Signals, Events, Objects, and Types

This chapter begins with a discussion of signals and signal handling. The topic of signals is an important one. A typical Gtk+ application will perform all of its useful work within the context of a signal handler, as we will see time and again throughout the course of this book. In addition to signals, we’ll also cover Gtk+ events and objects, defining what they are and how they can be used and manipulated by an application. The chapter will close with a short discussion on Gtk+ types.

Signals

Signals provide the mechanism by which a widget communicates useful information to a client about some change in its state.

In Chapter 2, “Hello Gtk+!,” we developed and discussed three “Hello World!” applications. Two of these were console-based, using standard I/O to display output to the screen and retrieve input from the user. We saw that flow of control in these programs was synchronous, meaning that statements were executed one after another, and when I/O was needed, the program would block in a routine such as fgets() until the input data needed by the application was entered by the user. The third of our “Hello World!” applications was also our first Gtk+ application. Two signal functions or callbacks were implemented in hellogtk+. Neither of these functions was called directly by hellogtk+. Instead, one of these functions was invoked by Gtk+ in response to the user pressing the “Print” button. The other was invoked in response to the application being closed (via a window manager control, for example).

An Example: GtkButton Signals

To better understand the functionality provided by signals, let’s take a closer look at how signals are used by the GtkButton widget class.

GtkButton, the widget class that implements push button in Gtk+, generates a signal whenever one of the following events is detected:

- The pointer enters the rectangular region occupied by the button.
- The pointer leaves the rectangular region occupied by the button.
- The pointer is positioned over the button, and a mouse button is pressed.
The pointer is positioned over the button, and a mouse button is released.

The user clicks the button (a combination of pressing and releasing a mouse button while the pointer is positioned over the button).

Each widget class implements signals needed to make that widget class useful to application designers. In addition, widget classes inherit signals from classes higher in the Gtk+ class hierarchy. For example, a signal is emitted when a push button is destroyed. This signal is actually generated by a superclass of GtButton. The signals implemented by a superclass represent functionality needed by many classes of widget. It is better to implement this functionality once in a superclass, allowing child classes to inherit the behavior, than it is to replicate the same functionality in each of the widget classes that need it.

Gtk+ does not force clients to use any of the signals that a class implements. However, in order to be useful, most applications will need to make use of at least one of the signals provided so that the widget can communicate useful information back to the client.

Handling Signals

Handling a signal in a Gtk+ application involves two steps. First, the application must implement a signal handler; this is the function that will be invoked by the widget when the signal triggers. Second, the client must register the signal handler with the widget. Registering a signal handler with a widget occurs after the application has created or instantiated the widget, by calling the Gtk+ routine gtk_signal_connect(). The prototype for this function is:

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

/* Signal handler */

void
print_signal_handler (GtkWidget *widget, gpointer func_data)
{
    printf ( "Signal triggered!\n" );
}

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

    /* Create a widget */
    button = gtk_button_new_with_label ( "Print" );

    /* Connect signal */
    gtk_signal_connect ( GTK_OBJECT ( button ),
                        "clicked",
                        GTK_SIGNAL_FUNC ( print_signal_handler ),
                        NULL );

    /* Run the main loop */
    gtk_main ();

    return 0;
}
```

Handling Signals

Handling a signal in a Gtk+ application involves two steps. First, the application must implement a signal handler; this is the function that will be invoked by the widget when the signal triggers. Second, the client must register the signal handler with the widget. Registering a signal handler with a widget occurs after the application has created or instantiated the widget, by calling the Gtk+ routine gtk_signal_connect(). The prototype for this function is:

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

/* Signal handler */

void
print_signal_handler (GtkWidget *widget, gpointer func_data)
{
    printf ( "Signal triggered!\n" );
}

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

    /* Create a widget */
    button = gtk_button_new_with_label ( "Print" );

    /* Connect signal */
    gtk_signal_connect ( GTK_OBJECT ( button ),
                        "clicked",
                        GTK_SIGNAL_FUNC ( print_signal_handler ),
                        NULL );

    /* Run the main loop */
    gtk_main ();

    return 0;
}
```

Handling Signals

Handling a signal in a Gtk+ application involves two steps. First, the application must implement a signal handler; this is the function that will be invoked by the widget when the signal triggers. Second, the client must register the signal handler with the widget. Registering a signal handler with a widget occurs after the application has created or instantiated the widget, by calling the Gtk+ routine gtk_signal_connect(). The prototype for this function is:

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

/* Signal handler */

void
print_signal_handler (GtkWidget *widget, gpointer func_data)
{
    printf ( "Signal triggered!\n" );
}

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

    /* Create a widget */
    button = gtk_button_new_with_label ( "Print" );

    /* Connect signal */
    gtk_signal_connect ( GTK_OBJECT ( button ),
                        "clicked",
                        GTK_SIGNAL_FUNC ( print_signal_handler ),
                        NULL );

    /* Run the main loop */
    gtk_main ();

    return 0;
}
```
The second argument to `gtk_signal_connect()` is the name of the signal we would like to associate with the signal handler. For those signals implemented by `GtkButton`, this will be one of the following strings:

- **enter**  The pointer entered the rectangular region occupied by the button.
- **leave**   The pointer left the rectangular region occupied by the button.
- **pressed** The pointer was positioned over the button, and a mouse button was pressed.
- **released** The pointer was positioned over the button, and a mouse button was released.
- **clicked** The user clicked the button (a combination of pressing and releasing the mouse button while the pointer was positioned over the button).

The third argument to `gtk_signal_connect()` is a pointer to the function that should be invoked by the widget when the signal specified by argument two, name, is triggered. The final argument to `gtk_signal_connect()` is a pointer to private data that will be passed to the signal handler by the widget when the signal handler is invoked.

Unfortunately, signal functions do not adhere to a single function prototype. The arguments passed to a signal handler will vary based on the widget generating the signal. The general form of a `Gtk+` signal handler is as follows:

```c
void
callback_func( GtkWidget *widget, gpointer callback_data );
```

I will describe the function prototypes for signal handlers in later chapters, along with the widgets that generate them. However, at this point, I can say a couple of things about callback function arguments that hold true regardless of the widget class involved:

- The first argument of the signal handler will always be a pointer to the widget that generated the signal.
- The callback_data argument will always be the last argument passed to the signal handler.
- Any arguments that are specific to the widget or to the signal will occur between the first and last arguments of the signal handler.

The final argument passed to the callback function, callback_data, contains a pointer to data that is private to the application and has no meaning whatsoever to the widget. The use of private callback data is a practice that `Gtk+` borrowed from `Xt/Motif`, and it has powerful implications for application design.

### Client Callback Data Example

To illustrate the use of client data, let’s design a simple application. Here’s the code:

**Listing 3.1** Passing Client Data to a Callback

```c
001  #include <stdio.h>
002  #include <time.h>
003  #include <gtk/gtk.h>
004
005  void
006  Update (GtkWidget *widget, char *timestr)
```
Chapter 3 • Signals, Events, Objects, and Types

```c
007  {
008      time_t timeval;
009
010      timeval = time( NULL );
011      strcpy( timestr, ctime( &timeval ) );
012  }
013
014  void
015  PrintAndExit (GtkWidget *widget, char timestr[][26])
016  {
017      int    i;
018
019      for ( i = 0; i < 4; i++ )
020          printf( "timestr[ %d ] is %s", i, timestr[ i ] );
021      gtk_main_quit ();
022  }
023
024  int
025  main( int argc, char *argv[] )
026  {
027      GtkWidget *window, *box, *button;
028
029      static char times[ 4 ][ 26 ] =
030          { "Unset
", "Unset
", "Unset
", "Unset
" };
031
032      gtk_set_locale ();
033
034      gtk_init (&argc, &argv);
035
036      window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
037
038      gtk_signal_connect (GTK_OBJECT(window), "destroy",
039          GTK_SIGNAL_FUNC(PrintAndExit), times);
040
041      gtk_window_set_title (GTK_WINDOW (window), "Signals 1");
042      gtk_container_border_width (GTK_CONTAINER (window), 0);
043
044      box = gtk_vbox_new (FALSE, 0);
045      gtk_container_add (GTK_CONTAINER (window), box);
046
047      button = gtk_button_new_with_label ("Update 0");
048      gtk_signal_connect (GTK_OBJECT (button), "clicked",
049          GTK_SIGNAL_FUNC(Update), &times[0]);
050      gtk_box_pack_start (GTK_BOX (box), button, TRUE, TRUE, 0);
051
052      button = gtk_button_new_with_label ("Update 1");
053      gtk_signal_connect (GTK_OBJECT (button), "clicked",
054          GTK_SIGNAL_FUNC(Update), &times[1]);
055      gtk_box_pack_start (GTK_BOX (box), button, TRUE, TRUE, 0);
056
057      button = gtk_button_new_with_label ("Update 2");
058      gtk_signal_connect (GTK_OBJECT (button), "clicked",
```
The purpose of this example is to illustrate how private data can be passed to a callback routine. On lines 029 and 030, we declare an array of four 26-character strings, 26 characters being what is needed to hold the value returned by the ctime(3) function. These strings are initialized to the value "Unset
" so that the callback routine that will be invoked when we exit, PrintAndExit(), has something sensible to print should the user not change one or more of the string’s values. On lines 048, 053, 058, and 083, we register the signal function Update() with the Gtk-Button that was created a line or two earlier, using gtk_signal_connect(). Each of these calls to gtk_signal_connect() is passed a different func_data argument; the first call is passed the address of the first cell in the array of times, the second call is passed the address of the second cell of times, and so forth.

Whenever the user clicks one of the buttons labeled “Update 0”, “Update 1”, “Update 2”, or “Update 3”, Update() will be invoked. The timestr argument will be set by Gtk+ to the private data assigned when the callback or signal function was registered.

This may be a silly example, but it illustrates a very important technique. Note that we have no logic inside of Update() that concerns itself with the button pressed by the user; we simply don’t need to know this. All we need to know is that the callback function is being passed a pointer to a string presumed to be big enough to hold the ctime(3) result that is going to be stuffed into it.

It is easy to extend this example to a real-life application such as a word processor or to any application that allows a user to manipulate more than one document at a time, such as a spreadsheet or a photo manipulation program like xv or GIMP. Whenever a callback is designed to manipulate data of some kind, try to make that data available to the callback function via the func_data argument. This will enable reuse of callbacks and minimize the need for maintaining global data.

Events

Events are similar to signals in that they are a method by which Gtk+ can tell an application that something has happened. Events and signals differ mainly in what it is they provide notification of. Signals make applications aware of somewhat abstract, high-level changes, such as GUI (not mouse) button presses, toggle button state changes, or the selection of a
row in a list widget. Events mainly provide a way for Gtk+ to pass along to the client any X11 events that have been received over the X server connection in which the client has expressed an interest.

Events and signals share the same Gtk+ APIs. To register a callback function for an event, use gtk_signal_connect(). The APIs involved will be discussed later in this chapter.

**Event Callback Function Prototypes**

The function prototype for event callbacks is slightly different than for signals:

```c
int callback_func( GtkWidget *widget, GdkEvent *event, gpointer callback_data );
```

*widget* is the Gtk+ widget to which the event pertains, *event* is a GDK data structure that contains information about the event, and *callback_data* is the application-specific data that was registered with the handler by the client at the time that gtk_signal_connect() was called.

Most event callbacks adhere to the preceding prototype, but there are variations. In the following section where individual events are described, I will provide the callback function prototype that is most appropriate for each event.

Table 3.1 defines each of the events supported by Gtk+ 1.2. Note that the names all start with GDK_ because the events all originate from within GDK code.

**Table 3.1** GDK Events

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GDK NOTHING</td>
<td>No event. (You should never see this value.)</td>
</tr>
<tr>
<td>GDK DELETE</td>
<td>This is a client message, likely from a window manager, requesting that a</td>
</tr>
<tr>
<td></td>
<td>window be deleted.</td>
</tr>
<tr>
<td>GDK DESTROY</td>
<td>Maps to the X11 DestroyNotify event. A window has been destroyed.</td>
</tr>
<tr>
<td>GDK EXPOSE</td>
<td>Maps to an X11 Expose or GraphicsExpose event. If Expose, some portion of</td>
</tr>
<tr>
<td></td>
<td>a window was exposed and is in need of a redraw. If GraphicsExpose, then X</td>
</tr>
<tr>
<td></td>
<td>protocol was CopyArea or CopyPlane, and the destination area could not</td>
</tr>
<tr>
<td></td>
<td>be completely drawn because some portion of the source was obscured or</td>
</tr>
<tr>
<td></td>
<td>unmapped.</td>
</tr>
<tr>
<td>GDK NO EXPOSE</td>
<td>Maps to an X11 NoExpose event. X protocol was CopyArea or CopyPlane, and</td>
</tr>
<tr>
<td></td>
<td>the destination area was completely drawn because all of source was</td>
</tr>
<tr>
<td></td>
<td>available.</td>
</tr>
</tbody>
</table>

Table 3.1  GDK Events (Continued)

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GDK_MOTION_NOTIFY</td>
<td>Maps to an X11 MotionNotify event. The pointer (controlled by mouse, keyboard, touchpad, or client via X protocol) was moved.</td>
</tr>
<tr>
<td>GDK_BUTTON_PRESS</td>
<td>Maps to an X11 ButtonPress event. A mouse button was pressed.</td>
</tr>
<tr>
<td>GDK_2BUTTONPRESS</td>
<td>GDK detected a mouse double-click while processing an X11 ButtonPress event.</td>
</tr>
<tr>
<td>GDK_3BUTTONPRESS</td>
<td>GDK detected a mouse triple-click while processing an X11 ButtonPress event.</td>
</tr>
<tr>
<td>GDK_BUTTON_RELEASE</td>
<td>Maps to an X11 ButtonRelease event.</td>
</tr>
<tr>
<td>GDK_KEY_PRESS</td>
<td>Maps to an X11 KeyPress event. Reports all keys, including Shift and Ctrl.</td>
</tr>
<tr>
<td>GDK_KEY_RELEASE</td>
<td>Maps to an X11 KeyRelease event. Reports all keys, including Shift and Ctrl.</td>
</tr>
<tr>
<td>GDK_ENTER_NOTIFY</td>
<td>Maps to an X11 EnterNotify event. The pointer has entered a window.</td>
</tr>
<tr>
<td>GDK_LEAVE_NOTIFY</td>
<td>Maps to an X11 LeaveNotify event. The pointer has left a window.</td>
</tr>
<tr>
<td>GDK_FOCUS_CHANGE</td>
<td>Maps to an X11 FocusIn or FocusOut event. A field in the event structure is used to indicate which a window has obtained or lost server focus.</td>
</tr>
<tr>
<td>GDK_CONFIGURE</td>
<td>Maps to an X11 ConfigureNotify event. Some change in the size, location, border, or stacking order of a window is being announced.</td>
</tr>
<tr>
<td>GDK_MAP</td>
<td>Maps to an X11 MapNotify event. A window’s state has changed to mapped.</td>
</tr>
<tr>
<td>GDK_UNMAP</td>
<td>Maps to an X11 UnmapNotify event. A window’s state has changed to unmapped.</td>
</tr>
<tr>
<td>GDK_PROPERTY_NOTIFY</td>
<td>Maps to an X11 PropertyNotify event. A property on a window has been changed or deleted.</td>
</tr>
<tr>
<td>GDK_SELECTION_CLEAR</td>
<td>Maps to an X11 SelectionClear event. See the following discussion.</td>
</tr>
<tr>
<td>GDK_SELECTION_REQUEST</td>
<td>Maps to an X11 SelectionRequest event. See the following discussion.</td>
</tr>
</tbody>
</table>
Note that there are X11 events that are not passed on to your Gtk+ application. For example, MappingNotify events are responded to by GDK by calling XRefreshKeyboardMapping(), which is the standard way for Xlib clients to handle the reception of this event. Unless you take extraordinary means to look for it, your application will never see a MappingNotify event.

In X11, clients must tell the server which events the client is interested in receiving by soliciting the events. If an event is not solicited by a client, it will not be sent. There are a few exceptions, however: MappingNotify, ClientMessage, and the Selection* events are all nonmaskable and will always be sent to the client.

In Gtk+/GDK, clients must also solicit the events in which they have interest. This is done on a per-widget basis, using a technique that is very similar to calling XSelectInput() from an Xlib program. In Gtk+, the routine to call is gtk_widget_set_events(). Here is its prototype:

