Lesson 3: Containers

1 Container Widgets and Packing

When you design an application with a graphical user interface, you put various widgets inside of one another and implicitly define a hierarchy of what's inside of what. This is a containment hierarchy. Some of the widgets you use have specific purposes, such as buttons, text entry boxes, and menus. If you design your GUI on paper, you draw these widgets where you want them and making them the sizes that you want them to be. However, getting them to be in those specific positions with their specific sizes using a library like GTK+ requires that you use widgets whose primary purpose is for laying out other widgets. These widgets are called *container* widgets.

In Lesson 2, we introduced the `GtkContainer` class, which is the ancestral class of all container widgets and which we now cover in more detail. Recall that containers can be partitioned into two categories: (1) those that can hold only a single child widget, and (2) those that can hold more than one. Containers that can contain only a single widget are called *decorator containers*, because their principal purpose is to add functionality and decorative effects to the child widget. Containers that can hold several children are called *layout containers*, because they are used primarily for laying out the child widgets within their (GDK) windows. Layout containers assign sizes and positions to their children.

1.1 The `GtkContainer` Class

Before we look at their concrete subclasses, we will examine what functionality and properties `GtkContainers` possess. They have just three properties:

"border-width"	`guint`	The width of the empty border outside the container's children.
"child"	`GtkWidget*`	Pointer to the container's child (if just a single child)
"resize-mode"	`GtkResizeMode`	Indicator for how the container is resized

Since a property consists of a string and a value, we will write the string value followed by the type of the value. Each of these properties has accessor and mutator methods in the `GtkContainer` class. For example, the accessor and mutator methods of the border-width" property are

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

The `GtkContainer` base class supports four signals. The signals and the conditions under which they are emitted by a container widget are:

`add`	This is emitted by the widget whenever a child is added to the container, either explicitly by your code or implicitly as a result of a packing operation.
`check-resize`	This is emitted when the container is checking whether it has to resize itself to make room for its children.
`remove`	This is emitted when a child has been removed from the container.
`set-focus-child`	This is emitted when a child of the container has been given focus by the window manager.

The GTK+ documentation has the signatures of the callbacks that are needed for each of these signals. To find them, do a search for the `GtkContainer` class, and on that page, search for the signal name, such as "set-focus-child". This will find an entry in the *Signals* section of the page. Click on the link (all signals are links in this section) and it will bring up the page with the callback's documentation. In this case you will see

```
The "set-focus-child" signal

    void    user_function    (GtkContainer *container,
                              GtkWidget    *widget,
                              gpointer      user_data)      : Run First

    container : the object which received the signal.
    user_data : user data set when the signal handler was connected.
```

This tells us that a callback to be connected to the "set-focus-child" signal must have the above three parameters, and that the user_data will be whatever was set in the g_signal_connect() function when the callback was connected to the signal. The "Run First" designation indicates that the default signal handler for a set-focus-child signal emitted on a container always runs *before* any user-defined callback. We will cover the details of signal handler invocation in a later lesson.

1.2 The GtkBin Class

GtkBin is the abstract base class of all decorator containers. Its derived classes include GtkWindow, GtkAlignment, GtkFrame, GtkButton, GtkItem, GtkComboBox, GtkEventBox, GtkExpander, GtkHandleBox, GtkToolItem, GtkScrolledWindow, and the GtkViewport . Buttons and combo boxes are familiar widgets; they are used as controls to cause actions to take place. Expanders and handle boxes are specialized containers that change the size, position, or appearance of their children. Scroll windows and viewports are important containers that provide scrolling for their child widgets and they will be covered in a later lesson. Frames are purely decorative – they help to organize the appearance of a window. Alignments serve a very important purpose – they control the size and alignment of their child widgets, meaning whether the child is centered, or bound to a side, and whether it grows or resizes with the containing window. Event boxes are also important, because many widgets do not have their own GDK window and therefore cannot catch events. Event boxes are widgets that can catch events and do nothing else, and when a windowless child is placed inside an event box, it is able to catch events, or more accurately, the event box can catch events on its behalf. We will cover the GtkAlignment, the GtkEventBox, and the GtkButton in this chapter.

The GtkBin class does not, in itself, provide much functionality. It has only one method:

```
    GtkWidget* gtk_bin_get_child ( GtkBin* bin);
```

which, given a GtkBin pointer, returns a pointer to its only child. This function is not needed most of the time, but sometimes it is precisely what one needs. Suppose that, in an application, when the user moves the cursor over an image, a pop-up window appears that provides information about the pixel at a given point. Since a GtkImage widget does not have its own GDK window, it cannot catch events. However, you can make the image a child of a GtkEventBox, which is a subclass of GtkBin that does have its own (GDK) window and therefore can catch events. A callback for the GtkEventBox can use the gtk_bin_get_child() function to get the GtkImage and from that get the information it needs to display in the pop-up window. The callback would look something like

```
gboolean button_pressed (GtkWidget *event_box,
                         GdkEventButton *event,
                         GtkWidget *label )
{
    GtkWidget  *image  = gtk_bin_get_child( GTK_BIN ( event_box ) );
    GdkPixbuf  *pixbuf = gtk_image_get_pixbuf(GTK_IMAGE ( image ));
    // get the pixels from the image and continue processing ...
    // ...
    return FALSE;
}
```

In a later chapter dealing with manipulation of images, we will see an application that does precisely this, and we will see how the callback works.

1.3 Layout Containers

The layout containers that are most useful for organizing your visual elements in a window are:

`GtkHBox`	Horizontal box with multiple children
`GtkVBox`	Vertical box with multiple children
`GtkTable`	Grid with rows and columns for multiple children
`GtkLayout`	Versatile scrollable container for multiple children

Horizontal and vertical boxes are one-dimensional containers; they let you pack widgets in a single dimension. Tables and layouts are two dimensional. Layouts are very powerful in that they create a blank canvas within which widgets can be positioned in various ways, and they also have built-in scrolling capability.

In addition to the above are the `GtkTextView` and the `GtkTreeView`. Although these are subclasses of the `GtkContainer`, they are somewhat special purpose: the `GtkTextView` is designed specifically for displaying text, and the `GtkTreeView` is designed to display lists and trees. These will be covered in chapters of their own.

1.4 Packing Boxes

Boxes are invisible, one-dimensional containers into which widgets can be *packed*. *Packing* refers to the process of putting widgets into boxes. There are two basic boxes[1]: a horizontal box `GtkHBox`, and a vertical box `GtkVBox`. I will refer to them as an *hbox* and *vbox* respectively. When widgets are packed into an hbox, the objects are inserted horizontally from left to right or right to left depending on which packing function used. In a vbox, widgets are packed from top to bottom or vice versa. You may use any combination of boxes inside or beside other boxes to create a layout.

1.4.1 Box Creation

To create a new hbox or vbox, use the appropriate `gtk_*_new()` function, which has two parameters:

```
    GtkWidget* gtk_hbox_new( gboolean homogeneous, guint spacing);
```

or

```
    GtkWidget* gtk_vbox_new( gboolean homogeneous, guint spacing);
```

If `homogeneous` is true, then all child widgets are given equal space in the dimension of the box (e.g., horizontal space in an hbox, vertical in a vbox.) GTK uses the size of the largest widget as a starting value. If, for example, an hbox has six buttons inside, and one needs to be 100 pixels long because it contains a label of that length, then no matter how small the other buttons' labels might be, they will each be given

[1]There is also a class called `GtkButtonBox`, with subclasses `GtkHButtnBox` and `GtkVButtonBox`. These are used for organizing buttons.

