CHAPTER 9

Menus and Toolbars

This chapter will teach you how to create pop-up menus, menu bars, and toolbars. You will begin by creating each manually, so you learn how the widgets are constructed. This will give you a firm understanding of all of the concepts on which menus and toolbars rely.

After you understand each widget, you will be introduced to `GtkUIManager`, which allows you to dynamically create menus and toolbars through custom XML files. Each user interface file is loaded, and each element applied to a corresponding `action` object, which tells the item how it will be displayed and how it will act.

In this chapter, you will learn the following:

- How to create pop-up menus, menu bars, and toolbars
- How to apply keyboard accelerators to menu items
- What the `GtkStatusBar` widget is and how you can use it to provide more information to the user about a menu item
- What types of menu and toolbar items are provided by GTK+
- How to dynamically create menus and toolbars with UI files
- How to create custom stock items with `GtkIconFactory`

Pop-up Menus

You will begin this chapter by learning how to create a pop-up menu. A pop-up menu is a `GtkMenu` widget that is displayed to the user when the right mouse button is clicked while hovering above certain widgets. Some widgets, such as `GtkEntry` and `GtkTextView`, already have pop-up menus built into the widget by default.

If you want to change the pop-up menu of a widget that offers one by default, you should edit the supplied `GtkMenu` widget in the pop-up callback function. For example, both `GtkEntry` and `GtkTextView` have a `populate-popup` signal, which receives the `GtkMenu` that is going to be displayed. You can edit this menu in any way you see fit before displaying it to the user.

Creating a Pop-up Menu

For most widgets, you will need to create your own pop-up menu. In this section, you are going to learn how to supply a pop-up menu to a `GtkProgressBar` widget. The pop-up menu we are going to implement is presented in Figure 9-1.

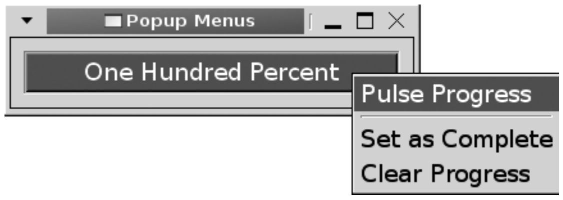


Figure 9-1. *A simple pop-up menu with three menu items*

The three pop-up menu items are used to pulse the progress bar, set it as 100 percent complete, and clear it. You will notice that, in Listing 9-1, an event box contains the progress bar. Because `GtkProgressBar`, like `GtkLabel`, is not able to detect GDK events by itself, we need to catch `button-press-event` signals using an event box.

Listing 9-1. *Simple Pop-up Menu (popupmenus.c)*

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

static void create_popup_menu (GtkWidget*, GtkWidget*);
static void pulse_activated (GtkMenuItem*, GtkProgressBar*);
static void clear_activated (GtkMenuItem*, GtkProgressBar*);
static void fill_activated (GtkMenuItem*, GtkProgressBar*);
static gboolean button_press_event (GtkWidget*, GdkEventButton*, GtkWidget*);

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

  gtk_init (&argc, &argv);

  window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  gtk_window_set_title (GTK_WINDOW (window), "Pop-up Menus");
  gtk_container_set_border_width (GTK_CONTAINER (window), 10);
  gtk_widget_set_size_request (window, 250, -1);
  /* Create all of the necessary widgets and initialize the pop-up menu. */
  menu = gtk_menu_new ();
  eventbox = gtk_event_box_new ();
  progress = gtk_progress_bar_new ();
  gtk_progress_bar_set_text (GTK_PROGRESS_BAR (progress), "Nothing Yet Happened");
  create_popup_menu (menu, progress);

  gtk_progress_bar_set_pulse_step (GTK_PROGRESS_BAR (progress), 0.05);
  gtk_event_box_set_above_child (GTK_EVENT_BOX (eventbox), FALSE);

  g_signal_connect (G_OBJECT (eventbox), "button_press_event",
                    G_CALLBACK (button_press_event), menu);

  gtk_container_add (GTK_CONTAINER (eventbox), progress);
  gtk_container_add (GTK_CONTAINER (window), eventbox);

  gtk_widget_set_events (eventbox, GDK_BUTTON_PRESS_MASK);
  gtk_widget_realize (eventbox);

  gtk_widget_show_all (window);
  gtk_main ();
  return 0;
}

/* Create the pop-up menu and attach it to the progress bar. This will make sure
 * that the accelerators will work from application load. */
static void
create_popup_menu (GtkWidget *menu,
                   GtkWidget *progress)
{
  GtkWidget *pulse, *fill, *clear, *separator;

  pulse = gtk_menu_item_new_with_label ("Pulse Progress");
  fill = gtk_menu_item_new_with_label ("Set as Complete");
  clear = gtk_menu_item_new_with_label ("Clear Progress");
  separator = gtk_separator_menu_item_new ();

  g_signal_connect (G_OBJECT (pulse), "activate",
                    G_CALLBACK (pulse_activated), progress);
  g_signal_connect (G_OBJECT (fill), "activate",
                    G_CALLBACK (fill_activated), progress);
  g_signal_connect (G_OBJECT (clear), "activate",
                    G_CALLBACK (clear_activated), progress);
  gtk_menu_shell_append (GTK_MENU_SHELL (menu), pulse);
  gtk_menu_shell_append (GTK_MENU_SHELL (menu), separator);
  gtk_menu_shell_append (GTK_MENU_SHELL (menu), fill);
  gtk_menu_shell_append (GTK_MENU_SHELL (menu), clear);

  gtk_menu_attach_to_widget (GTK_MENU (menu), progress, NULL);
  gtk_widget_show_all (menu);
}
```

In most cases, you will want to use `button-press-event` to detect when the user wants the pop-up menu to be shown. This allows you to check whether the right mouse button was clicked. If the right mouse button was clicked, `GdkEventButton`'s `button` member will be equal to `3`.

However, `GtkWidget` also provides the `popup-menu` signal, which is activated when the user presses built-in key accelerators to activate the pop-up menu. Most users will use the mouse to activate pop-up menus, so this is not usually a factor in GTK+ applications. Nevertheless, if you would like to handle this signal as well, you should create a third function that displays the pop-up menu that is called by both callback functions.

New menus are created with `gtk_menu_new()`. The menu is initialized with no initial content, so the next step is to create menu items.

In this section, we will cover two types of menu items. The first is the base class for all other types of menu items, `GtkMenuItem`. There are three initialization functions provided for `GtkMenuItem`: `gtk_menu_item_new()`, `gtk_menu_item_new_with_label()`, and `gtk_menu_item_new_with_mnemonic()`.

```
GtkWidget* gtk_menu_item_new_with_label (const gchar *label);
```

In most cases, you will not need to use the `gtk_menu_item_new()`, because a menu item with no content is not of much use. If you use that function to initialize the menu item, you will have to construct each aspect of the menu in code instead of allowing GTK+ to handle the specifics.

■ **Note** Menu item mnemonics are not the same thing as keyboard accelerators. A mnemonic will activate the menu item when the user presses Alt and the appropriate alphanumeric key while the menu has focus. A keyboard accelerator is a custom key combination that will cause a callback function to be run when the combination is pressed. You will learn about keyboard accelerators for menus in the next section.

The other type of basic menu item is `GtkSeparatorMenuItem`, which places a generic separator at its location. You can use `gtk_separator_menu_item_new()` to create a new separator menu item.

Separators are extremely important when designing a menu structure, because they organize menu items into groups so that the user can easily find the appropriate item. For example, in the File menu, menu items are often organized into groups that open files, save files, print files, and close the application. Rarely should you have many menu items listed without a separator in between them (e.g., a list of recent files might appear without a separator). In most cases, you should group similar menu items together and place a separator between adjacent groups.

After the menu items are created, you need to connect each menu item to the `activate` signal, which is emitted when the user selects the item. Alternatively, you can use the `activate-item` signal, which will additionally be emitted when a submenu of the given menu item is displayed. There will be no discernable difference between the two unless the menu item expands into a submenu.

Each `activate` and `activate-item` callback function receives the `GtkMenuItem` widget that initiated the action and any data you need to pass to the function. In Listing 9-2, three menu item callback functions are provided. They are used to pulse the progress bar, fill it to 100 percent complete, and clear all progress.

Now that you have created all of the menu items, you need to add them to the menu. `GtkMenu` is derived from `GtkMenuShell`, which is an abstract base class that contains and displays submenus and menu items. Menu items can be added to a menu shell with `gtk_menu_shell_append()`. This function appends each item to the end of the menu shell.

```
void gtk_menu_shell_append (GtkMenuShell *menu_shell,
                            GtkWidget *child);
```

Additionally, you can use `gtk_menu_shell_prepend()` or `gtk_menu_shell_insert()` to add a menu item to the beginning of the menu or insert it into an arbitrary position respectively. Positions accepted by `gtk_menu_shell_insert()` begin with an index of zero.

After setting all of the `GtkMenu`'s children as visible, you should call `gtk_menu_attach_to_widget()` so that the pop-up menu is associated to a specific widget. This function accepts the pop-up menu and the widget it will be attached to.