```c
void
gtk_widget_set_events (GtkWidget *widget, gint events)
```

The argument events is a bitmask used to indicate the types of events the client would like to receive notification of from Gtk+, and widget is the handle of the Gtk+ widget to which the event notification pertains. The X server will only send events specified in the events mask that belong to the window defined by the widget. This implies that widgets that

---

**Table 3.1** GDK Events (Continued)

<table>
<thead>
<tr>
<th>Event Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GDK_SELECTION_NOTIFY</td>
<td>Maps to an X11 SelectionNotify event. See the following discussion.</td>
</tr>
<tr>
<td>GDK_PROXIMITY_IN</td>
<td>Used by X Input Extension-aware programs that draw their own cursors.</td>
</tr>
<tr>
<td>GDK_PROXIMITY_OUT</td>
<td>Used by X Input Extension-aware programs that draw their own cursors.</td>
</tr>
<tr>
<td>GDK_DRAG_ENTER</td>
<td>Motif Drag and Drop top-level enter.</td>
</tr>
<tr>
<td>GDK_DRAG_LEAVE</td>
<td>Motif Drag and Drop top-level leave.</td>
</tr>
<tr>
<td>GDK_DRAG_MOTION</td>
<td>Motif Drag and Drop motion.</td>
</tr>
<tr>
<td>GDK_DRAG_STATUS</td>
<td>Motif Drag and Drop status message.</td>
</tr>
<tr>
<td>GDK_DROP_START</td>
<td>Motif Drag and Drop start.</td>
</tr>
<tr>
<td>GDK_DROP_FINISHED</td>
<td>Motif Drag and Drop finished.</td>
</tr>
<tr>
<td>GDK_CLIENT_EVENT</td>
<td>Maps to an X11 ClientMessage event which is a message or event that was sent by a client.</td>
</tr>
<tr>
<td>GDK_VISIBILITY_NOTIFY</td>
<td>Maps to an X11 VisibilityNotify event. A window has become fully or partially obscured, or it has become completely unobscured.</td>
</tr>
</tbody>
</table>
Events

Events do not create a window cannot receive events (we’ll return to this issue later in this book). Events that are not solicited for a window are not transmitted to the client by the X server.

Unless you plan to handle a specific event in your application, there is really no need for you to call this routine. This does not mean that events will not be solicited for the widget; it is very likely that one or more events will be solicited by the widget implementation.

The events bitmask can be constructed by OR’ing together one or more of the constants defined by GDK (see Table 3.2).

### Table 3.2  GDK Event Masks

<table>
<thead>
<tr>
<th>Mask</th>
<th>Event(s) Solicited</th>
</tr>
</thead>
<tbody>
<tr>
<td>GDK_EXPOSURE_MASK</td>
<td>Expose event (but not GraphicsExpose or NoExpose)</td>
</tr>
<tr>
<td>GDK_POINTER_MOTION_MASK</td>
<td>Fewer pointer motion events</td>
</tr>
<tr>
<td>GDK_POINTER_MOTION_HINT_MASK</td>
<td>All pointer motion events</td>
</tr>
<tr>
<td>GDK_BUTTON_MOTION_MASK</td>
<td>Pointer motion while any mouse button down</td>
</tr>
<tr>
<td>GDK_BUTTON1_MOTION_MASK</td>
<td>Pointer motion while mouse button 1 down</td>
</tr>
<tr>
<td>GDK_BUTTON2_MOTION_MASK</td>
<td>Pointer motion while mouse button 2 down</td>
</tr>
<tr>
<td>GDK_BUTTON3_MOTION_MASK</td>
<td>Pointer motion while mouse button 3 down</td>
</tr>
<tr>
<td>GDK_BUTTON_PRESS_MASK</td>
<td>Pointer button down events</td>
</tr>
<tr>
<td>GDK_BUTTON_RELEASE_MASK</td>
<td>Pointer button up events</td>
</tr>
<tr>
<td>GDK_KEY_PRESS_MASK</td>
<td>Key down events</td>
</tr>
<tr>
<td>GDK_KEY_RELEASE_MASK</td>
<td>Key up events</td>
</tr>
<tr>
<td>GDK_ENTER_NOTIFY_MASK</td>
<td>Pointer window entry events</td>
</tr>
<tr>
<td>GDK_LEAVE_NOTIFY_MASK</td>
<td>Pointer window leave events</td>
</tr>
<tr>
<td>GDK_FOCUS_CHANGE_MASK</td>
<td>Any change in keyboard focus</td>
</tr>
<tr>
<td>GDK_STRUCTURE_MASK</td>
<td>Any change in window configuration</td>
</tr>
<tr>
<td>GDKPROPERTY_CHANGE_MASK</td>
<td>Any change in property</td>
</tr>
<tr>
<td>GDK_VISIBILITY_NOTIFY_MASK</td>
<td>Any change in visibility</td>
</tr>
<tr>
<td>GDK_PROXIMITY_IN_MASK</td>
<td>Used by X Input Extension programs</td>
</tr>
<tr>
<td>GDK_PROXIMITY_OUT_MASK</td>
<td>Used by X Input Extension programs</td>
</tr>
<tr>
<td>GDK_SUBSTRUCTURE_MASK</td>
<td>Notify about reconfiguration of children</td>
</tr>
<tr>
<td>GDK_ALL_EVENTS_MASK</td>
<td>All of the above</td>
</tr>
</tbody>
</table>
The masks in the preceding table are not one-to-one with the events listed in Table 3.1. Some of the masks will lead to the reception of more than one event, and your callback function may have to check to see which event was received, depending on the application. Table 3.3 should clarify the mapping that exists between masks and events.

### Table 3.3  Event Mask-to-Event Mappings

<table>
<thead>
<tr>
<th>Mask</th>
<th>Event(s) Solicited</th>
</tr>
</thead>
<tbody>
<tr>
<td>GDK_EXPOSURE_MASK</td>
<td>GDK_EXPOSE</td>
</tr>
<tr>
<td>GDK_POINTER_MOTION_MASK</td>
<td>GDK_MOTION_NOTIFY</td>
</tr>
<tr>
<td>GDK_POINTER_MOTION_HINT_MASK</td>
<td>GDK_MOTION_NOTIFY</td>
</tr>
<tr>
<td>GDK_BUTTON_MOTION_MASK</td>
<td>GDK_MOTION_NOTIFY</td>
</tr>
<tr>
<td>GDK_BUTTON1_MOTION_MASK</td>
<td>GDK_MOTION_NOTIFY</td>
</tr>
<tr>
<td>GDK_BUTTON2_MOTION_MASK</td>
<td>GDK_MOTION_NOTIFY</td>
</tr>
<tr>
<td>GDK_BUTTON3_MOTION_MASK</td>
<td>GDK_MOTION_NOTIFY</td>
</tr>
<tr>
<td>GDK_BUTTON_PRESS_MASK</td>
<td>GDK_BUTTON_PRESS GDK_2BUTTON_PRESS GDK_3BUTTON_PRESS</td>
</tr>
<tr>
<td>GDK_BUTTON_RELEASE_MASK</td>
<td>GDK_BUTTON_RELEASE</td>
</tr>
<tr>
<td>GDK_KEY_PRESS_MASK</td>
<td>GDK_KEY_PRESS</td>
</tr>
<tr>
<td>GDK_KEY_RELEASE_MASK</td>
<td>GDK_KEY_RELEASE</td>
</tr>
<tr>
<td>GDK_ENTER_NOTIFY_MASK</td>
<td>GDK_ENTER_NOTIFY</td>
</tr>
<tr>
<td>GDK_BUTTON_RELEASE_MASK</td>
<td>GDK_BUTTON_RELEASE</td>
</tr>
<tr>
<td>GDK_LEAVE_NOTIFY_MASK</td>
<td>GDK_LEAVE_NOTIFY</td>
</tr>
<tr>
<td>GDK_FOCUS_CHANGE_MASK</td>
<td>GDK_FOCUS_CHANGE</td>
</tr>
<tr>
<td>GDK_STRUCTURE_MASK</td>
<td>GDK_DESTROY GDK_CONFIGURE GDK_MAP GDK_UNMAP</td>
</tr>
<tr>
<td>GDKPROPERTY_CHANGE_MASK</td>
<td>GDKPROPERTY_NOTIFY</td>
</tr>
<tr>
<td>GDK_VISIBILITY_NOTIFY_MASK</td>
<td>GDK_VISIBILITY_NOTIFY</td>
</tr>
<tr>
<td>GDK_PROXIMITY_IN_MASK</td>
<td>GDK_PROXIMITY_IN</td>
</tr>
<tr>
<td>GDK_PROXIMITY_OUT_MASK</td>
<td>GDK_PROXIMITY_OUT</td>
</tr>
</tbody>
</table>
What happens if you specify a mask that does not contain bits set by the widget? For example, the GtkButton widget selects GDK_BUTTONPRESS_MASK for its window when the buttons’ window is created. Let’s say your client calls gtk_set_widget_events(), and the mask you supply does not have the GDK_BUTTONPRESS_MASK bit set, as in the following code:

```c
button = gtk_button_new_with_label("Print");
gtk_signal_connect_object(GTK_OBJECT(button), "clicked",
                         GTK_SIGNAL_FUNC(PrintString), GTK_OBJECT(window));
gtk_widget_set_events(button, GDK_POINTER_MOTION_MASK);
gtk_signal_connect(GTK_OBJECT(button), "motion_notify_event",
                         GTK_SIGNAL_FUNC(MotionNotifyCallback), NULL);
```

In this case, button press events will be sent to the client and processed by the GtkButton widget, in addition to MotionNotify events that will be handled by the client in MotionNotifyCallback().

What about selecting an event that has already been selected by a widget? For example:

```c
button = gtk_button_new_with_label("Print");
gtk_signal_connect_object(GTK_OBJECT(button), "clicked",
                         GTK_SIGNAL_FUNC(PrintString), GTK_OBJECT(window));
gtk_widget_set_events(button, GDK_BUTTONPRESS_MASK);
gtk_signal_connect(GTK_OBJECT(button), "button_press_event",
                         GTK_SIGNAL_FUNC(ButtonPressCallback), NULL);
```

This too will not affect the widget. When a button press occurs, Gtk+ will first call ButtonPressCallback() and then call PrintString(). Note that we really did not need to call gtk_widget_set_events() to select GDK_BUTTONPRESS_MASK for the GtkButton widget because that event was already selected by the widget itself, but it didn’t hurt.

## Event Types

Earlier we introduced the function prototype for the callback function invoked by Gtk+ upon reception of a signal that the client has solicited and for which a signal function has been registered. The prototype, once again, is as follows:
Chapter 3 • Signals, Events, Objects, and Types

```c
int
callback_func( GtkWidget *widget, GdkEvent *event,
gpointer callback_data );
```

GdkEvent is actually a C union of structures, one structure for each signal type listed in Table 3.1:

```c
union _GdkEvent
{
    GdkEventType              type;
    GdkEventAny               any;
    GdkEventExpose            expose;
    GdkEventNoExpose          no_expose;
    GdkEventVisibility        visibility;
    GdkEventMotion            motion;
    GdkEventButton            button;
    GdkEventKey               key;
    GdkEventCrossing          crossing;
    GdkEventFocus             focus_change;
    GdkEventConfigure         configure;
    GdkEventProperty          property;
    GdkEventSelection         selection;
    GdkEventProximity         proximity;
    GdkEventClient            client;
    GdkEventDND              dnd;
};
```

The following describes each of the structures encapsulated within the GdkEvent union (with the only exceptions being GdkEventProximity, which is not covered, and GdkEventDND, which is an internal event type used in the implementation of Drag and Drop, also not discussed in this book). Each of the preceding names is a typedef for a struct that has the same name but is prefixed with '_'. For example:

```c
typedef struct _GdkEventExpose GdkEventExpose;
```

In each of the following structures, as well as in the preceding GdkEvent, GdkEventType is an enum that defines the events in Table 3.1. Thus, in a callback function that is supposed to process LeaveNotify events, the event type can be verified using code similar to the following:

```c
void
LeaveFunc( GtkWidget *widget, GdkEvent *event, gpointer callback_data )
{
    if (event==(GdkEvent *)NULL || event->type!=GDK_LEAVE_NOTIFY) {
        ErrorFunction( "LeaveFunc: NULL event or wrong type\n" );
        return;           /* bogus event */
    }
    /* event is good */
    ...
}
In the preceding routine, we leave the signal function if the event pointer is NULL or if the type of the event is not GDK_LEAVE_NOTIFY.