100 pixels if `homogeneous` is true, provided that the size negotiation can grant that much space to each, as per the discussion in Lesson 2 entitled *Size Requisition and Allocation.* Whatever the allocation is, though, each child receives the same amount. In an hbox, all widgets are made the same height, which is determined by the height of the tallest widget, whether or not the hbox is homogeneous. The analogous statement is true of vboxes.

The value of the `spacing` parameter is the amount of space to place *between* widgets within the box, also in the corresponding dimension. The hbox will make its size request to its parent based on the total space it needs to do this. For example, if there are six buttons in an hbox, and each button needs to be 100 pixels wide, and spacing is 20, then the hbox will ask for $6 \cdot 100 + 5 \cdot 20 = 700$ pixels (because there are 5 spaces between the 6 widgets.)

1.4.2 Box Packing

The two principal functions for packing boxes are `gtk_box_pack_start()` and `gtk_box_pack_end()`. The `gtk_box_pack_start()` function will pack from the top and work its way down in a vbox, and pack from left to right in an hbox. `gtk_box_pack_end()` will do the opposite, packing from bottom to top in a vbox, and right to left in an hbox.

The prototype for `gtk_box_pack_start()` is

```
void gtk_box_pack_start (GtkBox *box,
                         GtkWidget *child,
                         gboolean expand,
                         gboolean fill,
                         guint padding);
```

The parameters and their interactions are fairly complicated and depend upon whether the box was created to be homogeneous or not. In all cases, the parameters control only the widget being packed into the box with the given call, not all child widgets in the box . The effects of the parameters are summarized as follows. In all cases, the space and size referred to is always in the direction of the box, i.e., horizontal for hboxes and vertical for vboxes. All children have equal size in the other dimension.

`expand`	If `TRUE`, the new child is to be given its share of the extra space within the box. The extra space will be divided evenly among all children of the box that set `expand` to `TRUE`. For example, if there are 6 child widgets but only 4 have set `expand` to `TRUE`, then these 4 share the extra space in the box. If the box has 120 extra pixels, then each of the 4 will get 30 pixels extra.
`fill`	If `TRUE`, then any space given to the child by the `expand` option is actually allocated within the child, and if FALSE, then it is used as padding outside of it (on both sides). In a non-homogeneous box, this parameter has no effect if `expand` is set to `FALSE`.
`padding`	This is the amount of extra space in pixels to put on each side of the child in the direction of the box, over and above the global amount specified by the box's `spacing` property. If the child is at one of the ends of the box, then `padding` pixels are also put between the child and that edge of the box. For example, if a child requests padding of N pixels, then N pixels are placed on each side of the child, adding $2N$ pixels to the size of the box. If 6 children each request N pixels, then $12N$ pixels are added to its size.

Comments. To be sure that you understand how these parameters control widget size and position, the following observations might be useful.

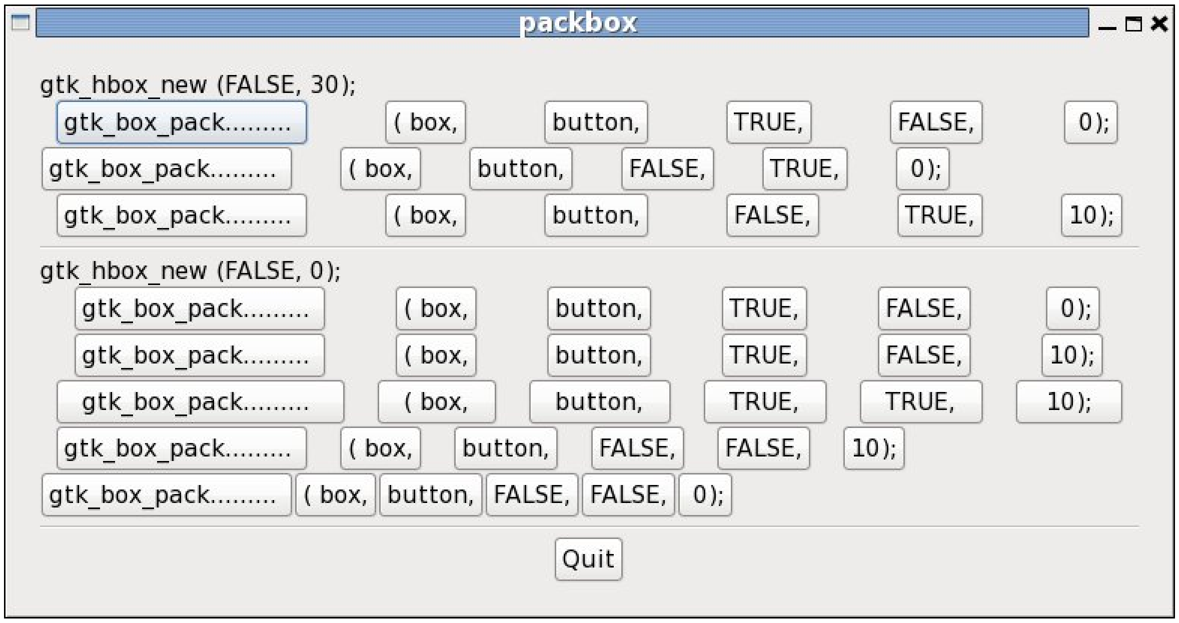
packbox
gtk_hbox_new (FALSE, 30);
gtk_box_pack......... (box, button, TRUE, FALSE, 0);
gtk_box_pack......... (box, button, FALSE, TRUE, 0);
gtk_box_pack......... (box, button, FALSE, TRUE, 10);
gtk_hbox_new (FALSE, 0);
gtk_box_pack......... (box, button, TRUE, FALSE, 0);
gtk_box_pack......... (box, button, TRUE, FALSE, 10);
gtk_box_pack......... (box, button, TRUE, TRUE, 10);
gtk_box_pack......... (box, button, FALSE, FALSE, 10);
gtk_box_pack......... (box, button, FALSE, FALSE, 0);
Quit

Figure 1: Screen-shot of packbox demo showing effects of parameters.

- In a homogeneous box, the size of the largest child is how big the remaining child allotments will be. If `fill` is `TRUE` for each of them, the children will be the same size. A child for which `fill` is `FALSE`, will appear smaller but still has the same allotment.
- In a non-homogeneous box, `fill` is ignored if `expand` is `FALSE`.
- In a homogeneous box, the value of the `expand` parameter is ignored; it is set to `TRUE` for each child packed into the box. If `fill` is `FALSE`, for a particular child, then any extra space is outside of that child as padding, and if it is `TRUE`, the space is within the child.
- In any box, the padding specified by the `padding` parameter is in addition to any spacing specified by the box's `spacing` parameter: `padding` pixels are added to each side of the widget. If extra space is allocated to a widget that set `expand` to `TRUE`, that space is in addition to the `padding`, either on the outside if `fill` is `FALSE`, or on the inside is fill is `TRUE`.

