Lesson 2: GTK+ Basics

1 A First GTK+ Program

We will begin with a very simple GTK+ program in order to demonstrate some of the key tasks that every GTK+ main program must perform. The program, `hello_world.c`, is found in many books and articles about GTK+ in one form or another because it contains most of the basic elements.

```
Listing 1: First GTK+ program: hello_world.c
1:  #include <gtk/gtk.h>
2:
3:  int main (int argc,
4:            char *argv[])
5:  {
6:      GtkWidget *window;
7:
8:      /* Initialize GTK+ and all of its supporting libraries. */
9:      gtk_init (&argc, &argv);
10:
11:     /* Create new window, give it a title and display to the user. */
12:     window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
13:     gtk_window_set_title (GTK_WINDOW (window), "Hello␣World");
14:     gtk_widget_show (window);
15:
16:     /* Hand control over to the main loop. */
17:     gtk_main ();
18:     return 0;
19: }
```

This program will not terminate normally; in order to terminate it, you will have to kill it with a signal such as Control-C issued from the terminal. If you click the close-box in the window when it runs, the window will disappear but the process will continue to run in a windowless state. (Use the `ps` command to see the list of running processes after closing the window to prove it to yourself.)

Note that line 1 includes `<gtk/gtk.h>` , the GTK+ header file that includes all GTK+ definitions as well as headers for all libraries required by GTK+. It must always be present in any GTK+ program. You will usually need no other header file in order to use the GTK+ libraries.

Building the Application A GTK+ program depends on several libraries, not just the GTK+ library. This fact is obscured because `<gtk/gtk.h>` includes many other header files. In fact if you open this file, you will see exactly what other headers are included. To find the file, you can use a command such as

```
find /usr/include -name gtk.h
```

which will search recursively through all directories in /usr/include for a file named `gtk.h`. The output will most likely be

```
/usr/include/gtk-2.0/gtk/gtk.h
```

If we look at the contents of the header file, we will see that it contains the following lines, among many others:

```
#include <gdk/gdk.h>
#include <gtk/gtkaboutdialog.h>
#include <gtk/gtkaccelgroup.h>
...
```

The remaining lines are all like the last two lines above, including various `gtk` headers. The first line is the important one at the moment; it says that this program has dependencies in the `<gdk.h>` header file and thus will need to link to the GDK library. If we look in `<gdk/gdk.h>` we will see that it has dependencies on the pango and cairo header files and therefore we may need to link to those libraries also. How then, can we possibly know what libraries we need?

The answer lies in a useful command named `pkg-config`. The `pkg-config` command is designed to output the compiler options necessary to compile and link programs that use software libraries such as GTK+. When a package like GTK+ is installed on the system, a file with a `.pc` extension is installed, usually in the directory `/usr/lib/pkgconfig`. The `pkg-config` command uses the information in that file to display the output needed to compile and link programs that use the library.

We can use `pkg-config` to display compiler flags for GTK+2 with the command

```
pkg-config --cflags gtk+-2.0
```

which will output

```
-I/usr/include/gtk-2.0 -I/usr/lib/gtk-2.0/include -I/usr/include/atk-1.0
-I/usr/include/cairo -I/usr/include/pango-1.0 -I/usr/include/glib-2.0
-I/usr/lib/glib-2.0/include -I/usr/include/freetype2 -I/usr/include/libpng12
```

This shows that GTK+2 depends on many header files, including those for ATK, Cairo, Pango, and GLib. We can get the linker flags using the command

```
pkg-config --libs gtk+-2.0
```

which will output

```
-L/lib -lgtk-x11-2.0 -lgdk-x11-2.0 -latk-1.0 -lgdk_pixbuf-2.0 -lm
-lpangocairo-1.0 -lpango-1.0 -lcairo -lgobject-2.0 -lgmodule-2.0
-ldl -lglib-2.0
```

This shows that the corresponding libraries are required, as well as some not mentioned above such as GObject and Gdk-Pixbuf.

If you are not familiar with the options supplied to `gcc`, please read my brief tutorial on `gcc`, which may be downloaded from http://www.compsci.hunter.cuny.edu/~sweiss/resources/The GCC Compilers.pdf. The compiler flags tell the compiler which directories to look in for the various included header files. The linker flags tell the linker which directories to look in to find the various required libraries (the "`L`" options) and which libraries are required for linking (the "`l`" options). We can combine these flags into a single call to `pkg-config` with the single command

```
pkg-config --cflags --libs gtk+-2.0
```

To compile and link the `helloworld.c` program, creating the executable `hello_world`, we can use the command

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

Putting the `pkg-config` command in backquotes (`` `...` ``) causes the shell to replace the command itself by its output, as if you typed

```
gcc -Wall -g -o hello_world hello_world.c \
-I/usr/include/gtk-2.0 -I/usr/lib/gtk-2.0/include -I/usr/include/atk-1.0 \
-I/usr/include/cairo -I/usr/include/pango-1.0 -I/usr/include/glib-2.0 \
-I/usr/lib/glib-2.0/include -I/usr/include/freetype2 -I/usr/include/libpng12\
-L/lib -lgtk-x11-2.0 -lgdk-x11-2.0 -latk-1.0 -lgdk_pixbuf-2.0 -lm \
-lpangocairo-1.0 -lpango-1.0 -lcairo -lgobject-2.0 -lgmodule-2.0 \
-ldl -lglib-2.0
```

The backslash "\" at the end of the first line is there in order to continue the command to the next line; it serves as an escape character for the newline character that follows. The `-g` option enables debugging output. Although there is space in this document to write this command on a single line, you ought to know how to break a command across multiple lines in the shell. Of course if it fits on a line, you can type it on a single line!

Because it is tedious to type this command over and over to build the executable, it is preferable to create a Makefile that automatically executes the command. If there is a Makefile in the current working directory and you type the command

```
make
```

then the Makefile will be executed, in much the same way that a shell script is executed by the shell. A Makefile can be named `Makefile` or `makefile`; either will work. If you want to name it `foo`, then you could also type

```
make -f foo
```

although you'd better have a good reason to do this! A simple Makefile for this program would look like

```
CC    = gcc
FLAGS = -Wall -g `pkg-config --cflags --libs gtk+-2.0`

all:
<T>hello_world

hello_world: hello_world.c
<T>$(CC) -o hello_world hello_world.c $(FLAGS)

clean:
<T>rm -f hello_world
```

In the listing above, the string "`<T>`" represents a tab character. I put it there so that you do not think there are space characters at the beginning of those lines. In a Makefile, the "recipes" must begin with a tab character. A Makefile is basically a list of rules. Each rule consists of a line describing the targets and a line describing the recipe. For example, the two lines

```
hello_world: hello_world.c
<T>$(CC)  -o hello_world  hello_world.c  $(FLAGS)
```

are a rule that states that the target `hello_world` depends upon the file `hello_world.c`, and that to create the target `hello_world`, the `gcc` compiler will be invoked with the command line

```
gcc  -o hello_world  hello_world.c  -Wall -g ‘pkg-config --cflags --libs gtk+-2.0‘
```

Notice that I substituted the variables `CC` and `FLAGS` by their values. In general, the `make` program lets you create variables and use them in rules. To use a variable named `FOO`, you write `$(FOO)`. Things can get much more complicated than this.