**GdkEventExpose**

```c
struct _GdkEventExpose
{
    GdkEventType type;    /* GDK_EXPOSE */
    GdkWindow *window;
    gint8 send_event;
    GdkRectangle area;
    gint count;           /* If non-zero, how many more events follow */
};
```

**Event Name String**
expose_event

**Callback Function Prototype**

```c
gint
func(GtkWidget *widget, GdkEventExpose *event, gpointer arg);
```

**Description**

Expose events are identified by a type field set to GDK_EXPOSE. Window identifies the window that needs repainting, and area defines the region that this expose event describes. If more than one region in a window becomes exposed, multiple expose events will be sent by the X server. The number of events pending for the window is identified by count. If your code ignores the area field and redraws the entire window in the expose signal function, then your code should wait until it receives an expose event with a count field equal to zero.

**GdkEventNoExpose**

```c
struct _GdkEventNoExpose
{
    GdkEventType type;                /* GDK_NO_EXPOSE */
    GdkWindow *window;
    gint8 send_event;
};
```

**Event Name String**

no_expose_event

**Callback Function Prototype**

```c
gint
func(GtkWidget *widget, GdkEventAny *event, gpointer arg);
```

**Description**

NoExpose events are received if CopyArea or CopyPlane X protocol is performed successfully. This will happen only if all values in the source image were able to be copied by the
X server, with no portions of the source window obscured, and if the graphics_exposures flag in the X GC used in the CopyArea or CopyPlane request was set to True.

XCopyArea is invoked by both gdk_draw_pixmap() and gdk_window_copy_area().

**GdkEventVisibility**

```c
struct _GdkEventVisibility
{
    GdkEventType type;        /* GDK_VISIBILITY_NOTIFY */
    GdkWindow *window;
    gint8 send_event;
    GdkVisibilityState state;
};
```

**Event Name String**

visibility_notify_event

**Callback Function Prototype**

```c
gint
func(GtkWidget *widget, GdkEventVisibility *event, gpointer arg);
```

**Description**

Visibility events are sent when the visibility of a window has changed. The state field of the event describes the nature of the change and can be one of the following values in Table 3.4.

**Table 3.4  Visibility Event States**

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>GDK_VISIBILITY_UNOBSCURED</td>
<td>Window was partially obscured, fully obscured, or not viewable, and became viewable and completely unobscured.</td>
</tr>
<tr>
<td>GDK_VISIBILITY_PARTIAL</td>
<td>Window was viewable and completely unobscured, or not viewable, and became viewable and partially unobscured.</td>
</tr>
<tr>
<td>GDK_VISIBILITY_FULLY_OBSCURED</td>
<td>Window was viewable and completely unobscured, or viewable and partially unobscured, or not viewable, and became viewable and fully unobscured.</td>
</tr>
</tbody>
</table>

**GdkEventMotion**

```c
struct _GdkEventMotion
{
    GdkEventType type;        /* GDK_MOTION_NOTIFY */
    GdkWindow *window;
    gint8 send_event;
};
```
Events

```c
struct Event {
    guint32 time;
    gdouble x;
    gdouble y;
    gdouble pressure;
    gdouble xtilt;
    gdouble ytilt;
    guint state;
    guint16 is_hint;
    GdkInputSource source;
    guint32 deviceid;
    gdouble x_root, y_root;
};
```

**Event Name String**

`motion_notify_event`

**Callback Function Prototype**

```c
gint
func(GtkWidget *widget, GdkEventMotion *event, gpointer arg);
```

**Description**

Motion notify events indicate that the pointer has moved from one location of the screen to another. The time field indicates the time of the event in server-relative time (milliseconds since the last server reset). If the window is on the same screen as the root (which is usually the case), then x and y are the pointer coordinates relative to the origin of the window; otherwise, they are both set to 0. x_root and y_root are the coordinates relative to the root window. Pressure is always set to the value 0.5, and xtilt and ytilt are always set to the value 0. Source is always set to GDK_SOURCE_MOUSE, and deviceid is always set to the value GDK_CORE_POINTER. Is_hint is set to 1 if the mask used to select the event was GDK_POINTER_MOTION_HINT_MASK; otherwise, it will be 0. If is_hint is 1e, then the current position information needs to be obtained by calling gdk_window_get_pointer(). State is used to specify the state of the mouse buttons and modifier keys just before the event. The values possible for state are constructed by OR’ing any of the following bits in Table 3.5.

**Table 3.5** Motion Event States

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>GDK_SHIFT_MASK</td>
<td>shift key is pressed.</td>
</tr>
<tr>
<td>GDK_LOCK_MASK</td>
<td>lock key is pressed.</td>
</tr>
<tr>
<td>GDK_CONTROL_MASK</td>
<td>control key is pressed.</td>
</tr>
<tr>
<td>GDK_MOD1_MASK</td>
<td>mod1 is pressed (typically Alt_L or Alt_R).</td>
</tr>
<tr>
<td>GDK_MOD2_MASK</td>
<td>mod2 is pressed (typically Num_Lock).</td>
</tr>
</tbody>
</table>
shift, lock, control, mod1 through mod5, and Button1 through Button5 are logical names in X11 and are subject to remapping by the user. The X11 user command for performing this remapping is xmodmap(1). The xmodmap(1) command can also be used to view the current logical name to keysym mapping, for example:

bash$ xmodmap -pm
xmodmap: up to 2 keys per modifier, (keycodes in parentheses):

<table>
<thead>
<tr>
<th>Shift_L (0x32), Shift_R (0x3e)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caps_Lock (0x42)</td>
</tr>
<tr>
<td>Control_L (0x25), Control_R (0x6d)</td>
</tr>
<tr>
<td>Alt_L (0x40), Alt_R (0x71)</td>
</tr>
<tr>
<td>Num_Lock (0x4d)</td>
</tr>
<tr>
<td>Scroll_Lock (0x4e)</td>
</tr>
</tbody>
</table>

GdkEventButton

struct _GdkEventButton
{
    GdkEventType type;    /* GDK_BUTTON_PRESS, GDK_2BUTTON_PRESS,
                            GDK_3BUTTON_PRESS, GDK_BUTTON_RELEASE */
    GdkWindow *window;
    guint8 send_event;
    guint32 time;
    gdouble x;
    gdouble y;
    gdouble pressure;
    gdouble xtilt;
    gdouble ytilt;
    guint state;
    guint button;

Table 3.5  Motion Event States (Continued)

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>GDK_MOD3_MASK</td>
<td>mod3 is pressed.</td>
</tr>
<tr>
<td>GDK_MOD4_MASK</td>
<td>mod4 is pressed.</td>
</tr>
<tr>
<td>GDK_MOD5_MASK</td>
<td>mod5 is pressed.</td>
</tr>
<tr>
<td>GDK_BUTTON1_MASK</td>
<td>Button1 is pressed (typically the left button).</td>
</tr>
<tr>
<td>GDK_BUTTON2_MASK</td>
<td>Button2 is pressed (typically the right button).</td>
</tr>
<tr>
<td>GDK_BUTTON3_MASK</td>
<td>Button3 is pressed (typically the middle button).</td>
</tr>
<tr>
<td>GDK_BUTTON4_MASK</td>
<td>Button4 is pressed.</td>
</tr>
<tr>
<td>GDK_BUTTON5_MASK</td>
<td>Button5 is pressed.</td>
</tr>
</tbody>
</table>

GdkEventButton

struct _GdkEventButton

GdkInputSource source;
guint32 deviceid;
gdouble x_root, y_root;  
};

Event Name Strings
button_press_event
button_release_event

Callback Function Prototype


gint
func(GtkWidget *widget, GdkEventButton *event, gpointer arg);

Description

Button events indicate that a mouse button press or release has occurred. The time field indicates the time of the event in server-relative time (milliseconds since server reset). If the window receiving the button press or release is on the same screen as the root (which is usually the case), then x and y are the pointer coordinates relative to the origin of window; otherwise, they are both set to zero. X_root and y_root are the coordinates of the press or release relative to the root window. Pressure is always set to the value 0.5, and xtilt and ytilt are always set to the value zero. Source is always GDK_SOURCE_MOUSE, and deviceid is always set to the value GDK_CORE_POINTER. State is used to specify the state of the mouse buttons and modifier keys just before the event. The values possible for state are the same values as those previously described for GdkEventMotion. Button indicates which button the event is for, with 1 indicating button 1, 2 indicating button 2, and so on.

GdkEventKey

struct _GdkEventKey
|
| GdkEventType type;               /* GDK_KEY_PRESS GDK_KEY_RELEASE */
| GdkWindow *window;
| guint8 send_event;
| guint32 time;
| guint state;
| guint keyval;
| guint length;
| gchar *string;
|
};

Event Name Strings

key_press_event
key_release_event

Callback Function Prototype


gint
func(GtkWidget *widget, GdkEventKey *event, gpointer arg);
**Description**

Key events indicate that a keyboard key press or key release has occurred. The time field indicates the time of the event in server-relative time (milliseconds since server reset). State is used to specify the state of the mouse buttons and modifier keys just before the event. The values possible for state are the same as those previously described for GdkEventMotion. Keyval indicates which key was pressed or released. Keyval is the keysym value that corresponds to the key pressed or released. Keysyms values are symbolic values that represent the keys on the keyboard. Keyboards generate hardware-dependent values that are mapped by Xlib to keysyms using a table provided by the X server. For example, the hardware code generated when the user presses the key labeled “A” is converted to the keysym value XK_A. It is this value (e.g., XK_A) that is stored inside the keyval field. String contains a string of ASCII characters that were obtained by GDK by calling the Xlib function XLookupString(). Usually, the string will be of length 1 and will correspond directly to the glyph or symbol displayed on the key that was pressed or released (e.g., for XK_A, the string will be “A”). However, clients can associate an arbitrarily long string with a key using XRebindKeysym(). The length of this string, which is limited to 16 characters by GDK, is stored in length, and string contains the value of the string (truncated if necessary to 16 characters) returned by XLookupString().

**GdkEventCrossing**

```c
struct _GdkEventCrossing
{
    GdkEventType type;  /* GDK_ENTER_NOTIFY GDK_LEAVE_NOTIFY */
    GdkWindow *window;
    guint8 send_event;
    GdkWindow *subwindow;
    guint32 time;
    gdouble x;
    gdouble y;
    gdouble x_root;
    gdouble y_root;
    GdkCrossingMode mode;
    GdkNotifyType detail;
    gboolean focus;
    guint state;
};
```

**Event Name Strings**

- enter_notify_event
- leave_notify_event

**Callback Function Prototype**

```c
gint
func(GtkWidget *widget, GdkEventCrossing *event, gpointer arg);
```
Description
Crossing events indicate that the mouse pointer has entered or left a window. The window into which the pointer has entered, or from which it has left, is indicated by window. If the event type is GDK_LEAVE_NOTIFY and the pointer began in a child window of window, then subwindow will be set to the GDK ID of the child window, or else it will be set to the value NULL. If the event type is GDK_ENTER_NOTIFY and the pointer ends up in a child window of window, then subwindow will be set to the GDK ID of the child window, or else it will be set to the value NULL. The time field indicates the time of the event in server-relative time (milliseconds since server reset). If window is on the same screen as the root (which is usually the case), then x and y specify the pointer coordinates relative to the origin of window; otherwise, they are both set to zero. X_root and y_root are the coordinates of the pointer relative to the root window. If the enter or leave event was caused by normal mouse movement or if it was caused by a pointer warp (that is, the client has explicitly moved the mouse), then mode will be set to GDK_CROSSING_NORMAL. Or, if the crossing event was caused by a pointer grab, mode will be set to GDK_CROSSING_GRAB. Finally, if the crossing event was caused by a pointer ungrab, then mode will be set to the value GDK_CROSSING_UNGRAB. If the receiving window is the focus window or is a descendant of the focus window (subwindow is not NULL), then focus will be set to TRUE; otherwise, it will be set to FALSE. State specifies the state of the mouse buttons and modifier keys just before the event. The values possible for state are the same as those previously described for GdkEventMotion.

The final field, detail, is a bit complicated to describe. Here we’ll simply list the GDK values that can be stored in this field and the X11 values to which they map. In practice, X11 client applications (and, by extension, Gtk+ applications) rarely, if ever, will make use of the data in this field (see Table 3.6).

Table 3.6  Event Crossing Event Detail Field

<table>
<thead>
<tr>
<th>X11 Value</th>
<th>GDK Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>NotifyInferior</td>
<td>GDK_NOTIFY_INFERIOR</td>
</tr>
<tr>
<td>NotifyAncestor</td>
<td>GDK_NOTIFY_ANCESTOR</td>
</tr>
<tr>
<td>NotifyVirtual</td>
<td>GDK_NOTIFY_VIRTUAL</td>
</tr>
<tr>
<td>NotifyNonlinear</td>
<td>GDK_NOTIFY_NONLINEAR</td>
</tr>
<tr>
<td>NotifyNonlinearVirtual</td>
<td>GDK_NOTIFY_NONLINEAR_VIRTUAL</td>
</tr>
</tbody>
</table>

GdkEventFocus

```c
struct _GdkEventFocus
{
    GdkEventType type;          /* GDK_FOCUS_CHANGE */
    GdkWindow *window;
    guint8 send_event;
    gint16 in;
};
```
**Event Name Strings**

- focus_in_event
- focus_out_event

**Callback Function Prototype**

```c
gint func(GtkWidget *widget, GdkEventFocus *event, gpointer arg);
```

**Description**

Focus events indicate a change in keyboard focus from one window to another. When keyboard focus changes, two events are sent. One event is sent for the window that had the keyboard focus just prior to the focus change. The other is sent for the window that just obtained the keyboard focus. The `in` field is used to define the type of focus change. If the X11 event type is `FocusIn`, then the window identified by `window` received focus, and `in` will be set to TRUE. Otherwise, the X11 event type was `FocusOut`, the window identified by `window` lost input focus, and `in` will be set to FALSE.