Figure 1 demonstrates the principles. The figure is a screen-shot of a program that creates a vertical box containing 13 child widgets[2]. Eight of these widgets are horizontal boxes packed in different ways. The first three are each created to be non-homogeneous with a spacing of 30 pixels. That is the purpose of the label "`gtk_hbox_new(FALSE,30);`" above those three boxes. The next five are non-homogeneous boxes with a spacing of 0 pixels.

Each hbox has six child widgets and within a single hbox, all six children are packed using the same parameters to `gtk_box_pack_start()`. The children are buttons whose labels are the words of the call to `gtk_box_pack_start()`. For example, in the first hbox, the children were each packed with the call

```
gtk_box_pack_start (GTK_BOX (box), button, TRUE, FALSE, 0);
```

so the buttons have labels "`gtk_box_pack.......`", "`(box,`", "`button,`" and so on. Some words are deleted to make the window smaller. If you study this figure you can verify the behavior of the packing function.

[2]This program is contained in the GTK+ 2 tutorial written by Tony Gale and Ian Main, and modified slightly by Stewart Weiss. The link is http://developer.gnome.org/gtk-tutorial/2.90/book1.html.

First, one of the hboxes determined the width of the window, forcing it to be as wide as it is. Which one? The third hbox is the culprit. In addition to 30 pixels of spacing, each child demands 10 pixels of padding. This is a total of $5 \times 30 + 6 \times 2 \times 10 = 270$ pixels added to the lengths of the buttons.

The eighth hbox is what it would look like if no spacing and no padding were added, so the third is 270 pixels wider than the eighth. Also, `expand` is `FALSE` in this third hbox for all of the children and `fill`, although `TRUE`, is ignored. So the buttons are the minimum sizes that they need to be.

Comparing the first hbox to the third. It has `expand` set to `TRUE`, and `padding`=0. The spacing is 30, but there are 120 extra pixels to be shared. Since `fill` is `FALSE`, the space is outside the widgets. This is why the first hbox looks the same as the third – the extra 120 pixels are distributed among six widgets, each of which pads itself with 10 pixels on each side.

The second hbox's widgets set `expand` to `FALSE`, so the first button aligns against the left border of the window, and the remaining ones are 30 pixels apart from each other.

Looking at the fourth and fifth hboxes, they appear to be identical in terms of size and position to each other and to the third. This is for the same reason that the first is identical to the third.

The sixth is different because `fill` is `TRUE` for those widgets. The extra space is put *inside* the widgets, not as padding. However, because `padding` was set to 10 pixels, there are 20 pixels between adjacent buttons and 10 at the ends.

Lastly, the seventh is different because `spacing` = 0 and `padding` = 10, so they are packed 20 pixels apart and do not expand.

Everything said about `gtk_box_pack_start()` is also true about `gtk_box_pack_end()` except that it packs in the opposite direction, meaning right to left for hboxes, and bottom to top for vboxes. If for example, you want to put a button or a status bar at the bottom of a window and a few other widgets at the top, then you would use a non-homogeneous vbox and then use `gtk_box_pack_start()` to put the upper widgets at the top, and `gtk_box_pack_end()` to put the button or status bar at the bottom. To prevent them from resizing as the window is resized, you would pack them with `expand` and `fill` both false.

GTK provides a quick and dirty method called `gtk_box_pack_start_defaults()`, but it is deprecated since 2.14 and should not be used.

1.5 Tables

Tables, objects of the `GtkTable` class, provide a grid within which you can place widgets. The nice thing about tables is that widgets can span multiple rows and/or columns, and they can be of uniform or non-uniform row height or column width. Tables are a suitable container when you want to organize child widgets within a grid or in any regular two-dimensional pattern. Tables consist of *cells*, which are the individual units of the grid. A table with 5 rows and 4 columns has 20 cells.

To create a new table, use

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

The first two parameters are pretty obvious – they specify the number of rows and columns in the table. The third parameter determines whether or not homogeneous spacing is used. It also has implications about how individual cells are resized when the table is resized. If `homogeneous` is true, then all rows are of equal height and all columns are of equal width, and the heights and widths are determined by the tallest and widest widgets within the table. If `homogeneous` is false, then rows may be of unequal heights and columns of unequal widths, and the height of a row is determined by the tallest widget in the row, and the width of a column, by the widest widget within the column.

Although there are several methods in the `GtkTable` class, I will describe just two of them in detail here. These are the most important, and knowing just these two, you can pretty much do 90% of the common programming tasks associated with tables.

To insert a child widget into a table, you can use either `gtk_table_attach()`, or `gtk_table_attach_defaults()`. The former requires that you specify all of the positioning and resizing parameters, whereas the latter supplies default values for you.

```
void gtk_table_attach( GtkTable *table,
                       GtkWidget *child,
                       guint left_attach,
                       guint right_attach,
                       guint top_attach,
                       guint bottom_attach,
                       GtkAttachOptions xoptions,
                       GtkAttachOptions yoptions,
                       guint xpadding,
                       guint ypadding);
```

The first parameter is the table into which to insert the widget, and the second is the widget to insert. To understand the next four parameters, you need to know the table's coordinate system. A table has a coordinate system in which the upper-left corner is at (0,0) and the lower-right corner is at (rows, columns), where these are the current number of rows and columns respectively (yes, the table can change dynamically.)

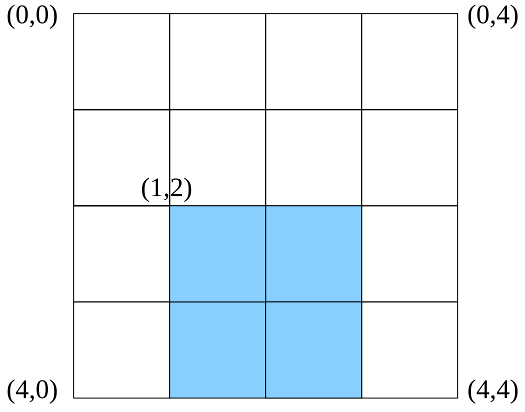

Figure 2: GtkTable coordinate system

With this in mind, the next four parameters are the left column coordinate, the right column coordinate, the top row coordinate and the bottom row coordinate of the widget. If they are (1, 3, 2, 4), then the widget will appear in the shaded ares in Figure 2, occupying four cells.

The next two parameters specify the attachment options in the x-direction and y-direction respectively, and they are bit-wise-OR of three boolean flags. According to the API documentation:

GTK_EXPAND If this flag is set, and the table's `homogeneous` flag is `FALSE`, then the widget is allowed to grow if the table grows in the given axis. If it is not set, the widget will remain the same size.

GTK_SHRINK If this flag is set, and the table's `homogeneous` flag is `FALSE`, then the widget is allowed to shrink if the table is reduced in size in the given axis. If it is not set, the widget will remain the same size.

GTK_FILL This flag determines whether any extra space given to the child widget will be within the widget or not. If it is false, the extra space is outside the widget but within the allocated grid cells. If true, it is within the widget.

You will not see the effect of the attachment parameters if the table is homogeneous. The last two parameters specify the padding around the widget in the x and y directions respectively, in pixels, both left and right, and above and below, respectively.

I have more to say about the `GTK_EXPAND` parameter, verified by experiments I have performed with tables. The behavior I observed does not seem to be documented anywhere. For simplicity, I only describe behavior in the x-direction, but it applies to the y-direction as well.