```
void gtk_menu_attach_to_widget (GtkMenu *menu,
                                GtkWidget *attach_widget,
                                GtkMenuDetachFunc detacher);
```

The last parameter of `gtk_menu_attach_widget()` accepts a `GtkMenuDetachFunc`, which can be used to call a specific function when the menu is detached from the widget.

Pop-up Menu Callback Functions

After creating the necessary widgets, you need to handle the `button-press-event` signal, which is shown in Listing 9-2. In this example, the pop-up menu is displayed every time the right mouse button is clicked on the progress bar.

Listing 9-2. *Callback Functions for the Simple Pop-up Menu (popupmenus.c)*

```
static gboolean
button_press_event (GtkWidget *eventbox,
                    GdkEventButton *event,
                    GtkWidget *menu)
{
  if ((event->button == 3) && (event->type == GDK_BUTTON_PRESS))
  {
    gtk_menu_popup (GTK_MENU (menu), NULL, NULL, NULL, NULL,
                    event->button, event->time);
    return TRUE;
  }

  return FALSE;
}

static void
pulse_activated (GtkMenuItem *item,
                 GtkProgressBar *progress)
{
  gtk_progress_bar_pulse (progress);
  gtk_progress_bar_set_text (progress, "Pulse!");
}

static void
fill_activated (GtkMenuItem *item,
                GtkProgressBar *progress)
{
  gtk_progress_bar_set_fraction (progress, 1.0);
  gtk_progress_bar_set_text (progress, "One Hundred Percent");
}

static void
clear_activated (GtkMenuItem *item,
                 GtkProgressBar *progress)
{
  gtk_progress_bar_set_fraction (progress, 0.0);
  gtk_progress_bar_set_text (progress, "Reset to Zero");
}
```

In the `button-press-event` callback function in Listing 9-2, you can use `gtk_menu_popup()` to display the menu on the screen.

```
void gtk_menu_popup (GtkMenu *menu,
                     GtkWidget *parent_menu_shell,
                     GtkWidget *parent_menu_item,
                     GtkMenuPositionFunc func,
                     gpointer func_data,
                     guint button,
                     guint32 event_time);
```

In Listing 9-2, all parameters were set to `NULL` except for the mouse button that was clicked to cause the event (`event->button`) and the time when the event occurred (`event->time`). If the pop-up menu was activated by something other than a button, you should supply `0` to the button parameter.

■ **Note** If the action was invoked by a `popup-menu` signal, the event time will not be available. In that case, you can use `gtk_get_current_event_time()`. This function returns the timestamp of the current event or `GDK_CURRENT_TIME` if there are no recent events.

Usually, `parent_menu_shell`, `parent_menu_item`, `func`, and `func_data` are set to `NULL`, because they are used when the menu is a part of a menu bar structure. The `parent_menu_shell` widget is the menu shell that contains the item that caused the pop-up initialization. Alternatively, you can supply `parent_menu_item`, which is the menu item that caused the pop-up initialization.

`GtkMenuPositionFunc` is a function that decides at what position on the screen the menu should be drawn. It accepts `func_data` as an optional last parameter. As previously stated, these parameters are not frequently used in applications, so they can safely be set to `NULL`. In our example, the pop-up menu was already associated with the progress bar, so it will be drawn in the correct location.

Keyboard Accelerators

When creating a menu, one of the most important things to do is to set up keyboard accelerators. A keyboard accelerator is a key combination created from one accelerator key and one or more modifiers such as Ctrl or Shift. When the user presses the key combination, the appropriate signal is emitted.

Listing 9-3 is an extension of the progress bar pop-up menu application that adds keyboard accelerators to the menu items. The progress bar is pulsed when the user presses Ctrl+P, filled with Ctrl+F, and cleared with Ctrl+C.

Listing 9-3. *Adding Accelerators to Menu Items (accelerators.c)*

```
static void
create_popup_menu (GtkWidget *menu,
                   GtkWidget *window,
                   GtkWidget *progress)
{
  GtkWidget *pulse, *fill, *clear, *separator;
  GtkAccelGroup *group;

  /* Create a keyboard accelerator group for the application. */
  group = gtk_accel_group_new ();
  gtk_window_add_accel_group (GTK_WINDOW (window), group);
  gtk_menu_set_accel_group (GTK_MENU (menu), group);

  pulse = gtk_menu_item_new_with_label ("Pulse Progress");
  fill = gtk_menu_item_new_with_label ("Set as Complete");
  clear = gtk_menu_item_new_with_label ("Clear Progress");
  separator = gtk_separator_menu_item_new ();

  /* Add the necessary keyboard accelerators. */
  gtk_widget_add_accelerator (pulse, "activate", group, GDK_P,
                              GDK_CONTROL_MASK, GTK_ACCEL_VISIBLE);
  gtk_widget_add_accelerator (fill, "activate", group, GDK_F,
                              GDK_CONTROL_MASK, GTK_ACCEL_VISIBLE);
  gtk_widget_add_accelerator (clear, "activate", group, GDK_C,
                              GDK_CONTROL_MASK, GTK_ACCEL_VISIBLE);

  g_signal_connect (G_OBJECT (pulse), "activate",
                    G_CALLBACK (pulse_activated), progress);
  g_signal_connect (G_OBJECT (fill), "activate",
                    G_CALLBACK (fill_activated), progress);
  g_signal_connect (G_OBJECT (clear), "activate",
                    G_CALLBACK (clear_activated), progress);

  gtk_menu_shell_append (GTK_MENU_SHELL (menu), pulse);
  gtk_menu_shell_append (GTK_MENU_SHELL (menu), separator);
  gtk_menu_shell_append (GTK_MENU_SHELL (menu), fill);
  gtk_menu_shell_append (GTK_MENU_SHELL (menu), clear);

  gtk_menu_attach_to_widget (GTK_MENU (menu), progress, NULL);
  gtk_widget_show_all (menu);
}
```

Keyboard accelerators are stored as an instance of GtkAccelGroup. In order to implement accelerators in your application, you need to create a new accelerator group with gtk_accel_group_new(). This accelerator group must be added to the GtkWindow where the menu will appear for it to take effect. It must also be associated with any menus that take advantage of its accelerators. In Listing 9-3, this is performed immediately after creating the GtkAccelGroup with gtk_window_add_accel_group() and gtk_menu_set_accel_group().

It is possible to manually create keyboard accelerators with GtkAccelMap, but in most cases, gtk_widget_add_accelerator() will provide all of the necessary functionality. The only problem that this method presents is that the user cannot change keyboard accelerators created with this function during runtime.

```
void gtk_widget_add_accelerator (GtkWidget *widget,
                                 const gchar *signal_name,
                                 GtkAccelGroup *group,
                                 guint accel_key,
                                 GdkModifierType mods,
                                 GtkAccelFlags flags);
```

To add an accelerator to a widget, you can use gtk_widget_add_accelerator(), which will emit the signal specified by signal_name on the widget when the user presses the key combination. You need to specify your accelerator group to the function, which must be associated with the window and the menu as previously stated.

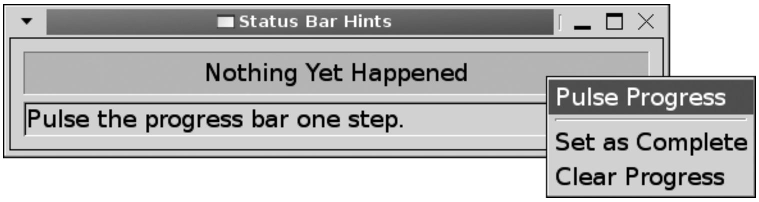
An accelerator key and one or more modifier keys form the complete key combination. A list of available accelerator keys is available in <gdk/gdkkeysyms.h>. This header file is not included in <gtk/gtk.h>, so it must explicitly be included. Modifiers are specified by the GdkModifierType enumeration. The most often used modifiers are GDK_SHIFT_LOCK, GDK_CONTROL_MASK, and GDK_MOD1_MASK, which correspond to the Shift, Ctrl, and Alt keys respectively.

■**Tip** When dealing with key codes, you need to be careful because you many need to supply multiple keys for the same action in some cases. For example, if you want to catch the number 1 key, you will need to watch for GDK_1 and GDK_KP_1—they correspond to the 1 key at the top of the keyboard and the 1 key on the numeric keypad.

The last parameter of gtk_widget_add_accelerator() is an accelerator flag. There are three flags defined by the GtkAccelFlags enumeration. The accelerator will be visible in a label if GTK_ACCEL_VISIBLE is set. GTK_ACCEL_LOCKED will prevent the user from modifying the accelerator. GTK_ACCEL_MASK will set both flags for the widget accelerator.

Status Bar Hints

Usually placed along the bottom of the main window, the GtkStatusbar widget can be used to give the user further information about what is going on in the application. A status bar can also be very useful with menus, because you can provide more information to the user about the functionality of the menu item that the mouse cursor is hovering over. A screenshot of a status bar can be viewed in Figure 9-2.



Nothing Yet Happened

Pulse the progress bar one step.

Figure 9-2. *A pop-up menu with status bar hints*

The Status Bar Widget

While the status bar can only display one message at a time, the widget actually stores a stack of messages. The currently displayed message is on the top of the stack. When you pop a message from the stack, the previous message is displayed. If there are no more strings left on the stack after you pop a message from the top, no message is displayed on the status bar.

New status bar widgets are created with `gtk_statusbar_new()`. This will create a new `GtkStatusbar` widget with an empty message stack. Before you are able to add or remove a message from the new status bar's stack, you must retrieve a context identifier with `gtk_status_bar_get_context_id()`:

```
guint gtk_statusbar_get_context_id (GtkStatusBar *statusbar,
                                    const gchar *description);
```

The context identifier is a unique unsigned integer that is associated with a context description string. This identifier will be used for all messages of a specific type, which allows you to categorize messages on the stack.

For example, if your status bar will hold hyperlinks and IP addresses, you could create two context identifiers from the strings "URL" and "IP". When you push or pop messages to and from the stack, you have to specify a context identifier. This allows separate parts of your application to push and pop messages to and from the status bar message stack without affecting each other.

■ **Tip** It is important to use different context identifiers for different categories of messages. If one part of your application is trying to give a message to the user while the other is trying to remove its own message, you do not want the wrong message to be popped from the stack!

After you generate a context identifier, you can add a message to the top of the status bar's stack with `gtk_statusbar_push()`. This function returns a unique message identifier for the string that was just added. This identifier can be used later to remove the message from the stack, regardless of its location.

```
guint gtk_statusbar_push (GtkStatusBar *statusbar,
                          guint context_id,
                          const gchar *message);
```

There are two ways to remove a message from the stack. If you want to remove a message from the top of the stack for a specific context ID, you can use `gtk_statusbar_pop()`. This function will remove the message that is highest on the status bar's stack with a context identifier of `context_id`.