**GdkEventConfigure**

```c
struct _GdkEventConfigure {
    GdkEventType type;                /* GDK_CONFIGURE */
    GdkWindow *window;
    gint8 send_event;
    gint16 x, y;
    gint16 width;
    gint16 height;
};
```

**Event Name String**

- configure_event

**Callback Function Prototype**

```c
gint func(GtkWidget *widget, GdkEventConfigure *event, gpointer arg);
```

**Description**

Configure events indicate a change in the size and/or location of a window. The `window` field identifies the window that was moved or resized. The `x` and `y` fields identify the new `x` and `y` locations of the window in the root window coordinate space. `Width` and `height` identify the width and the height of the window.

**GdkEventProperty**

```c
struct _GdkEventProperty {
    GdkEventType type;                /* GDKPROPERTY_NOTIFY */
};
```
Event Name String
property_notify_event

Callback Function Prototype

\[
gint\;\text{func}(\text{GtkWidget }*\text{widget}, \text{GdkEventProperty }*\text{event}, \text{gpointer } \text{arg});\]

Description

Property events indicate a change of a property. Properties are named data associated with a window. This data is stored on the X server to which your Gtk+ client is connected. Properties have a unique ID that either is predefined or is assigned by the X server at the time the property is installed. This ID is identified by the atom field in the preceding event struct. Several standard properties are used to help the window manager do its job. For example, when you call \text{gtk_set_window_title()}, GDK will set the XA_WM_NAME atom of the window to the character string that was passed to \text{gtk_set_window_title()}. The window manager will be notified of this property change by the X server via a PropertyNotify event. Upon receiving the event, the window manager will redraw the title displayed in the window title bar decoration that the window manager has placed around your application’s top-level window.

While the major use of properties is in satisfying window manager protocols (such as specifying window titles and icon pixmaps) or client notification of window deletion, properties can also be used as a form of interprocess communication among cooperating clients. GDK provides routines that allow you to create, modify, and destroy properties.

The time field stores the time that the event occurred in server-relative time (milliseconds since server reset). State identifies the type of change that has occurred. PropertyNewValue indicates that the value of the property identified by atom has changed. PropertyDelete indicates that the property identified by atom no longer exists on the server.
Event Name Strings
- selection_clear_event
- selection_request_event
- selection_notify_event

Callback Function Prototype

```c
gboolean func(GtkWidget *widget, GdkEventSelection *event, gpointer arg);
```

Selections are an important form of interprocess communication available to all X clients, including those written in Gtk+. Selections provide the mechanism by which copy and paste operations among clients are performed. The classic example of such an operation is highlighting text in an `xterm(1)` and using mouse button 2 to paste the highlighted text in another `xterm` window. The `xterm` in which the text is highlighted is referred to as the owner client, and the `xterm` into which the text is pasted is referred to as the requestor client. Owner clients have data currently selected, and requestor clients want that data. Selections allow the requestor to become aware that data is available and provide the mechanism by which the owner can convert the data into a form that is useful by the requestor. A client (or a widget) can either be an owner or a requestor, as the need arises.

Selection Protocol. Basically, the protocol between the owner and requestor is as follows. In this example, we’ll assume we are performing copy and paste between text edit (entry) widgets in two different clients. We’ll refer to the text edit widget in client 1 as entry-1 and the text edit widget in client 2 as entry-2.

When the user selects text in entry-1 (we ignore here how that is done), client 1 will call `gtk_selection_owner_set()` to obtain ownership of the selection atom named `GDK_SELECTION_PRIMARY` (XA_PRIMARY). If successful and client 1 did not already own the selection, a SelectionClear (GDK_SELECTION_CLEAR) event will be sent to the previous owner (perhaps client 2, but this could be any X11 client connected to the X server, written using Xlib, Motif, or any other X11 toolkit). The client receiving the SelectionClear event will respond by unhighlighting the previously selected text. Notice that all we have done so far is switch the ownership of the primary selection atom from one client to another. No data has been transferred at this point.

Assume now that the text edit widget in client 2 (entry-2) obtains focus, and the user initiates a paste operation in some application-specific way. Client 2 now takes on the role of requestor and calls `gtk_selection_convert()` to obtain the data. `Gtk_selection_convert()` will call `gdk_selection_convert()`, which in turn will call `XConvertSelection()`. `XConvertSelection()` is passed a window ID, the GDK_SELECTION_PRIMARY atom, and a target atom. The target atom is used to indicate the data type to which the requestor would like the selected data to be converted, if necessary or even possible, by the owner prior to transferring the selected data to the X server. A base set of targets is predefined by X11’s Inter-Client Communication Conventions Manual (ICCCM). Table 3.7 illustrates some predefined target atoms in X11.
After the requestor has successfully called gtk_selection_convert(), the owner receives a SelectionRequest (GDK_SELECTION_REQUEST) event. Selection identifies the selection to which the request pertains. Usually, selection will be GDK_SELECTION_PRIMARY unless the owner is supporting multiple selections. Target is the target atom (for example, one of the atoms listed in Table 3.7). Property identifies the atom or property where the selected data should be placed. Requestor identifies the window of the client that is making the request.

Now that the owner has received the GDK_SELECTION_REQUEST event, it attempts to convert the selection it owns to the requested type. If the owner is unable to perform the conversion (for example, the data associated with the selection is text and the requestor wants it converted to a colormap), then the owner creates and sends to the requestor a GDK_SELECTION_NOTIFY event with the property field set to GDK_NONE. If the conversion was successful, the owner also sends a GDK_SELECTION_NOTIFY event but with the property field set to the same value.

### Table 3.7 Predefined Target Atoms in X11

<table>
<thead>
<tr>
<th>Atom</th>
<th>X11 Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>XA_ARC</td>
<td>XArc</td>
</tr>
<tr>
<td>XA_POINT</td>
<td>XPoint</td>
</tr>
<tr>
<td>XA_ATOM</td>
<td>Atom</td>
</tr>
<tr>
<td>XA_RGB_COLOR_MAP</td>
<td>Atom (standard colormap)</td>
</tr>
<tr>
<td>XA_BITMAP</td>
<td>Pixmap (depth 1)</td>
</tr>
<tr>
<td>XA_RECTANGLE</td>
<td>XRectangle</td>
</tr>
<tr>
<td>XA_CARDINAL</td>
<td>int</td>
</tr>
<tr>
<td>XA_STRING</td>
<td>char *</td>
</tr>
<tr>
<td>XA_COLORMAP</td>
<td>Colormap</td>
</tr>
<tr>
<td>XA_VISUALID</td>
<td>VisualID</td>
</tr>
<tr>
<td>XA_CURSOR</td>
<td>Cursor</td>
</tr>
<tr>
<td>XA_WINDOW</td>
<td>Window</td>
</tr>
<tr>
<td>XA_DRAWABLE</td>
<td>Drawable</td>
</tr>
<tr>
<td>XA_WM_HINTS</td>
<td>XWMHints</td>
</tr>
<tr>
<td>XA_FONT</td>
<td>Font</td>
</tr>
<tr>
<td>XA_INTEGER</td>
<td>int</td>
</tr>
<tr>
<td>XA_WM_SIZE_HINTS</td>
<td>XSizeHints</td>
</tr>
<tr>
<td>XA_PIXMAP</td>
<td>Pixmap (depth 1)</td>
</tr>
</tbody>
</table>
received by the owner in the GDK_SELECTION_REQUEST event. The selection and target fields in any GDK_SELECTION_NOTIFY event should be the same values as those received in the GDK_SELECTION_REQUEST event.

The final major portion of the selection protocol happens back on the requestor. The requestor will receive a GDK_SELECTION_NOTIFY event. If the property field is GDK_NONE, then the requestor knows that the selection failed. Otherwise, the selection was successful, and the requestor then reads the property specified in the property field for the converted data.

GdkEventClient

struct _GdkEventClient
{
    GdkEventType type;        /* GDK_CLIENT_EVENT */
    GdkWindow *window;
    gint8 send_event;
    GdkAtom message_type;
    guint8 data_format;
    union {
        char b[20];
        short s[10];
        long l[5];
    } data;
};

Event Name String

client_event

Callback Function Prototype

int
func(GtkWidget *widget, GdkEventClient *event, gpointer arg);

Client events provide a mechanism by which one client can send an event to some other client executing on the same X server. An example of this was illustrated when we discussed selections. The owner of a selection, in response to a GDK_SELECTION_REQUEST event, will send a GDK_SELECTION_NOTIFY event to the requestor client to indicate the result of the request.

Any event type can be sent by a client to another client using this mechanism. In practice, however, use of client events is generally restricted to selections, where it is needed to satisfy the selection protocol, or to window managers which use them to notify clients of some pending event, such as the destruction of a window.

Client events are never selected by the receiving client; they will always be sent to the receiving client regardless of the event mask associated with the receiving clients’ window.

Message_type is an atom that is used to identify the type of the message sent. It is up to the clients that send and receive messages of this type to agree on the value of this field. Data_format specifies the format of the message sent in the event and must have one of these values: 8, 16, or 32. This is necessary so that the X server can do the necessary swapping of
Events

bytes. Data contains the actual data sent in the message, either 20 8-bit chars, 10 16-bit shorts, or 5 32-bit longs.

GdkEventAny

struct _GdkEventAny
{
    GdkEventType type;             /* any event type is possible here */
    GdkWindow *window;
    gint8 send_event;
};

Event Name Strings
destroy_event
delete_event
map_event
unmap_event
no_expose_event

Callback Function Prototype

gint
func(GtkWidget *widget, GdkEventAny *event, gpointer arg);

GdkEventAny is a convenient, event-independent means by which the type, window, and send_event fields of any event can be accessed. Generally, your signal functions will map to a specific type of event, and you will never make use of this type. However, several events only communicate type and window information, and so they make use of GdkEventAny to pass event information into a callback function, perhaps at the cost of decreased code clarity. Event types that have GdkEventAny * in their callback prototypes include GDK_DESTROY, GDK_DELETE, GDK_UNMAP, GDK_MAP, and GDK_NO_EXPOSE.

Signal and Event APIs

Each widget in Gtk+ supports signals that, when triggered, represent a change in the state of the widget. Signal functions, or callbacks, are the way that the logic of your application is connected to the occurrence of these events.

As a programmer, you are free to register none, one, or multiple callbacks for any signal supported by an object. Gtk+ will invoke each of the signal functions registered for an object in the order they were registered by the programmer. In addition, a “class function” associated with the signal is also invoked by Gtk+. This class function is what would normally be executed by Gtk+ for that widget. Unless you are overriding the behavior of the widget, you generally need not be concerned with the class function. But there are times when you might, and I will present an example in this chapter. You can control whether or not your callback function is called after all the class functions by registering your callback with gtk_signal_connect_after(). It is up to the widget designer to determine what the
default is for a given widget; the choices include calling the class function before, after, or both before and after your callbacks for the widget have been called.

Let’s look at the functions that are available to application programmers for use in creating, controlling, and destroying signals. In doing so, we will discuss a few interesting tidbits about signals not covered so far.

**Signal Lookup**

The first function is `gtk_signal_lookup()`. The prototype for this function is as follows:

```c
 gint
 gtk_signal_lookup (gchar *name, gint object_type)
```

What `gtk_signal_lookup()` does is search the widget hierarchy for the signal identified by name, starting with the object type specified by `object_type` and searching recursively higher to include the object type’s parents if needed. If the search is successful, then the signal identifier, a unique number that identifies the signal, will be returned. If the search is not successful, then `gtk_signal_lookup()` returns 0.

To use this function, you need to know what object types and signal names are. Let’s start with object types. Each widget class in Gtk+ has an object type, defined by the widget programmer. The naming convention for object types seems to be `GTK_OBJECT_*`, where * is replaced with the name of the widget class. For example, the object type that corresponds to the GtkButton widget class is `GTK_OBJECT_BUTTON`. The object type is defined in the header file for the widget class, usually `gtk.h`, where `type` is the name of the widget class. Again, using GtkButton as our example, the header file in which the `GTK_OBJECT_BUTTON` macro is defined is named `gtk-button.h`. The object macro is defined to be a call to a function also defined by the widget writer. There is a convention for the naming of this function, too; in this case it is `gtk_*_get_type()`, which for the GtkButton class would be `gtk_button_get_type()`.

Now let’s turn to signal names. In the example code presented earlier in this chapter, we connected a callback routine to the “destroy” signal of the window object that represented our application’s main window with the following code:

```c
             gtk_signal_connect (GTK_OBJECT(window), "destroy",
                        GTK_SIGNAL_FUNC(PrintAndExit), times);
```

Here, destroy is an example of a signal name. Another signal that we connected to our application was the “clicked” signal, defined by the GtkWidget widget class. Each widget class defines some number of signals that are specific to the widget class. Signals that are common to more than one class will be defined in a parent class, from which the widget classes that share that signal can inherit.

When I introduce a widget class in the chapters that follow, I will specify the name of the object type macro that corresponds to the widget class as well as the name and behavior of each signal supported by the widget class.

