assumes the event has already been handled and will not continue. By returning FALSE, you are telling GTK+ to continue handling the event. FALSE is the default return value for the function, so you do not need to use the delete-event signal in most cases. This is only useful if you want to override the default signal handler.

For example, in many applications, you may want to confirm the exit of the program. By using the following code, you can prevent the application from exiting if the user does not want to quit.

```
static gboolean
delete_event (GtkWidget *window,
              GdkEvent *event,
              gpointer data)
{
  gboolean answer = /* Ask the user if exiting is desired. */

  if (answer)
    return FALSE;
  else
    return TRUE;
}
```

By returning FALSE from the delete-event callback function, gtk_widget_destroy() is automatically called on the widget. As stated before, this signal will automatically continue with the action, so there is no need to connect to it unless you want to override the default.

In addition, the callback function includes the GdkEvent parameter. GdkEvent is a C union of the GdkEventType enumeration and all of the available event structures. Let's first look at the GdkEventType enumeration.

Event Types

The GdkEventType enumeration provides a list of available event types. These can be used to determine the type of event that has occurred, since you may not always know what has happened.

For example, if you connect the button-press-event signal to a widget, there are three different types of events that can cause the signal's callback function to be run: GDK_BUTTON_PRESS, GDK_2BUTTON_PRESS, and GDK_3BUTTON_PRESS. Double-clicks and triple-clicks emit the GDK_BUTTON_PRESS as a second event as well, so being able to distinguish between different types of events is necessary.

In Appendix B, you can see a complete list of the events available to you. It shows the signal name that is passed to g_signal_connect(), the GdkEventType enumeration value, and a description of the event.

Let's look at the delete-event callback function from Listing 2-2. We already know that delete-event is of the type GDK_DELETE, but let us assume for a moment that we did not know that. We can easily test this by using the following conditional statement:

```
static gboolean
delete_event (GtkWidget *window,
              GdkEvent *event,
              gpointer data)
{
  if (event->type == GDK_DELETE)
    return FALSE;

  return TRUE;
}
```

In this example, if the event type is GDK_DELETE, FALSE is returned, and gtk_widget_destroy() will be called on the widget. Otherwise, TRUE is returned, and no further action is taken.

Using Specific Event Structures

Sometimes, you may already know what type of event has been emitted. In the following example, we know that a key-press-event will always be emitted:

```
g_signal_connect (G_OBJECT (widget), "key-press-event"
                  G_CALLBACK (key_press), NULL);
```

In this case, it is safe to assume that the type of event will always be GDK_KEY_PRESS, and the callback function can be declared as such.

```
static gboolean
key_press (GtkWidget *widget,
           GdkEventKey *event,
           gpointer data)
```

Since we know that the type of event is a GDK_KEY_PRESS, we will not need access to all of the structures in GdkEvent. We will only have a use for GdkEventKey, which we can use instead of GdkEvent in the callback function. Since the event is already cast as GdkEventKey, we will have direct access to only the elements in that structure.

```
typedef struct
{
  GdkEventType type;         // GDK_KEY_PRESS or GDK_KEY_RELEASE
  GdkWindow *window;         // The window that received the event
  gint8 send_event;          // TRUE if the event used XSendEvent
  guint32 time;              // The length of the event in milliseconds
  guint state;               // The state of Control, Shift, and Alt
  guint keyval;              // The key that was pressed <gdk/gdkkeysyms.h>
  gint length;               // The length of string
  gchar *string;             // A string approximating the entered text
  guint16 hardware_keycode;  // Raw code of the key that was pressed or released
  guint8 group;              // The keyboard group
  guint is_modifier : 1;     // Whether hardware_keycode was mapped (since 2.10)
} GdkEventKey;
```

There are many useful properties in the GdkEventKey structure that we will use throughout the book. At some point it would be useful for you to browse some of the GdkEvent structures in the API documentation. We will cover a few of the most important structures in this book, including GdkEventKey and GdkEventButton.

The only variable that is available in all of the event structures is the event type, which defines the type of event that has occurred. It is a good idea to always check the event type to avoid handling it in the wrong way.

Further GTK+ Functions

Before continuing on to further examples, I would like to draw your attention to a few functions that will come in handy in later chapters and when you create your own GTK+ applications.

GtkWidget Functions

The GtkWidget structure contains many useful functions that you can use with any widget. This section outlines a few that you will need in a lot of your applications.