```
void gtk_statusbar_pop (GtkStatusBar *statusbar,
                        guint context_id);
```

It is also possible to remove a specific message from the status bar's message stack with `gtk_statusbar_remove()`. To do this, you must provide the context identifier of the message and the message identifier of the message you want to remove, which was returned by `gtk_statusbar_push()` when it was added.

```
void gtk_statusbar_remove (GtkStatusBar *statusbar,
                           guint context_id,
                           guint message_id);
```

`GtkStatusbar` has one property, `has-resize-grip`, which will place a graphic in the corner of the status bar for resizing the window. The user will be able to grab the resize grip and drag it to resize its parent window. You can also use the built-in function `gtk_statusbar_set_has_resize_grip()` to set this property.

Menu Item Information

One useful role of the status bar is to give the user more information about the menu item the mouse cursor is currently hovering over. An example of this was shown in the previous section in Figure 9-2, which is a screenshot of the progress bar pop-up menu application in Listing 9-4.

To implement status bar hints, you should connect each of your menu items to `GtkWidget`'s `enter-notify-event` and `leave-notify-event` signals. Listing 9-4 shows the progress bar pop-up menu application you have already learned about, except status bar hints are provided when the mouse cursor moves over a menu item.

Listing 9-4. *Displaying More Information About a Menu Item (statusbarhints.c)*

```
static void
create_popup_menu (GtkWidget *menu,
                   GtkWidget *progress,
                   GtkWidget *statusbar)
{
  GtkWidget *pulse, *fill, *clear, *separator;

  pulse = gtk_menu_item_new_with_label ("Pulse Progress");
  fill = gtk_menu_item_new_with_label ("Set as Complete");
  clear = gtk_menu_item_new_with_label ("Clear Progress");
  separator = gtk_separator_menu_item_new ();

  g_signal_connect (G_OBJECT (pulse), "activate",
                    G_CALLBACK (pulse_activated), progress);
  g_signal_connect (G_OBJECT (fill), "activate",
                    G_CALLBACK (fill_activated), progress);
  g_signal_connect (G_OBJECT (clear), "activate",
                    G_CALLBACK (clear_activated), progress);

  /* Connect signals to each menu item for status bar messages. */
  g_signal_connect (G_OBJECT (pulse), "enter-notify-event",
                    G_CALLBACK (statusbar_hint), statusbar);
  g_signal_connect (G_OBJECT (pulse), "leave-notify-event",
                    G_CALLBACK (statusbar_hint), statusbar);
  g_signal_connect (G_OBJECT (fill), "enter-notify-event",
                    G_CALLBACK (statusbar_hint), statusbar);
  g_signal_connect (G_OBJECT (fill), "leave-notify-event",
                    G_CALLBACK (statusbar_hint), statusbar);
  g_signal_connect (G_OBJECT (clear), "enter-notify-event",
                    G_CALLBACK (statusbar_hint), statusbar);
  g_signal_connect (G_OBJECT (clear), "leave-notify-event",
                    G_CALLBACK (statusbar_hint), statusbar);

  g_object_set_data (G_OBJECT (pulse), "menuhint",
                     (gpointer) "Pulse the progress bar one step.");
  g_object_set_data (G_OBJECT (fill), "menuhint",
                     (gpointer) "Set the progress bar to 100%.");
  g_object_set_data (G_OBJECT (clear), "menuhint",
                     (gpointer) "Clear the progress bar to 0%.");

  gtk_menu_shell_append (GTK_MENU_SHELL (menu), pulse);
  gtk_menu_shell_append (GTK_MENU_SHELL (menu), separator);
  gtk_menu_shell_append (GTK_MENU_SHELL (menu), fill);
  gtk_menu_shell_append (GTK_MENU_SHELL (menu), clear);
  gtk_menu_attach_to_widget (GTK_MENU (menu), progress, NULL);
  gtk_widget_show_all (menu);
}

/* Add or remove a status bar menu hint, depending on whether this function
 * is initialized by a proximity-in-event or proximity-out-event. */
static gboolean
statusbar_hint (GtkMenuItem *menuitem,
                GdkEventProximity *event,
                GtkStatusbar *statusbar)
{
  gchar *hint;
  guint id = gtk_statusbar_get_context_id (statusbar, "MenuItemHints");

  if (event->type == GDK_ENTER_NOTIFY)
  {
    hint = (gchar*) g_object_get_data (G_OBJECT (menuitem), "menuhint");
    gtk_statusbar_push (statusbar, id, hint);
  }
  else if (event->type == GDK_LEAVE_NOTIFY)
    gtk_statusbar_pop (statusbar, id);

  return FALSE;
}
```

When implementing status bar hints, you first need to figure out what signals are necessary. We want to be able to add a message to the status bar when the mouse cursor moves over the menu item and remove it when the mouse cursor leaves. From this description, using `enter-notify-event` and `leave-notify-event` is a good solution.

One advantage of using these two signals is that we only need one callback function, because the prototype for each receives a `GdkEventProximity` object. From this object, we can discern between `GDK_ENTER_NOTIFY` and `GDK_LEAVE_NOTIFY` events. You will want to return `FALSE` from the callback function, because you do not want to prevent GTK+ from handling the event; you only want to enhance what is performed when it is emitted.

Within the `statusbar_hint()` callback function, you should first retrieve a context identifier for the menu item messages. You can use whatever string you want, as long as your application remembers what was used. In Listing 9-4, `"MenuItemHints"` was used to describe all of the menu item messages added to the status bar. If other parts of the application used the status bar, using a different context identifier would leave the menu item hints untouched.

```
guint id = gtk_statusbar_get_context_id (statusbar, "MenuItemHints");
```

If the event type is `GDK_ENTER_NOTIFY`, you need to show the message to the user. In the `create_popup_menu()` function, a data parameter was added to each menu item called `"menuhint"`. This is a more in-depth description of what the menu item does, which will be displayed to the user.

```
hint = (gchar*) g_object_get_data (G_OBJECT (menuitem), "menuhint");
gtk_statusbar_push (statusbar, id, hint);
```

Then, with `gtk_statusbar_push()`, the message can be added to the status bar under the `"MenuItemHints"` context identifier. This message will be placed on the top of the stack and displayed to the user. You may want to consider processing all GTK+ events after calling this function, since the user interface should reflect the changes immediately.

However, if the event type is `GDK_LEAVE_NOTIFY`, you need to remove the last menu item message that was added with the same context identifier. The most recent message can be removed from the stack with `gtk_statusbar_pop()`.

Menu Items

Thus far, you have learned about flat menus that display label and separator menu items. It is also possible to add a submenu to an existing menu item. GTK+ also provides a number of other `GtkMenuItem` objects. Figure 9-3 shows a pop-up menu that contains a submenu along with image, check, and radio menu items.

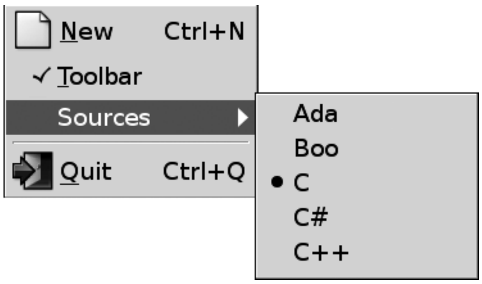


Figure 9-3. *Image, check, and radio menu items*

Submenus

Submenus in GTK+ are not created by a separate type of menu item widget but by calling `gtk_menu_item_set_submenu()`. This function calls `gtk_menu_attach_to_widget()` to attach the submenu to the menu item and places an arrow beside the menu item to show that it now has a submenu. If the menu item already has a submenu, it will be replaced with the given `GtkMenu` widget.

```
void gtk_menu_item_set_submenu (GtkMenuItem *menuitem,
                                GtkWidget *submenu);
```

Submenus are very useful if you have a list of very specific options that would clutter an otherwise organized menu structure. When using a submenu, you can use the `activate-item` signal provided by the `GtkMenuItem` widget, which will be emitted when the menu item displays its submenu.

In addition to `GtkMenuItem` and menu item separators, there are three other types of menu item objects: image, check, and radio menu items; these are covered in the remainder of this section.

Image Menu Items

GtkImageMenuItem is very similar to its parent class GtkMenuItem except it shows a small image to the left of the menu item label. There are four functions provided for creating a new image menu item.

The first function, gtk_image_menu_item_new() creates a new GtkImageMenuItem object with an empty label and no associated image. You can use image menu item's image property to set the image displayed by the menu item.

```
GtkWidget* gtk_image_menu_item_new ();
```

Additionally, you can create a new image menu item from a stock identifier with gtk_image_menu_item_new_from_stock(). This function creates the GtkImageMenuItem with the label and image associated with stock_id. This function accepts stock identifier strings that are listed in Appendix D.

```
GtkWidget* gtk_image_menu_item_new_from_stock (const gchar *stock_id,
                                               GtkAccelGroup *accel_group);
```

The second parameter of this function accepts an accelerator group, which will be set to the default accelerator of the stock item. If you want to manually set the keyboard accelerator for the menu item as we did in Listing 9-3, you can specify NULL for this parameter.

Also, you can use gtk_image_menu_item_new_with_label() to create a new GtkImageMenuItem initially with only a label. Later, you can use the image property to add an image widget. GTK+ also provided the function gtk_image_menu_item_set_image(), which allows you to edit the image property of the widget.

```
GtkWidget* gtk_image_menu_item_new_with_label (const gchar *label);
```

Also, GTK+ provides gtk_image_menu_item_new_with_mnemonic(), which will create an image menu item with a mnemonic label. As with the previous function, you will have to set the image property after the menu item is created.

Check Menu Items

GtkCheckMenuItem allows you to create a menu item that will display a check symbol beside the label, depending on whether its Boolean active property is TRUE or FALSE. This would allow the user to view whether an option is activated or deactivated.

As with GtkMenuItem, three initialization functions are provided: gtk_check_menu_item_new(), gtk_check_item_new_with_label(), and gtk_check_menu_item_new_with_mnemonic(). These functions create a GtkCheckMenuItem with no label, with an initial label, or with a mnemonic label, respectively.

