Chapter 9. Handling input from the Keyboard

It is desirable to make as many operations of your application accessible through the keyboard as possible. Many users will prefer using the keyboard instead of having to constantly resort to the mouse, and making your program able to be controlled from the keyboard will increase its accessibility for disabled users. GTK+ automatically handles keyboard input in such widgets `GtkText` and `GtkEntry`, and also takes care of such standard keyboards as allowing the arrow keys to navigate within a `GtkCList` widget. However, other types of keyboard input need to be customized by the application. We saw one example of this in Chapter 5, where we set up menu accelerators using `GtkItemFactory`. These keyboard bindings were set up in an *accelerator group*, which allows setting up keyboard bindings that are global within a toplevel window. GTK+ also allows defining user configurable keys for particular widgets using a *key binding* mechanism, and if we need to special handling we can handle raw key events directly.

As was mentioned when we described `GtkItemFactory`, there are two values associated with a key press, the *key symbol* and the set of *modifiers*. The key symbol describes an abstract key on the keyboard. It is represented by an enumeration value such as `GDK_a`, `GDK_F1`, or conceivably, `GDK_Thai_ngognu`. (You probably don't have the last key on your keyboard, but it would be found on some Thai keyboards.) The complete set of these values is found in the header file `gdk/gdkkeysyms.h`. The modifiers associated with a key press are the set of keys such as Shift and Control held down when the key was pressed. GTK+ represents the set of modifiers as a bitwise combination of a number of flags. The possible flags are:

```
GDK_CONTROL_MASK
GDK_SHIFT_MASK
GDK_MOD1_MASK
GDK_MOD2_MASK
GDK_MOD3_MASK
GDK_MOD4_MASK
GDK_MOD5_MASK
```

`GDK_MOD1_MASK` traditionally is used for the Alt key, and in fact, the `<Alt>` tag in an itemfactory accelerator description maps to this flag. The mapping of `GDK_MOD2_MASK` throughn `GDK_MOD5_MASK` to keyboard keys is basically system-dependent and cannot be relied on.

9.1. Accelerator Groups

In Chapter 5 we created an accelerator group, passed it to our item factory, and then attached it to the toplevel window with `gtk_window_add_accelerator_group()`. In this case, all the acclerators were added to the accelerator group by the item factory. In some cases, we may also want to add accelerators to widgets directly. As an example of this, in our addressbook example, we set up keyboard accelerators so that the user can jump to a particular widget. For each pair of a label and an entry, we underline a character in the label and set up an accelerator so that pressing Alt plus that character jumps to the entry.

This is done in the `make_pair()` utility function which packs a label together with another widget into a `GtkHBox` widget.

```
GtkWidget *
make_pair (gchar *labeltext,
           GtkAccelGroup  *accel_group,
           GtkWidget *widget)
{
  GtkWidget *util_hbox;
  GtkWidget *label;
  guint keysym;

  util_hbox = gtk_hbox_new (FALSE, 2);

  label = gtk_label_new (NULL);
  gtk_box_pack_start (GTK_BOX (util_hbox), label,  FALSE, FALSE, 0);

  gtk_box_pack_start (GTK_BOX (util_hbox), widget, TRUE, TRUE, 0);

  keysym = gtk_label_parse_uline (GTK_LABEL (label), labeltext);
  if (keysym)
    gtk_accel_group_add (accel_group, keysym, GDK_MOD1_MASK, 0,
                         GTK_OBJECT (widget), "grab-focus");

  return util_hbox;
}
```

A call to `make_pair()` looks like:

```
pair = make_pair ("_Month:", book->accel_group„ book->month_combo);
```

the accelerator is specified by putting '_' character before the key corresponding the accelerator in the `labeltext` argument, just as we did for for the item factory. Inside *make_pair()*, we first determine the accelerator key using the convenience function:

```
guint gtk_label_parse_uline(GtkLabel *label, const gchar *string);
```

This function scans the `string` argument for '_' characters, draws underlines in the corresponding positions, and then returns the key symbol corresponding to the first underline character in the string. Once we have the key symbol, we add the accelerator using

```
void gtk_accel_group_add(GtkAccelGroup *accel_group, guint accel_key,
GdkModifierType accel_mods, GtkAccelFlags accel_flags, GtkObject *ob-
ject, const gchar *accel_signal);
```