Let’s now take a quick look at how `gtk_signal_lookup()` might be called. To make the example familiar, we’ll simply modify the earlier example to use `gtk_signal_lookup()` to val-
Signal Lookup

idate the signal name we pass to gtk_signal_connect(). Note that this is sort of a contrived example, but it does illustrate how to call gtk_signal_lookup().

```c
if ( gtk_signal_lookup( "destroy", GTK_OBJECT_WINDOW ) )
    /* The "destroy" signal is implemented, go ahead and register the signal function with the widget */
    gtk_signal_connect (GTK_OBJECT(window), "destroy",
                        GTK_SIGNAL_FUNC(PrintAndExit), times);
else
    fprintf( stderr, "'destroy' is not implemented\n" );
```

The following is another way to make the call to gtk_signal_lookup():

```c
GtkObject *object;
object = GTK_OBJECT(window);
if ( gtk_signal_lookup( "destroy", GTK_OBJECT_TYPE(object) ) )
```

The GTK_OBJECT_TYPE macro takes a GtkWidget * argument. I’ll discuss objects in detail later in this chapter. Notice that the preceding code promotes reusability. We can use this strategy to define a function that can be called to search for support for a given signal name in an arbitrary widget:

```c
/* Returns 0 if signal name is not defined, otherwise 1 */
gint HasSignal( GtkWidget *widget, char *name )
{
    GtkWidget *object;
    int retval = 0;
    object = GTK_OBJECT(widget);
    if ( object != (GtkWidget *) NULL )
        retval = gtk_signal_lookup( name,
                        GTK_OBJECT_TYPE( object ) );
    return( retval );
}
```

Gtk+ also defines a function that takes a signal number and returns that signal’s character string name. Here is its prototype:

```c
gchar*
gtk_signal_name (gint signal_num)
```

Gtk+ maintains a global table of signal names. A signal number in Gtk+ is merely an index into this table, so what this function really does is return the string that is stored in the table indexed by signal_num (or 0 if signal_num is not a valid index into the table).
Emitting Signals

Although in practice this may not be a very common thing to do, Gtk+ does give a client the ability to cause events and signals to trigger. This can be done by calling one of the gtk_signal_emit* functions:

```c
void gtk_signal_emit (GtkWidget *object, gint signal_type, ...)
void gtk_signal_emit_by_name (GtkWidget *object, gchar *name, ...)
```

The first argument to either function is the object from which the signal or the event will be generated. The second argument to gtk_signal_emit() is the type of the signal. This can be found by calling gtk_signal_lookup(), as previously described (or the function HasSignal(), as previously developed). The second argument to gtk_signal_emit_by_name() is the event name; gtk_signal_emit_by_name() will do the lookup operation itself. If you already have the signal type value, it is more efficient to call gtk_signal_emit() to avoid the overhead incurred by gtk_signal_emit_by_name() to look up the event name string and convert to the signal type value accepted by gtk_signal_emit().

The remaining arguments to the gtk_signal_emit* functions will vary in type and number based on the signal being emitted. For example, the prototype for the map_event (GDK_MAP) callback function is as follows:

```c
gint func(GtkWidget *widget, GdkEventAny *event, gpointer arg);
```

The call to gtk_signal_emit_by_name() would then be as follows:

```c
GdkEventAny event;
gint retval;
...
gtk_signal_emit_by_name( GDK_OBJECT(window), "map_event", &event, &retval );
```

The third argument to gtk_signal_emit_by_name() is a pointer to a GdkEventAny struct, and it is passed as the second argument to the signal callback function. The fourth parameter is a pointer to hold the value returned by the callback function, which is of type gint. If the callback function being invoked is void, we would simply omit the final argument to the gtk_signal_emit* function (as in the example that follows).

Note that the application making the preceding call would need to fill in the fields of event, including the event type, the window ID, and the send_event fields. The third and final argument to the callback is the application-specific pointer or data that was passed to gtk_signal_connect().

As a second example, the callback function for the GtkButton widget “pressed” signal has the following function prototype:
To invoke this handler, we would call `gtk_signal_emit_by_name()` as follows:

```c
gtk_signal_emit_by_name (GTK_OBJECT (button), "pressed");
```

Since there is no return value from the callback (the function is `void`), we need not pass a pointer to hold the return value, and so we pass `NULL` instead. Also, the callback function has no arguments (except for the obligatory widget pointer and application-private data that all callback functions are passed), so we pass no additional arguments to the `gtk_signal_emit*` function.

Some widget signal functions do take arguments. For example, the callback function invoked by the `GtkCList` widget (which we will talk about in detail later in this book) when a row is selected by the user has the following function prototype:

```c
void
select_row_callback(GtkWidget *widget, gint row, gint column,
                     GdkEventButton *event, gpointer data);
```

The function `select_row_callback()` takes three arguments—row, column, and event—in addition to the widget and data arguments that are passed to every signal function. The call to `gtk_signal_emit_by_name()` in this case would be as follows:

```c
GtkWidget *clist;
int      row, column;
...

gtk_signal_emit_by_name (GTK_OBJECT (clist), "select_row", row,
                          column, NULL);
```

The value `NULL` will be passed as the “event” argument to `select_row_callback()`.

**Emitting Signals—An Example**

Now might be a good time to provide some example code. This example creates a top-level window with a `GtkDrawingArea` widget child. Every second, the application generates and handles a synthetic mouse motion event. It also handles actual mouse motion events that occur in the same window when the user moves the mouse over the window.

```c
#include <stdio.h>
#include <time.h>
#include <gtk/gtk.h>
#include <unistd.h>
#include <signal.h>

static GtkWidget *drawing;

void
AlarmFunc( int foo )
{
    GdkEvent event;
    gint retval;
    ...

gtk_signal_emit_by_name (GTK_OBJECT (drawing), "press", foo,
                          NULL);
```
014      gtk_signal_emit( GTK_OBJECT(drawing),
015                       gtk_signal_lookup( "motion_notify_event", 
016                       GTK_OBJECT_TYPE(drawing) ), &event, &retval );
017
018      alarm(1L);
019  }
020
021  static void
022  motion_notify_callback( GtkWidget *w, GdkEventMotion *event, char *arg )
023  {
024      static int count = 1;
025
026      fprintf( stderr, "In motion_notify_callback %s %03d\n", arg, count++ );
027      fflush( stderr ) ;
028  }
029
030  void
031  Exit (GtkWidget *widget, gpointer arg)
032  {
033      gtk_main_quit ();
034  }
035
036  int
037  main( int argc, char *argv[] )
038  {
039      GtkWidget *window, *box;
040      struct sigaction old, act;
041      gtk_set_locale ();
042      gtk_init (&argc, &argv);
043      window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
044      gtk_signal_connect (GTK_OBJECT(window), "destroy", 
045                      GTK_SIGNAL_FUNC(Exit), NULL);
046      gtk_window_set_title (GTK_WINDOW (window), "Events 3");
047      gtk_container_border_width (GTK_CONTAINER (window), 0);
048      box = gtk_vbox_new (FALSE, 0);
049      gtk_container_add (GTK_CONTAINER (window), box);
050      drawing = gtk_drawing_area_new ();
051      gtk_widget_set_events (drawing,
052                      GDK_POINTER_MOTION_MASK);
053      gtk_signal_connect( GTK_OBJECT(drawing), "motion_notify_event", 
054                       GTK_SIGNAL_FUNC(motion_notify_callback), "Hello World" );
055      gtk_box_pack_start (GTK_BOX (box), drawing, TRUE, TRUE, 0);
056
057      gtk_widget_show_all (window);
Signal Lookup

066 act.sa_handler = AlarmFunc;
067 act.sa_flags = 0;
068 sigaction( SIGALRM, &act, &old );
070 alarm( 1L );
071
072 gtk_main ();
073
074 sigaction( SIGALRM, &old, NULL );
075 return( 0 );
076 }

Analysis of the Sample
On line 058, a GtkDrawingArea widget is created, and then on lines 059 and 060, the event mask for the GtkDrawingArea widget is set to GDK_POINTER_MOTION_MASK, enabling motion_notify event notification for the widget. On lines 061 and 062, the signal callback function motion_notify_callback(), implemented on lines 023 through 029, is registered with Gtk+ to be invoked when motion_notify_events in the GtkDrawingArea widget are received.

058 drawing = gtk_drawing_area_new ();
059 gtk_widget_set_events (drawing, GDK_POINTER_MOTION_MASK);
060 gtk_signal_connect( GTK_OBJECT(drawing), "motion_notify_event",
061     GTK_SIGNAL_FUNC(motion_notify_callback), "Hello World" );

On lines 067 through 070, we use POSIX signal function sigaction(2) to register a SIGALRM signal handler named AlarmFunc(), which is implemented on lines 009 through 020. Then, on line 070, we call alarm(2) to cause the SIGALRM signal to fire one second later. When SIGALRM is triggered, AlarmFunc() is entered.

009 void
010 AlarmFunc( int foo )
011 {
012     GdkEvent event;
013     gint retval;
014
015     gtk_signal_emit( GTK_OBJECT(drawing),
016         gtk_signal_lookup("motion_notify_event"),
017         GTK_OBJECT_TYPE(drawing) ), &event, &retval );
018
019     alarm(1L);
020 }

In AlarmFunc(), we call gtk_signal_emit() to generate a motion_notify_event on the window associated with the GtkDrawingArea widget named drawing. Doing this will cause our signal callback function, motion_notify_callback(), to be called by Gtk+. Motion_notify_callback() simply prints a message that includes a serial number and the application-dependent data that was registered with the signal callback function, the string “Hello World”.

There are two reasons why I made use of alarm(2) in this example. The first is that alarm() provides a convenient method by which an asynchronous event can be generated at
a fixed interval, giving me an opportunity to generate the motion_notify_events needed to illustrate the main idea of this example. The second reason for using alarm() is to point out a possible point of confusion with regards to terminology. It is important to note that signals in Gtk+/GDK are not the same thing as UNIX signals, as described in signal(7) and handled by UNIX functions such as signal(2) and sigaction(2).

There are certainly times when sending a signal to yourself is appropriate. I will give one such example when I discuss the GtkDrawingArea later in this book.

### Controlling Signals

Gtk+ provides a few functions that allow applications to control signals in a variety of ways. The first of these functions is gtk_signal_emit_stop():

```c
void
gtk_signal_emit_stop (GtkWidget *object, gint signal_type);
```

The function gtk_signal_emit_stop() stops the emission of a signal. A signal emission is defined as the invocation of all signal callback functions that have been registered with a widget for a given signal type. For example, should an application register with a widget a dozen callback functions for an event or signal, the emission of that signal will begin once the event occurs and will continue until each of the callback functions registered by the application has been called. The argument signal_type is obtained in the same way as the argument of the same name passed to gtk_signal_emit(). If you’d rather identify the signal by name instead of by signal_type, call gtk_signal_emit_stop_by_name():

```c
void
gtk_signal_emit_stop_by_name (GtkWidget *object, char *name);
```

An example should make this clear. I modified the preceding example slightly so that five different signal callback functions are registered with the GtkDrawingAreaWidget:

```c
117     gtk_signal_connect( GTK_OBJECT(drawing), "motion_notify_event",
118             GTK_SIGNAL_FUNC(motion_notify_callback1), "Hello World1" );
119     gtk_signal_connect( GTK_OBJECT(drawing), "motion_notify_event",
120             GTK_SIGNAL_FUNC(motion_notify_callback2), "Hello World2" );
121     gtk_signal_connect( GTK_OBJECT(drawing), "motion_notify_event",
122             GTK_SIGNAL_FUNC(motion_notify_callback3), "Hello World3" );
123     gtk_signal_connect( GTK_OBJECT(drawing), "motion_notify_event",
124             GTK_SIGNAL_FUNC(motion_notify_callback4), "Hello World4" );
125     gtk_signal_connect( GTK_OBJECT(drawing), "motion_notify_event",
126             GTK_SIGNAL_FUNC(motion_notify_callback5), "Hello World5" );
```

Each signal callback function (motion_notify_callback1(), etc.) will be invoked by the GtkDrawingArea widget after AlarmFunc() calls gtk_signal_emit(). I then modified each callback function slightly to generate a random number in the range [0,100]. If the random number falls below 50, then the signal callback function makes a call to gtk_signal_emit_stop_by_name() to stop the emission of the signal. For example:
The effect of this change is that, should one of the signal callback functions generate a random number below 50, the remaining signal callback functions will not be invoked for the signal emission because signal emission will be stopped. In testing this function, motion_notify_callback1() was called approximately half as often as motion_notify_callback2(), motion_notify_callback3() was called approximately half as often as motion_notify_callback2(), and so on. At least this demonstrates that my random number macro was performing approximately as it should have been.

Note that we do not need to reconnect the signal callback functions after an emission is stopped. The next time the signal is generated, all functions are once again eligible for invocation. Also, calling gtk_signal_emit_stop*() for a signal that is not being emitted is a no-op.

I mentioned that connecting a signal with gtk_signal_connect() will cause the registered signal function to be invoked after all signal functions previously registered with the widget for that signal and prior to the default class signal function implemented for the widget. However, applications can arrange to have signal callback functions invoked after the class signal callback function by registering the callback with gtk_signal_connect_after():

```c
gint
gtk_signal_connect_after (GtkWidget *object, gchar *name, 
                        GtkSignalFunc func, gpointer func_data)
```

A slightly different way to connect a signal to a signal callback function is to call gtk_signal_connect_object(), which has the following prototype:

```c
gint
gtk_signal_connect_object (GtkWidget *object, gchar *name, 
                          GtkSignalFunc func, GtkWidget *slot_object)
```

The major difference between gtk_signal_connect_object() and the other signal connection functions—gtk_signal_connect() and gtk_signal_connect_after()—is reflected in the function prototypes of the gtk_signal_connect* functions and in the function prototypes of the signal callback functions that are invoked.
The final argument to `gtk_signal_connect()` and `gtk_signal_connect_after()` is application private data. The first argument to a callback function registered using `gtk_signal_connect()` and `gtk_signal_connect_after()` is the widget or object with which the signal callback function was registered. In contrast, `gtk_signal_connect_object()` takes as its final argument a `GtkWidget` pointer, which is the first (and only) argument passed to the signal callback function when the signal or event is triggered. The effect is that an event happening in the widget with which the signal callback function was registered will cause a callback function to be invoked as though the event or signal happened in some other object.