1. If any widget in a given row has the `GTK_EXPAND` bit set in the x-direction, then all widgets in its column will also expand horizontally, and will do so equally. Otherwise, the column in which that widget is placed would have "crooked" boundaries (and the corresponding statement is true about the y-direction.)

2. The `GTK_EXPAND` parameter makes the widget *very greedy*. If there are adjacent cells that are unoccupied and the `GTK_EXPAND` bit is set, then the widget will expand to occupy those cells, provided that all of the widgets in its column can also expand in that same direction. Figure 3 illustrates what happens. This table has 5 rows and 3 columns. Buttons are attached to the first and third columns but not to the second. Buttons 2, 5, 8, 11, and 14 would have been attached to the second column, but for this example, they are not, so the second column's space is free in rows 1, 3, 4, and 5. Button 4 is attached to two cells in the second row: (2,1) and 2,2), so it takes up column 2 in its row.

   All widgets are attached with `GTK_FILL` set in both-directions so that you can see how much space is available in their cells. In Table 5.1 on the left, no button except Button 4 is attached with `GTK_EXPAND` in the x-direction. Notice that in this table, the second column is therefore empty, except for Button 4, which was attached to columns 1 and 2 in its row. Because Button 4 has its `GTK_EXPAND` bit set, it expands to take up any extra space. Notice that the buttons in column 1 have expanded to be much larger than the ones in column 3. None of the buttons in column 1 have `GTK_EXPAND` set, but they are expanding anyway. I am reasonably certain that the algorithm used when a widget spanning N columns has the `GTK_EXPAND` bit set horizontally, is that, if none of the columns has the `GTK_EXPAND` bit set explicitly, then all columns that it spans become expandable equally, and that if one or more do, then those that have it set explicitly share the space equally.

   A slight change was made to the table on the right, named Table 5.2. The `GTK_EXPAND` bit was turned on for Button 1 in the x-direction. This made Button 1 greedy, and it grabbed the space in column 2. Because Button 1 expanded, and all of the other buttons in its columnwere able to expand, all did so (except for Button 4, which already had that real estate.) So even though Button 4 is the only widget that was attached across two columns, all of the buttons in column 1 span both columns.

Figure 4 illustrates that the behavior described above cannot take place if there is any obstruction in the cells into which expansion could take place. Table 5.3 has a slight change relative to Table 5.1. Button 15 was moved from the lower right hand corner into the cell to its left. It does not have the `GTK_EXPAND` bit set in the horizontal dimension, but because Button 4 does, it is expandable. In Table 5.4, Button 1 was made expandable, just as it was in Table 5.2, but it does not expand to occupy column 2 space, nor do any other buttons in its column. This is because Button 15 prevents Button 13 from expanding, so no others can expand.

Notice though that the buttons occupy a different proportion of the row than they did before. My conjecture about the algorithm, as stated above, is that because Button 1 set its `GTK_EXPAND` bit, column 1 was expandable, so the `GTK_EXPAND` bit on Button 4 did not divide equally among columns 1 and 2, but was given all to column 1.

One other note: if a widget in a given cell does not have the `GTK_FILL` bit set in a given direction, then the widget will be centered along that dimension in its allotted space. For example, if there are 100 pixels available for it horizontally, but it is only 40 pixels wide, and `GTK_FILL` is not set in the x-direction, it will have 30 pixels on its left and right (plus any padding that has been set).

If you do not want to deal with any of this and are willing to accept the default values for the last six parameters, you can use the following function. It sets the attachment parameters to (`GTK_EXPAND | GTK_FILL`) in both directions and sets the padding to 0. In many cases, this is sufficient.

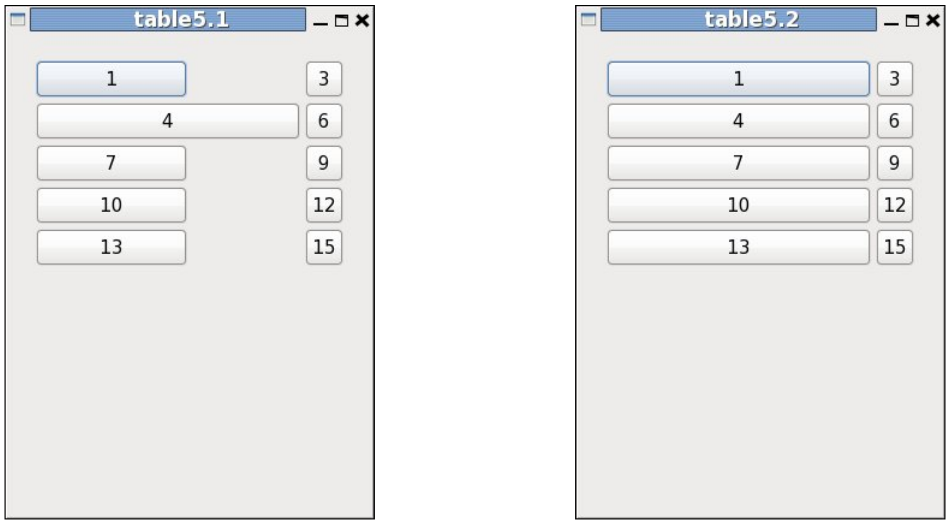

Figure 3: Expanding across unoccupied cells.

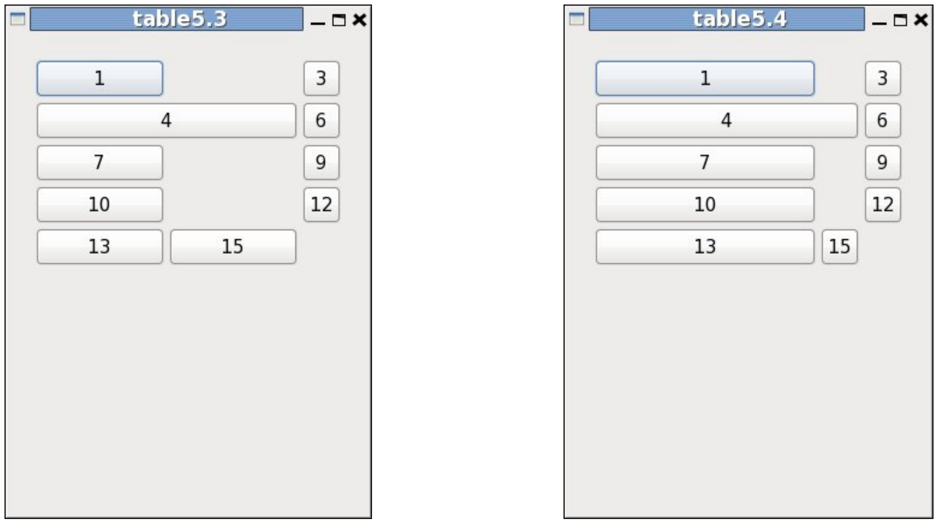

Figure 4: Prevented from expanding across unoccupied cells.

```
void gtk_table_attach_defaults (GtkTable *table,
                                GtkWidget *widget,
                                guint left_attach,
                                guint right_attach,
                                guint top_attach,
                                guint bottom_attach);
```

The GtkTable class has a few other methods that you might find useful include:

```
void      gtk_table_set_homogeneous        (GtkTable *table,
                                             gboolean homogeneous);
gboolean  gtk_table_get_homogeneous        (GtkTable *table);
```

If the program needs to dynamically change whether or not a table is homogeneous, it can use the first function, and it can use the second to check whether it is or not.

```
void      gtk_table_set_row_spacing        (GtkTable *table,
                                             guint row,
                                             guint spacing);
```

This changes the space between a given table row and the subsequent row. There is a similar function for columns, and there are functions that can change the spacing for all rows and for all columns.