```
GtkWidget* gtk_check_menu_item_new ();
GtkWidget* gtk_check_menu_item_new_with_label    (const gchar *label);
GtkWidget* gtk_check_menu_item_new_with_mnemonic (const gchar *label);
```

As previously stated, the current state of the check menu item is held by the active property of the widget. GTK+ provides two functions, gtk_check_menu_item_set_active() and gtk_check_menu_item_get_active() to set and retrieve the active value.

As with all check button widgets, you are able to use the toggled signal, which is emitted when the user toggles the state of the menu item. GTK+ takes care of updating the state of the check button, so this signal is simply to allow you to update your application to reflect the changed value.

GtkCheckMenuItem also provides gtk_check_menu_item_set_inconsistent(), which is used to alter the inconsistent property of the menu item. When set to TRUE, the check menu item will display a third, "in between" state that is neither active nor inactive. This can be used to show the user that a choice must be made that has yet to be set or that the property is both set and unset for different parts of a selection.

Radio Menu Items

GtkRadioMenuItem is a widget derived from GtkCheckMenuItem. It is rendered as a radio button instead of a check button by setting check menu item's draw-as-radio property to TRUE. Radio menu items work the same way as normal radio buttons.

The first radio button should be created with one of the following functions. You can set the radio button group to NULL, because it is not necessary since requisite elements will be added to the group by referencing the first element. These functions create an empty menu item, a menu item with a label, and a menu item with a mnemonic, respectively.

```
GtkWidget* gtk_radio_menu_item_new               (GSList *group);
GtkWidget* gtk_radio_menu_item_new_with_label    (GSList *group,
                                                  const gchar *text);
GtkWidget* gtk_radio_menu_item_new_with_mnemonic (GSList *group,
                                                  const gchar *text);
```

All other radio menu items should be created with one of the following three functions, which will add it to the radio button group associated with group. These functions create an empty menu item, a menu item with a label, and a menu item with a mnemonic, respectively.

```
GtkWidget* gtk_radio_menu_item_new_from_widget            (GtkRadioMenuItem *group);
GtkWidget* gtk_radio_menu_item_new_from_widget_with_label (GtkRadioMenuItem *group,
                                                           const gchar *text);
GtkWidget* gtk_radio_menu_item_new_from_widget_with_mnemonic
                                                          (GtkRadioMenuItem *group,
                                                           const gchar *text);
```

Menu Bars

GtkMenuBar is a widget that organizes multiple pop-up menus into a horizontal or vertical row. Each root element is a GtkMenuItem that pops down into a submenu. An instance of GtkMenuBar is usually displayed along the top of the main application window to provide access to functionality provided by the application. An example menu bar is shown in Figure 9-4.

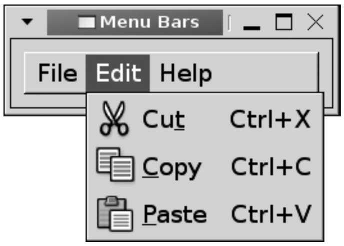


Figure 9-4. *A menu bar with three menus*

In Listing 9-5, a `GtkMenuBar` widget is created with three menus: File, Edit, and Help. Each of the menus is actually a `GtkMenuItem` with a submenu. A number of menu items are then added to each submenu.

Listing 9-5. *Creating Groups of Menus (menubars.c)*

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

int main (int argc,
          char *argv[])
{
  GtkWidget *window, *menubar, *file, *edit, *help, *filemenu, *editmenu, *helpmenu;
  GtkWidget *new, *open, *cut, *copy, *paste, *contents, *about;
  GtkAccelGroup *group;

  gtk_init (&argc, &argv);

  window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  gtk_window_set_title (GTK_WINDOW (window), "Menu Bars");
  gtk_widget_set_size_request (window, 250, -1);

  group = gtk_accel_group_new ();
  menubar = gtk_menu_bar_new ();
  file = gtk_menu_item_new_with_label ("File");
  edit = gtk_menu_item_new_with_label ("Edit");
  help = gtk_menu_item_new_with_label ("Help");
  filemenu = gtk_menu_new ();
  editmenu = gtk_menu_new ();
  helpmenu = gtk_menu_new ();

  gtk_menu_item_set_submenu (GTK_MENU_ITEM (file), filemenu);
  gtk_menu_item_set_submenu (GTK_MENU_ITEM (edit), editmenu);
  gtk_menu_item_set_submenu (GTK_MENU_ITEM (help), helpmenu);
  gtk_menu_shell_append (GTK_MENU_SHELL (menubar), file);
  gtk_menu_shell_append (GTK_MENU_SHELL (menubar), edit);
  gtk_menu_shell_append (GTK_MENU_SHELL (menubar), help);

  /* Create the File menu content. */
  new = gtk_image_menu_item_new_from_stock (GTK_STOCK_NEW, group);
  open = gtk_image_menu_item_new_from_stock (GTK_STOCK_OPEN, group);
  gtk_menu_shell_append (GTK_MENU_SHELL (filemenu), new);
  gtk_menu_shell_append (GTK_MENU_SHELL (filemenu), open);

  /* Create the Edit menu content. */
  cut = gtk_image_menu_item_new_from_stock (GTK_STOCK_CUT, group);
  copy = gtk_image_menu_item_new_from_stock (GTK_STOCK_COPY, group);
  paste = gtk_image_menu_item_new_from_stock (GTK_STOCK_PASTE, group);
  gtk_menu_shell_append (GTK_MENU_SHELL (editmenu), cut);
  gtk_menu_shell_append (GTK_MENU_SHELL (editmenu), copy);
  gtk_menu_shell_append (GTK_MENU_SHELL (editmenu), paste);

  /* Create the Help menu content. */
  contents = gtk_image_menu_item_new_from_stock (GTK_STOCK_HELP, group);
  about = gtk_image_menu_item_new_from_stock (GTK_STOCK_ABOUT, group);
  gtk_menu_shell_append (GTK_MENU_SHELL (helpmenu), contents);
  gtk_menu_shell_append (GTK_MENU_SHELL (helpmenu), about);

  gtk_container_add (GTK_CONTAINER (window), menubar);
  gtk_window_add_accel_group (GTK_WINDOW (window), group);

  gtk_widget_show_all (window);
  gtk_main ();
  return 0;
}
```

New `GtkMenuBar` widgets are created with `gtk_menu_bar_new()`. This will create an empty menu shell into which you can add content.

After you create the menu bar, you can define the pack direction of the menu bar items with `gtk_menu_bar_set_pack_direction()`. Values for the `pack-direction` property are defined by the `GtkPackDirection` enumeration and include `GTK_PACK_DIRECTION_LTR`, `GTK_PACK_DIRECTION_RTL`, `GTK_PACK_DIRECTION_TTB`, or `GTK_PACK_DIRECTION_BTT`. These will pack the menu items from left to right, right to left, top to bottom, or bottom to top, respectively. By default, child widgets are packed from left to right.

GtkMenuBar also provides another property called child-pack-direction, which sets what direction the menu items of the menu bar's children are packed. In other words, it controls how submenu items are packed. Values for this property are also defined by the GtkPackDirection enumeration.

Each child item in the menu bar is actually a GtkMenuItem widget. Since GtkMenuBar is derived from GtkMenuShell, you can use gtk_menu_shell_append() to add an item to the bar as shown in the following line.

```
gtk_menu_shell_append (GTK_MENU_SHELL (menubar), file);
```

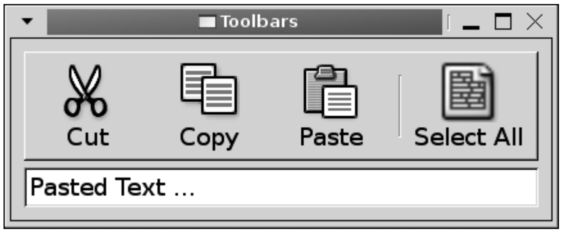
You can also use gtk_menu_shell_prepend() or gtk_menu_shell_insert() to add an item to the beginning or in an arbitrary position of the menu bar.

You next need to call gtk_menu_item_set_submenu() to add a submenu to each of the root menu items. Each of the submenus is a GtkMenu widget created in the same way as pop-up menus. GTK+ will then take care of showing submenus to the user when necessary.

```
gtk_menu_item_set_submenu (GTK_MENU_ITEM (file), filemenu);
```

Toolbars

A GtkToolbar is a type of container that holds a number of widgets in a horizontal or vertical row. It is meant to allow easy customization of a large number of widgets with very little trouble. Typically, toolbars hold tool buttons that can display an image along with a text string. However, toolbars are actually able to hold any type of widget. A toolbar holding four tool buttons and a separator is shown in Figure 9-5.



Pasted Text ...

Figure 9-5. *A toolbar showing both images and text*

In Listing 9-6, a simple toolbar is created that shows five tool items in a horizontal row. Each toolbar item displays an icon and a label that describes the purpose of the item. The toolbar is also set to display an arrow that will provide access to toolbar items that do not fit in the menu.

In this example, a toolbar is used to provide cut, copy, paste, and select-all functionality to a `GtkEntry` widget. The `main()` function creates the toolbar, packing it above the `GtkEntry`. It then calls `create_toolbar()`, which populates the toolbar with tool items and connects the necessary signals.