The `accel_key` and `accel_mods` arguments specify the key and modifier combination that activate the keyboard accelerator. The `accel_symbol` argument specifies a signal to emit on `object` when the accelerator is pressed. This signal must be an *action* signal; action signals are specially flagged when they are created. Their special characteristic is that they can be directly emitted without any special setup. The signal must also not have any parameters, since there is no way of specifying parameters here. The signal we use here is *GtkWidget::grab-focus*, which simply sets the keyboard focus to the given widget (the focused widget is the widget to which keyboard input goes by default. We'll describe this more later in the chapter.) The `accel_flags` argument is used to specify some options that affect the way menu accelerators work. We can safely leave it as 0 here.

One unusual feature of GTK+'s accelerators is that menu accelerators can be changed on the fly. The user does that by simply pressing the new accelerator key while the mouse is over the menu item. GTK+ automatically takes care of supporting this feature while the application is running, but a bit of intervention from the application programmer is needed to store the values from one invocation of an application to another. (Note that the GNOME application framework automatically provides this support, so if you are using GNOME you don't need to worry about this issue.) `GtkItemFactory` provides a pair of functions for writing the current menu accelerators to a file and loading them back:

```
void gtk_item_factory_dump_rc(const gchar *file_name, GtkPatternSpec
*path_pspec, gboolean modified_only);
```

```
void gtk_item_factory_parse_rc(const gchar *file_name);
```

The `path_pspec` argument to `gtk_item_factory_dump_items` provides a way of dumping only some of the accelerators based on the item factory pathnames, while *mod-*

ified_only specifies whether all accelerators, or only the ones that were modified by the user should be written out. In most cases, using NULL and TRUE for these two values should work fine.

After initializing GTK+ but before creating our interface, we call gtk_item_factory_parse_rc:

```
accel_file_name = g_strconcat (g_get_home_dir(), "/", ".addressbook.accels", NULL);
gtk_item_factory_parse_rc (accel_file_name);
```

and at the end of our main() function, just before exiting, we callgtk_item_factory_dump_rc: gtk_item_factory_dump_rc (accel_file_name, NULL, TRUE); g_free (accel_file_name); You can try, for example, changing the accelerators for the “Back” and “Forward” entries in the “Entry”menu to <Control>b and <Control>f. If you do that, the following lines will be written to the file .addressbook.accels in your home directory:

```
(menu-path "<main>/Entry/Back" "<Control>b")
(menu-path "<main>/Entry/Forward" "<Control>f")
```

And if you quit and restart the addressbook program, your accelerators will be remembered.

9.2. Key bindings

Accelerators provide a way of connecting a key press on a toplevel window to an action on a particular widget. There are also key bindings that only take effect in particular widgets. As an example, once you activate a menu, you can navigate around the menu using the arrow keys. These types of keyboard control are handled by creating *key bindings*. Key bindings are similar to accelerators in that they bind a key and modifier combination to action signals, but they provide a somewhat different set of functionality that is more appropriate to their intended use. The following table shows some of the differences between accelerator groups and key bindings.

Accelerator Groups	Key Bindings
Connects key press on toplevel to signal on some widget.	Connects key press on a to signal(s) on the same widget.
Only one signal is activated.	Any number of signals can be activated.
Signal cannot have parameters.	Signals can have parameters.

Can be changed by the user on the fly.	Can only be changed through code or a RC file.

While key bindings fill a similar role to accelerator groups, in many ways, the manner in which they are implemented is in many ways similar to GTK+'s style system. We define groups of key bindings called *binding sets* that are analagous to a RC styles, and attach them to them to widgets using the same types of attachment (matching against class, class path, or widget path) that we used for RC styles in Chapter 8. Just as with styles, the definitions and attachments can either be done in our application's code or in an RC file.