The function `gtk_signal_connect_object_after()` is analogous to `gtk_signal_connect_after()` in that the signal callback function will be invoked after the default widget class signal function for the widget has been invoked:

```c
 guint
 gtk_signal_connect_object_after (GtkWidget *object, gchar *name,
   GtkSignalFunc func, GtkWidget *slot_object)
```

The classic example of `gtk_signal_connect_object()` is in tying together the press of a Quit, Cancel, or Dismiss `GtkButton` object with the destruction of the dialog or window in which the button is being displayed. The following code fragment taken from `testgtk.c`, an example application that is a part of the `Gtk+` distribution, illustrates how this can be done:

```c
 GtkWidget *button, *box2;
...

button = gtk_button_new_with_label("Close");
gtk_box_pack_start(GTK_BOX(box2), button, TRUE, TRUE, 0);
gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
   GTK_SIGNAL_FUNC (gtk_widget_destroy), GTK_OBJECT (window));
```

Here, the clicked signal supported by the `GtkButton` class is registered with the `GtkButton` instance defined by `button`. When the "clicked" signal is triggered, the function `gtk_widget_destroy()` will be invoked. The function prototype for `gtk_widget_destroy()` is as follows:

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

Note that `gtk_widget_destroy()` takes only one argument, which in this case is a widget to destroy. The widget argument passed to `gtk_widget_destroy()` is the same object that was passed as the last argument to `gtk_signal_connect_object()`.

It is likely that the only time you will ever use this technique is when handling the destruction of simple dialogs such as those used to display an error or warning message to the user. There is little need for an application signal callback function to deal with the cancellation or dismissal of such a dialog, and so the preceding technique works well. However, if you have a dialog that allows users to make changes to data, you’ll want to register an application-specific signal callback function with the "clicked" signal of "Cancel" or "Dismiss" button so that your application will have the opportunity to verify the cancellation operation.
Gtk+ supplies two functions that can be used by an application to disconnect a previously registered signal callback function from a signal or event. The first of these is `gtk_signal_disconnect()`:

```c
void
gtk_signal_disconnect (GtkObject *object, gint id);
```

The argument `object` is the object with which the signal was registered, corresponding to the first argument that was passed to the `gtk_signal_connect*` family of functions. As I pointed out earlier, more than one signal callback function can be registered with a given signal, so the `id` argument to `gtk_signal_disconnect()` is needed to identify which of the registered signal callback functions is to be disconnected. The argument `id` is the value returned by the `gtk_signal_connect*` function used to connect the signal callback function to the signal. In the following example, a signal callback function is connected to a clicked signal, and then is immediately disconnected, to illustrate the techniques involved:

```c
GtkWidget *button;

button = gtk_button_new_with_label("Close");

id = gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
    GTK_SIGNAL_FUNC(gtk_widget_destroy), GTK_OBJECT (window));

gtk_signal_disconnect (GTK_OBJECT (button), id);
```

`gtk_disconnect_by_data()` performs the same operation as `gtk_signal_disconnect()`, but instead of identifying the signal callback function by its `id`, the signal function is identified by the application data passed as the `func_data` argument to `gtk_signal_connect()` or `gtk_signal_connect_after()`, or by the `slot_object` argument passed to either `gtk_signal_connect_object()` or `gtk_signal_connect_object_after()`. Here is the function prototype:

```c
void
gtk_signal_disconnect_by_data (GtkObject *object, gpointer data);
```

Note that multiple signal callback functions can be disconnected with a call to `gtk_signal_disconnect_by_data()`, as every signal callback function registered with the object or application data to which data pertains will be disconnected by this function.

Gtk+ also allows an application to temporarily block the invocation of a signal callback function. This can be done by calling `gtk_signal_handler_block()`:

```c
void
gtk_signal_handler_block (GtkObject *object, gint id);
```

The arguments passed to `gtk_signal_handler_block()` are analogous to those passed to `gtk_signal_disconnect()`. The first argument, `object`, is the object with which the signal being blocked was registered by calling one of the `gtk_signal_connect*` functions. The argument `id` is the value returned by the `gtk_signal_connect*` function that registered the signal callback function with the object.
A similar function, `gtk_signal_handler_block_by_data()`, performs the same task as `gtk_signal_handler_block()`, but the data argument is used to identify the signal callback function(s) to be blocked. This is similar to how the data argument `gtk_signal_disconnect_by_data()` is used to identify the signal callback functions to be disconnected. Here is the function prototype for `gtk_signal_handler_block_by_data()`:

```c
void
gtk_signal_handler_block_by_data (GtkObject *object, gint data);
```

The argument blocking a signal handler function is not the same as stopping signal emission by calling `gtk_signal_emit_stop()`. When signal emission is stopped, it is only for the emissions corresponding to the triggering of a single event. The next time the event or signal is triggered, each and every signal callback function is once again eligible for invocation. When a signal callback function is blocked, it will not be invoked until it has been unblocked, no matter how many times the signal or event is triggered.

Each signal callback function registered with an object maintains a “blocked” count that starts at 0 and is incremented each time the signal is blocked by a call to a `gtk_signal_handler_block*` function.

Signal callback functions that are blocked can be unblocked at any time by a call to `gtk_signal_handler_unblock()`:

```c
void
gtk_signal_handler_unblock (GtkObject *object, gint id);
```

This decrements an object’s blocked count by one. When the blocked count goes to zero, the signal callback function for the specified object, identified by id, will become eligible for invocation the next time the signal or event is triggered. The function

```c
void
gtk_signal_handler_unblock_by_data (GtkObject *object, gint data);
```

is analogous to `gtk_signal_disconnect_by_data()` in that it has the ability to unblock more than one blocked signal callback function.

The final function that operates on signals that I will discuss in this section is `gtk_signal_handlers_destroy()`:

```c
void
gtk_signal_handlers_destroy (GtkObject *object);
```

`gtk_signal_handlers_destroy()` destroys all signal callback functions that have been registered with the specified object. This will not, however, destroy the class signal and event functions that are implemented by the object or widget.

**Objects**

The very first argument passed to `gtk_signal_connect()` is a pointer to variables of type `GtkWidget`. Example code in the preceding section made use of a macro named `GTK_OBJECT` to coerce variables that were declared as `GtkWidget *` to `GtkObject *`
so that the code would conform to the function prototype of the routine being called. Objects have played a part in nearly every function that has been discussed so far in this chapter. However, until now, I’ve not really defined yet what an object is. Obtaining a basic understanding of objects is the main idea behind this section.

Many of you will no doubt have some previous experience with C++, Smalltalk, or some other object-oriented language or programming paradigm. Perhaps you have heard about object-oriented programming but have no actual experience in its use. Or perhaps you have no idea at all what I am talking about when I use the terms “object” and “object-oriented programming.”

It is not within the scope of this book to present an in-depth look at object-oriented systems or design. Gtk+ is a C-based toolkit, and C is not considered to be an object-oriented language, although object-oriented designs can in fact be implemented in C.

For us, a widget is the practical manifestation of what it is that we talk about when we refer to objects in Gtk+. Widgets, as we will come to see in the chapters that follow, are characterized by both visual representation and functionality. Visual representation defines how the widget appears in the user interface of the application. A widget’s functionality defines how that widget will respond to input events directed towards it by Gtk+.

**Button Widgets as Objects**

The GtkButton widget can be used to illustrate both of these widget attributes. Visually, buttons are simply rectangular areas in a window that have labels that identify the action that the application will perform when the button is clicked. The button’s label can be a text string, which is usually the case, or it can be in the form of a pixmap that graphically represents the operation that will be performed by the application when the button is clicked. Figure 3.1 illustrates instances of the GtkButton widget.

![Figure 3.1 Button Widgets](image)

Functionally speaking, a GtkButton widget will invoke an application-registered callback function when any one of the following events occur (these events were mentioned earlier in this chapter but are repeated here for convenience):

- The pointer enters the rectangular region occupied by the button.
- The pointer leaves the rectangular region occupied by the button.
- The pointer is positioned over the button, and a mouse button is pressed.
- The pointer is positioned over the button, and a mouse button is released.
• The user clicks the button (a combination of pressing and releasing a mouse button while the pointer is positioned over the button).

The behavior of a widget often corresponds to visual change, as is the case with the Gtk-Button widget, which will change its appearance after one of the preceding events has occurred. For example, as the pointer enters the rectangular region occupied by the button, the widget will redraw the button in a different color (a lighter shade of gray) to provide visual feedback to the user that the pointer is in a region owned by the button (see Figure 3.2). Should the user press mouse button 1 while the pointer is positioned over a GtkButton widget, the widget will redrew itself as shown in Figure 3.3.

Figure 3.2  Pointer Positioned Over Button 3

Figure 3.3  Button 3 Clicked

Let’s now take a look at another Gtk+ widget, the GtkToggleButton, and see how it compares to the GtkButton widget.

Toggle buttons are used by an application to represent a value that can have one of two states. Examples include On or Off, Up or Down, and Left or Right.

In the nontoggled state, a GtkToggleButton widget has an appearance much like that of a GtkButton widget (see Figure 3.4). GtkToggleButton widgets are rectangular in shape and have a label. Like GtkButton, a GtkToggleButton widget’s label can be either a text string or a pixmap. Visually, a user would be hard-pressed to tell a button from a toggle button in a user interface at first glance.
Figure 3.4 GtkToggleButton Widgets

Functionally, GtkToggleButton and GtkWidget are closely related. Both respond basically the same way to the pointer entering or leaving the region occupied by a widget instance. The GtkToggleButton widget supports the same signals as GtkWidget, plus a new signal, “toggled.” The GtkToggleButton widget will emit this signal after the user positions the pointer over a toggle button and presses mouse button 1, the same condition that leads GtkWidget to emit a “clicked” signal. In fact, a GtkToggleButton widget can also emit a “clicked” signal if the application so desires.

As the pointer enters the area occupied by a GtkToggleButton widget, the widget will redraw itself in a lighter shade of gray, just as a GtkWidget widget does. However, GtkToggleButton’s visual response to presses and releases is different. In the toggled state, a toggle button will appear as in Figure 3.5, which corresponds to the pressed state of GtkWidget. In the untoggled state, a toggle button will appear as in Figure 3.4, corresponding to the unpressed state of GtkWidget. The transition between the toggled and untoggled state occurs at the time of the button release (assuming the pointer is still within the area of the button at the time of release; otherwise, the toggle button widget will revert to its prior state).

Figure 3.5 GtkToggleButton Widget in Toggled State

We’ve now established that GtkWidget and GtkToggleButton share much in terms of look, feel, and functionality. So, how does this relate to objects?
Widgets in Gtk+ are organized as a hierarchy of classes. Each class in the Gtk+ widget class hierarchy is ultimately a descendant of the class named GtkObject. Refer to the appendix for a listing of the Gtk+ class hierarchy as of Gtk+ 1.2.

The GtkObject class represents a parent class from which all classes in the widget hierarchy inherit basic behavior. GtkObjects’ contribution is minimal but important. Among the functionality provided by GtkObject is the signal mechanism. As one descends the hierarchy, visual representation (if any) and functionality become increasingly specialized. Each node in the class hierarchy diagram that has descendants provides a base class from which those descendants can, if they choose, inherit their look and feel or functionality. A child class will always replace some (perhaps all) of the look and feel or functionality of its parent class or introduce new look and feel or functionality that was not present in the parent class.

Such is the case with GtkButton (the parent) and GtkToggleButton (the child). Much of the implementation of GtkToggleButton is inherited from GtkButton. GtkToggleButton overrides the semantics of button presses and button releases and introduces the “toggled” signal, but essentially, a toggle button is really a button for the most part.

Object API

GtkObject implements an API that can be used by widget implementations and client developers. Here I just focus on a few of the application-level functions in this API so we can obtain a better understanding of what objects are from the perspective of an application. The first routine is gtk_object_destroy():

```c
void
gtk_object_destroy( GtkObject *object )
```

gtk_object_destroy() takes an object as a parameter and destroys it. This routine can be called from any place that gtk_widget_destroy() is called. We saw one example of the use of gtk_widget_destroy() earlier in this chapter when I discussed gtk_signal_connect_object().

For example, let’s say you have a GtkButton widget that you need to destroy. You can perform the destruction using either of the following techniques:

```c
GtkButton *button;
...

gtk_widget_destroy( GTK_WIDGET( button ) );
```
or

```c
GtkButton *button;
...

gtk_object_destroy( GTK_OBJECT( button ) );
```

To be complete, you could declare the button as GtkObject, in which case:
GtkObject *button;
...

gtk_widget_destroy( GTK_WIDGET( button ) );

or

GtkObject *button;
...

/* No cast needed, it is already an object */

gtk_object_destroy( button );

Finally, we could declare the button as GtkWidget, and then it would be:

GtkWidget *button;
...

/* No cast needed, it is already a widget */

gtk_widget_destroy( button );

or

GtkWidget *button;
...

gtk_object_destroy( GTK_OBJECT( button ) );

Regardless of how it’s done, in the end, the button will be destroyed. Notice the use of the casting macros. If a routine expects an object and you have a widget, use GTK_OBJECT to convert the widget to an object. And, going the other way, use GTK_WIDGET to cast an object to a widget when a widget is needed.

These casting macros are not restricted to just GtkWidget and GtkObject. All widget classes in the widget hierarchy implement a macro that can be used to convert from one widget class to another. In later chapters, I will point out the macro name when I discuss the corresponding widget class, but as a general rule of thumb, the name of the macro can be formed by taking the widget class name, converting it to all uppercase, and inserting an underscore (_) after the initial GTK. For example, the casting macro for GtkButton is GTK_BUTTON. It is not actually this easy; additional underscores are added in some cases in which the class name is formed by a concatenation of words. For example, the casting macro for the class GtkDrawingArea is GTK_DRAWINGAREA.

These “casting” macros do not perform just a simple C-style cast. They also make sure that the item being cast is non-NULL and that the class to which the object is being cast either is
Chapter 3 • Signals, Events, Objects, and Types

of the same class (making the cast a no-op) or is a super-class in the widget instance hierarchy. Thus, a cast from any widget class (e.g., GtkButton) to GtkWidget will be successful because all buttons inherit from GtkWidget. A cast from GtkWidget to GtkWidget will fail because GtkList does not inherit from GtkWidget. The casting macros generate warning output if, for whatever reason, the cast being performed is illegal.

As you become moderately experienced as a Gtk+ programmer, deciding when and when not to use the casting macros will become somewhat second nature.