Listing 9-6. *Creating a GtkToolbar Widget (toolbars.c)*

```
static void select_all (GtkEditable*);

/* Create a toolbar with Cut, Copy, Paste and Select All toolbar items. */
static void
create_toolbar (GtkWidget *toolbar,
                GtkWidget *entry)
{
  GtkToolItem *cut, *copy, *paste, *selectall, *separator;

  cut = gtk_tool_button_new_from_stock (GTK_STOCK_CUT);
  copy = gtk_tool_button_new_from_stock (GTK_STOCK_COPY);
  paste = gtk_tool_button_new_from_stock (GTK_STOCK_PASTE);
  selectall = gtk_tool_button_new_from_stock (GTK_STOCK_SELECT_ALL);
  separator = gtk_separator_tool_item_new ();

  gtk_toolbar_set_show_arrow (GTK_TOOLBAR (toolbar), TRUE);
  gtk_toolbar_set_style (GTK_TOOLBAR (toolbar), GTK_TOOLBAR_BOTH);

  gtk_toolbar_insert (GTK_TOOLBAR (toolbar), cut, 0);
  gtk_toolbar_insert (GTK_TOOLBAR (toolbar), copy, 1);
  gtk_toolbar_insert (GTK_TOOLBAR (toolbar), paste, 2);
  gtk_toolbar_insert (GTK_TOOLBAR (toolbar), separator, 3);
  gtk_toolbar_insert (GTK_TOOLBAR (toolbar), selectall, 4);

  g_signal_connect_swapped (G_OBJECT (cut), "clicked",
                            G_CALLBACK (gtk_editable_cut_clipboard), entry);
  g_signal_connect_swapped (G_OBJECT (copy), "clicked",
                            G_CALLBACK (gtk_editable_copy_clipboard), entry);
  g_signal_connect_swapped (G_OBJECT (paste), "clicked",
                            G_CALLBACK (gtk_editable_paste_clipboard), entry);
  g_signal_connect_swapped (G_OBJECT (selectall), "clicked",
                            G_CALLBACK (select_all), entry);
}

/* Select all of the text in the GtkEditable. */
static void
select_all (GtkEditable *entry)
{
  gtk_editable_select_region (entry, 0, -1);
}
```

New toolbars are created with gtk_toolbar_new(), which was called before the create_toolbar() function shown in Listing 9-6. This creates an empty GtkToolbar widget in which you can add tool buttons.

GtkToolbar provides a number of properties for customizing how it appears and interacts with the user including the orientation, button style, and the ability to give access to items that do not fit in the toolbar.

If all of the toolbar items cannot be displayed on the toolbar because there is not enough room, then an overflow menu will appear if you set gtk_toolbar_set_show_arrow() to TRUE. If all of the items can be displayed on the toolbar, the arrow will be hidden from view.

```
void gtk_toolbar_set_show_arrow (GtkToolbar *toolbar,
                                 gboolean show_arrow);
```

Another GtkToolbar property is the style by which all of the menu items will be displayed, which is set with gtk_toolbar_set_style(). You should note that this property can be overridden by the theme, so you should provide the option of using the default style by calling gtk_toolbar_unset_style(). There are four toolbar styles, which are defined by the GtkToolbarStyle enumeration:

- GTK_TOOLBAR_ICONS: Show only icons for each tool button in the toolbar.
- GTK_TOOLBAR_TEXT: Show only labels for each tool button in the toolbar.
- GTK_TOOLBAR_BOTH: Show both icons and labels for each tool button, where the icon is located above its label.
- GTK_TOOLBAR_BOTH_HORIZ: Show both icons and labels for each tool button, where the icon is to the left of the label. The label text of a tool item will only be shown if the is-important property for the item is set to TRUE.

Another important property of the toolbar is the orientation that can be set with gtk_toolbar_set_orientation(). There are two possible values defined by the GtkOrientation enumeration, GTK_ORIENTATION_HORIZONTAL and GTK_ORIENTATION_VERTICAL, which can be used to make the toolbar horizontal (default) or vertical.

Toolbar Items

Listing 9-6 introduces three important tool item types: GtkToolItem, GtkToolButton, and GtkSeparatorToolItem. All tool buttons are derived from the GtkToolItem class, which holds basic properties that are used by all tool items.

If you are using the GTK_TOOLBAR_BOTH_HORIZ style, then an essential property installed in GtkToolItem is the is-important setting. The label text of the toolbar item will only be shown for this style if this property is set to TRUE.

As with menus, separator tool items are provided by GtkSeparatorToolItem and are created with gtk_separator_tool_item_new(). Separator tool items have a draw property, which will draw a separator when set to TRUE. If you set draw to FALSE, it will place padding at its location without any visual separator.

■ **Tip** If you set the `expand` property of a `GtkSeparatorToolItem` to `TRUE` and its `draw` property to `FALSE`, you will force all tool items after the separator to the end of the toolbar.

Most toolbar items are of the type `GtkToolButton`. `GtkToolButton` provides a number of initialization functions including `gtk_tool_button_new_from_stock()`. This function accepts a stock identifier; a list of stock items available in GTK+ 2.10 can be found in Appendix D. Unlike most initialization functions, this method returns a `GtkToolItem` object instead of a `GtkWidget`.

Alternatively, you can use `gtk_tool_button_new()` to create a `GtkToolButton` with a custom icon and label. Each of these properties can be set to `NULL`.

```
GtkToolItem* gtk_tool_button_new (GtkWidget *icon,
                                  const gchar* label);
```

It is possible to manually set the label, stock identifier, and icon after initialization with `gtk_tool_button_set_label()`, `gtk_tool_button_set_stock_id()`, and `gtk_tool_button_set_icon_widget()`. These functions provide access to tool button's `label`, `stock-id`, and `icon-widget` properties.

Additionally, you can define your own widget to use instead of the default `GtkLabel` widget of the tool button with `gtk_tool_button_set_label_widget()`. This will allow you to embed an arbitrary widget, such as an entry or combo box, into the tool button. If this property is set to `NULL`, the default label will be used.

```
void gtk_tool_button_set_label_widget (GtkToolButton *button,
                                       GtkWidget *label_widget);
```

After you create the toolbar items, you can insert each `GtkToolItem` into the toolbar with `gtk_toolbar_insert()`. You do not have to cast the `GtkToolItem`, since the initialization functions do not return a `GtkWidget`.

```
void gtk_toolbar_insert (GtkToolbar *toolbar,
                         GtkToolItem *item,
                         gint pos);
```

The third parameter of `gtk_toolbar_insert()` accepts the position to insert the item into the toolbar. Tool button positions are indexed from zero. A negative position will append the item to the end of the toolbar.

Toggle Tool Buttons

`GtkToggleToolButton` is derived from `GtkToolButton` and, therefore, only implements initialization and toggle abilities itself. Toggle tool buttons provide the functionality of a `GtkToggleButton` widget in the form of a toolbar item. It allows the user to view whether the option is set or unset.

Toggle tool buttons are tool buttons that remain depressed when the `active` property is set to `TRUE`. You can use the `toggled` signal to receive notification when the state of the toggle button has been changed.

There are two ways to create a new GtkToggleToolButton. The first is with gtk_toggle_tool_button_new(), which will create an empty tool button. You can then use the functions provided by GtkToolButton to add a label and image.

```
GtkToolItem* gtk_toggle_tool_button_new ();
GtkToolItem* gtk_toggle_tool_button_new_from_stock (const gchar *stock_id);
```

Alternatively, you can use gtk_toggle_tool_button_new_from_stock(), which will create a tool button with the label and image associated with the stock identifier. If the stock identifier is not found, the image and label will be set to the error stock item.

Radio Tool Buttons

GtkRadioToolButton is derived from GtkToggleToolButton, so it inherits the active property and toggled signal. Therefore, the widget only needs to give a way for you to create new radio tool buttons and add them to a radio group.

The first radio tool button should be created with gtk_radio_tool_button_new() or gtk_radio_tool_button_new_from_stock(), where the radio group is set to NULL. This will create a default initial radio group for the radio tool button.

```
GtkToolItem* gtk_radio_tool_button_new            (GSList *group);
GtkToolItem* gtk_radio_tool_button_new_from_stock (GSList *group,
                                                    const gchar *stock_id);
```

GtkRadioToolButton inherits functions from GtkToolButton, which provides functions and properties that can then be used to set the label of the radio tool button if necessary.

All requisite elements should be created with gtk_radio_tool_button_from_widget() or gtk_radio_tool_button_new_with_stock_from_widget(). Setting group as the first radio tool button will add all requisite items added to the same group.

```
GtkToolItem* gtk_radio_tool_button_new_from_widget (GtkRadioToolButton *group);
GtkToolItem* gtk_radio_tool_button_new_with_stock_from_widget
                                                   (GtkRadioToolButton *group,
                                                    const gchar *stock_id);
```

GtkRadioToolButton provides one property, group, which is another radio tool button that belongs to the radio group. This allows you to link all of the radio buttons together so that only one will be selected at a time.

Menu Tool Buttons

GtkMenuToolButton, derived from GtkToolButton, allows you to attach a menu to a tool button. The widget places an arrow beside the image and label that provides access to the associated menu. For example, you could use GtkMenuToolButton to add a list of recently opened files to a GTK_STOCK_OPEN toolbar button. Figure 9-6 is a screenshot of a menu tool button that is used for this purpose.

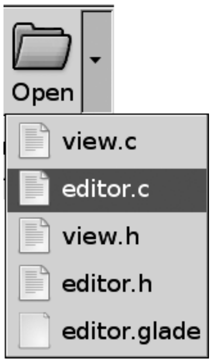


Figure 9-6. *A menu tool button showing recently opened files*

Listing 9-7 shows you how to implement a menu tool button. The actual tool button is created in a similar way as any other `GtkToolButton` except there is an extra step of attaching a menu to the `GtkMenuToolButton` widget.

Listing 9-7. *Using GtkMenuToolButton*

```
GtkToolItem *open;
GtkWidget *recent;

recent = gtk_menu_new ();
/* Add a number of menu items where each corresponds to one recent file. */

open = gtk_menu_tool_button_new_from_stock (GTK_STOCK_OPEN);
gtk_menu_tool_button_set_menu (GTK_MENU_TOOL_BUTTON (open), GTK_MENU (recent));
```

In Listing 9-7, the menu tool button was created with a default stock icon and label with `gtk_menu_tool_button_new_from_stock()`. This function accepts a stock identifier and will apply the appropriate label and icon.