The ability to define bindings in a RC file means that it is possible for the user to customize the bindings that are used for widgets without having to modify GTK+. As an example, consider the example mentioned above where arrow keys are bound to navigatation in menus. emacs users might prefer to navigate about in menus using the emacs keys for directional motion: <Control>b, <Control>f, <Control>n, and <Control>p. They that make the change by simply adding the following lines to their RC files:

```
binding "emacs-menus" {
  bind "<control>b" { "move-current" ( parent ) }
  bind "<control>f" { "move-current" ( child ) }
  bind "<control>p" { "move-current" ( prev ) }
  bind "<control>n" { "move-current" ( next ) }
}

class "GtkMenuShell" binding "emacs-menus"
```

This code declares a binding set called `"emacs-menus"`, which contains bindings for the four directional keys. Each directional key is bound to the `GtkMenuShell::move-current` signal, an action signal that the `GtkMenuShell` class defines for this purpose. (The `GtkMenuShell` class is a common base class for `GtkMenu` and `GtkMenuBar` widgets.) The signal takes a single parameter indicating the direction of motion — this distinguishes the four bindings. The parameter to the signal is an enumeration value of type GtkMenuDirectionType, which, in C, has the values `GTK_MENU_DIR_PARENT`, `GTK_MENU_DIR_CHILD`, `GTK_MENU_DIR_NEXT`, and `GTK_MENU_DIR_PREV`. GTK+'s RC parsing code provides the convenient abbreveviations you see in the RC file excerpt.

Once we've defined the binding set, we use a `class` statement to attach the binding set to all widgets that derive from the `GtkMenuShell` class. We can also use the `widget` and `widget_class` statements, as was demonstrated for RC files, though they tend not to be very useful for binding sets. When specifying binding sets via `class`, `widget`, or `widget_class` patterns, we can also give a priority for the pattern, using the syntax

```
class "GtkMenuShell" binding : rc "my-menus"
```

The possible priorities are, in order of increasing priority, `gtk`, `application`, and `rc`. When finding the binding set that applies to a given keystroke, the binding sets attached to the widget are searched in order of priority, from highest to lowest. When we specify a pattern in a RC file, the default priority is the highest priority: `rc`. The bindings built into GTK+ are given the `gtk` priority. The priority system ensures that even if the RC file is parsed before a class is initialized, the bindings from the RC file will override the default bindings. (For bindings of equal priority, later patterns override earlier ones). The `application` priority is a priority intermediate betwen `gtk` and `rc` which may be useful for applications that want to establish bindings that will override the default bindings but will not override the user's bindings.

The most common time to define new bindings from a program is when setting up the default bindings for a new widget. However, it may occasionally be useful to create bindings at other times. As an example of this, we'll add an extra feature to the editable widgets in our addressbook: each time an editable widget gets the focus, we'll save the contents of the widget, and then, if the user hits the ESC key after changing the contents, we'll restore the contents of the widget to value when the widget got the focus. The code to do this is shown in Figure 9-1.

Figure 9-1. Adding a revert capability to editable widgets

```
static gboolean
editable_focus_in_cb (GtkWidget *widget, GdkEventFocus *key)
{
  gchar *str = gtk_editable_get_chars (GTK_EDITABLE (widget), 0, -1);
  gtk_object_set_data_full (GTK_OBJECT (widget), "editable-revert-data",
                              str, (GDestroyNotify)g_free);
  return TRUE;
}

static gboolean
editable_key_press_cb (GtkWidget *widget, GdkEventKey *event)
{
  if (gtk_bindings_activate (GTK_OBJECT (widget), event->keyval, event-
>state))
    {
      gtk_signal_emit_stop_by_name (GTK_OBJECT (widget), "key_press_event");
      return TRUE;
    }
  else
```

```
    return FALSE;
}

static void
revert_cb (GtkEditable *editable, gpointer data)
{
  gchar *text = gtk_object_get_data (GTK_OBJECT (editable), "editable-
revert-data");
  gint position = 0;

  if (text)
    {
      gtk_editable_delete_text (editable, 0, -1);
      gtk_editable_insert_text (editable, text, strlen (text), &position);
    }
}

void
add_editable_revert (GtkEditable *editable)
{
  static gboolean initialized = FALSE;

  g_return_if_fail (GTK_IS_EDITABLE (editable));

  if (!initialized)
    {
      GtkBindingSet *binding_set;
      initialized = TRUE;

      gtk_object_class_user_signal_new (gtk_type_class (gtk_editable_get_type ()),
                                        "revert",
                                        GTK_RUN_ACTION,
                                        gtk_signal_default_marshaller,
                                        GTK_TYPE_NONE /* 'void' re-
turn.*/ ,

                                        0 /* No extra arguments */);

      binding_set = gtk_binding_set_by_class (gtk_type_class (gtk_editable_get_type ()));

      gtk_binding_entry_add_signal (binding_set, GDK_Escape, 0, "re-
vert", 0);
    }

  gtk_signal_connect (GTK_OBJECT (editable), "focus_in_event",
                      GTK_SIGNAL_FUNC (editable_focus_in_cb), NULL);
```

```
  gtk_signal_connect (GTK_OBJECT (editable), "revert",
                      GTK_SIGNAL_FUNC (revert_cb), NULL);
  gtk_signal_connect (GTK_OBJECT (editable), "key_press_event",
                      GTK_SIGNAL_FUNC (editable_key_press_cb), NULL);
}
```