1.5.1 Example

The following listing shows how a table can be used to organize a window. Most comments have been removed to save space.

```

void on_button_click( GtkWidget *widget,
                      gpointer   data )
{
    g_print ("Button number %s was clicked\n", (char *) data);
}

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

    gtk_init (&argc, &argv);

    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title (GTK_WINDOW (window), "Table Demo 1");
    gtk_container_set_border_width (GTK_CONTAINER (window), 10);

    table = gtk_table_new (3, 4, FALSE);
    gtk_container_add (GTK_CONTAINER (window), table);

    button = gtk_button_new_with_label ("button 1");
    g_signal_connect (G_OBJECT (button), "clicked",
                          G_CALLBACK (on_button_click),
                      (gpointer) "1");

    // Insert first button into cell (1,1) of the table
    gtk_table_attach (GTK_TABLE (table), button,
                      0, 1, 0, 1, 0, 0, 2, 2);

    button = gtk_button_new_with_label ("button 2");
    g_signal_connect (G_OBJECT (button), "clicked",
                      G_CALLBACK (on_button_click), (gpointer) "2");

    // Insert second button into cell (1,2) of the table
    gtk_table_attach (GTK_TABLE (table), button,
                      1, 2, 0, 1, 0, 0, 2, 2);

    // Create quit button, insert into cells (2,1) and (2,2) and
    // connect the button click to gtk_main_quit on the window widget
    button = gtk_button_new_with_label ("Quit");
    gtk_table_attach (GTK_TABLE (table), button, 0, 2, 1, 2);
    g_signal_connect(G_OBJECT (window), "destroy",
                                G_CALLBACK (gtk_main_quit), NULL);

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

    label = gtk_label_new("Column␣3␣Row␣1");
    gtk_table_attach (GTK_TABLE (table), label,
                      2, 3, 0, 1, 0, 0, 10, 10);

    label = gtk_label_new("Column␣3␣Row␣2");
    gtk_table_attach (GTK_TABLE (table), label,
                      2, 3, 1, 2, 0, 0, 10, 10);

    // Create a label-less frame, put a GtkLabel into it, and
    // attach it to the table, spanning the top two rows of column 4.
    // Make it expand and fill with 0 padding so that we can see the
    // cell's boundaries.
    frame = gtk_frame_new(NULL);
    label = gtk_label_new("Column␣4\nusing␣two␣rows");
    gtk_container_add (GTK_CONTAINER (frame), label);
    gtk_table_attach (GTK_TABLE (table), frame, 3, 4, 0, 2,
                      GTK_EXPAND|GTK_FILL, GTK_EXPAND|GTK_FILL, 0, 0);

    frame = gtk_frame_new(NULL);
    label = gtk_label_new("This␣is␣the␣third␣row␣of␣the␣table.");
    gtk_container_add (GTK_CONTAINER (frame), label);
    gtk_table_attach  (GTK_TABLE (table), frame, 0, 4, 2, 3,
                       GTK_EXPAND|GTK_FILL, GTK_EXPAND|GTK_FILL, 0, 0);

    gtk_widget_show_all (window);
    gtk_main ();

    return 0;
}
```

The table has three rows and four columns and is non-homogeneous, so that the sizes of the cells can vary. It demonstrates how a widget can span both vertical and horizontal cells. It also introduces the `GtkFrame` widget, which is a window-less widget whose primary purpose is to create a rectangular frame with an optional label around its child. By attaching the frame with the `GTK_EXPAND` and `GTK_FILL` options enabled, we can see the size of the cells it occupies.

1.6 Event Boxes

The `GtkEventBox`, a subclass of `GtkBin`, exists because some widgets do not have their own GDK windows and as a consequence cannot react to events. An event box has its own GDK window and can catch events for a window-less child widget. This is how you can turn a widget that cannot catch events, such as a `GtkImage`, `GtkButton`, or a `GtkLabel`, to name just a few, into one that can. The complete list of window-less widgets, as of GTK+-2.24, is

```
GtkAlignment, GtkArrow, GtkBin, GtkBox, GtkButton, GtkCheckButton, GtkFixed,
GtkImage, GtkLabel, GtkMenuItem, GtkNotebook, GtkPaned, GtkRadioButton,
GtkRange, GtkScrolledWindow, GtkSeparator, GtkTable, GtkToolbar,
GtkAspectFrame, GtkFrame, GtkVBox, GtkHBox, GtkVSeparator, GtkHSeparator
```

Event boxes are relatively simple to use and do not have many methods of their own. You create a new event box with `gtk_event_box_new()`. The following example program demonstrates a simple use of an event box.

1.6.1 A First Event Box Example

The two listings below demonstrate a relatively simple use of an event box. All error-checking and most comments have been removed to save space. The first listing is the callback function that handles the prescribed events, which in this case are button press and button release events. The event count is passed as the third parameter to the callback; because it is a pointer, it is dereferenced in the callback in order to increment it.

```
Listing: Callback for Button Events
gboolean on_button_event (GtkWidget      *eventbox,
                          GdkEventButton *event,
                          guint          *count)
{
    (*count)++;
    switch ( event->type ) {
    case GDK_BUTTON_PRESS:
        g_print( "Button press event " );
        break;
    case GDK_BUTTON_RELEASE:
        g_print( "Button release event " );
        break;
    case GDK_2BUTTON_PRESS:
        g_print( "2Button press event " );
        break;
    case GDK_3BUTTON_PRESS:
        g_print( "3Button press event " );
        break;
    default:
        g_print( "Some other event " );
    }
    g_print( " #%d at (%d,%d)\n",
              *count, (gint) event->x, (gint) event->y );
    return TRUE;
}
```

The callback checks which type of event occurred and prints a message on the standard output accordingly. The message includes the type of event, the current value of the event counter, and the coordinates relative to the event box where the event occurred.

```
Listing: A Simple Event Box Example
int main (int argc, char *argv[])
{
    GtkWidget     *window;
    GtkWidget     *eventbox;
    GtkWidget     *image;
    guint         event_count = 0;

    gtk_init (&argc, &argv);

    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title (GTK_WINDOW (window), "Event Box Demo 1");
    gtk_container_set_border_width (GTK_CONTAINER (window), 10);
    gtk_window_set_resizable(GTK_WINDOW (window), FALSE);

    eventbox = gtk_event_box_new();

    // Create an image from a file
    image    =  gtk_image_new_from_file (argv[1]);

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

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

    g_signal_connect (G_OBJECT (eventbox), "button_press_event",
                      G_CALLBACK (on_button_event),
                      (gpointer) &event_count);

    g_signal_connect (G_OBJECT (eventbox), "button_release_event",
                      G_CALLBACK (on_button_event),
                      (gpointer) &event_count);

    gtk_widget_set_events (eventbox, GDK_BUTTON_PRESS_MASK);
    gtk_widget_show_all (window);

    gtk_main ();
    return 0;
}
```

The program creates a top level window with an event box inside it. Inside the event box, it places a `GtkImage` widget, which was instantiated by loading an image from a file using `gtk_image_new_from_file()`. Since at this point, we do not yet know how to create file chooser dialog boxes, the image file must be supplied as a command line argument. Therefore, the program must be run from the command line. The acceptable image formats may vary from one system to another, but usually, you can supply any of JPG, GIF, BMP, PNG, and TIF. It will even animate if given GIF animations.