Alternatively, you can create a menu tool button with `gtk_menu_tool_button_new()`, which accepts an icon widget and the label text. You can set either of these parameters to `NULL` if you want to set them at a later time using `GtkToolButton` properties.

```
GtkToolItem* gtk_menu_tool_button_new (GtkWidget *icon,
                                       const gchar *label);
```

What makes `GtkMenuToolButton` unique is that an arrow to the right of the tool button provides the user with access to a menu. The tool button's menu is set with `gtk_menu_tool_button_set_menu()` or by setting the `menu` property to a `GtkMenu` widget. This menu is displayed to the user when the arrow is clicked.

Dynamic Menu Creation

While it is possible to manually create every menu and toolbar item, doing so can take up a large amount of space and cause you to have to code monotonously for longer than necessary. In order to automate menu and toolbar creation, GTK+ allows you to dynamically create menus from XML files.

Creating UI Files

User interface files are constructed in XML format. All of the content has to be contained between `<ui>` and `</ui>` tags. One type of dynamic UI that you can create is a `GtkMenuBar` with the `<menubar>` tag shown in Listing 9-8.

Listing 9-8. *Menu UI File (menu.ui)*

```
<ui>
  <menubar name="MenuBar">
    <menu name="FileMenu" action="File">
      <menuitem name="FileOpen" action="Open"/>
      <menuitem name="FileSave" action="Save"/>
      <separator/>
      <menuitem name="FileQuit" action="Quit"/>
    </menu>
    <menu name="EditMenu" action="Edit">
      <menuitem name="EditCut" action="Cut"/>
      <menuitem name="EditCopy" action="Copy"/>
      <menuitem name="EditPaste" action="Paste"/>
      <separator/>
      <menuitem name="EditSelectAll" action="SelectAll"/>
      <menuitem name="EditDeselect" action="Deselect"/>
    </menu>
    <menu name="HelpMenu" action="Help">
      <menuitem name="HelpContents" action="Contents"/>
      <menuitem name="HelpAbout" action="About"/>
    </menu>
  </menubar>
</ui>
```

While not necessary, you should add the `name` attribute to every `menubar`, `menu`, and `menuitem`. The `name` attribute can be used to access the actual widget. If `name` is not specified, using the `"action"` field can access the widget.

Each `<menubar>` can have any number of `<menu>` children. Both of these tags must be closed according to normal XML rules. If a tag does not have a closing tag (e.g., `<menuitem/>`), you must place a forward slash character (`/`) at the end of the tag so the parser knows the tag has ended.

The `action` attribute is applied to all elements except top-level widgets and separators. When loading the UI file to associate a `GtkAction` object to each element, `GtkUIManager` uses the `action` attributes. `GtkAction` holds information about how the item is drawn and what callback function should be called, if any, when the item is activated.

Separators can be placed in a menu with the `<separator/>` tag. You do not need to provide name or action information for separators, because a generic `GtkSeparatorMenuItem` will be added.

In addition to menu bars, you can create toolbars in a UI file with the `<toolbar>` tag, as shown in Listing 9-9.

Listing 9-9. *Toolbar UI File (toolbar.ui)*

```
<ui>
  <toolbar name="Toolbar">
    <toolitem name="FileOpen" action="Open"/>
    <toolitem name="FileSave" action="Save"/>
    <separator/>
    <toolitem name="EditCut" action="Cut"/>
    <toolitem name="EditCopy" action="Copy"/>
    <toolitem name="EditPaste" action="Paste"/>
    <separator/>
    <toolitem name="EditSelectAll" action="SelectAll"/>
    <toolitem name="EditDeselect" action="Deselect"/>
    <separator/>
    <toolitem name="HelpContents" action="Contents"/>
    <toolitem name="HelpAbout" action="About"/>
  </toolbar>
</ui>
```

Each toolbar can contain any number of `<toolitem>` elements. Tool items are specified in the same manner as menu items, with an `"action"` and an optional `"name"`. You can use the same `"name` for elements in separate UI files, but you should not use the same names if, for example, the toolbar and menu bar are located in the same file.

However, you *can* and should use the same `"action"` for multiple elements. This will cause each element to be drawn in the same way and to be connected to the same callback function. The advantage of this is that you need to define only one `GtkAction` for each item type. For example, the same `"action"` will be used for the Cut element in the UI files in Listings 9-8 through 9-10.

■Tip While the toolbar, menu bar, and pop-up menu were split into separate UI files, you can include as many of these widgets as you want in one file. The only requirement is that the whole file content is contained between the `<ui>` and `</ui>` tags.

In addition to toolbars and menu bars, it is possible to define pop-up menus in a UI file, as illustrated in Listing 9-10. Notice that there are repeated actions in Listings 9-8, 9-9, and 9-10. Repeating actions allows you to define only a single `GtkAction` object instead of separate objects for each instance of an `"action"`.

Listing 9-10. *Pop-up UI File (popup.ui)*

```
<ui>
  <popup name="EntryPopup">
    <menuitem name="EditCut" action="Cut"/>
    <menuitem name="EditCopy" action="Copy"/>
    <menuitem name="EditPaste" action="Paste"/>
    <separator/>
    <menuitem name="EditSelectAll" action="SelectAll"/>
    <menuitem name="EditDeselect" action="Deselect"/>
  </popup>
</ui>
```

The last type of top-level widget supported by UI files is the pop-up menu, denoted by the `<popup>` tag. Since a pop-up menu is the same thing as a normal menu, you can still use `<menuitem>` elements as children.

Loading UI Files

After you create your UI files, you need to load them into your application and retrieve the necessary widgets. To do this, you need to utilize the functionality provided by `GtkActionGroup` and `GtkUIManager`.

`GtkActionGroup` is a set of items with name, stock identifier, label, keyboard accelerator, tooltip, and callback functions. The name of the each action can be set to an `action` parameter from a UI file to associate it with a UI element.

`GtkUIManager` is an object that allows you to dynamically load one or more user interface definitions. It will automatically create an accelerator group based on associated action groups and allow you to reference widgets based on the `name` parameter from the UI file.

In Listing 9-11 `GtkUIManager` is used to load the menu bar and toolbar from the UI files in Listings 9-8 and 9-9. The resulting application is shown in Figure 9-7.

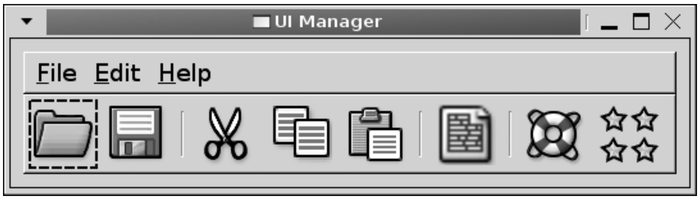


Figure 9-7. *A menu bar and a toolbar that are dynamically loaded*

Each of the menu and tool items in the application are connected to empty callback functions, because this example is only meant to show you how to dynamically load menus and toolbars from UI definitions. You will implement callback functions with actual content in the two exercises found at the end of this chapter.

Listing 9-11. *Loading a Menu with GtkUIManager (uimanager.c)*

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

/* All of the menu item callback functions have a GtkMenuItem parameter, and
 * receive the same gpointer value. There is only one callback function shown
 * since all of the rest will be formatted in the same manner. */
static void open (GtkMenuItem *menuitem, gpointer data);

#define NUM_ENTRIES 13
static GtkActionEntry entries[] =
{
  { "File", NULL, "_File", NULL, NULL, NULL },
  { "Open", GTK_STOCK_OPEN, NULL, NULL,
     "Open an existing file", G_CALLBACK (open) },
  { "Save", GTK_STOCK_SAVE, NULL, NULL,
     "Save the document to a file", G_CALLBACK (save) },
  { "Quit", GTK_STOCK_QUIT, NULL, NULL,
    "Quit the application", G_CALLBACK (quit) },
  { "Edit", NULL, "_Edit", NULL, NULL, NULL },
  { "Cut", GTK_STOCK_CUT, NULL, NULL,
    "Cut the selection to the clipboard", G_CALLBACK (cut) },
  { "Copy", GTK_STOCK_COPY, NULL, NULL,
    "Copy the selection to the clipboard", G_CALLBACK (copy) },
  { "Paste", GTK_STOCK_PASTE, NULL, NULL,
    "Paste text from the clipboard", G_CALLBACK (paste) },
  { "SelectAll", GTK_STOCK_SELECT_ALL, NULL, NULL,
    "Select all of the text", G_CALLBACK (selectall) },
  { "Deselect", NULL, "_Deselect", "<control>d",
    "Deselect all of the text", G_CALLBACK (deselect) },
  { "Help", NULL, "_Help", NULL, NULL, NULL },
  { "Contents", GTK_STOCK_HELP, NULL, NULL,
    "Get help on using the application", G_CALLBACK (help) },
  { "About", GTK_STOCK_ABOUT, NULL, NULL,
    "More information about the application", G_CALLBACK (about) }
};

int main (int argc,
          char *argv[])
{
  GtkWidget *window, *menubar, *toolbar, *vbox;
  GtkActionGroup *group;
  GtkUIManager *uimanager;

  gtk_init (&argc, &argv);

  window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  gtk_window_set_title (GTK_WINDOW (window), "UI Manager");
  gtk_widget_set_size_request (window, 250, -1);

  /* Create a new action group and add all of the actions to it. */
  group = gtk_action_group_new ("MainActionGroup");
  gtk_action_group_add_actions (group, entries, NUM_ENTRIES, NULL);

  /* Create a new UI manager and build the menu bar and toolbar. */
  uimanager = gtk_ui_manager_new ();
  gtk_ui_manager_insert_action_group (uimanager, group, 0);
  gtk_ui_manager_add_ui_from_file (uimanager, "menu.ui", NULL);
  gtk_ui_manager_add_ui_from_file (uimanager, "toolbar.ui", NULL);

  /* Retrieve the necessary widgets and associate accelerators. */
  menubar = gtk_ui_manager_get_widget (uimanager, "/MenuBar");
  toolbar = gtk_ui_manager_get_widget (uimanager, "/Toolbar");
  gtk_toolbar_set_style (GTK_TOOLBAR (toolbar), GTK_TOOLBAR_ICONS);
  gtk_window_add_accel_group (GTK_WINDOW (window),
                              gtk_ui_manager_get_accel_group (uimanager));

  vbox = gtk_vbox_new (FALSE, 0);
  gtk_box_pack_start_defaults (GTK_BOX (vbox), menubar);
  gtk_box_pack_start_defaults (GTK_BOX (vbox), toolbar);

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

  gtk_main ();
  return 0;
}
```

The first thing you need to do when using `GtkUIManager` to dynamically load menus and toolbars is to create an array of actions. It is possible to manually create every `GtkAction`, `GtkToggleAction`, or `GtkRadioAction` object, but there is a much easier way.

GtkActionEntry is a structure that holds an action name, stock identifier, label, accelerator, tooltip, and callback function. The content of the GtkActionEntry structure can be viewed in the following code snippet.