In general you should know how to create Makefiles for your projects. If you are unfamiliar with them, now is the time to learn the basics. A very good introduction to the basics of Makefiles is found at the website http://www.eng.hawaii.edu/Tutor/Make/.

Programming Convention Sprinkled throughout these notes you will find paragraphs labeled “Programming Convention.” GTK+ programmers tend to follow a few conventions that make for good style in any program, not just GTK+ programs. For the most part, I will follow those conventions in these notes, and where appropriate, I will introduce a convention prior to using it. The first one you will discover is the format of function declarations and definitions, particularly those with multiple parameters. Rather than writing all of the parameters on a single line, even if space permits, the parameters are written one per line, as follows:

```
GtkWidget* gtk_table_new (guint rows,
                          guint columns,
                          gboolean homogeneous);
```

Sometimes when there are just two parameters, they will be written on a single line. I will also follow the style used by the authors of GTK when there are multiple declarations. For example:

```
GtkWidget *        gtk_window_new                (GtkWindowType type);
void               gtk_window_set_title          (GtkWindow *window,
                                                  const gchar *title);
void               gtk_window_set_wmclass        (GtkWindow *window,
                                                  const gchar *wmclass_name,
                                                  const gchar *wmclass_class);
```

Here the return types, function names, and parameter lists are aligned in columns. This makes it easier to read. The above is a fragment of the documentation for the `GtkWindow` class.

2 Tasks in Creating GTK Programs

The program in Listing 1 demonstrates all but one of the key steps in creating a GTK+ application. There are basically seven different steps:

1. Initialize the GTK environment;
2. Create the widgets and set their attributes;

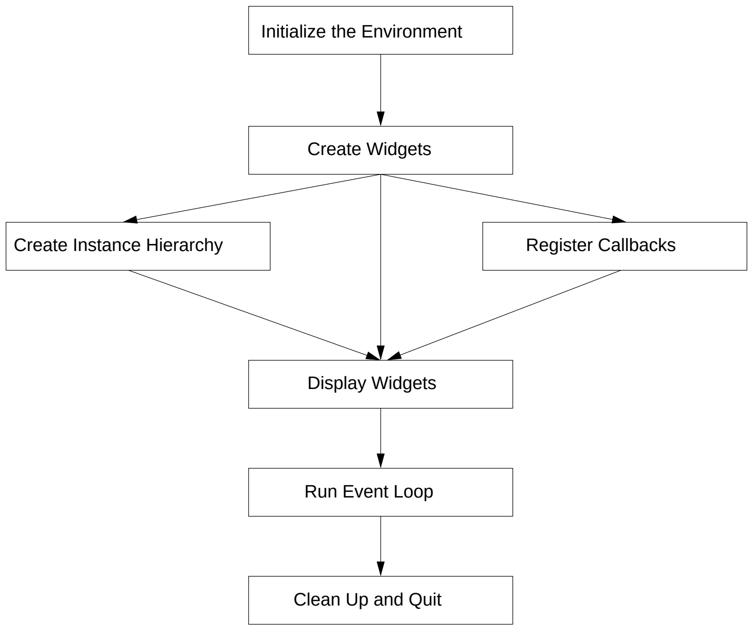

Figure 1: Flow graph for Listing 1

3. Register the callback functions;
4. Create the instance hierarchy;
5. Display the widgets;
6. Enter the main event loop;
7. Clean up and quit.

These steps are not linear in sequence. The flow graph below shows what steps must precede others. In the remainder of this document, I will summarize and give examples of each of these steps. Some of them are trivial for you as programmer because they involve just a single function call, but they are all critical and cannot be omitted.

In this first program, there are no callback functions, so step 3 is not required.

2.1 Initializing the Environment

The call to `gtk_init()` in line 9 initializes the environment:

```
void gtk_init (int &argc, char &argv);
```

Note that you pass it the addresses of `argc` and `argv`, not `argc` and `argv` themselves. This is because `gtk_init()` strips any arguments that it recognizes from the list and passes the rest to your program. You must call `gtk_init()` before any other functions in the GTK+ libraries.

2.2 The GTK Object Hierarchy

Although GTK is written in C and provides an interface in C, it is object-oriented. It uses clever techniques to provide all of the important features of object-oriented programming, including private data, encapsulation, and virtual function calls. The top of the object hierarchy for all libraries is the `GObject`. Everything is derived from `GObject`. `GObject` provides the methods that allow objects to be created and destroyed, to be referenced and unreferenced, and to emit signals. The `GObject` class has a set of properties inherited by all derived classes. The `GInitiallyUnowned` class is derived from the `GObject` class. You will never have to use it and for now we will ignore it.

The `GtkObject` is the top of the GTK+ object hierarchy and is derived from `GInitiallyUnowned`. This too is a class that we will make little use of. It exists for backwards compatibility with GTK+1.0. It has been eliminated completely from GTK+3.

2.2.1 Widgets

The single most important `GtkObject` is the `GtkWidget`[1]. The word "widget" comes from the word "gadget"; in computer science, the word has come to mean something that responds to mouse clicks, key presses, or other types of user actions. In short, widgets are things like windows, buttons, menus, and edit-boxes.

The `GtkWidget` derives from `GtkObject` and inherits all of its properties and methods. The `GtkWindow` is a widget that derives from `GtkWidget` and the `GtkDialog` is a kind of window that derives from `GtkWindow`.

You may think that all widgets are either containers like windows and boxes of various kinds, or controls like buttons and menus, but this is not the case. In fact, some widgets are neither, like a `GtkRange`, which is a range of values. There are also objects that are not widgets, such as the `GtkCellRenderer` class.

Widgets can be grouped together into composite widgets. This is how menus and menu bars are constructed. A menu is a collection of widgets that act together. Menu bars are collections of menus.

Widgets have properties, such as their state and visibility. Some properties are inherited from the ancestral `GtkWidget` class because they are part of all widgets. Others are specific to particular kinds of widgets. Windows, for example, have properties not shared by rulers or calendars. Widgets also have methods, which are the functions that act upon them, and these too are inherited from ancestor classes or are specific to the class itself.

2.2.2 Object Types and Casting

The `GObject` hierarchy provides a set of macros for type checking and type casting. It is a pretty intuitive set of macros to use. For example, to cast a pointer to a `GtkWidget` into a `GObject`, the `G_OBJECT()` macro is used, as in

```
    G_OBJECT( Widget_ptr )
```

The result is a pointer to a `GObject`.

In most programs, all widgets are declared and created using `GtkWidget*` pointers. If a window or a button is created, it will be through a `GtkWidget*`; therefore, when one wants to use properties of these things that are not inherited from the `GtkWidget` class but are part of the derived class, the window or button will have to be cast to its own type. In the example program, we see that the window object is declared as a `GtkWidget`, but to set its title, we have to cast it using `GTK_WINDOW()` because windows have titles, not widgets in general.

In the program, the program's main and only window is declared as a widget in line 6:

```
    GtkWidget *window;
```

[1]In GTK+3, `GtkWidget` derives directly from `GInitiallyUnowned`.