This code provides a function `add_editable_revert()` that adds the revert capability. The first thing that this function does is checks if it is the first time that the function has been called, and if so, performs some setup. The setup consists of first creating a *user signal*. A user signal is a signal that is created by an application instead of by a widget class. It is associated with a particular class, but since the class doesn't know about its existence there is no default handler for it. Creating the user signal is done using the function:

guint **gtk_object_class_user_signal_new**(GtkObjectClass **klass*, const gchar **name*, GtkSignalRunType *signal_flags*, GtkSignalMarshaller *marshaller*, GtkType *return_val*, guint *nparams*, ...);

The *klass* and *name* specify the object class in which to create the new signal and the name of the new signal. The *signal_flags* argument is a bitfield which gives options for the newly created signal. The value used here, `GTK_RUN_ACTION`, indicates that it is an action signal suitable for being used for an accelerator or key binding. For details about the *marshaller*, *return_value* and *nparams* arguments, see the documentation for creating signals. The values specified in our example are appropriate for creating a signal with no arguments other than the standard *object* and *user_data* arguments, and no return value.

Now that we've added a `::revert` signal to the `GtkEditable`, we set up binding of the Escape key to that signal using:

void **gtk_binding_entry_add_signal**(GtkBindingSet **binding_set*, guint *keyval*, guint *modifiers*, const gchar **signal_name*, guint *n_args*, ...);

This function binds a single signal to the key combination identified by the *keyval* and *modifiers* arguments. If you want to set things up so that more than one signal is activated for the key and modifier combination, you can call this function multiple times with the same *keyval* and *modifiers* arguments. The signal name is given by the *signal_name* argument, while the signal parameters are specified by a variable argument list. We get the binding set in which we establish our binding using the convenience function:

```
GtkBindingSet *gtk_binding_set_by_class(gpointer object_class);
```

This function returns a binding set that is bound to the specified class with the priority `gtk`. If the function is called multiple times for the same class, it will return the same binding set.

Most of the rest of the code in the example is concerned with actually implementing the "revert" action. When we get a `GtkWidget::focus_in_event` signal for our widget, we retrieve the text from the editable widget and store it as object data on the widget. (We use `gtk_object_set_data_full()` so that we can use the *destroy* parameter to pass in a function to free the data when the widget is destroyed, or when we set a different set of data on the same key.) In the handler for `GtkEditable::revert`, we retrieve the text that we saved, and set it as the text for the editable widget.

The last thing that we do is a workaround for the fact that `GtkEditable` doesn't actually use the key binding system itself, but instead intercepts the key presses directly. To get around this problem, we connect our own handler to `GtkWidget::key_press_event`, which activates the bindings for the widget and if a binding was found (indicated by a `TRUE` return value from `gtk_bindings_activate`, we stop the emission of the signal and return `TRUE` from our event handler to suppress further handling of the key-press signal.

9.3. Handling Key Events directly

The accelerator and key binding mechanism provide powerful mechanisms for handling key presses. However, they don't cover all possible types of keyboard input and there are also some cases in which these mechanisms are simply too complicated. As an example of the latter case, consider making Escape close a dialog. While we could do this by creating an accelerator group, and adding an accelerator to that, that's a fair bit of overhead. Instead, we can just catch the raw keypress event and react to that.