Here is a source code snippet that illustrates casting at a few different levels:

Listing 3.2 Object/Widget Casting Example

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

void PrintAndExit (GtkWidget *widget, char *foo)
{
    if ( foo )
        printf( "%s\n", foo );
}

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

    gtk_set_locale ();

    gtk_init (&argc, &argv);

    /* button */
    button = (GtkWidget *) gtk_button_new_with_label("foo");
    gtk_signal_connect (GTK_OBJECT(button), "destroy",
        GTK_SIGNAL_FUNC(PrintAndExit), "button, object destroy");
    gtk_object_destroy( GTK_OBJECT( button ) );

    /* button */
    button = (GtkWidget *) gtk_button_new_with_label("foo");
    gtk_signal_connect (GTK_OBJECT(button), "destroy",
        GTK_SIGNAL_FUNC(PrintAndExit), "button, widget destroy");
    gtk_widget_destroy( GTK_WIDGET( button ) );

    ...
```
The application basically creates and destroys six buttons. Here, I only show the lines pertaining to creating the button and storing its handle in a variable of GtkWidget *. The full application contains code that creates and destroys instances of GtkWidget, storing them as GtkWidget * and as GtkWidget *. Our first need for a cast occurs on line 039. Here, the return value from gtk_button_new_with_label() is GtkWidget *, and I am required to cast this result to (GtkButton *) to eliminate a compile-time warning from gcc(1). Note that all *new() functions return a handle of type GtkWidget * because buttons, scrollbars, labels, toggle buttons, and so on, are all widgets. I personally feel that using “GtkButton * button;” to declare a variable that is going to hold a widget handle to a GtkButton to be better style, but adding the casts is annoying, so I suggest all widgets be declared as GtkWidget *. There are other good reasons for doing this, but avoiding the need for adding casts all over the place is reason enough.

On lines 041 and 042, we register a signal callback function with Gtk+ for the “destroy” signal. The user data argument is a string that identifies the operation; in this case, “button, object destroy” means we have stored the widget in a “button” variable (i.e., a variable of type GtkButton *) and we are going to call gtk_object_destroy() as opposed to gtk_widget_destroy() to destroy the widget. On line 044, we destroy the widget by making a call to gtk_object_destroy(). Note that the argument is cast by the GTK_OBJECT macro because the widget instance was stored in a GtkWidget * variable, but gtk_object_destroy() requires a variable of GtkWidget *.

The same basic logic prevails on lines 046 through 051, except this time, the button is destroyed with a call to gtk_widget_destroy(), requiring us to cast the button variable from a GtkWidget * to a GtkWidget * using GTK_WIDGET.

Object Attributes

In Gtk+, objects have attributes. When you instantiate a GtkWidget widget with a call to gtk_button_new_with_label(), for example, you are setting the button widget’s label to the string that was passed in as an argument. Actually, there is more going on than just this, but from an application’s perspective, this is effectively what happens.

Usually, perhaps ideally, applications will not make use of the following functions. However, it is worthwhile to look at them because it will strengthen your concept of what an object is in Gtk+.

The first function we’ll look at is gtk_object_query_args():

```
GtkArg *
gtk_object_query_args (GtkType type, guint32 **flags, guint *nargs);
```

The function gtk_object_query_args() can be used to obtain the list of attributes supported by a widget class. This can only be done after the application has instantiated at least one instance of the class being queried. The argument type defines the class to be queried. The best way to obtain the value of type is to call a routine provided by the class implementation. For the GtkButton class, this is gtk_button_get_type(). For other classes, it will be named
gtk_*_get_type() by convention (the actual names are documented along with the widget classes as they are discussed later in this book).

The second argument, flags, is a pointer to an array of unallocated guint32 (or guint) values. You can pass (guint32 **) NULL here or the address of a variable of type guint32 *:

type = gtk_button_get_type();
args = gtk_object_query_args (type, (guint32 **) NULL, ... );
or

guint32 *flags;
args = gtk_object_query_args (type, &flags, ... );

The second argument is ignored if NULL. If non-NULL, gtk_object_query_args() will allocate an array of 32-bit unsigned ints, each corresponding to an attribute supported by the widget class. The number of elements in this array is stored in the value returned in nargs, a pointer to an unsigned int, which is the third and final argument to gtk_object_query_args(). Once you are done with the flags array, you must free it by calling g_free():

g_free( flags );

The flags in Table 3.8 are supported by Gtk+.

**Table 3.8 Flags Supported by gtk_object_query_args()**

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>GTK_ARG_READABLE</td>
<td>Attribute’s value can be read</td>
</tr>
<tr>
<td>GTK_ARG_WRITABLE</td>
<td>Attribute’s value can be written</td>
</tr>
<tr>
<td>GTK_ARG_CONSTRUCT</td>
<td>Can be specified at object construction time</td>
</tr>
<tr>
<td>GTK_ARG_CONSTRUCT_ONLY</td>
<td>Must be specified at object construction time</td>
</tr>
<tr>
<td>GTK_ARG_CHILD_ARG</td>
<td>Attribute applies to children of widget (used by containers)</td>
</tr>
<tr>
<td>GTK_ARG_READWRITE</td>
<td>Same as GTK_ARG_READABLE</td>
</tr>
</tbody>
</table>

The flags relevant to applications include GTK_ARG_READABLE, GTK_ARG_WRITABLE, and GTK_ARG_READWRITE. These flags specify whether an application can query the value of an argument, change its value, or do either, respectively. The remaining flags are relevant to the widget writer and do not concern us here.

gtk_object_query_args() returns a pointer to an array of type GtArg. The array will have nargs entries in it. Once you are finished with the array, it must also be freed with a quick call to g_free().
If you are curious about the contents of GtkArg, the structure is defined in gtktypeutils.h. However, you can, and should, access the fields in this structure using accessor macros defined by Gtk+. Table 3.9 lists the possible simple data types that an attribute can have, along with the accessor macros that can be used to obtain the data for each type.

Table 3.9  Nonaggregate Accessor Macros

<table>
<thead>
<tr>
<th>Type</th>
<th>Accessor Macro</th>
</tr>
</thead>
<tbody>
<tr>
<td>gchar</td>
<td>GTK_VALUE_CHAR(a)</td>
</tr>
<tr>
<td>guchar</td>
<td>GTK_VALUE_UCHAR(a)</td>
</tr>
<tr>
<td>gboolean</td>
<td>GTK_VALUE_BOOL(a)</td>
</tr>
<tr>
<td>gint</td>
<td>GTK_VALUE_INT(a)</td>
</tr>
<tr>
<td>guint</td>
<td>GTK_VALUE_UINT(a)</td>
</tr>
<tr>
<td>glong</td>
<td>GTK_VALUE_LONG(a)</td>
</tr>
<tr>
<td>gulong</td>
<td>GTK_VALUE_ULONG(a)</td>
</tr>
<tr>
<td>gfloat</td>
<td>GTK_VALUE_FLOAT(a)</td>
</tr>
<tr>
<td>gdouble</td>
<td>GTK_VALUE_DOUBLE(a)</td>
</tr>
<tr>
<td>gchar *</td>
<td>GTK_VALUE_STRING(a)</td>
</tr>
<tr>
<td>gint</td>
<td>GTK_VALUE_ENUM(a)</td>
</tr>
<tr>
<td>guint</td>
<td>GTK_VALUE_FLAGS(a)</td>
</tr>
<tr>
<td>gpointer</td>
<td>GTK_VALUE_BOXED(a)</td>
</tr>
<tr>
<td>gpointer</td>
<td>GTK_VALUE_POINTER(a)</td>
</tr>
<tr>
<td>GtkWidget *</td>
<td>GTK_VALUE_OBJECT(a)</td>
</tr>
</tbody>
</table>

Accessor macros are also defined for the following aggregate types in Table 3.10.

Table 3.10  Aggregate Accessor Macros

<table>
<thead>
<tr>
<th>Accessor Macro</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>GTK_VALUE_SIGNAL(a)</td>
<td>struct {</td>
</tr>
<tr>
<td></td>
<td>GtkSignalFunc f;</td>
</tr>
<tr>
<td></td>
<td>gpointer d;</td>
</tr>
<tr>
<td></td>
<td>} signal_data;</td>
</tr>
</tbody>
</table>
To determine the type of an attribute, use the macro GTK_FUNDAMENTAL_TYPE(), passing the type field of the GtkArg struct from which data is to be accessed:

\[
\text{GTK_FUNDAMENTAL_TYPE( a.type );}
\]

The following code snippet illustrates how to call gtk_object_query_args(). Lines 008, 009, and 010 declare the variables needed for the call to gtk_object_query_args().

```c
008    GtkArg *args;
009    guint nArgs;
010    guint32 *flags = (guint32 *) NULL;
```

On line 020, we call gtk_object_query_args(). Then, on lines 029 through 080, we iterate through the array of GtkArg structs returned. For each arg, we determine its type using GTK_FUNDAMENTAL_TYPE (line 032). Then, in the switch statement, we print that type as a string to stdout:

```c
020    args = gtk_object_query_args( gtk_button_get_type(), &flags, &nArgs );
021    if ( args == (GtkArg *) NULL ) {
022        fprintf( stderr, "Unable to query widget's args\n" );
023        exit( 1 );
024    }
025
029    for ( i = 0; i < nArgs; i++ ) {
030        printf( "Name: '%s', type: ", args[i].name );
```
switch( GTK_FUNDAMENTAL_TYPE (args[i].type) ) {
    case GTK_TYPE_CHAR :
        printf( "GTK_TYPE_CHAR, " );
        break;
    case GTK_TYPE_UCHAR :
        printf( "GTK_TYPE_UCHAR, " );
        break;
    case GTK_TYPE_BOOL :
        printf( "GTK_TYPE_BOOL, " );
        break;
    ...
}

Following the switch, on lines 081 through 101, we interpret the corresponding entry in the flags array that was returned. Remember, if NULL is sent as the second argument to gtk_object_query_args(), then no flags will be returned.

switch( flags[i] ) {
    case GTK_ARG_READABLE :
        printf( "GTK_ARG_READABLE\n" );
        break;
    case GTK_ARG_WRITABLE :
        printf( "GTK_ARG_WRITABLE\n" );
        break;
    case GTK_ARG_CONSTRUCT :
        printf( "GTK_ARG_CONSTRUCT\n" );
        break;
    case GTK_ARG_CONSTRUCT_ONLY :
        printf( "GTK_ARG_CONSTRUCT_ONLY\n" );
        break;
    case GTK_ARG_CHILD_ARG :
        printf( "GTK_ARG_CHILD_ARG\n" );
        break;
    case GTK_ARG_READWRITE :
        printf( "GTK_ARG_READWRITE\n" );
        break;
}

Finally, on lines 106 through 109, the flags and args pointers are freed by a call to g_free().
Getting and Setting Object Attributes

Now that we know how to obtain a list of the attributes supported by a widget class, let’s discuss how to get and set the values of attributes in a widget or object instance. To retrieve attribute data from a widget, we need only make minor changes to the preceding source. Two routines can be used to read attribute data. The first is gtk_object_arg_get():

```c
void gtk_object_arg_get (GtkObject *object, GtkArg *arg, GtkArgInfo *info)
```

The first argument, object, is the widget from which object data is to be retrieved. The second argument, arg, is effectively the element in the vector returned by gtk_object_query_args() that corresponds to the attribute being queried. You can use gtk_object_query_args() to obtain this value, or you can allocate a GtkArg variable on the stack and set the name field to the attribute you want to query, for example:

```c
GtkArg myArg;
GtkWidget *myButton;
...

type myArg.name = "GtkButton::label";
gtk_object_arg_get( GTK_OBJECT( myButton ), &myArg, NULL );
...
```

The final argument, info, should always be passed as NULL. In fact, there are no examples of gtk_object_arg_get() usage in the Gtk+ source code where this argument is set to anything but NULL. gtk_object_arg_get() will retrieve the value internally if you pass NULL, so perhaps this argument will be deprecated in a future version of Gtk+.

On return, myArg will contain the data that was requested. If the data could not be obtained for whatever reason (for example, the attribute does not exist), gtk_object_arg_get() will generate output to the console, for example:

```
Gtk-WARNING **: gtk_object_arg_get(): could not find argument "Yabbadabba" in the 'GtkButton' class ancestry
```

The type field in the GtkArg struct will be set to GTK_TYPE_INVALID. This can be checked using code similar to the following:

```c
if ( GTK_FUNDAMENTAL_TYPE (myArg.type) == GTK_TYPE_INVALID )
    /* Attribute could not be read for some reason */
else
    /* Attribute was read */
```

The second routine that can be used to retrieve attribute values is as follows:
Objects

void
gtk_object_getv (GtkObject *object, guint n_args, GtkArg *args)

This routine is nearly identical to gtk_object_arg_get(), except that it can be used to
retrieve multiple attributes with a single function call. The argument n_args holds the number
of elements in args; args is a vector of GtkArg structs.

The following code snippet illustrates how to call gtk_object_arg_get() using the return
value from gtk_object_query_args(). The majority of the code is the same as in the previous
listing. Here I’ll just show the loop used to obtain the attribute values, one for each element
in the array of GtkArg elements returned by gtk_object_query_args():

017 widget = gtk_button_new_with_label( "This is a test" );
018
019 args = gtk_object_query_args( gtk_button_get_type(), &flags, &nArgs );
020
021 if ( args == (GtkArg *) NULL ) {
022    fprintf( stderr, "Unable to query widget's args\n" );
023    exit( 1 );
024 }
025
026 for ( i = 0; i < nArgs; i++ ) {
027
028    printf( "Name: '%s', value: ", args[i].name );
029
030    if ( flags[i] == GTK_ARG_READABLE
031            || flags[i] == GTK_ARG_READWRITE ) {
032        gtk_object_arg_get( GTK_OBJECT( widget ), &args[i],
033                             NULL );
034
035        switch( GTK_FUNDAMENTAL_TYPE (args[i].type) ) {
036            case GTK_TYPE_CHAR :
037                printf( "%c
",
038                    GTK_VALUE_CHAR (args[i]) );
039                break;
040            case GTK_TYPE_UCHAR :
041                printf( "%c
",
042                    GTK_VALUE_UCHAR (args[i]) );
043                break;
044            case GTK_TYPE_BOOL :
045                printf( "%s\n",
046                    (GTK_VALUE_BOOL(args[i])==TRUE?
047                    "TRUE":"FALSE"));
048                break;
049
050            case GTK_TYPE_STRING :
051                printf( "%s\n",
052                    GTK_VALUE_STRING (args[i]) );
053                g_free (GTK_VALUE_STRING (args[i]));
054                break;
055            ...
056        }
057    }
058
059}
Chapter 3 • Signals, Events, Objects, and Types

On line 017, we create an instance of the GtkWidget class. We need to pass an object to gtk_object_arg_get() to identify the object we are querying, and we also need, in this example, to create an instance of GtkWidget so that gtk_object_query_args() can do its job.

On line 019, we call gtk_object_query_args() to obtain a list of the attributes supported by the GtkWidget widget class. Then, on lines 026 through 105, we iterate through the array returned by gtk_object_query_args(). For each element, we make a call to gtk_object_arg_get(); this occurs on line 033. We then switch on the type field set by gtk_object_arg_get(); accessing this value using the GTK_FUNDAMENTAL_TYPE macro as previously described. In the switch statement, we simply use the type to determine the format string passed to printf and use the accessor macro needed to retrieve the value from the GtkWidget element. Note the use of g_free(), which is needed to release storage allocated by Gtk+ for GTK_TYPE_STRING attributes, as shown on lines 074 through 078.

Gtk+ provides two routines for setting attribute values in a widget. They are:

```c
void
gtk_object_set (GtkWidget *object, const gchar *first_arg_name, ...)
```

and

```c
void
gtk_object_setv (GtkWidget *object, guint n_args, GtkWidget_Arg *args)
```

Both can be used to set multiple attributes. gtk_object_setv() would be the more convenient routine to call after obtaining a GtkWidget vector from gtk_object_query_args(), although this is not required, of course. In all other cases, gtk_object_set() is probably the easiest of the two to use.

gtk_object_setv() takes the very same arguments as gtk_object_getv(). The only difference is that the elements in the args vector need to contain that data to which the attribute is being set and the type. To do this, use the accessor macros used to read data from a GtkWidget struct. For example, to change the label of a button, we might code the following:
Objects

GtkArg arg;
GtkWidget *widget;
...

arg.type = GTK_TYPE_STRING;
arg.name = "GtkButton::label";
GTK_VALUE_STRING(arg) = "Yabba Dabba Doo";
gtk_object_setv( GTK_OBJECT( widget ), 1, &arg );

The function gtk_object_set() accepts a variable argument list. Each attribute to be set is specified in the argument list by its name, such as GtkButton::label, followed by a variable number of arguments that specify the value of that attribute. In some cases, a single argument can be used to specify a value, for example, a button label value is a string. In some cases, the attribute being set is an aggregate, and in this case, the value arguments will correspond to the fields of a structure or the elements of a table.

The final argument to gtk_object_set() must be NULL to indicate the end of the argument list (if you forget the NULL, gtk_object_set() will read beyond the stack, leading to unpredictable behavior).

The preceding example, using gtk_object_set(), is reduced to the following:

GtkWidget *widget;
gtk_object_set( GTK_OBJECT( widget ), "GtkButton::label", "Yabba Dabba Doo", NULL );

Associating Client Data with an Object or Widget

Gtk+ allows applications to associate an arbitrary amount of indexed data with a widget instance. An index is nothing more than a character string used to uniquely identify the data. The data associated with an index is of type gpointer. Gtk+ maintains one list of indexed data per object or widget; there is no practical limit to the number of data items that can be attached to the list. The only restriction is that each entry on the list must have a unique index. Adding an entry using an index that corresponds to an entry already on the list will cause Gtk+ to replace that entry’s data with the newly specified data.

Let’s take a quick look at the functions involved, and then we’ll discuss how this facility might be useful to an application.

To add an entry to an object’s list, applications can use gtk_object_set_data() or gtk_object_set_data_full().

The first function, gtk_object_set_data(), takes an object, a key, and a data value as arguments:

```c
void
gtk_object_set_data (GtkWidget *object, const gchar *key, gpointer data)
```

An item on the object’s data list will be added by Gtk+ as a result of making this call.

The second function is gtk_object_set_data_full():

```c
void
gtk_object_set_data_full (GtkWidget *object, const gchar *key, gpointer data)
```
void gtk_object_set_data_full (GtkObject *object, const gchar *key,
gpointer data, GtkDestroyNotify destroy)

gtk_object_set_data_full() takes the same arguments plus an additional argument named
destroy, which is a pointer to a function that will be called by Gtk+ should the data indexed
by key be destroyed. Destruction means that the entry indexed by key was removed from
the list. The function prototype for destroy is as follows:

void
DestroyFunc ( gpointer data )

You may pass NULL as the last argument to gtk_object_set_data_full(), but then the call
effectively becomes equivalent to calling gtk_object_set_data().

If an entry indexed by key already exists on the object’s list prior to calling either
gtk_object_set_data() or gtk_object_set_data_full(), then the gpointer stored by that entry
will be replaced by data. A new entry on the list will not be created because indexes on the
list must be unique.

To retrieve data from an object’s list, call gtk_object_get_data():

gpointer
gtk_object_get_data (GtkObject *object, const gchar *key)

The function gtk_object_get_data() takes an object and a key. If there is no entry on the
object’s list indexed by key, then NULL is returned. Otherwise, the data that is stored on
the list indexed by key will be returned.

To remove an entry from an object’s list, call gtk_object_remove_data() or
gtk_object_remove_no_notify():

void
gtk_object_remove_data (GtkObject *object, const gchar *key)

void
gtk_object_remove_no_notify (GtkObject *object, const gchar *key)

Either function will remove the entry indexed by key from the list maintained by object,
if such an entry exists. If gtk_object_remove_data() was called, the destroy function regist-
ered with the entry, if any, will be invoked, and a copy of the data stored by that entry will
be passed as an argument as previously discussed. If gtk_object_remove_no_notify() is
used, then the destroy function will not be invoked.

Gtk+ supports the following two convenience functions:

void
gtk_object_set_user_data (GtkObject *object, gpointer data)

gpointer
gtk_object_get_user_data (GtkObject *object)
Calling one of these functions is equivalent to calling gtk_object_set_data() or gtk_object_get_data(), respectively, with a key argument that has been set to user_data.

Please be aware that some widget implementations will add a user_data entry, so setting or removing this entry may lead to incorrect behavior of the widget and your application. Calling gtk_object_get_data() or gtk_object_get_user_data() and obtaining a NULL return value cannot be taken as an indication that the list does not contain an entry indexed by user_data. It could be that the entry exists and, at the time of calling, is storing a NULL pointer as its data item. Therefore, until Gtk+ provides a routine that can be used to test for the existence of an item on a list indexed by key, I recommend playing it safe and avoid adding, setting, or removing entries keyed by user_data. Also, take reasonable precautions to ensure that keys used by your application are unique and do not collide with keys that might be in use internally by a widget implementation.

When to Use Client Data
How might one use indexed data in an application? An obvious application would be a word processor, a text editor, or for that matter, any application that allows the concurrent editing of more than one document. An image-editing tool such as The GIMP is an example of such an application.

The GIMP allows users to display and edit more than one image at a time. Each image being edited has an associated set of attributes, including width, height, image type (RGB, grayscale), the name of the file from which the image was read and to which it will be saved by default, and a flag that indicates whether or not the image is dirty and needs to be saved before the user exits The GIMP. Some of this information is reflected in the title bar of the window displaying the image data (Figure 3.6).

So how might The GIMP maintain information about images currently being edited? A convenient method for organizing this data would be to maintain a data structure for each image being edited. A possible candidate data structure is the following:
Now that we have a way to represent this data, where should we store this data structure? Whatever method we choose, we must be able to easily associate the image currently being edited or displayed by the user with the data about that image.

One possibility would be to store it in a global linked list. Whenever the user selects a window and it is brought to the front, we search the linked list for the entry with a “win” field that contains the window handle of the window that was raised; this record will contain the information about the image being edited in the window. This is a perfectly fine solution. The only problem is that the application will need to maintain code needed to support the linked list.

An alternate solution would be to use indexed data. To associate image data with a window, we simply use `gtk_object_set_data()` at the time the image is created or opened. For example, the routine that creates a new image and its window might perform the following:

```c
ImageData *imageData;
GtkWidget *dialog;

imageData = (ImageData *) malloc( sizeof( ImageData ) );
imageData->filename = (gchar *) NULL;
imageData->dirty = FALSE;
/* set defaults */
imageData->width = imageData->height = 250;
imageData->type = FILL_BG;
imageData->type = IMAGE_RGB;
/* create a window */
dialog = CreateGIMPImageDialog( imageData );
imageData->win = GTK_WIDGET(dialog)->window;
/* associate the image data with the dialog */
gtk_object_set_data( GTK_OBJECT( dialog ), "image_data",
(gpointer) imageData );
...
In the preceding, CreateGIMPImageDialog() is a hypothetical routine that creates a dialog or window using the image attributes passed to it as an argument. For example, the width and height fields are used to define the size of the window.

There are two advantages in using the preceding technique. First, we didn’t need to provide the linked list code; gtk_object_set_data() takes care of this for us. Second, the image data is tightly coupled with the dialog being used to display it. The result is that finding the image data that corresponds to a dialog is a straightforward task.

For example, we could associate a signal function with the dialog widget that will fire when the window becomes destroyed, as follows:

```c
gtk_signal_connect (GTK_OBJECT (dialog), "destroy", GTK_SIGNAL_FUNC(HandleDestroy), NULL);
```

HandleDestroy() can then retrieve the image_data entry from the dialog and, if the image data is “dirty”, give the user the opportunity to save changes to a file:

```c
void
HandleDestroy (GtkWidget *widget, gpointer data)
{
    ImageData *ptr;
    /* get the image data attached to the widget */
    ptr = gtk_object_get_data ( GTK_OBJECT ( dialog ), "image_data" );
    /* if we found it, and the data is dirty, give user opportunity to
     * save it */
    if ( ptr != (ImageData *) NULL && ptr->dirty == TRUE )
        LetUserSaveData ( ptr );
    /* free the image data */
    if ( ptr != (ImageData *) NULL )
        free ( ptr );
}
```

Well, that ends my coverage of objects in this chapter. You should now have a good idea of what an object is and be aware of some of the ways that objects can be used in a Gtk+ application.

### Types

You may have noticed that the code snippets and function prototypes presented in this chapter make use of nonstandard C types such as gpointer, gint, and gchar * . These types, which are defined by Glib in glib.h, are intended to aid in the portability of Gtk+, GDK, and Glib and the applications that make use of these toolkits.
You should get in the habit of using these types, particularly when declaring variables that will be passed as arguments to the Glib, GDK, or Gtk+ APIs. Using C language types such as void *, int, or char * is acceptable in other cases. Declaring a loop index variable as int as opposed to gint will not lead to any problems, unless perhaps the index variable is used as an argument to a Glib function that requires gint, for example. While perhaps unlikely, it is not guaranteed that a gint will map to an int in all implementations.

Table 3.11 lists the basic types defined by Glib for UNIX and Linux.

### Table 3.11 Glib Types

<table>
<thead>
<tr>
<th>C Language Type</th>
<th>Glib Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>gchar</td>
</tr>
<tr>
<td>signed char</td>
<td>gint8</td>
</tr>
<tr>
<td>unsigned char</td>
<td>guint8</td>
</tr>
<tr>
<td>unsigned char</td>
<td>guchar</td>
</tr>
<tr>
<td>short</td>
<td>gshort</td>
</tr>
<tr>
<td>signed short</td>
<td>gint16</td>
</tr>
<tr>
<td>unsigned short</td>
<td>guint16</td>
</tr>
<tr>
<td>unsigned short</td>
<td>gushort</td>
</tr>
<tr>
<td>int</td>
<td>gint</td>
</tr>
<tr>
<td>int</td>
<td>gboolean</td>
</tr>
<tr>
<td>signed int</td>
<td>gint32</td>
</tr>
<tr>
<td>unsigned int</td>
<td>guint32</td>
</tr>
<tr>
<td>unsigned int</td>
<td>guint</td>
</tr>
<tr>
<td>long</td>
<td>glong</td>
</tr>
<tr>
<td>unsigned long</td>
<td>gulong</td>
</tr>
<tr>
<td>float</td>
<td>gfloat</td>
</tr>
<tr>
<td>double</td>
<td>gdouble</td>
</tr>
<tr>
<td>void *</td>
<td>gpointer</td>
</tr>
<tr>
<td>const void *</td>
<td>gconstpointer</td>
</tr>
</tbody>
</table>

Where more than one Glib type maps to the same C type (for example, gboolean and gint both map to int in the preceding table), avoid interchanging Glib types. In other words, if a function prototype mandates the use of a gboolean, do not use a gint in its place; use a gboolean.
Summary

In this chapter, we discussed signals and signal handling. Signals are the way in which widgets communicate changes back to your application and are a required part of any meaningful Gtk+ application. You will, as a Gtk+ programmer, do much of your programming within the context of signal functions. We also covered Gtk+ events and objects and described the associated functions for each. Events are low-level when compared to signals, corresponding to events that exist at the X protocol level. Many (most) of the events we discussed are intercepted by widgets on behalf of your application and are translated into their higher level signal counterparts. Some applications, however, can make good use of events (this is especially true for applications that involve interactive graphics of some kind). We also discussed objects. Objects are fundamental to the architecture of the Gtk+ toolkit. In a practical sense, you will find yourself using the terms “object” and “widget” interchangeably. All Gtk+ widgets are descendants of GtkWidget in the object/widget hierarchy. This chapter described what objects are as well as the API that exists for manipulating them. The chapter ended with a short discussion ofGtk+ data types. For the sake of portability, you should strive to use the Gtk+ types (e.g., use “guint” instead of “unsigned int”), although, as I will illustrate time and again in this book, use of the Gtk+ types is by no means a requirement.