```
typedef struct
{
  const gchar *name;
  const gchar *stock_id;
  const gchar *label;
  const gchar *accelerator;
  const gchar *tooltip;
  GCallback    callback;
} GtkActionEntry;
```

The action name string must be the same as the action attribute of a menu or tool item in a UI definition for it to be used. Any of the attributes except for the action name can safely be set to NULL if they are not needed. If you specify a stock identifier, you do not need to specify a label or an accelerator unless you want to override their default values.

The keyboard accelerator is specified as a string that spells out its value. Acceptable keyboard accelerators include "<Control>a", "<Shift><Control>x", "F3", and so on. Some of the modifiers can also be abbreviated. For example, the Control key can be referenced with "<Ctrl>" or "<Ctl>". In short, the accelerator must be of the form that it can be parsed by gtk_accelerator_parse().

After you create lists of actions, you need to create a new GtkActionGroup that will hold all of the actions with gtk_action_group_new(). The name specified to this function will be used when associating key bindings with the actions.

An array of GtkActionEntry objects can be added to a GtkActionGroup by calling gtk_action_group_add_actions(). This function accepts the array of entries, the number of entries, and an optional data parameter that will be passed to each callback function.

```
void gtk_action_group_add_actions (GtkActionGroup *group,
                                   const GtkActionEntry *entries,
                                   guint n_entries,
                                   gpointer data);
```

If you need to pass different data parameters to different callback functions, you will have to manually create each GtkAction and add it to the group with gtk_action_group_add_action() or gtk_action_group_add_action_with_accel().

The next step is to create the GtkUIManager with gtk_ui_manager_new(). This object will be used to load the UI definitions and connect each item to its corresponding GtkAction. You then need to use gtk_ui_manager_insert_action_group() to add all of your action groups to the GtkUIManager. This function will add all of the actions from the group to the UI manager. Then, it will be able to match actions to elements in UI definitions to create appropriate widgets.

```
void gtk_ui_manager_insert_action_group (GtkUIManager *uimanager,
                                         GtkActionGroup *group,
                                         gint pos);
```

The third parameter of this function is an integer that states the position of the action group within the UI manager. Actions with the same name in groups with a lower position will take preference over those with higher positions.

Next, you will want to use gtk_ui_manager_add_ui_from_file() to load any number of UI files. In Listing 9-11, the menu.ui and toolbar.ui files were loaded with respect to the executable. The third parameter of this function is an optional GError object.

```
guint gtk_ui_manager_add_ui_from_file (GtkUIManager *uimanager,
                                       const gchar *filename,
                                       GError **error);
```

This function will load the content of each file. Each element is then matched up with objects added from an action group. The UI manager will then create all of the appropriate widgets according to the UI definition. An error will be output to the terminal if an action does not exist.

After the UI manager creates the widgets, you can load them based on name paths or the action if the name parameter does not exist, as shown in the following code. The two top-level widgets were the menu bar and toolbar found at "/MenuBar" and "/Toolbar". They are loaded with gtk_ui_manager_get_widget().

```
GtkWidget* gtk_ui_manager_get_widget (GtkUIManager *self,
                                      const gchar *path);
```

You have to give the absolute path to any widget when a path is required. In the absolute path, the <ui> element is omitted. The path is then built with the name attribute of each item. For example, if you wanted to access the GTK_STOCK_OPEN element in the menu bar, you call gtk_ui_manager_get_widget(), which would return the "/MenuBar/FileMenu/FileOpen" menu item.

Additional Action Types

Menu and toolbar items with stock images and keyboard accelerators are great, but what about using toggle buttons and radio buttons with GtkUIManager? For this, GTK+ provides GtkToggleActionEntry and GtkRadioActionEntry. The content of GtkToggleActionEntry follows:

```
typedef struct
{
  const gchar *name;
  const gchar *stock_id;
  const gchar *label;
  const gchar *accelerator;
  const gchar *tooltip;
  GCallback callback;
  gboolean is_active;
} GtkToggleActionEntry;
```

Note One advantage of using UI definitions is that the actual definition does not know anything about how the action is going to be implemented in your application. Because of this, the user can redesign a menu structure without needing to know how each action will be implemented.

In addition to `GtkActionEntry`, GTK+ provides `GtkToggleActionEntry`, which will create a toggle menu or tool item. This structure includes an additional member—`is_active`, which defines whether the button is initially set as active.

Adding an array of `GtkToggleActionEntry` objects is similar to adding normal actions except you have to use `gtk_action_group_add_toggle_actions()`. This function accepts an array of `GtkToggleActionEntry` objects, the number of actions in the array, and a pointer that will be passed to every callback function.

```
void gtk_action_group_add_toggle_actions (GtkActionGroup *group,
                                          const GtkToggleActionEntry *entries,
                                          guint num_entries,
                                          gpointer data);
```

Additionally, `GtkRadioActionEntry` allows you to create a group of radio actions. The `value` member is a unique integer that can be used to activate a specific radio menu item or radio tool button.

```
typedef struct
{
  const gchar *name;
  const gchar *stock_id;
  const gchar *label;
  const gchar *accelerator;
  const gchar *tooltip;
  gint value;
} GtkRadioActionEntry;
```

The radio actions are added to the action group with `gtk_action_group_add_radio_actions()`, which will group all of the radio buttons together. This function works the same as `gtk_action_group_add_toggle_actions()` except you need to specify two additional parameters.

```
void gtk_action_group_add_radio_actions (GtkActionGroup *group,
                                         const GtkRadioActionEntry *entries,
                                         guint num_entries,
                                         gint value,
                                         GCallback on_change,
                                         gpointer data);
```

The `value` parameter is the identifier assigned to the action that should be initially activated or set to `-1` to deactivate all by default. The callback function `on_change()` is called when the `changed` signal is emitted on a radio button.

Placeholders

When creating UI files, you may want to mark a location in a menu where other menu items can be added at a later time. For example, if you want to add a list of recent files to the File menu, you may not know how many files will be available for the list.

For this situation, GTK+ provides the `<placeholder>` tag. In the following line of code, a `<placeholder>` tag is defined that can be used to mark the location in the File menu that recent file menu items can be added.

```
<placeholder name="FileRecentFiles"/>
```

Within your application, you can use `gtk_ui_manager_add_ui()` to add new user interface information at the location of the placeholder. This function first accepts a unique unsigned integer that was returned by a call to `gtk_ui_manager_new_merge_id()`. You have to retrieve a new merge identifier every time you add a widget to the user interface.

```
void gtk_ui_manager_add_ui (GtkUIManager *uimanager,
                            guint merge_id,
                            const gchar *path,
                            const gchar *name,
                            const gchar *action,
                            GtkUIManagerItemType type,
                            gboolean top);
```

The next parameter of `gtk_ui_manager_add_ui()` is a path to the point where the new item should be added; this would be `"/MenuBar/File/FileRecentFiles"`, which is the path to the placeholder. Then, you should specify a name and action for the new widget followed by the type of UI item that is being added. UI item types are defined by the following `GtkUIManagerItemType` enumeration options:

- `GTK_UI_MANAGER_AUTO`: GTK+ will determine what type of widget is to be added.
- `GTK_UI_MANAGER_MENUBAR`: Add a `GtkMenuBar` widget. The location of the placeholder should be a direct child of a `<ui>` tag.
- `GTK_UI_MANAGER_MENU`: Add a `GtkMenu` as a child of a top-level widget.
- `GTK_UI_MANAGER_TOOLBAR`: Add a `GtkMenuBar`. The location of the placeholder should be a direct child of a `<ui>` tag.
- `GTK_UI_MANAGER_PLACEHOLDER`: Add a new placeholder, which can be added at any location in the user interface.
- `GTK_UI_MANAGER_POPUP`: Add a `GtkMenuBar`. This requires that the placeholder is located as a direct child of a `<ui>` tag.
- `GTK_UI_MANAGER_MENUITEM`: Add a `GtkMenuItem` as a child of a top-level widget.
- `GTK_UI_MANAGER_TOOLITEM`: Add a `GtkToolItem` as a child of a top-level `GtkToolbar` widget.
- `GTK_UI_MANAGER_SEPARATOR`: Add a separator into any type of top-level widget.
- `GTK_UI_MANAGER_ACCELERATOR`: Add a keyboard accelerator to a menu or toolbar.

The last parameter of gtk_ui_manager_add_ui() is a Boolean variable that positions the new UI element with respect to the given path. If set to TRUE, the UI element is inserted before the path. Otherwise, it is inserted after the path.

Custom Stock Items

From the last section, you will notice that GtkActionEntry accepts a stock identifier to add an image to the item. Because of this, you will, at some point, need to create your own custom stock icons that can be used for nonstandard menu and toolbar items. New stock items are created with three objects: GtkIconSource, GtkIconSet, and GtkIconFactory. Let us work from the bottom up.

GtkIconSource is an object that holds a GdkPixbuf or an image filename. It is meant to hold one variant of an image. For example, if you have an image that will be displayed differently when it is enabled or disabled, you would need to have multiple icon sources, one for each state. You may need multiple icon sources for different icon sizes, different languages, or different icon states.

Multiple icon sources are organized with GtkIconSet, which holds all of the GtkIconSource objects for one stock image. In some cases, your icon set may only have one image. While this is usually not the case, you can use gtk_icon_set_new_from_pixbuf() to skip the step of creating an icon source.