We catch a key press event by connecting to the `GtkWidget::key_press_event` signal. A handler for this signal has the prototype:

```
gboolean key_press_event_cb(GtkWidget *widget, GdkEventKey *event, gpointeruser_data);
```

this prototype adds an *event* argument to the standard *widget* and *user_data* arguments. The *event* argument points to a GdkEventKey structure which contains additional information about the event. This structure looks like, inpart:

```
struct _GdkEventKey
{
  [...]
```

```
  guint state;
  guint keyval;
  gchar *string;
};
```

The `state` field is a bitfield containing the modifiers that were held down when they key was pressed. the `field` gives the key symbol that was pressed. The `string` field gives a string which is the translation of the keypress. This may simply be a direct translation of the key, but when the user is inputting internationalized text, many key strokes may be combined into a single string of characters. A simple example of this is the use of dead keys for typing accented characters. In order to get the character ü, the user would first type a special "dead key" then the u key. An application listening to keypress events would first see an event with the key symbol GDK_dead_diaeresis and the `string` field of the even structure set to `NULL`, and then another event with the key symbol GDK_u and the `string` field set to "ü".

The return value from the signal indicates whether the keystroke was handled or not by the handler or not. The return value for the signal is used in some cases in deciding whether to do any more processing for the keystroke. For instance, if the focus widget handles the keystroke, then the global accelerators will not be activated. (See Figure 9-2.) Note that the return value of your *handler* will not be the return value for the signal unless it is the last handler that is run. You can ensure that this is the case by using `gtk_signal_emit_stop_by_name()`.

We set up Escape to close our message dialogs with the following code:

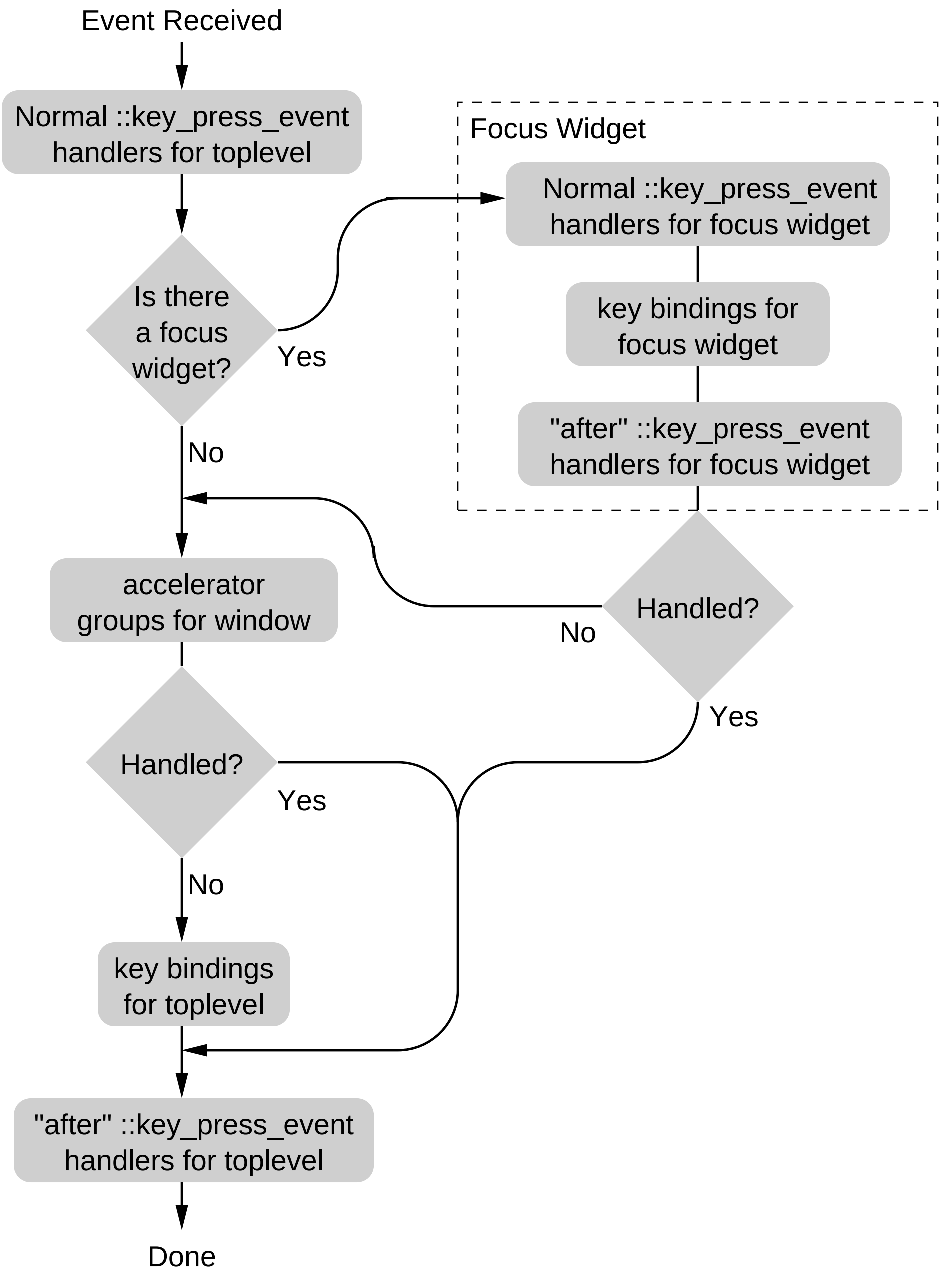
```
static gboolean
msgbox_key_press_cb (GtkWidget *widget, GdkEventKey *event, gpointer data)
{
  if (event->keyval == GDK_Escape)
    {
      gtk_signal_emit_stop_by_name (GTK_OBJECT (widget), "key_press_event");
      gtk_object_destroy (GTK_OBJECT (widget));
      return TRUE;
    }

  return FALSE;
}


  gtk_signal_connect (GTK_OBJECT (dialog), "key_press_event",
                      GTK_SIGNAL_FUNC (msgbox_key_press_cb), NULL);
```