It is possible to destroy a widget by explicitly calling gtk_widget_destroy() on the object. When invoked, gtk_widget_destroy() will drop the reference count on the widget and all of its children recursively. The widget, along with its children, will then be destroyed, and all memory freed.

```
void gtk_widget_destroy (GtkWidget *widget);
```

Generally, this is only called on top-level widgets. It is usually only used to destroy dialog windows and to implement menu items that quit the application. It will be used in the next example in this chapter to quit the application when a button is clicked

You can use `gtk_widget_set_size_request()` to set the minimum size of a widget. It will force the widget to be either smaller or larger than it would normally be. It will not, however, resize the widget so that it is too small to be functional or able to draw itself on the screen.

```
void gtk_widget_set_size_request (GtkWidget *widget,
                                  gint width,
                                  gint height);
```

By passing `-1` to either parameter, you are telling GTK+ to use its natural size, or the size that the widget would normally be allocated to if you do not define a custom size. This can be used if you want to specify either only the height or only the width parameter. It will also allow you to reset the widget to its original size.

There is no way to set a widget with a width or height of less than 1 pixel, but by passing `0` to either parameter, GTK+ will make the widget as small as possible. Again, it will not be resized so small that it's nonfunctional or unable to draw itself.

Because of internationalization, there is a danger by setting the size of any widget. The text may look great on your computer, but on a computer using a German translation of your application, the widget may be too small or large for the text. Themes also present issues with widget sizing, because widgets are defaulted to different sizes depending on the theme. Therefore, it is best to allow GTK+ to choose the size of widgets and windows in most cases.

You can use `gtk_widget_grab_focus()` to force a widget to grab keyboard focus. This will only work on widgets that can handle keyboard interaction. One example of a use for `gtk_widget_grab_focus()` is sending the cursor to a text entry when the search toolbar is shown in Firefox. This could also be used to give focus to a `GtkLabel` that is selectable.

```
void gtk_widget_grab_focus (GtkWidget *widget);
```

Often, you will want to set a widget as inactive. By calling `gtk_widget_set_sensitive()`, the specified widget and all of its children are disabled or enabled. By setting a widget as inactive, the user will be prevented from interacting with the widget. Most widgets will also be grayed out when set as inactive.

```
void gtk_widget_set_sensitive (GtkWidget *widget,
                               gboolean sensitive);
```

If you want to re-enable a widget and its children, you need only to call this function on the same widget. Children are affected by the sensitivity of their parents, but they only reflect the parent's setting instead of changing their properties.

GtkWindow Functions

You have now seen two examples using the `GtkWindow` structure. You have learned how to add border padding between the inner edge of the window and its child. You have also learned how to set the title of a window and add a child widget. Now, let us explore a few more functions that will allow you to further customize windows.

All windows are set as resizable by default. This is desirable in most applications, because each user will have different size preferences. However, if there is a specific reason for doing so, you can use gtk_window_set_resizable() to prevent the user from resizing the window.

```
void gtk_window_set_resizable (GtkWindow *window,
                               gboolean resizable);
```

■ **Caution** You should note that the ability to resize is controlled by the window manager, so this setting may not be honored in all cases!

The note directly above brings up an important point. Much of what GTK+ does interacts with the functionality provided by the window manager. Because of this, not all of your window settings may be followed on all window managers. This is because your settings are merely hints given that are then either used or ignored. You should keep in mind that your requests may or may not be honored when designing applications with GTK+.

The default size of a GtkWindow can be set with gtk_window_set_default_size(), but there are a few things to watch out for when using this function. If the minimum size of the window is larger than the size you specify, this function will be ignored by GTK+. It will also be ignored if you have previously set a larger size request.

```
void gtk_window_set_default_size (GtkWindow *window,
                                  gint width,
                                  gint height);
```

Unlike gtk_widget_set_size_request(), gtk_window_set_default_size() only sets the initial size of the window—it does not prevent the user from resizing it to a larger or smaller size. If you set a height or width parameter to 0, the window's height or width will be set to the minimum possible size. If you pass -1 to either parameter, the window will be set to its natural size.