2.3 Creating Widgets and Setting Their Attributes

The program creates a window in line 12 using the instruction

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

This creates a window of fixed size using the default width and height of 200 pixels. The area of the window includes the title bar and window border. This created a top-level window. A top-level window is not under the control of the programmer; the window manager (e.g., Gnome) has control over such things as its placement and its decorations. Your program can make requests, but they do not have to be honored by the window manager, which has the much larger responsibility to manage all currently running graphical applications. If you want a non-top-level window that you can control completely, you use the `GTK_WINDOW_POPUP` type, which we will discuss later.

For many widgets *W* there is a function of the form `gtk_`*W*`_new` that creates an instance of a *W* widget. Each of these functions is of the form

```
GtkWidget* gtk_W_new (parameters);
```

returning a pointer to a `GtkWidget` (not a W.) For some, there are no parameters, whereas for others, there are. Some examples include

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

for creating a window. Usually it is a top level window, which is the type to supply.

```
GtkWidget* gtk_button_new (void);
```

for creating a button. It takes no arguments.

```
GtkWidget* gtk_calendar_new (void);
```

for creating a calendar, also with no arguments.

```
GtkWidget* gtk_table_new (guint rows,
                          guint columns,
                          gboolean homogeneous);
```

for creating a table. It is given the numbers of rows and columns and a boolean indicating whether to use uniform spacing or not.

```
GtkPrinter* gtk_printer_new (const gchar *name,
                             GtkPrintBackend *backend,
                             gboolean virtual_);
```

for creating a printer connection.

An almost complete list of the widgets that can be created with the `gtk_*_new` call follows. I have omitted some of the more specialized widgets, and I have not included those widgets that are derived from other widgets included in the list. For example, only a single button is listed, not radio buttons or check boxes, which derive from them.

Widget Type	Name in Call	Parameter Prototype
button	`button`	`(void)`
calendar	`calendar`	`(void)`
combo-box	`combobox`	`(void)`
entry	`entry`	`(void)`
event	`eventbox`	`(void)`
frame	`frame`	`(const gchar* )`
horizontal box	`hbox`	`(gboolean, gint)`
horizontal ruler	`hruler`	`(void)`
horizontal scale	`hscale`	`(GtkAdjustment*)`
horizontal scrollbar	`hscrollbar`	`(GtkAdjustment*)`
label	`label`	`(const gchar*)`
layout	`layout`	`(GtkAdjustment*, GtkAdjustment*)`
list	`list`	`(void)`
menu	`menu`	`(void)`
menu bar	`menubar`	`(void)`
menu item	`menuitem`	`(void)`
notebook	`notebook`	`(void)`
printer	`printer`	`(const gchar*, GtkPrintBackend*, gboolean)`
socket	`socket`	`(void)`
status bar	`statusbar`	`(void)`
table	`table`	`(guint, guint, gboolean)`
text widget	`text`	`(GtkAdjustment*, GtkAdjustment*)`
text buffer	`textbuffer`	`(GtkTextTagTable*)`
tool bar	`toolbar`	`(void)`
tool item	`toolitem`	`(void)`
tree	`tree`	`(void)`
tree item	`treeitem`	`(void)`
tree path	`treepath`	`(void)`
vertical box	`vbox`	`(gboolean, gint)`
view port	`viewport`	`(GtkAdjustment*, GtkAdjustment*)`
vertical ruler	`vruler`	`(void)`
vertical scale	`vscale`	`(GtkAdjustment*)`
vertical scrollbar	`vscrollbar`	`(GtkAdjustment*)`
window	`window`	`(GtkWindowType)`

After creating a widget, you should set its attributes. For some widgets, the attributes can include the size, position, border widths, text that appears inside (e.g., it's title), its name, which is used by GTK in looking up styles in gtkrc files, a topic to be covered later, and so on.

To determine what attributes can be set for a given widget, it is useful to look at their positions in the widget hierarchy, remembering that any attribute defined by a widget's ancestor is inherited by the widget. The hierarchy is depicted in the Appendix.

In the hello_world program, after creating a window with `gtk_window_new`, the title of the window is set using

```
gtk_window_set_title (GTK_WINDOW (window), "Hello World");
```

We could also have set the resizable property with a call such as

```
gtk_window_set_resizable( GTK_WINDOW(window), FALSE);
```

which prevents the window from being resized.

Sometimes, the attributes to be set are not members of the widget class itself, but are inherited from a parent or higher ancestor. An example is the border width of a window. Windows are not the only widgets that have borders; in general, borders are a property of containers, an abstract base class from which windows and buttons and other such things are derived. The border of a container is the empty space inside of the container that surrounds its children. To set the border width of a window or a button, you have to call the ancestor's method, casting as necessary:

```
    gtk_container_set_border_width (GTK_CONTAINER (window), 10);
```

Notice that the ancestor's method expects a pointer to an instance of the ancestor (`gtk_container`), so the window widget pointer is cast using the macro `GTK_CONTAINER` to a `gtk_container*`. I will say more about this in *Defining the Instance Hierarchy* below.

2.3.1 About Sizes

Sizes are always given in logical pixels. For example, in the call to set the border width of a container, the integer is the number of logical pixels. The size of a logical pixel is determined by two factors: the physical dimensions of the screen and the current display resolution. For example, if the monitor's width is 14.75 inches and the resolution is set to 1152 by 864, then the number of logical pixels per inch is $1152/14.75 = 78$ pixels per inch.

2.4 Registering Callback Functions

In this program, there are no callback functions and hence nothing to do with them right now. The next program will demonstrate what to do, so for now, we skip this topic.

2.5 Defining the Instance Hierarchy

When you define the user interface for your program, you visually lay out the various windows and controls. Your layout determines the hierarchical relationships that the various widgets will have. In general, there is a transitive, non-reflexive, non-symmetric "contains" relationship among widgets. If widget A directly contains widget B, then A is a parent of B. If A contains B and B contains C, then A contains C by transitivity. This leads naturally to a tree of containments.

GTK manages the allocation of memory and resources required by the various widgets in your application. It frees you from having to figure out exactly how big everything needs to be, and it takes care of resizing events as they take place. But this does not happen for free; in order for GTK to do this for you, you have to do something for GTK in return. You have to tell GTK the parent-child relationships that exist in the application by telling it what is contained in what, and in what relative positions. In effect, by adding a widget into another widget viewed as a container, you are telling GTK that the former is a child of the latter.

This first program has just a single widget, the main window, so you do not have to do anything to define the containment hierarchy within the program. The next program will demonstrate what is required.

2.6 Showing The Widgets

Showing a widget is easy in this case since there is only a single widget to display. The call is simply

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

which sets the widget's visibility property to true. Line 14 uses this function:

```
gtk_widget_show( window);
```

Notice that the window does not need to be cast because it is a widget, not a window. The window may not appear at once because requests are queued. Furthermore, a widget will not be displayed on the screen unless its parent is visible as well.

You can hide a widget with the symmetric call

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

which hides a widget. If it has any children, then, because their parent is hidden, they will be hidden because of the rule cited above, not because they are marked as invisible.