The program uses `gtk_widget_set_events()`

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

to set a GDK mask on the widget, defining which events it should catch. The second argument must be a value of the `GdkEventMask` enumeration. *This function must be called before the event box is realized.* An alternative is to call `gtk_widget_add_events()` *after realizing the widget*, also passing it an event mask. The `GdkEventMask` value `GDK_BUTTON_PRESS_MASK` enables the event box to catch any button press or button release event, including single, double, and triple button presses and button releases. If you build and run this you will see that

- If you single click, a button press and release are caught;
- If you double click, five events are caught: `GDK_BUTTON_PRESS`, `GDK_BUTTON_RELEASE`, `GDK_BUTTON_PRESS`, `GDK_2BUTTON_PRESS`, `GDK_BUTTON_RELEASE`;
- If you triple click, eight events are caught: `GDK_BUTTON_PRESS`, `GDK_BUTTON_RELEASE`, `GDK_BUTTON_PRESS`, `GDK_2BUTTON_PRESS`, `GDK_BUTTON_RELEASE` , `GDK_BUTTON_PRESS`, `GDK_3BUTTON_PRESS`, `GDK_BUTTON_RELEASE`.

1.6.2 Changing the Cursor

If you ran this program you would see that it does not give the user any hint that the mouse button can be clicked over the image. A good user interface lets the user know when the mouse is hovering over something that is clickable. In fact, a good interface should also give the user an idea about what might happen if a mouse button *is clicked* at that point. The previous program does neither.

It is a relatively simple task to change the appearance of the cursor when it is over a widget that can respond to events. You do this by *binding* a cursor to the widget. Any widget that has a GDK window can have a cursor bound to it with

```
void     gdk_window_set_cursor      (GdkWindow *window,
                                     GdkCursor *cursor);
```

The function takes a pointer to a GDK window and a pointer to the specific `GdkCursor` that should be bound to that window. Once that is done, whenever the cursor is over the window, it will take on the appearance of the cursor that is bound to this window.

Because this function is called on a GDK window, it cannot be called until after the widget is realized, using either `gtk_widget_realize()`, or `gtk_widget_show()`, which will realize the widget as one of its steps. The GDK window of a widget is the `window` member of the `GtkWidget` structure, i.e., if `my_widget` is a pointer to a widget in your program that has a GDK window, then `my_widget->window` is the GDK window of that widget (once it has been realized.)

Creating a cursor is also easy, if you are willing to use any of the pre-existing cursors built into GDK. The function

```
GdkCursor * gdk_cursor_new (GdkCursorType cursor_type);
```

creates a new cursor of the given type, which can be selected from the set of built-in cursor types for the display. It returns a pointer to the new cursor. The API documentation has a complete list of the built-in cursor types together with icons that show what they look like. For example, there is a cursor type named `GDK_CROSSHAIR` that seems suitable to use over an image. To change the cursor for the event box in the preceding program, right before the call to `gtk_main()`, we can insert the code

```
gdk_window_set_cursor(eventbox->window,
                      gdk_cursor_new ( GDK_CROSSHAIR ) );
```

We could also bind a cursor to the top-level window with the code

```
gdk_window_set_cursor(window->window,
                      gdk_cursor_new( GDK_RIGHT_PTR) );
```

1.6.3 More on Event Boxes

Two methods specific to event boxes (i.e., not inherited from ancestor classes) are:

```
void         gtk_event_box_set_above_child       (GtkEventBox *event_box,
                                                   gboolean above_child);
gboolean     gtk_event_box_get_above_child       (GtkEventBox *event_box);
```

The notion of one widget being above or below another has to do with which one gets the events first. The widget that is on top gets the events before the one that is below. If the event box is above its child widget, then the event box will get the events first. If the child widget is above, then the child will get them first. Which is above which does not matter if the child has no children of its own and is a window-less widget, because the event box will get the events in either case. When you put a widget into an event box, it is, by default, "below" the event box, which works out for most cases.

There are situations in which you might want the child to be above the event box. For one, the child might be another container with descendants of its own, and some of these might want to catch their own events. In this situation, the event box acts like a fallback, catching only those events not caught within widgets nested within its boundaries. For this reason `gtk_event_box_set_above_child()` lets you control whether the event box or the child is above the other, and the "get" method can be used to get their current order.

The following listing is designed to let the user alter the relative stacking order of two nested event boxes. It has two event boxes and an image. The image is within `eventbox2`, which is within `eventbox1`, which is in the top-level window. The program is invoked on the command line as follows:

```
eventbox_demo2 1|2 imagefile;
```

If the user supplies a 1, then `eventbox1` is above `eventbox2`, and vice versa if a 2 is supplied. Only the relevant parts of the program are shown.

```
    ...
    eventbox1 = gtk_event_box_new ();
    g_object_set_data(G_OBJECT(eventbox1), "number", (gpointer) 1);

    eventbox2 = gtk_event_box_new ();
    g_object_set_data(G_OBJECT(eventbox2), "number", (gpointer) 2);

    image    =  gtk_image_new_from_file (argv[2]);

    gtk_container_add (GTK_CONTAINER (eventbox2), image);
    gtk_container_add (GTK_CONTAINER (eventbox1), eventbox2);
    gtk_container_add (GTK_CONTAINER (window),    eventbox1);

    // Get command line argument for which box to put on top
    which_box_on_top = strtol(argv[1], (char **) NULL, 10);

    switch (which_box_on_top) {
    case 1:   // Put eventbox1 above eventbox2
        gtk_event_box_set_above_child(GTK_EVENT_BOX(eventbox1), TRUE);
        break;
    case 2:   // Put eventbox2 above eventbox1
        gtk_event_box_set_above_child(GTK_EVENT_BOX(eventbox1), FALSE);
        break;
    default:  // Put eventbox2 above eventbox1  by default
        gtk_event_box_set_above_child(GTK_EVENT_BOX(eventbox1), FALSE);
    };

    // Put eventbox2 above image
    gtk_event_box_set_above_child(GTK_EVENT_BOX(eventbox2), TRUE);

    g_signal_connect (G_OBJECT (window), "destroy",
                      G_CALLBACK (destroy), NULL);
    g_signal_connect (G_OBJECT (eventbox1), "button_press_event",
                        G_CALLBACK (on_button_pressed),
                        (gpointer) &click_count);
    g_signal_connect (G_OBJECT (eventbox2), "button_press_event",
                        G_CALLBACK (on_button_pressed),
                        (gpointer) &click_count);

    gtk_widget_set_events (eventbox1, GDK_BUTTON_PRESS_MASK);
    gtk_widget_set_events (eventbox2, GDK_BUTTON_PRESS_MASK);
    gtk_widget_show_all (window);
    ...
```

The callback is listed below:

```
gboolean on_button_pressed (GtkWidget      *eventbox,
                            GdkEventButton *event,
                            guint          *count)
{
    (*count)++;
    int box_id = (int) g_object_get_data(G_OBJECT(eventbox), "number");
    g_print( "Eventbox %d: Button click #%d at (%d,%d)\n",
             box_id, *count, (gint) event->x, (gint) event->y );
    return TRUE;
}
```

If you run this program (`eventbox_demo2` in the demos directory), you will see that the upper event box always catches the events.