You can *request* that the window manager move the window to the specified location with gtk_window_move(). However, the window manager is free to ignore this request. This is true of all "request" functions that require action from the window manager.

```
void gtk_window_move (GtkWindow *window,
                      gint x,
                      gint y);
```

By default, the position of the window on the screen is calculated with respect to the top-left corner of the screen, but you can use gtk_window_set_gravity() to change this assumption.

```
void gtk_window_set_gravity (GtkWindow *window,
                             GdkGravity gravity);
```

This function defines the gravity of the widget, which is the point that layout calculations will consider (0, 0). Possible values for the `GdkGravity` enumeration include `GDK_GRAVITY_NORTH_WEST`, `GDK_GRAVITY_NORTH`, `GDK_GRAVITY_NORTH_EAST`, `GDK_GRAVITY_WEST`, `GDK_GRAVITY_CENTER`, `GDK_GRAVITY_EAST`, `GDK_GRAVITY_SOUTH_WEST`, `GDK_GRAVITY_SOUTH`, `GDK_GRAVITY_SOUTH_EAST`, and `GDK_GRAVITY_STATIC`.

North, south, east, and west refer to the top, bottom, right, and left edges of the screen. They are used to construct multiple gravity types. `GDK_GRAVITY_STATIC` refers to the top-left corner of the window itself, ignoring window decorations.

If your application has more than one window, you can set one as the parent with `gtk_window_set_transient_for()`. This allows the window manager to do things such as center the child above the parent or make sure one window is always on top of the other. We will explore the idea of multiple windows and transient relationships in Chapter 5 when discussing dialogs.

```
void gtk_window_set_transient_for (GtkWindow *window,
                                   GtkWindow *parent);
```

You can set the icon that will appear in the task bar and title bar of the window by calling `gtk_window_set_icon_from_file()`. The size of the icon does not matter, because it will be resized when the desired size is known. This allows for the best quality possible of the scaled icon.

```
gboolean gtk_window_set_icon_from_file (GtkWindow *window,
                                        const gchar *filename,
                                        GError **err); // NULL
```

`TRUE` is returned if the icon was successfully loaded and set. Therefore, unless you want in-depth information on why the icon loading failed, it is safe to pass `NULL` to the third parameter for now. We will discuss the `GError` structure in Chapter 4.

Process Pending Events

At times, you may want to process all pending events in an application. This is extremely useful when you are running a piece of code that will take a long time to process. This will cause your application to appear frozen, because widgets will not be redrawn if the CPU is taken up by another process. For example, in an integrated development environment that I have created called OpenLDev, I have to update the user interface while a build command is being processed. Otherwise, the window would lock up, and no build output would be shown until the build was complete.

The following loop is the solution for this problem. It is the answer to a great number of questions presented by new GTK+ programmers.

```
while (gtk_events_pending ())
  gtk_main_iteration ();
```

The loop calls `gtk_main_iteration()`, which will process the first pending event for your application. This is continued while `gtk_events_pending()` returns `TRUE`, which tells you whether there are events waiting to be processed.

Using this loop is an easy solution to the freezing problem, but a better solution would be to use coding strategies that avoid the problem altogether. For example, you can use idle functions, which will be covered in Chapter 6, to call a function only when there are no actions of greater importance to process.

Buttons

The `GtkButton` widget is a special type of container that turns its child into a clickable entity. It is only capable of holding one child. However, that child can be a container itself, so the button can theoretically be the ancestor of large amounts of children. This allows the button to hold, for example, a label and an image at the same time.

Because the purpose of a `GtkButton` widget is to make the child clickable, you will almost always need to use the `clicked` signal to get notification of when the button is activated. You will use this signal in the following example.

The `GtkButton` widget is usually initialized with `gtk_button_new_with_label()`, which creates a new button with a `GtkLabel` as its child. If you want to create an empty `GtkButton` and add your own child at a later time, you can use `gtk_button_new()`, although this is not what you will want to do in most cases.

Figure 2-4 shows a button with mnemonic capabilities. You can recognize a mnemonic label by the underlined character. In the case of the button below, when Alt+C is pressed, the button will be clicked.

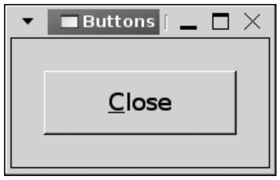

Figure 2-4. *A GtkButton widget with a mneumonic label*

The function `gtk_button_new_with_mnemonic()` will initialize a new button with mnemonic label support. When the user presses the Alt key along with the specified accelerator key, the button will be activated. An accelerator is a key or set of keys that can be used to activate a predefined action.

■ **Note** When the mnemonic option is available for a widget that provides some type of user interaction, it is recommended that you take advantage of that capability. Even if you do not use keyboard shortcuts, some users prefer to navigate user interfaces using a keyboard instead of a mouse.

Listing 2-3 is a simple demonstration of `GtkButton` capabilities using the `clicked` signal. When the button is pressed, the window will be destroyed, and the application will quit. The button in this example also takes advantage of the mnemonic and keyboard accelerator features. You saw a screenshot of this example in Figure 2-4.