2.7 Starting the Main Event Loop

Every GTK+ application must have a `gtk_main()` call, which puts the program into an event driven state. `gtk_main()` wrests control from the program and puts it in the hands of the behind-the-scenes event manager. When `gtk_main()` is running, all events and signals are processed as they occur. The only way for the program to terminate is via some outside event.

3 A Second GTK+ Application

The next application, from the Krause textbook, pages 23-24, introduces two features: *containment* and *signal handling*. It is displayed in Listing 2. It can be compiled and linked in the same way as the first program. Unlike the first program, this one will terminate itself when the user clicks the close-box in the main window.

3.1 The GtkLabel Widget

This program puts a label inside the main window using the `GtkLabel` widget. A `GtkLabel` widget is a widget that can contain non-editable, formatted text, wrapped or unwrapped. One obvious use for such a widget is to display labels for other widgets. The `GtkLabel` widget supports dozens of methods, such as setting the text, setting various text attributes (whether it is selectable, how it is aligned, wrapped or unwrapped, and so on), adding mnemonics, and retrieving the text in the widget.

To create the label, the program declares a widget pointer named label in line 9:

```
GtkWidget *window, *label;
```

and in line 25, it creates the widget and sets its text simultaneously with the function

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

in which, if `str` is not `NULL`, it becomes the text of the label. If it is `NULL` the widget has no text.

Notice that in general GTK+ uses GLib types instead of the standard C types, preferring gchar to char. The call in line 25 is

```
    label = gtk_label_new ("Hello World");
```

To illustrate the use of one `GtkLabel` method besides its constructor, the program makes the widget text selectable, so that it can be copied into the clipboard for use outside of the application using `Ctrl-C`. The function to make the text selectable is

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

which is used in line 26:

```
    gtk_label_set_selectable (GTK_LABEL (label), TRUE);
```

Notice that the function expects a `GtkLabel*` for its first argument, so the label variable, which is of type `Widget*`, must be cast to a `GtkLabel*`. The macro `GTK_LABEL` performs the cast. The second argument is set to `TRUE` to turn on the selectable property.

The label text can be changed at any time with the function

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

The text can be retrieved with `gtk_label_get_text()`:

```
    const gchar* gtk_label_get_text (GtkLabel *label);
```

Consult the GTK+ Reference Manual for other methods related to labels. You will see functions to add mnemonics (keyboard shortcuts), change the text properties and more.

```
Listing 2: Second GTK+ program: hello_world2.c
1:  #include <gtk/gtk.h>
2:
3:  void destroy (GtkWidget*, gpointer);
4:  gboolean delete_event (GtkWidget*, GdkEvent*, gpointer);
5:
6:  int main (int argc, char *argv[])
8:  {
9:      GtkWidget *window, *label;
10:
11:     gtk_init (&argc, &argv);
12:
13:     window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
14:     gtk_window_set_title (GTK_WINDOW (window), "Hello␣World!");
15:     gtk_container_set_border_width (GTK_CONTAINER (window), 10);
16:     gtk_widget_set_size_request (window, 200, 100);
17:
19:     g_signal_connect (G_OBJECT (window), "destroy",
20:                       G_CALLBACK (destroy), NULL);
21:     g_signal_connect (G_OBJECT (window), "delete_event",
22:                       G_CALLBACK (delete_event), NULL);
23:
24:     /* Create a new GtkLabel widget that is selectable. */
25:     label = gtk_label_new ("Hello␣World");
```

```
    gtk_label_set_selectable (GTK_LABEL (label), TRUE);

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

    gtk_main ();
    return 0;
}

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

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

3.2 About Container Widgets

In the first program, there was a single widget, and hence nothing was contained in anything else. In general, a GTK+ user interface is constructed by nesting widgets inside widgets. There is a natural parent-child relationship in containment: the contained object is the child of the containing object. This leads to a tree of containments in which the internal nodes are *container widgets*. So, for example, you might have a `GtkWindow` containing a `GtkTable` containing a `GtkLabel`. If you wanted an image instead of a textual label inside the table, you might replace the `GtkLabel` widget with a `GtkImage` widget. In this program, we want to put a label widget inside a window widget, and so the label would be the child of the main window.

In general, to put one widget inside another, we have to use the methods of the container class from which the parent was derived. In particular, a `GtkWindow` is derived indirectly from a `GtkContainer`. The `GtkContainer` class is an abstract base class with two major types of concrete subclasses, one that can have a single child, and others that can have multiple children. The `GtkContainer` class encapsulates several properties and methods that are characteristic of things that contain other things. For example, two of its properties are its border width and whether it can be resized. Methods include setting border width, adding and removing children, changing focus, and altering the inter-relationships among its children.

The `GtkBin` is the subclass that can only contain one child. Windows, buttons, frames, and combo-boxes are subclasses of `GtkBin`. You may wonder why a button is a `GtkBin`, but a button is simply a container that contains a label. Because a window is a type of `GtkBin`, it can only contain one widget.

The `GtkBox` is one of the subclasses that can contain multiple children. So is the `GtkTable` and the `GtkTree` (and a long list of others as well). There are three kinds of boxes: *horizontal boxes*, *vertical boxes*, and *button boxes*. Horizontal boxes and vertical boxes provide enough flexibility so that you can layout many windows with them alone.

Because a window is a type of `GtkBin`, it can have only a single child. If you only want to add a single widget to a window, you can use the `gtk_container_add()` method:

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

On line 29 of the listing, the call to add the label is

```
gtk_container_add (GTK_CONTAINER (window), label);
```

Notice that the window must be cast to a container to use this method.

Note. When you have several buttons, edit boxes, and other items to add, you will need to add a container object to the window of the type that can have multiple children, such as the GtkBox subclasses, the GtkTable, the GtkAlignment, the GtkFixed, or the GtkLayout. Also, if you have a window-less widget that needs to respond to events, it can be placed into a special GtkEventBox, which is a GtkBin class that provides a home for such widgets and allows them to appear to handle those events. Labels are a good example of this; labels have no windows and cannot respond to events, so if you wanted to make them do things when they are clicked, you would put them into event boxes. We will cover event boxes and the other types of containers in subsequent lessons.

We use one other GtkContainer method in this example program to demonstrate how to set the border width of the window. Remember that the border is the region inside the window near the edge, like a margin. The border width is the number of pixels surrounding on the inside edge of the window that cannot be used by any nested widgets. The border creates space around the child widget. The function is:

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

and it is used on line 15 to set a width of 10 pixels all around the window:

```
gtk_container_set_border_width (GTK_CONTAINER (window), 10);
```

Size Requisition and Allocation

On line 16, the program sets the window size to 200 by 100 pixels with the call