The remaining two event box methods are

```
void        gtk_event_box_set_visible_window    (GtkEventBox *event_box,
                                                  gboolean visible_window);
gboolean    gtk_event_box_get_visible_window    (GtkEventBox *event_box);
```

The GDK window of an event box can be made "invisible." By default when it is created, it is visible. This is usually how you would want it to be. The visibility of the event box's GDK window determines whether or not your application can draw on it: if it is invisible, it will be a `GDK_INPUT_ONLY` window, which means that it can only receive events. If it is visible, it is a `GDK_INPUT_OUTPUT` window that can be rendered or drawn upon. According to the API documentation,

> "Creating a visible window may cause artifacts that are visible to the user, especially if the user is using a theme with gradients or pixmaps."

On the other hand, if you make it invisible, you should make sure that it is above its child widget, because if it is below, this can lead to other problems, also described in the manuals. It is best to accept the defaults until you gain experience.

1.7 The GtkAlignment Class

If you run `eventbox_demo2`, you will see that you can resize the window, making it larger, and that the image does not get larger but the event box does. Try clicking near, but outside of, the image to see this. You will be able to click where the click is meaningless, which is clearly undesirable. A `GtkAlignment` widget can be used to prevent this.

The `GtkAlignment` widget, a subclass of `GtkBin`, controls the alignment and size of its (only) child widget. It has four parameters: `xscale`, `yscale`, `xalign`, and `yalign`.

The scale settings specify how much the child widget should expand to fill the space allocated to the `GtkAlignment`. These are floating point values between 0 and 1.0. A value of 0 means that the child does not expand at all to fill available space in the given dimension; a value of 1.0 means that the child fills all available space in that dimension. If `xscale` is 0.5, then the widget will occupy 50% of the available space in the horizontal dimension. More generally, if the scale is f and the available space is p pixels, the child will be given $f \cdot p$ extra pixels.

(picture here)

The align settings are used to place the child widget within the available area. They are floating point values ranging from 0.0 (top or left) to 1.0 (bottom or right). If the scale settings are both set to 1.0, the alignment settings have no effect, because the child widget fills the box and therefore there is no choice about placement. If the `xalign` is 0.25, for example, it means that 25% of the extra space will be to the left of the child, and 75% of the space will be to its right. If `xalign` is 0.5, it is centered.

1.7.1 Example 1

The listing below shows the parts of the event box example that would change if a `GtkAlignment` were used to constraint the event box so that it "hugged" the image no matter how the window was resized. The alignment parameters force the event box to remain the same size as the image and be centered within the main window horizontally and vertically.

```
Listing of Event Box Inside an Alignment
int  main (int argc, char *argv[])
{
    GtkWidget         *window;           // window widget
    GtkWidget         *image;            // the GtkImage
    GtkWidget         *eventbox;
    GtkWidget         *alignment;

    // stuff omitted here ...

    eventbox = gtk_event_box_new();
    gtk_widget_set_events ( eventbox, GDK_BUTTON_PRESS_MASK );

    image    =  gtk_image_new_from_file (argv[1]);

    alignment = gtk_alignment_new ( 0.5, 0.5, 0, 0 );
    gtk_container_add (GTK_CONTAINER (eventbox), image );
    gtk_container_add (GTK_CONTAINER (alignment), eventbox);
    gtk_container_add (GTK_CONTAINER (window), alignment);

    // connect signals to there callbacks here
    gtk_widget_show_all (window);

    // stuff omitted here...

    gtk_main();
    return 0;
}
```

When the top-level window is resized by the user, the alignment widget grows with it, but the alignment's child, the event box, does not grow; it stays the smallest size it can be yet still containing the image, because the alignment's `xscale` and `yscale` parameters were set to 0. The align values of 0.5 in each dimension keep the event box centered on the top-level window. Note though that if the image is so large that it would exceed the available space on the screen, this program would not work very well. It would really need scroll bars. Scrolling and scroll bars are the subject of a later chapter.

1.7.2 Example 2

Alignments can be used to compensate for the limitations of various other layout containers, and tables in particular. For example, a `GtkTable` does not have methods that force a child widget to be positioned within a table cell with particular gravity, such as left- or right-aligned, or top- or bottom-aligned. Remember that a widget within a table cell will generally be centered on the available space. If you want a column of buttons to "hug" the right edge of the table, you could use an alignment to solve this. Or if you want a particular button to sink to the bottom of the window no matter how it is resized, you could use an alignment to compensate for the table's lack of methods to do this. The following listing shows how to accomplish this. Some comments are removed to save space. Figure 5 shows the window that it displays.

```
Listing table_demo4
#include <gtk/gtk.h>
#include <libgen.h> // for basename()

#define BUTTON_WIDTH    70
#define BUTTON_HEIGHT   30

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

    GtkWidget *window;
    GtkWidget *table;
    GtkWidget *frame;
    GtkWidget *label;
    GtkWidget *horiz_align;
    GtkWidget *horiz_align2;
    GtkWidget *vert_align;
    GtkWidget *apply_button;
    GtkWidget *close_button;
    GtkWidget *help_button;
    GtkWidget *ok_button;

    gtk_init(&argc, &argv);

    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_widget_set_size_request (window, 300, 250);
    gtk_window_set_resizable(GTK_WINDOW(window), TRUE);
    gtk_window_set_title(GTK_WINDOW(window), basename(argv[0]));
    gtk_container_set_border_width(GTK_CONTAINER(window), 10);

    table = gtk_table_new(5, 4, FALSE);

    // A label will go into the upper-left corner. To force it to
    // stay to the left no matter how large the table columns are,
    // we can put it into an alignment with 0 scaling, and 0 xalign
    // to keep all extra space to the right. The alignment will go
    // into cell (1,1). If the attachment options do not include
    // GTK_FILL, then the alignment will not fill the table cell
    // and will therefore stay centered in the cell. This will
    // prevent the label from staying to the left. If they include
    // GTK_EXPAND, the alignment's functionality will be thwarted,
    // and the label will expand, shifting  to the right.
    label = gtk_label_new("Stuff");
    horiz_align = gtk_alignment_new(0, 0, 0, 0);
    gtk_container_add(GTK_CONTAINER(horiz_align), label);
    gtk_table_attach(GTK_TABLE(table), horiz_align, 0, 1, 0, 1,
                     GTK_FILL, 0, 0, 0);

    frame = gtk_frame_new(NULL);
    gtk_table_attach(GTK_TABLE(table), frame, 0, 2, 1, 3,
                     GTK_FILL | GTK_EXPAND,
                     GTK_FILL | GTK_EXPAND, 2, 2);

    // Create two buttons that will be in the first two rows of
    // the table, in the leftmost column. Force them to be the
    // same size using gtk_widget_set_size_request().
    // The upper button will not need an alignment because the button
    // below it will be set to expand to take up all extra space.
    // This will force the upper button to remain "in place".
    apply_button = gtk_button_new_with_label("Apply");
    gtk_widget_set_size_request(apply_button,
                                BUTTON_WIDTH, BUTTON_HEIGHT);
    gtk_table_attach(GTK_TABLE(table), apply_button, 3, 4, 1, 2,
                     0, GTK_FILL, 1, 1);

    // Use a vertical alignment for the second button. This will force
    // the button to stay at the top of its row. With the row set to
    // expand vertically, it will expand as the window is resized.
    close_button = gtk_button_new_with_label("Close");
    gtk_widget_set_size_request(close_button,
                                BUTTON_WIDTH, BUTTON_HEIGHT);

    vert_align = gtk_alignment_new(0, 0, 0, 0);
    gtk_container_add(GTK_CONTAINER(vert_align), close_button);
    gtk_table_attach(GTK_TABLE(table), vert_align, 3, 4, 2, 3,
                     0, GTK_FILL | GTK_EXPAND, 1, 1);

    // Set up bottom row of table. The Help button should have bottom
    // gravity, so use the alignment to force it down. Notice that the
    // alignment has the y-align set to 1, forcing all extra space to
    // be above the button.
    horiz_align2 = gtk_alignment_new(0, 1, 0, 0);
```

