1

RGtk2: Overview

As the name implies, the RGtk2 package is an interface, or binding, between R and GTK+, a mature, cross-platform GUI toolkit. The letters *GTK* stand for the *GIMP ToolKit*, with the word *GIMP* recording the origin of the library as part of the GNU Image Manipulation Program. GTK+ provides the same widgets on every platform, though it can be customized to emulate platform-specific look and feel. The library is written in C, which facilitates access from languages like R that are also implemented in C. GTK+ is licensed under the *Lesser GNU Public License* (LGPL), while RGtk2 is under the *GNU Public License* (GPL). The package is available from the Comprehensive R Archive Network (CRAN) at http://CRAN.R-project.org/package=RGtk2.

The name RGtk2 also implies that there exists a package named RGtk, which is indeed the case. The original RGtk is bound to the previous generation of GTK+, version 1.2. RGtk2 is based on GTK+ 2.0, the current generation. This book covers RGtk2 specifically, although many of the fundamental features of RGtk2 are inherited from RGtk.

RGtk2 provides virtually all of the functionality in GTK+ to the R programmer. In addition, RGtk2 interfaces with several other libraries in the GTK+ stack: Pango for font rendering; Cairo for vector graphics; GdkPixbuf for image manipulation; libglade for designing GUI layouts from an XML description; ATK for accessible interfaces; and GDK, an abstraction over the native windowing system, supporting either X11 or Windows. These libraries are multi-platform and extensive and have been used for many major projects, such as the Linux versions of Firefox and Open Office.

The API of each of these libraries is mapped to R in a way that is consistent with R conventions and familiar to the R user. Much of the RGtk2 API consists of autogenerated R functions that call into one of the underlying libraries. For example, the R function gtkContainerAdd eventually calls the C function gtk_container_add. The naming convention is that the C name has its underscores removed and each following letter capitalized (camelback style).

The full API for GTK+ is quite large, and complete documentation of it is beyond our scope. However, the GTK+ documentation is algorithmically converted into the R help format during the generation of RGtk2. This conveniently allows the programmer to refer to the appropriate documentation within an R session, without having to consult a web page, such as http://library.gnome.org/devel/gtk/stable/, which lists the C API of the stable versions GTK+.

In this chapter, we give an overview of how RGtk2 maps the GTK+ API, including its classes, constructors, methods, properties, signals and enumerations, to an R-level API that is relatively familiar to, and convenient for, an R user. A simple GUI will be gradually constructed to demonstrate the API.

1.1 Objects and Classes

In any toolkit, all widget types have functionality in common. For example, they are all drawn on the screen in a consistent style. They can be hidden and shown again. To formalize this relationship and to simplify implementation by sharing code between widgets, GTK+, like many other toolkits, defines an inheritance hierarchy for its widget types. In the parlance of object-oriented programming, each type is represented by a *class*.

For specifying the hierarchy, GTK+ relies on GObject, a C library that implements a class-based, single-inheritance object-oriented system. A GObject class encapsulates behaviors that all instances of the class share. Every class has at most one parent from which it inherits the behaviors of its ancestors. A subclass can override some specific inherited behaviors. The interface defined by a class consists of constructors, methods, properties, and signals.

Single inheritance can be restrictive when a class performs multiple roles in a program. To circumvent this, GTK+ adopts the popular concept of the *interface*, which is essentially a contract that specifies which methods, properties and signals a class must implement. As with languages like Java and C#, a class can *implement* multiple interfaces, and an interface can be composed of other interfaces. An interface allows the programmer to treat all instances of implementing classes in a similar way. However, unlike class inheritance, the implementation of the methods, properties and signals is not shared.

We explain the constructors, methods, properties and signals of classes and interfaces in the following sections and demonstrate them in the construction of a simple "Hello World" GUI, shown in Figure **??**. A more detailed and technical explanation of GObject is available in Section **??**.

Figure 1.1: "Hello World" in GTK+. A window containing a single button displaying a label with the text Hello World.