```
gtk_widget_set_size_request (window, 200, 100);
```

which uses the widget method

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

This sets the minimum size of the window. It is a request to the window manager to draw it at least this large, possibly larger. The window will not be drawn this size if doing so makes it too small to be functional. By passing -1 in either size parameter, you tell GTK+ to use the natural size of the window, which is the size GTK+ would calculate it needs to draw all of the nested widgets within it. If one parameter is -1, the window is sized using the other and scaled appropriately.

In order to understand how large widgets can or cannot be, you need to understand how GTK determines the sizes of widgets. A container acts as a negotiator between its parent and its children; children make size requests, usually implicitly, but parents may impose size limitations. The container finds a happy medium.

A widget makes a size request by virtue of its properties. For example, a label contains text of a particular size and so it needs a minimum height and width for its display. The same is true of an icon or an image. The container must add up the size requests of all of its children and present a request to the parent. If the request can be satisfied, then all of the children receive an allocation that satisfies their needs. If the request cannot be satisfied, then the children are given less space than they requested.

Requests proceed from the top down. The highest level widget in the tree, usually a window, asks its child container how much space it needs. The container in turn, asks each of its children, which in turn ask their children , which in turn ask theirs, and so on, until the leaf nodes in this tree are reached. The leaf nodes present their requests, and their parents compute their requests, and their parents, theirs, and so on, until the request reaches the top level. If it can be satisfied, the allocations take place, in a top-to-bottom direction. The top-level widget tells its children how much space they get, and they in turn tell their children how much they get, and so on, until all widgets have been given their allocations.

Each child widget must "live with" what it gets. Life is tough for a widget, and it does not always get what it asked for. On the other hand, windows almost always expand to satisfy the requests of their children, so under normal circumstances, the windows will be as large as necessary to grant the requests of the children. Even if a window is set to be non-resizable, it will still expand to accommodate the sizes of its children; it will not let the user resize it though.

3.3 Showing Widgets

Sometimes it is convenient to display each widget as it is created, and other times it is easier to wait until just before the event loop to display them all. If you have placed many widgets inside a container, rather than calling `gtk_widget_show()` for each widget, you first create them all, and then you can use the following function:

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

where widget is the top-level container itself. This function will recursively descend into the container, showing each widget and any of its children. In our example program, the call is made on line 30:

```
gtk_widget_show_all(window);
```

One advantage of doing it this way is that the window appears all at once, already drawn, instead of appearing in pieces, which can be disconcerting to the user.

3.4 Signals and Callback Functions

3.4.1 Signals and Events

All GUIs depend upon the propagation of events from the hardware and/or operating system to the application, so that it can handle those events. Events usually represent inputs from a user, such as mouse clicks, mouse drags, or key presses, but they can also represent changes of focus caused by an application becoming active, or remapping of screen regions to different applications.

To be concrete, when you press a mouse button to interact with a GUI, an event is generated by the window manager (e.g., Gnome) and placed by the window manager in the event queue of the window within which the button was pressed. The window manager keep tracks of the stacking order of the windows on the screen and determines from the coordinates of the mouse pointer which window generated the event. The event is a structure containing the information needed to specifically handle the mouse click. This includes the values for the x and the y coordinates of the mouse pointer when the mouse button was pressed and the identity of the button (e.g., left, right, or middle).

Technically, the widget that owns the window within which the event was generated, does not receive an event packet; it receives a *signal. GTK distinguishes between events and signals.* Events are derived from the underlying GDK event system, which in turn comes from X Windows (or whatever window manager exists on the system). Signals, which were added to GTK because events were not robust and flexible enough, are a repackaging of the information contained in the event. When a widget receives the signal, it is said to *"emit the signal."* The distinction between events and signals is summarized well by Havoc Pennington[2]:

> "Events are a stream of messages received from the X server. They drive the Gtk main loop; which more or less amounts to "wait for events, process them" (not exactly, it is really more general than that and can wait on many different input streams at once). Events are a Gdk/Xlib concept.
>
> "Signals are a feature of `GtkObject` and its subclasses. They have nothing to do with any input stream; really a signal is just a way to keep a list of callbacks around and invoke them ("emit" the signal). There are lots of details and extra features of course. Signals are emitted by object instances, and are entirely unrelated to the Gtk main loop. Conventionally, signals are emitted "when something changes" about the object emitting the signal.
>
> "Signals and events only come together because `GtkWidget` happens to emit signals when it gets events. This is purely a convenience, so you can connect callbacks to be invoked when a particular widget receives a particular event. There is nothing about this that makes signals and events inherently related concepts, any more than emitting a signal when you click a button makes button clicking and signals related concepts."

To summarize, an event is a notification that is generated by the X Window system, passed up to the application through the GDK library. It corresponds to an actual action such as a mouse movement or a key press. Signals are notifications emitted by widgets. Events can trigger signals. For example, when the mouse button is pressed, a button-press-event is issued by the window manager on the button, and this causes a "clicked" signal to be emitted by the button. Not every event triggers a signal; for example, there are various drag and drop events that are not converted to signals. Conversely, not every signal is the result of a GDK event; widgets can synthesize their own signals.

When GDK generates an event, it is placed in a queue that is processed by the GTK+ main event loop. GTK+ monitors GDK's event queue; for each event received, it decides which widget (if any) should receive the event. The `GtkWidget` base class defines signals for most event types (such as "button_press_event"); it also defines a generic "event" signal.

Signals and events are both notifications to the program that something happened requiring the program's attention, and callback functions are the way that the program can respond to them. The functions are called *callbacks* because the calling is backwards – the callback function is a function in your program's code that is called by the run-time system (the operating system in a sense) when something happens.

You should remember that *objects emit signals* and that the set of signals that an object can emit consists of all signals that it inherits from ancestor classes, as well as signals that are specific to the class of which it is an instance. For example, a button, being a descendant of the widget class, can emit a "hide" signal because all widgets can emit the hide signal, but buttons can also emit the "clicked" signal, which is a button signal in particular.

There are two steps to creating callback functions. The first is to create the function definition. The second is to connect the function to the signal or event that it is supposed to handle and register that connection. We begin by looking at how callbacks are connected to signals and events.

3.4.2 Registering Callback Functions

To connect a callback function to a signal, the `g_signal_connect()` function is used:

[2]Havoc Pennington, "*GTK+/Gnome Application Development*, New Riders Publishing, ISBN 0-7357-0078-8

```
gulong g_signal_connect( gpointer *object_emitting_signal,
                         const gchar *name_of_signal,
                         GCallback function,
                         gpointer function_data );
```

The first argument is the widget that will be emitting the signal, cast to a `GObject*`, and the second is the name of the signal to catch, as a string. The third is the callback function to be called when the signal is caught. It must be cast to a `GCallBack` using the macro `G_CALLBACK()`. The fourth is a pointer to the data to be passed to the callback function. In our program, there are two calls to `g_signal_connect()`:

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

In both cases, the widget to be monitored is the main window, cast as a `GObject`. The `GObject` class defines the destroy signal, and all widgets inherit it. When the window emits the destroy signal, the `destroy()` callback function in this program will be run, with no data passed to it since the fourth parameter is `NULL`.

The second call connects the `delete_event()` function in the program to the "delete-event" signal emitted by the window[3]. When the user clicks the close box in the main window, it generates a delete-event. Any attempt to close the window, such as with the close menu item that appears when you right-click the title-bar, will also generate the delete-event. As noted above, the delete-event generated by GDK is converted to a delete-event signal for GTK.

In this simple program, the return value from the calls to `g_signal_connect()` was not used. The return value is an unsigned integer that acts as an identifier for the signal, like a signal id. If for some reason, you wanted to temporarily block the signal, you could use `g_signal_handler_block()`, passing this id. There are calls to unblock and even disconnect the signal as well, and all require this id.

Later we will see another means of connecting signals that allows us to make clicks on one widget cause a different widget to be closed.

3.4.3 Callback Functions

Callback functions can take several forms, but the simplest and most common is

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

Here, the first argument is a pointer to the widget that emitted the signal and the second argument is a pointer to the data to be used by the function. The actual form of a callback function depends upon the widget and the event or signal; you have to consult the reference manual to determine the parameters.

Callbacks for events are different from callbacks for signals. Event handling callbacks usually have three parameters, one of which is the name of the event. Callbacks for signals may only have two parameters. The easiest way to find the form of the function is to use the GTK+ (version#) Reference Manual: click on the Index hyperlink, and use the browser's find function to look up the exact name of the signal. For example, to find the "delete-event" signal, enter "delete-event". Then click on the hyperlink and the section with the prototype for that callback will open.

In our program, there is a callback for the delete-event and a callback for the destroy signal. The `destroy()` function has two parameters:

[3]The delete-event signal can be written as "delete-event" or "delete_event" – either works.