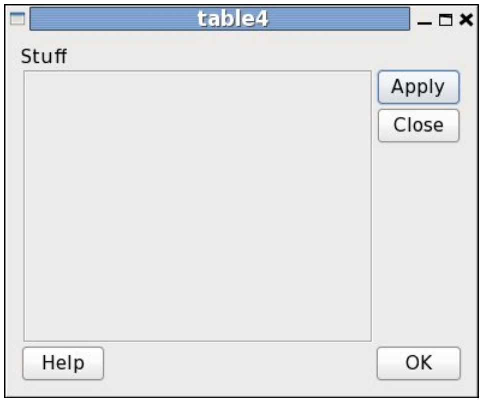

Figure 5: An application using alignments within a table.

```
    help_button = gtk_button_new_with_label("Help");
    gtk_container_add(GTK_CONTAINER(horiz_align2), help_button);
    gtk_widget_set_size_request(help_button,
                                BUTTON_WIDTH, BUTTON_HEIGHT);
    gtk_table_attach(GTK_TABLE(table), horiz_align2, 0, 1, 4, 5,
                     GTK_FILL, GTK_FILL, 0, 0);

    // The OK button should be all the way to the right. It will stay
    // in the same vertical position as the Help button because of the
    // fact that the Close button is set to expand but it is not.
   ok_button = gtk_button_new_with_label("OK");
    gtk_widget_set_size_request(ok_button,
                                BUTTON_WIDTH, BUTTON_HEIGHT);
    gtk_table_attach(GTK_TABLE(table), ok_button, 3, 4, 4, 5,
                     GTK_FILL, GTK_FILL, 0, 0);

    gtk_container_add(GTK_CONTAINER(window), table);

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

    gtk_widget_show_all(window);
    gtk_main();

    return 0;
}
```

The important principle to bear in mind in using alignments is the following:

If the `GTK_FILL` bit is not set in the given direction, then the alignment will not be given the full size of the cell to work with, and it will be positioned in the center of the available space. For example, if you put a `GtkAlignment` into a cell that is 200 pixels wide, and it contains a button that is 100 pixels wide, but `GTK_FILL` is *not* set in the x-direction, then the alignment will have 50 pixels to its left and right, so it will not be able to force the button to any edge of the cell.

It is also important to remember what `GTK_EXPAND` will do. This bit will cause a cell to grow to take up any space available in the given dimension. If `GTK_FILL` is set in that dimension, then the alignment will grow. This may not be a problem if it is set to force its child widget to the left, with 0 scaling, but it will be a problem for other settings. In the program, the `Close` button is attached with `GTK_EXPAND` set in the vertical dimension. This is okay because it is within an alignment that fills that dimension and has top gravity, so the button stays at the top of the cell. The expand property is necessary to keep the other rows from growing – it steals any space vertically as the window is resized.

1.7.3 The GtkPaned Widgets

Consider how email clients are usually organized. There is usually a list of email messages in a rectangular scrollable window at the top, and directly below that, a place where the current message is on display. Most clients let the user change the space allocated to the list by dragging a control up or down. The dragging might work by pulling on the border of the upper window, or by pulling on some type of other visual element. The `GtkPaned` class can be packed with two children that share the space in this way. The `GtkPaned` widget stays the same size but lets the user change the boundary between the two children. The `GtkPaned` class has two subclasses, `GtkHpaned` and `GtkVPaned`, which are horizontally and vertically paned. Here we show an example of how a `GtkVPaned` widget can be used.

```
Listing pane_demo1
int main( int argc, char *argv[] )
{
    GtkWidget *window;
    GtkWidget *vpaned;
    GtkWidget *label;
    GtkWidget *frame;
    gchar      *contents;
    GError     *error = NULL;

    gtk_init (&argc, &argv);

    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title (GTK_WINDOW (window), "Paned Window Demo ");
    gtk_container_set_border_width (GTK_CONTAINER (window), 10);
    gtk_widget_set_size_request (GTK_WIDGET (window), 500, 500);

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

    // create a vpaned widget and add it to the toplevel window
    vpaned = gtk_vpaned_new ();
    gtk_container_add (GTK_CONTAINER (window), vpaned);
    gtk_widget_show (vpaned);

    // Read some text from a file, insert it into a label,
    // and pack into top pane
    if (! g_file_get_contents( "sometext", &contents, NULL, &error)) {
        g_printf(error->message);
        g_clear_error(&error);
        exit(1);
    }
    label = gtk_label_new(contents);
    frame = gtk_frame_new(NULL);
    gtk_container_add(GTK_CONTAINER(frame), label);
    gtk_paned_pack1 (GTK_PANED (vpaned),frame , TRUE, FALSE);

   // Fill a label with some other text and pack into bottom pane
```

```
    label = gtk_label_new("Resize the upper pane by pulling down"
                          "\non the control below it.");
    frame = gtk_frame_new(NULL);
    gtk_container_add(GTK_CONTAINER(frame), label);
    gtk_paned_pack2 (GTK_PANED (vpaned), frame, FALSE, FALSE);
    gtk_widget_show_all (window);

    gtk_main ();
    return 0;
}
```

The horizontal and vertical pane widgets have only a single method each, their respective contructors:

```
GtkWidget *    gtk_vpaned_new     (void);
GtkWidget *    gtk_hpaned_new     (void);
```

Note that they take no arguments. The parent class provides a few methods, the most important being the packing methods:

```
void           gtk_paned_pack1    (GtkPaned *paned,
                                   GtkWidget *child,
                                   gboolean resize,
                                   gboolean shrink);
void           gtk_paned_pack2    (GtkPaned *paned,
                                   GtkWidget *child,
                                   gboolean resize,
                                   gboolean shrink);
```

which pack a widget into the top (or left) and bottom (or right) panes respectively. These methods provide the most flexibility, as you can see. The `resize` and `shrink` parameters control whether or not the pane can be resized or shrunk. See the API documentation for details.