```
GtkIconSet* gtk_icon_set_new_from_pixbuf (GdkPixbuf *pixbuf);
```

After you have created all of the necessary icon sets, they are added to a GtkIconFactory, which is used to organize all of the stock items for a particular theme. Icon factories are added to a global list that GTK+ searches through to find stock items.

In this section, a number of new stock items are going to be created. Figure 9-8 is a screenshot of the new stock items that are created in Listing 9-12.

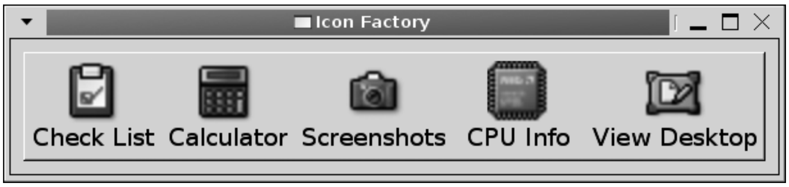


Figure 9-8. *Custom images added to the global icon factory*

In Listing 9-12, five new stock items are created including "check-list", "calculator", "screenshot", "cpu", and "desktop". A toolbar item is then created from each of the new stock items and displayed on the screen.

Listing 9-12. *Using GtkIconFactory (iconfactory.c)*

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

#define ICON_LOCATION "/path/to/icons/"

typedef struct
{
  gchar *location;
  gchar *stock_id;
  gchar *label;
} NewStockIcon;

const NewStockIcon list[] =
{
  { ICON_LOCATION"checklist.png", "check-list", "Check _List" },
  { ICON_LOCATION"calculator.png", "calculator", "_Calculator" },
  { ICON_LOCATION"camera.png", "screenshot", "_Screenshots" },
  { ICON_LOCATION"cpu.png", "cpu", "CPU _Info" },
  { ICON_LOCATION"desktop.png", "desktop", "View _Desktop" },
  { NULL, NULL, NULL }
};

static void add_stock_icon (GtkIconFactory*, gchar*, gchar*);

int main (int argc,
          char *argv[])
{
  GtkWidget *window, *toolbar;
  GtkIconFactory *factory;
  gint i = 0;

  gtk_init (&argc, &argv);

  window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  gtk_window_set_title (GTK_WINDOW (window), "Icon Factory");
  gtk_container_set_border_width (GTK_CONTAINER (window), 10);

  factory = gtk_icon_factory_new ();
  toolbar = gtk_toolbar_new ();
  /* Loop through the list of items and add new stock items. */
  while (list[i].location != NULL)
  {
    GtkToolItem *item;

    add_stock_icon (factory, list[i].location, list[i].stock_id);
    item = gtk_tool_button_new_from_stock (list[i].stock_id);
    gtk_tool_button_set_label (GTK_TOOL_BUTTON (item), list[i].label);
    gtk_tool_button_set_use_underline (GTK_TOOL_BUTTON (item), TRUE);
    gtk_toolbar_insert (GTK_TOOLBAR (toolbar), item, i);
    i++;
  }

  gtk_icon_factory_add_default (factory);
  gtk_toolbar_set_style (GTK_TOOLBAR (toolbar), GTK_TOOLBAR_BOTH);
  gtk_toolbar_set_show_arrow (GTK_TOOLBAR (toolbar), FALSE);
  gtk_container_add (GTK_CONTAINER (window), toolbar);

  gtk_widget_show_all (window);
  gtk_main ();
  return 0;
}

/* Add a new stock icon from the given location and with the given stock id. */
static void
add_stock_icon (GtkIconFactory *factory,
                gchar *location,
                gchar *stock_id)
{
  GtkIconSource *source;
  GtkIconSet *set;

  source = gtk_icon_source_new ();
  set = gtk_icon_set_new ();

  gtk_icon_source_set_filename (source, location);
  gtk_icon_set_add_source (set, source);
  gtk_icon_factory_add (factory, stock_id, set);
}
```

Creating a new icon factory, source, or set is as simple as calling `gtk_icon_factory_new()`, `gtk_icon_source_new()`, or `gtk_icon_set_new()`. Each of these functions creates an empty object that is not of any use in its current state.

In Listing 9-12, the icon source is initialized to an image found at the specified filename with `gtk_icon_source_set_filename()`. Alternatively, you can create the icon source out of a `GdkPixbuf` object with `gtk_icon_source_set_pixbuf()`.

Since we only needed one icon source in Listing 9-12 for each stock item, there was no need for further customization. However, it is possible to set the icon to be displayed for a specific size with `gtk_icon_source_set_size()`. This will tell GTK+ to only use this icon if the application needs the specified size.

```
void gtk_icon_source_set_size (GtkIconSource *source,
                               GtkIconSize size);
```

■ **Caution** If you need to set the icon source size, it will have no effect unless you pass `FALSE` to `gtk_icon_source_set_size_wildcarded()`. Otherwise, the icon source will be used for all sizes. This also goes for icon states, which must be unset with `gtk_icon_source_set_state_wildcarded()`.

Additionally, you can define the icon to be shown during a specific state defined by the `GtkIconState`. If you use `gtk_icon_source_set_state()`, you will want to make sure to define icons for all five states defined by the enumeration.

```
void gtk_icon_source_set_state (GtkIconSource *source,
                                GtkIconState state);
```

After you create your icon sources, you will need to add them all to an icon set with `gtk_icon_set_add_source()`. This function accepts the `GtkIconSet` and the icon source that will be added.

```
Void gtk_icon_set_add_source (GtkIconSet *iconset,
                              const GtkIconSource *source)
```

If you unset any wildcards in the icon sources, you will want to make sure to define stock icons for every possible state or size. Adding a single icon source with both the state and the size indicated as the wildcards usually does this. If there is a more specific icon, it will be used. If the appropriate icon is not found, the wildcard icon will be used. This wildcard image may be lightened or altered in some other way to fit the occasion.

Next, you need to add each `GtkIconSet` to the icon factory with `gtk_icon_factory_add()`. This function accepts the stock identifier that will be used to reference the icon. Normally, you will want to name this `"myapp-iconname"`, where `"myapp"` is replaced by the name of your application.

```
void gtk_icon_factory_add (GtkIconFactory *factory,
                           const gchar *stock_id,
                           GtkIconSet *iconset);
```

If the stock identifier already exists, the new item will replace it, so by using your application name in the stock identifier, you avoid overriding any default stock items.

Any stock items added to your icon factory are not available until you add it to the global list of icon factories with `gtk_icon_factory_add_default()`. Normally, a separate icon factory will exist for each graphical library that includes its default icons.

Test Your Understanding

The following two exercises give an overview of what you have learned about menus and toolbars throughout the chapter.

In addition to completing them, you may want to create examples of pop-up menus with other widgets that do not support them by default. Also, after finishing both of these exercises, you should expand them by creating your own stock icons that are used in place of the default items.

Exercise 9-1. Toolbars

In Chapter 7, you created a simple text editor using the `GtkTextView` widget. In this exercise, expand on that application and provide a toolbar for actions instead of a vertical box filled with `GtkButton` widgets.

While manual toolbar creation is possible, in most applications, you will want to utilize the `GtkUIManager` method of toolbar creation. Therefore, use that method in this exercise. You should also make use of built-in stock items or create your own with `GtkIconFactory`.

Oftentimes, it is advantageous for an application to provide the toolbar as a child of a handle box. Do this for your text editor, placing the toolbar above the text view. Also, set up the toolbar so that the textual descriptor is shown below every tool button.

This first exercise taught you how to build your own toolbars. It also showed you how to use the `GtkHandleBox` container. In the next exercise, you will reimplement the Text Editor application with a menu bar.

Exercise 9-2. Menu Bars

In this exercise, implement the same application as in Exercise 9-1, except use a menu bar this time. You should continue to use `GtkUIManager`, but the menu does not need to be contained by a `GtkHandleBox`.

Since tooltips are not shown for menu items automatically, use a status bar to provide more information about each item. The menu bar should contain two menus: File and Edit. You should also provide a Quit menu item in the File menu.

Summary

In this chapter, you learned two methods for creating menus, toolbars, and menu bars. The first method was the manual method, which was more difficult but introduced you to all of the necessary widgets.

The first example showed you how to use basic menu items to implement a pop-up menu for a progress bar. This example was expanded on in order to provide keyboard accelerators and more information to the user with the `GtkStatusbar` widget. You also learned about submenus as well as image, toggle, and radio menu items.

The next section showed you how to use menu items with submenus to implement a menu bar with a `GtkMenuShell`. This menu bar could be displayed horizontally or vertically and forward or backward.

Toolbars are simply a horizontal or vertical list of buttons. Each button contains an icon and label text. You learned about three additional types of toolbar buttons: toggles, radio buttons, and tool buttons with a supplemental menu.

Then, after much hard work, you were taught how to create dynamically loadable menus. Each menu or toolbar is held in a UI definition file, which is loaded by the `GtkUIManager` class. The UI manager associates each object with the appropriate action and creates the widgets according to the UI definition.

Last, you learned how to create your own custom stock icons. It is necessary to create your own icons, because arrays of actions require a stock identifier to add an icon to an action.

In the next chapter, we are going to take a short break from coding and cover the design of graphical user interfaces with the Glade User Interface Builder. This application creates user interface XML files, which can be dynamically loaded when your application starts. You will then learn how to handle these files programmatically with Libglade.