Listing 2-3. *The GtkButton Widget (buttons.c)*

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

static void destroy (GtkWidget*, gpointer);

int main (int argc,
          char *argv[])
{
  GtkWidget *window, *button;

  gtk_init (&argc, &argv);

  window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  gtk_window_set_title (GTK_WINDOW (window), "Buttons");
  gtk_container_set_border_width (GTK_CONTAINER (window), 25);
  gtk_widget_set_size_request (window, 200, 100);

  g_signal_connect (G_OBJECT (window), "destroy",
                    G_CALLBACK (destroy), NULL);

  /* Create a new button that has a mnemonic key of Alt+C. */
  button = gtk_button_new_with_mnemonic ("_Close");
  gtk_button_set_relief (GTK_BUTTON (button), GTK_RELIEF_NONE);

  /* Connect the button to the clicked signal. The callback function recieves the
   * window followed by the button because the arguments are swapped. */
  g_signal_connect_swapped (G_OBJECT (button), "clicked",
                            G_CALLBACK (gtk_widget_destroy),
                            (gpointer) window);

  gtk_container_add (GTK_CONTAINER (window), button);
  gtk_widget_show_all (window);

  gtk_main ();
  return 0;
}

/* Stop the GTK+ main loop function. */
static void
destroy (GtkWidget *window,
         gpointer data)
{
  gtk_main_quit ();
}
```

In Listing 2-3, `gtk_widget_destroy()` is called on the main window when the button is clicked. This is a very simple example, but it has a practical use in most applications.

The GNOME Human Interface Guidelines, which can be viewed or downloaded at `http://developer.gnome.org/projects/gup/hig`, state that preferences dialogs should apply settings immediately after a setting is changed.

Therefore, if you create a preferences dialog, there is a good chance that you will only need one button. The purpose of the button would be to destroy the window that contains the button and save the changes.

After creating the button, `gtk_button_set_relief()` can be used to add a certain magnitude of relief around the `GtkButton`. Relief is a type of 3-D border that distinguishes the button from surrounding widgets. Values of the `GtkReliefStyle` enumeration follow:

- `GTK_RELIEF_NORMAL`: Add relief around all edges of the button.
- `GTK_RELIEF_HALF`: Add relief around only half of the button.
- `GTK_RELIEF_NONE`: Add no relief around the button.

Listing 2-3 introduces `g_signal_connect_swapped()`, a new signal connection function. This function swaps the position of the object on which the signal is being emitted and the data parameter when running the callback function.

```
g_signal_connect_swapped (G_OBJECT (button), "clicked",
                          G_CALLBACK (gtk_widget_destroy),
                          (gpointer) window);
```

This allows you to use `gtk_widget_destroy()` on the callback function, which will call `gtk_widget_destroy (window)`. If the callback function only receives one parameter, the object will be ignored.

Widget Properties

GObject provides a property system, which allows you to customize how widgets interact with the user and how they are drawn on the screen. In this section, you will learn how to use styles, resource files and GObject's property system.

Every class derived from the `GObject` class can install any number of properties. In GTK+, these properties store information about how the widget will act. For example, `GtkButton` has a property called `relief` that defines the relief style used by the button.

In the following code, `g_object_get()` is used to retrieve the current value stored by the button's `relief` property. This function accepts a `NULL`-terminated list of properties and variables to store the returned value.

```
g_object_get (button, "relief", &value, NULL);
```

Each object can have many properties, so a full list will not be found in this book. For more information on properties available for a specific widget, you should reference the API documentation.

Setting Widget Properties

Setting a new value for a property is easily done with `g_object_set()`. In this example, the `relief` property of the button was set to `GTK_RELIEF_NORMAL`:

```
g_object_set (button, "relief", GTK_RELIEF_NORMAL, NULL);
```

Functions are provided to set and retrieve many of the properties of each widget. However, not every property has that option. These functions will become extremely important when you learn about the `GtkTreeView` widget in Chapter 8, because many objects used in that chapter do not provide get or set functions for any properties.

It is also possible to monitor a specific property with GObject's `notify` signal. You can monitor a property by connecting to the `notify::property-name` signal. The example in Listing 2-4 calls `property_changed()` when the `relief` property is changed.

Listing 2-4. *Using the Notify Property*

```
g_signal_connect (G_OBJECT (button), "notify::relief",
                  G_CALLBACK (property_changed), NULL);

...