```
void destroy (GtkWidget *window, gpointer data)
{
    gtk_main_quit ();
}
```

The `destroy()` callback calls `gtk_main_quit()`. `gtk_main_quit()` does several things, such as causing all resources used by the application to be returned to the system, eventually terminating the application. Because no data was connected to the function when it was registered with `g_signal_connect()`, the `data` parameter will have a `NULL` pointer. In this case, the callback had no need for any data.

The `delete_event()` function has three parameters:

```
gboolean delete_event (GtkWidget *window,
                              GdkEvent *event,
                              gpointer data)
{
    return FALSE;
}
```

Because `delete_event()` handles an event, it is passed not only the widget pointer and user-data pointer, but the `GdkEvent` type as its second parameter. In addition, because it handles events, it must return a boolean value. This value indicates whether or not the handler handled the event. If `TRUE` is returned, it is telling GTK+ that it handled the event and GTK+ will do nothing more. If `FALSE` is returned, it is telling GTK+ that the event has not been handled. In this case, GTK+ runs the default handler for the delete-event signal. The default handler will issue a destroy signal on the widget that emitted the event. The destroy signal will be handled by the `destroy()` callback.

In general, your callbacks for events should return `FALSE`, unless you want to handle them yourself and stop GTK+ from continuing to handle it. The default handler for an event always runs last, so returning `FALSE` is a way to let the default handler do its job.

You should experiment with the second program by commenting out the `g_signal_connect()` calls one at a time. You will see that it was not necessary to have a handler for the delete-event signal. Why do we have it? Consider modifying the body of that function by including a bit of code that asks the user whether he or she really wants to quit. If the user clicks an `OK` button, then it would return `FALSE`, otherwise it would return `TRUE`. That is the real reason for having a delete-event handler.

3.4.4 Events and Event Types

It is easy to know when a signal is the result of an event – the signal name will always be of the form "something-event." All signals with names of this form are caused by GDK events. All events are associated with a `GdkWindow`. As mentioned above, they also come to be associated with a `GtkWidget` when the GTK+ main loop passes events from GDK to the GTK+ widget tree.

There are thirty or so different events that GDK can generate. The Appendix contains a list of all of them. For each GDK event type, there is a corresponding GTK signal name. The GDK events and the corresponding signal names are also in the Appendix.

Sometimes you do not know what type of event took place when you receive a signal. For example, the "button-press-event" is emitted by any of three different GDK event types: `GDK_BUTTON_PRESS`, `GDK_2BUTTON_PRESS`, and `GDK_3BUTTON_PRESS`, The callback has to inspect the event structure to determine which took place. For example, for a button press, the callback may be something like this:

```
Listing 3: A callback that inspects the event structure
static gboolean button_press_event (GtkWidget *window,
                                    GdkEvent *event,
                                    gpointer data)
{
    if ( event->type == GDK_BUTTON_PRESS )
        do_single_click(window, data);
    else if ( event->type == GDK_2BUTTON_PRESS )
        do_double_click(window, data);
    else
        do_triple_click(window, data);

    return FALSE;
}
```

3.5 Processing Signals and Events

This is a relatively easy part for you as programmer because GTK takes control of signal and event handling within the `gtk_main()` function. After everything else has been taken care of in your program, it must call `gtk_main()`. This function enters a wait state in which it listens for events and signals directed at your program. Each time an event or signal is transmitted to your program, `gtk_main()` determines its type and which callback function must be invoked. When the callback finishes, `gtk_main()` resumes.

3.6 Quitting

Usually, the signal handling functions will respond to the signal to close the program and issue a `gtk_main_quit()` call, which terminates the program. It is also possible to make `gtk_main_quit()` the callback of the event itself, when the only action is quitting, as in

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

Regardless of how it is done, `gtk_main_quit()` is needed in order to clean up resources and notify the operating system that the process has terminated.

4 A Third GTK+ Example

When a window's size, position, or stack order is changed, GDK generates a `GDK_CONFIGURE` event. A widget can indicate its readiness to handle these events by calling the `gtk_widget_add_events()` function:

```
void gtk_widget_add_events (GtkWidget *widget,
                            gint events);
```

This function tells the GTK library that the given widget should be notified if the given event occurs. The `GDK_CONFIGURE` event is of type `gint`. The program can use a callback function to extract the `x` and `y` coordinates of the upper-left corner of the window out of the event structure, using the two lines

```
x = event->configure.x;
y = event->configure.y;
```

The complete program follows. If you are not familiar with C, the `sprintf()` function prints to the C string which is given as its first argument, so in this case, the values of integers `x` and `y` are written as strings into the 10-character string named `buf`. The `sprintf()` function automatically adds the `NULL` character to the end of `buf`, which is then used as the title of the window.

Listing 4: Using event data to change title bar

```
void frame_callback(GtkWindow *window,
                    GdkEvent *event,
                    gpointer data)
{
    int x, y;
    char buf[10];
    x = event->configure.x;
    y = event->configure.y;
    sprintf(buf, "%d, %d", x, y);
    gtk_window_set_title(window, buf);
}


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

    gtk_init(&argc, &argv);

    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);
    gtk_window_set_default_size(GTK_WINDOW(window), 230, 150);
    gtk_window_set_title(GTK_WINDOW(window), "Simple");
    gtk_widget_add_events(GTK_WIDGET(window), GDK_CONFIGURE);

    g_signal_connect_swapped(G_OBJECT(window), "destroy",
        G_CALLBACK(gtk_main_quit), G_OBJECT(window));

    g_signal_connect(G_OBJECT(window), "configure-event",
        G_CALLBACK(frame_callback), NULL);

    gtk_widget_show(window);
    gtk_main();

    return 0;
}
```

5 A Fourth Example

In many applications, clicking a button has an effect on a widget other than the button itself. For example, a window may have a button that when clicked, causes the an action to be taken which also might include closing the window itself. This is what happens when, in a dialog box, you click the "OK" or "CANCEL" button. If a button's callback function is run when a button is clicked, how could it cause a window to close? The answer is that with what you know so far, there is no way to accomplish this.

GTK+ solves this problem with a special function,