When we receive a key press event for the dialog, we check to see if if the key pressed was the Escape key, and if so, destroy the dialog, stop the signal emission and return `TRUE` to indicate that the key was handled. Otherwise, we return `FALSE` and let normal handling take place.

9.4. Interactions between the keypress mechanisms

So far in this chapter, we've discussed three different mechanisms for handling key presses; when we have a toplevel window with multiple widgets, and several of these mechanisms in effect, how is it decided who actually gets the key press? Figure 9-2 gives a flowchart of the steps that are taken to determine this. While this figure may look a little complicated, in practice things work in a pretty straightforward manner.

Figure 9-2. An overview of the handling of a key press



There are two widgets involved for every key press. The first widget is the toplevel widget in which the key press occurs. The second widget is the *focus* widget within the toplevel widget. This is the widget that the appears to receive the user's keystrokes. When the user clicks inside an entry, a dark border is drawn around the entry indicating it has the focus, and all keystrokes are sent to that widget until the user explicitely moves the focus elsewhere by clicking in another widget or using the Tab key to move the focus.

A program can set the focus to a particular widget using the function:

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

We saw another way of moving the focus earlier in the chapter when we set up accelerators to emit the `GtkWidget::grab_focus` signal.

All keystrokes received by GTK+ are first sent to the toplevel widget. So, the way to make sure you get a keystroke before anybody else is to attach to the `::key_press_event` signal on the toplevel widget. The default handler for key presses on a toplevel window does a number of things. First, it checks to see if there is a focus widget for the toplevel, and if so, forwards the keystroke to that widget. If there is no focus widget, or the focus widget does not handle the keystroke (indicated from the return value of the signal emission), then the toplevel procedes to check for an accelerator for the widget, then check key bindings if no accelerator was found.

Within the focus widget, the handling is similar, except that accelerators are not checked. For both the toplevel window and the focus widget it is possible to install signal handlers with `gtk_signal_connect_after()` that will be called after the default handler, so this is shown in the diagram. It is, however, seldom useful to do that, since by the point these handlers get called, the keystroke has probably been handled in some other way.

Most applications will use accelerators as there primary method of handling key input. Some applications, especially applications such as games which need non-standard handling of keys, will connect directly to `GtkWidget::key_press_event`. The binding mechanism provides a very flexible mechanism for establishing customizable bindings, but unless you are writing your own widgets, you will seldom need to deal directly with this facililty.