Container Widgets and Packing

As mentioned in Chapter 2, there are two kinds of containers, those that can hold no more than one child widget, and those that can hold more than one. Containers that can contain a single widget are called ***decorator containers***, because their principal purpose is to add functionality and decorative effects to the child. Containers that can hold several children are called ***layout containers***, because they are used for laying out the child widgets within their (GDK) windows.

The `GtkContainer` base class supports four signals: `add`, `check_resize`, `remove`, and `set_focus_child`. You should consult the GTK+ documentation or the textbook for details.

`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` . The last two of these will be covered in a later chapter on scrolling. The decorator containers that we will examine are the `GtkExpander` and the `GtkEventBox`; the rest can be looked up in the GTK+ documentation.

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

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

which, given the `GtkBin`, returns a pointer to its only child. If you want to retrieve the text of a label inside a window, and all you have is a pointer to the window widget, you can use this to get the label widget and then get its text.

Layout Containers

There are several choices for organizing your visual elements in a window; some of the most useful classes are:

`GtkHBox`	Horizontal box with multiple children
`GtkVBox`	Vertical box with multiple children
`GtkTable`	Grid with rows and columns for multiple children
`GtkFixed`	Fixed positioning container for multiple children
`GtkLayout`	Versatile scrollable container for multiple children
`GtkAlignment`	Container to control relative position of single child widget

We will start with horizontal and vertical boxes.

Packing Boxes

Boxes are invisible containers into which widgets can be packed. There are two basic boxes: 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 a horizontal box, the objects are inserted horizontally from left to right or right to left depending on the call used. In a vertical box, 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.

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);
```

If *homogeneous* is true, then all widgets in the box 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 that long, then no matter how small the other buttons' labels might be, they will each be given 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.

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* = 20, then the *hbox* will ask for $6*100+5*20=700$ pixels.

Box Packing

The two functions related to 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 how the box was created. I will start with the case in which the box is not homogeneous.

Case 1: Non-homogeneous Box

If the *expand* parameter is false, then any extra space in the box is placed at the growing end of the box; i.e., at the right end of an *hbox* unless `gtk_box_pack_end()` is used, or at the top or bottom of a *vbox* depending on which packing function is used. For example, an hbox that is 600 pixels long with six children whose total length is 480 pixels has an extra 120 pixels inside.

If *expand* is true and the box has more space than is necessary to display all of the child widgets, then the extra space is divided equally among those child widgets that have set this parameter to true. *This does not mean that each of these children will be equal size*; it means that each of them will be given an equal amount of extra space in addition to whatever part of their size request was granted. For example, if the box is 600 pixels long and it has six children, and the total length of its children is 480, then the extra 120 pixels is divided equally among the children and each gets an extra 20 pixels.

Furthermore, if *expand* is true, the *fill* parameter determines whether the extra space given to that child is within the child widget or outside the child widget, providing space on either side of it. If *fill* is true, the extra space is within the widget; if false, it is outside the widget. If *expand* is false, the *fill* parameter is ignored.

The padding parameter is the number of extra pixels to place between the child widget and its neighbors in the box, over and above any space that might be between them because of the spacing parameter or the expansion of the widgets caused by setting expand to be true. If a child is at an end o f the box, then this many pixels are placed on the outside of the child as well. Note that each widget gets padding pixels on each side, so that double the number of pixels ends up between adjacent children that have requested padding. For example, if the hbox of size 600 has six children totaling 480 pixels in length, and each asked for 5 pixels of padding, then 10 pixels will be between pairs and 5 will be at each end.

Case 2: Homogeneous Box

If a box is *homogeneous*, then the value of the *expand* parameter is ignored. In this case, every child is given an equal amount of space in the box. If *fill* is false, then the space is outside of the child widget, and if it is true, it is within the widget. The *padding* specified by the padding parameter is added to the space allocated to each widget, using the same rule as described in the preceding section; *padding* pixels are added to the left and right of each widget, expanding the box as needed, if the request can be granted.

Tables

Tables 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 on a grid or any regular two-dimensional pattern.

To create a new table, use

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

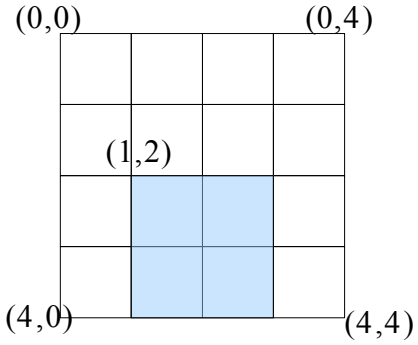
The first two parameters are pretty obvious -- they specify the number of rows and columns in the table. The third parameter determines whether or not homogeneous spacing is used. It also has implications about how children are resized when the containing widget is resized. If *homogeneous* is true, 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 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. The methods I am omitting are also relatively easy to understand, unlike what you are about to see.

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.)



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 are in the above illustration.

The next two parameters specify the attachment options in the x-direction and y-direction respectively, and they are bitwise-OR of three boolean flags:

`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 determined 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 between the widgets in the x and y directions respectively, in pixels.

If you 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.

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