```
    gulong g_signal_connect_swapped(gpointer *object_emitting_signal,
                                    const gchar *name_of_signal,
                                    GCallback function,
                                    gpointer function_data );
```

Unlike `g_signal_connect()`, just prior to starting the callback function, the pointer to the object emitting the signal is swapped with the data pointer, so that the callback receives, in its first argument, the data pointer. By putting a pointer to the window in the data pointer, the callback will be invoked on the window object. If we want to close the window, we can use this function as follows:

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

When the button emits a clicked signal, `gtk_widget_destroy()` will be called on the window passed to it in the fourth parameter.

The third program will create a button widget and connect it to the clicked signal so that it closes the top-level window and thereby terminates the application. To create a button widget with a mnemonic label, we use the following method:

```
button = gtk_button_new_with_mnemonic ("_Close");
```

This puts the word "`Close`" in the button. The underscore preceding the "C" turns Alt-C into a keyboard accelerator that activates the button. The user can type Alt-C to terminate the application. The program is in Listing 3.

The other ways to create a button can be found in the GTK+ Reference Manuals. They include creating a button with a label, or creating a stock button. We will cover stock items later.

Listing 5: GTK+ Program, adding a button

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

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

    gtk_init (&argc, &argv);

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

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

    button = gtk_button_new_with_mnemonic ("_Close");
    gtk_button_set_relief (GTK_BUTTON (button), GTK_RELIEF_NONE);

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

```

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

        gtk_main ();
        return 0;
    }
```

6 GObject Properties

The `GObject` base class has methods that allow you to set and retrieve the properties of an object. It also lets you define arbitrary key-value pairs that can be added to an object. Since all widgets inherit this feature, it is a means of adding user-defined data to widgets. The four relevant methods are `g_object_get()`, `g_object_set()`, `g_object_get_data()`, and `g_object_set_data()`.

To illustrate, to retrieve the "relief" property of a button, you could use the call

```
GtkReliefStyle relief_value;
g_object_get( G_OBJECT(button), "relief", &relief_value, NULL);
```

To set the relief of the button, you would use

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

In general, the functions take a pointer to a `GObject` followed by a NULL-terminated list of property names as strings, and either a variable in which to store the value (for get), or a value (for set).

The ability to add key-value pairs is based on keys represented as strings, and pointers for their values. Each object has a table of strings with associated pointers. The method setting is

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

and for retrieving is

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

If you need to pass data to a widget, you should use these methods. We will see examples of this in the upcoming lessons. To give you some idea of the power of these methods, suppose that you want to store in a drawing widget, a user's choice of default font or a pen color for drawing. You could create a key named "user_font_choice" and use these methods to set its value and retrieve it.

A Appendix

A.1 Object Hierarchy

The level of indentation indicates the depth of the object in the hierarchy; if an object is indented relative to the one immediately preceding it, it is a subclass of the preceding one. This is essentially a tree turned sideways.

```
GObject
    GInitiallyUnowned
        GtkObject
            GtkWidget
                GtkContainer
                    GtkBin
                        GtkWindow
                            GtkDialog
                                GtkAboutDialog
                                GtkColorSelectionDialog
                                GtkFileChooserDialog
                                GtkFileSelection
                                GtkFontSelectionDialog
                                GtkInputDialog
                                GtkMessageDialog
                                GtkPageSetupUnixDialog
                                GtkPrintUnixDialog
                                GtkRecentChooserDialog
                            GtkAssistant
                            GtkPlug
                        GtkAlignment
                        GtkFrame
                            GtkAspectFrame
                        GtkButton
                            GtkToggleButton
                                GtkCheckButton
                                    GtkRadioButton
                            GtkColorButton
                            GtkFontButton
                            GtkLinkButton
                            GtkOptionMenu
                            GtkScaleButton
                                GtkVolumeButton
                        GtkItem
                            GtkMenuItem
                                GtkCheckMenuItem
                                    GtkRadioMenuItem
                                GtkImageMenuItem
                                GtkSeparatorMenuItem
                                GtkTearoffMenuItem
                            GtkListItem
                            GtkTreeItem
                        GtkComboBox
                            GtkComboBoxEntry
                        GtkEventBox
                        GtkExpander
                        GtkHandleBox
                        GtkToolItem
                            GtkToolButton
                                GtkMenuToolButton
                                GtkToggleToolButton
                                    GtkRadioToolButton
                            GtkSeparatorToolItem
                        GtkScrolledWindow
    GtkViewport
GtkBox
    GtkButtonBox
        GtkButtonBox
            GtkHButtonBox
            GtkVButtonBox
        GtkVBox
            GtkColorSelection
            GtkFileChooserWidget
            GtkFontSelection
            GtkGammaCurve
            GtkRecentChooserWidget
        GtkHBox
            GtkCombo
            GtkFileChooserButton
            GtkInfoBar
            GtkStatusbar
    GtkCList
        GtkCTree
    GtkFixed
    GtkPaned
        GtkHPaned
        GtkVPaned
    GtkIconView
    GtkLayout
    GtkList
    GtkMenuShell
        GtkMenuBar
        GtkMenu
            GtkRecentChooserMenu
    GtkNotebook
    GtkSocket
    GtkTable
    GtkTextView
    GtkToolbar
    GtkTree
    GtkTreeView
GtkMisc
    GtkLabel
        GtkAccelLabel
        GtkTipsQuery
    GtkArrow
    GtkImage
    GtkPixmap
GtkCalendar
GtkCellView
GtkDrawingArea
    GtkCurve
GtkEntry
    GtkSpinButton
GtkRuler
    GtkHRuler
    GtkVRuler
GtkRange
```

```
                GtkScale
                    GtkHScale
                    GtkVScale
                GtkScrollbar
                    GtkHScrollbar
                    GtkVScrollbar
            GtkSeparator
                GtkHSeparator
                GtkVSeparator
            GtkHSV
            GtkInvisible
            GtkOldEditable
                GtkText
            GtkPreview
            GtkProgress
                GtkProgressBar
        GtkAdjustment
        GtkCellRenderer
            GtkCellRendererText
                GtkCellRendererAccel
                GtkCellRendererCombo
                GtkCellRendererSpin
            GtkCellRendererPixbuf
            GtkCellRendererProgress
            GtkCellRendererToggle
        GtkFileFilter
        GtkItemFactory
        GtkTooltips
        GtkTreeViewColumn
        GtkRecentFilter
GtkAccelGroup
GtkAccelMap
AtkObject
    GtkAccessible
GtkAction
    GtkToggleAction
        GtkRadioAction
    GtkRecentAction
GtkActionGroup
GtkBuilder
GtkClipboard
GtkEntryBuffer
GtkEntryCompletion
GtkIconFactory
GtkIconTheme
GtkIMContext
    GtkIMContextSimple
    GtkIMMulticontext
GtkListStore
GMountOperation
    GtkMountOperation
GtkPageSetup
GtkPrinter
GtkPrintContext
```

```
            GtkPrintJob
            GtkPrintOperation
            GtkPrintSettings
            GtkRcStyle
            GtkRecentManager
            GtkSettings
            GtkSizeGroup
            GtkStatusIcon
            GtkStyle
            GtkTextBuffer
            GtkTextChildAnchor
            GtkTextMark
            GtkTextTag
            GtkTextTagTable
            GtkTreeModelFilter
            GtkTreeModelSort
            GtkTreeSelection
            GtkTreeStore
            GtkUIManager
            GtkWindowGroup
            GtkTooltip
            GtkPrintBackend
        GInterface
            GtkBuildable
            GtkActivatable
            GtkOrientable
            GtkCellEditable
            GtkCellLayout
            GtkEditable
            GtkFileChooser
            GtkTreeModel
            GtkTreeDragSource
            GtkTreeDragDest
            GtkTreeSortable
            GtkPrintOperationPreview
            GtkRecentChooser
            GtkToolShell
```

A.2 Events in GDK

The following enumeration is defined in the GDK Reference Manual.