static void
property_changed (GObject *button,
                  GParamSpec *property,
                  gpointer data)
{
  /* Handle the property change ... */
}
```

■ **Caution** While it is acceptable to use either a dash or an underscore when typing signal names, you must always use dashes when using the notify signal. For example, if you need to monitor GtkWidget's can-focus property, `notify::can_focus` is not acceptable! Remember that notify is the signal name, and can-focus is the name of the widget property.

The callback function receives a new type of object called `GParamSpec`, which holds information about the property that was changed. For now, all you need to know is that you can retrieve the name of the property that was changed with `property->name`. You will learn more about the `GParamSpec` structure in Chapter 11 when you learn how to add properties to your own custom widgets.

In addition to the property system, every GObject has a table that associates a list of strings to a list of pointers. This allows you to add data to an object that can easily be accessed, which is useful when you need to pass additional data to a signal handler. To add a new data field to an object, all you have to do is call g_object_set_data(). This function accepts a unique string that will be used to point to data. If an association already exists with the same key name, the new data will replace the old.

```
void g_object_set_data (GObject *object,
                        const gchar *key,
                        gpointer data);
```

When you need to access the data, you can call g_object_get_data(), which returns the pointer associated with key. You should use this method of passing data instead of trying to pass arbitrary pieces of data with g_signal_connect().

Test Your Understanding

In Chapter 2, you have learned about the window and label widgets. It is time to put that knowledge into practice. In the following two exercises, you will employ your knowledge of the structure of GTK+ applications, signals, and the GObject property system.

Exercise 2-1. Using Events and Properties

This exercise will expand on the first two examples in this chapter by creating a GtkWindow that has the ability to destroy itself. You should set your first name as the title of the window. A selectable GtkLabel with your last name as its default text string should be added as the child of the window.

Other properties of this window are that it should not be resizable and the minimum size should be 300 pixels by 100 pixels. Functions to perform these tasks can be found in this chapter.

Next, by looking at the API documentation, connect the key-press-event signal to the window. In the skey-press-event callback function, switch the window title and the label text. For example, the first time the callback function is called, the window title should be set to your last name and the label text to your first.

You may also find this function useful:

```
gint g_ascii_strcasecmp (const gchar *str1, const gchar *str2);
```

When the two strings in g_ascii_strcasesmp() are the same, 0 is returned. If str1 is less than str2, a negative number is returned. Otherwise, a positive number is returned.

Once you have completed Exercise 2-1, you can find a description of the solution in Appendix F, or the solution's complete source code is downloadable at www.gtkbook.com.

Exercise 2-2. GObject Property System

In this exercise, you will expand on Exercise 2-1, but the title, height, and width of the window should be set by using the functions provided by GObject. Also, within the callback function, all operations involving the window title and label text should be performed with the functions provided by GObject. Additionally, you should monitor the window's title with the notify signal. When the title is changed, you should notify the user in the terminal output.

Hint: You can use a function provided by GLib, `g_message()`, to output a message to the terminal. This function follows the same formatting supported by `printf()`.

Once you have completed both of these exercises, you are ready to move on to the next chapter, which covers container widgets. These widgets allow your main window to contain more than just a single widget, which was the case in all of the examples in this chapter.

However, before you continue, you should know about `www.gtkbook.com`, which can be used to supplement the content of *Foundations of GTK+ Development*. This web site is filled with downloads, links to further GTK+ information, C refresher tutorials, API documentation, and more. You can use it as you go through this book to aid in your quest to learn GTK+.

Summary

In this chapter, you learned about the most basic GTK+ widget and applications. The first application was a simple "Hello World" example that showed the fundamental calls required by all GTK+ applications. These include the following:

- Initialize GTK+ with `gtk_init()`.
- Create your top-level `GtkWindow`.
- Show the `GtkWindow`.
- Move into the main loop with `gtk_main()`.

In the second example, you learned the purpose of signals, events, and callback functions within GTK+ applications. The `GtkContainer` structure was introduced as it relates to `GtkWindow`. You also saw the purpose of the widget hierarchy system implemented by the GObject library.

You then saw useful functions that relate to `GtkWidget`, `GtkWindow`, and `GtkLabel`. Many of these will be used throughout the book. In fact, both of the exercises required that you put a few of them into practice.

The last example introduced you to the `GtkButton` widget. `GtkButton` is a type of container that makes its child widget a clickable button. It can be used to display labels, mnemonics, or arbitrary widgets. Buttons will be covered in further detail in Chapter 4.

In the next chapter, you will learn more about the `GtkContainer` structure and how it relates to the vast array of container widgets at your disposal.