```
typedef enum
{
  GDK_NOTHING           = -1,  /* a special code to indicate a null event. */
  GDK_DELETE            = 0,   /* the window manager has requested that the toplevel
                                  window be hidden or destroyed, usually when the
                                  user clicks on a special icon in the title bar. */
  GDK_DESTROY           = 1,   /* the window has been destroyed. */
  GDK_EXPOSE            = 2,   /* all or part of the window has become visible and
                                  needs to be redrawn. */
  GDK_MOTION_NOTIFY     = 3,   /* the pointer (usually a mouse) has moved. */
  GDK_BUTTON_PRESS      = 4,   /* a mouse button has been pressed. */
```

```
GDK_2BUTTON_PRESS     = 5,   /* a mouse button has been double-clicked (clicked
                                twice within a short period of time). Note that
                                each click also generates a GDK_BUTTON_PRESS event.
                             */
GDK_3BUTTON_PRESS     = 6,   /* a mouse button has been clicked 3 times in a short
                                period of time. Note that each click also generates
                                a GDK_BUTTON_PRESS event. */
GDK_BUTTON_RELEASE    = 7,   /* a mouse button has been released. */
GDK_KEY_PRESS         = 8,   /* a key has been pressed. */
GDK_KEY_RELEASE       = 9,   /* a key has been released. */
GDK_ENTER_NOTIFY      = 10,  /* the pointer has entered the window. */
GDK_LEAVE_NOTIFY      = 11,  /* the pointer has left the window. */
GDK_FOCUS_CHANGE      = 12,  /* the keyboard focus has entered or left the window.
                             */
GDK_CONFIGURE         = 13,  /* the size, position or stacking order of the window
                                has changed. Note that GTK+ discards these events
                                for GDK_WINDOW_CHILD windows. */
GDK_MAP               = 14,  /* the window has been mapped. */
GDK_UNMAP             = 15,  /* the window has been unmapped. */
GDK_PROPERTY_NOTIFY   = 16,  /* a property on the window has changed or was
                                deleted. */
GDK_SELECTION_CLEAR   = 17,  /* the application has lost ownership of a
                                selection. */
GDK_SELECTION_REQUEST = 18,  /* another application has requested a selection. */
GDK_SELECTION_NOTIFY  = 19,  /* a selection has been received. */
GDK_PROXIMITY_IN      = 20,  /* an input device has moved into contact with a
                                sensing surface. */
GDK_PROXIMITY_OUT     = 21,  /* an input device has moved out of contact with a
                                sensing surface. */
GDK_DRAG_ENTER        = 22,  /* the mouse has entered the window while a drag is
                                in progress. */
GDK_DRAG_LEAVE        = 23,  /* the mouse has left the window while a drag is in
                                progress. */
GDK_DRAG_MOTION       = 24,  /* the mouse has moved in the window while a drag is
                                in progress. */
GDK_DRAG_STATUS       = 25,  /* the status of the drag operation initiated by the
                                window has changed. */
GDK_DROP_START        = 26,  /* a drop operation onto the window has started. */
GDK_DROP_FINISHED     = 27,  /* the drop operation initiated by the window has
                                completed. */
GDK_CLIENT_EVENT   = 28,  /* a message has been received from another
                                application. */
GDK_VISIBILITY_NOTIFY = 29,  /* the window visibility status has changed. */
GDK_NO_EXPOSE         = 30,  /* indicates that the source region was completely
                                available when parts of a drawable were copied. */
GDK_SCROLL            = 31,  /* the scroll wheel was turned. */
GDK_WINDOW_STATE      = 32,  /* the state of a window has changed. See
                                GDKWindowState for the possible window states. */
GDK_SETTING           = 33,  /* a setting has been modified. */
GDK_OWNER_CHANGE      = 34,  /* the owner of a selection has changed. */
GDK_GRAB_BROKEN       = 35,  /* a pointer or keyboard grab was broken. Added 2.8 */
GDK_DAMAGE            = 36,  /* the content of the window has been changed.
                                This event type was added in 2.14. */
GDK_EVENT_LAST               /* helper variable for decls */
```

```
    }  GDKEventType;
```

The following table shows which GDK events are converted to GTK+ signals. The column labeled "Propagated?" indicates whether or not that event is propagated to the parent widget. Propagation is covered in a later chapter. Roughly, if an event is propagated and the widget does not handle the event, then the event is sent to the parent widget. If it is not propagated and not handled, then the event is ignored. The column labeled "Grabbed?" will be explained later.

Event Type	GtkWidget Signal	Propagated?	Grabbed?
GDK_DELETE	delete_event	No	No
GDK_DESTROY	destroy_event	No	No
GDK_EXPOSE	expose_event	No	No
GDK_MOTION_NOTIFY	motion_notify_event	Yes	Yes
GDK_BUTTON_PRESS	button_press_event	Yes	Yes
GDK_2BUTTON_PRESS	button_press_event	Yes	Yes
GDK_3BUTTON_PRESS	button_press_event	Yes	Yes
GDK_BUTTON_RELEASE	button_release_event	Yes	Yes
GDK_KEY_PRESS	key_press_event	Yes	Yes
GDK_KEY_RELEASE	key_release_event	Yes	Yes
GDK_ENTER_NOTIFY	enter_notify_event	No	Yes
GDK_LEAVE_NOTIFY	leave_notify_event	No	Yes
GDK_FOCUS_CHANGE	focus_in_event, focus_out_event	No	Yes
GDK_CONFIGURE	configure_event	No	No
GDK_MAP	map_event	No	No
GDK_UNMAP	unmap_event	No	No
GDK_PROPERTY_NOTIFY	property_notify_event	No	No
GDK_SELECTION_CLEAR	selection_clear_event	No	No
GDK_SELECTION_REQUEST	selection_request_event	No	No
GDK_SELECTION_NOTIFY	selection_notify_event	No	No
GDK_PROXIMITY_IN	proximity_in_event	Yes	Yes
GDK_PROXIMITY_OUT	proximity_out_event	Yes	Yes
GDK_CLIENT_EVENT	client_event	No	No
GDK_VISIBILITY_NOTIFY	visibility_notify_event	No	No
GDK_NO_EXPOSE	no_expose_event	No	No