1.2 Constructors

The first step in our example is to create a top-level window to contain our GUI. Creating an instance of a GTK widget requires calling a single R function, known as a constructor. Following R conventions, the constructor for a class has the same name as the class, except the first character is lowercase. The following statement constructs an instance of the `GtkWindow` class:

```
window <- gtkWindow("toplevel", show = FALSE)
```

The first argument to the constructor for `GtkWindow` instructs the window manager to treat the window as top-level. The `show` argument is the last argument for every widget constructor. It indicates whether the widget should be made visible immediately after construction. The default value of `show` is `TRUE`. In this case we want to defer showing the window until after we finish constructing our simple GUI.

At the GTK+ level, a class usually has multiple constructors, each implemented as a separate C function. In `RGtk2`, the names of these functions all end with `New`. The "meta" constructor `gtkWindow`, called above, automatically delegates to one of the low-level constructors, based on the provided arguments.

A GTK+ object created by the R user has an R-level object as its proxy. Thus, `window` is a reference to a `GtkWindow` instance. The class hierarchy of a proxy object is represented by the `class` attribute. One interprets the attribute according to S3 conventions, so that the class names are in order from most to least derived:

```
class(window)
```

```
[1] "GtkWindow"         "GtkBin"            "GtkContainer"
[4] "GtkWidget"         "GtkObject"         "GInitiallyUnowned"
[7] "GObject"           "RGtkObject"
```

We find that the `GtkWindow` class inherits methods, properties, and signals from the `GtkBin`, `GtkContainer`, `GtkWidget`, `GtkObject`, `GInitiallyUnowned`, and `GObject` classes. Every type of `GTK+` widget inherits from the base `GtkWidget` class, which implements the general characteristics shared by all widget classes, e.g., properties storing the location and background color; methods for hiding, showing and painting the widget. We can also query `window` for the interfaces it implements:

```
interface(window)
```

```
[1] "AtkImplementorIface" "GtkBuildable"
```

When the underlying GTK+ object is destroyed, i.e., deleted from memory, the class of the proxy object is set to `<invalid>`, indicating that it can no longer be manipulated.

1.3 Methods

The next steps in our example are to create a "Hello World" button and to place the button in the window that we have already created. This depends on an understanding of how of one programmatically manipulates widgets by invoking methods. Methods are functions that take an instance of their class as the first argument and instruct the widget to perform an action.

Although class information is stored in the style of S3, `RGtk2` introduces its own mechanism for method dispatch. The call `obj$method(...)` resolves to a function call `f(obj,...)`. The function is found by looking for any function that matches the pattern *classNameMethodName*, the concatenation of one of the names from `class(obj)` or `interface(obj)` with the method name. The search begins with the interfaces and proceeds through each character vector in order.

For instance, if `win` is a `gtkWindow` instance, then to resolve the call `win$add(widget)` `RGtk2` considers `gtkBuildableAdd`, `atkImplementorIfaceAdd`, `gtkWindowAdd`, `gtkBinAdd` and finally finds `gtkContainerAdd`, which is called as `gtkContainerAdd(win, widget)`. The `$` method for `RGtk2` objects does the work.

We take advantage of this convenience when we add the "Hello World" button to our window and set its size:

```
button <- gtkButton("Hello World")
window$add(button)
window$setDefaultSize(200, 200)
```

The above code calls the `gtkContainerAdd` and `gtkWindowSetDefaultSize` functions with less typing and less demands on the memory of the user.

Understanding this mechanism allows us to add to the `RGtk2` API. For instance, we can add to the button API with

```
gtkButtonSayHello <- function(obj, target)
  obj$setLabel(paste("Hello", target))
button$sayHello("World")
button$getLabel()
```

```
[1] "Hello World"
```

Some common methods are inherited by most widgets, as they are defined in the base `gtkWidget` class. These include the methods `show` to specify that the widget should be drawn; `hide` to hide the widget until specified; `destroy` to destroy a widget and clear up any references to it; `getParent` to find the parent container of the widget; `modifyBg` to modify the background color of a widget; and `modifyFg` to modify the foreground color.

1.4 Properties

The GTK+ API uses properties to store object state. Properties are similar to R attributes and even more so to S4 slots. They are inherited, typed, self-describing and encapsulated, so that an object can intercept access to the underlying data. A list of properties that a widget has is returned by its `getPropInfo` method. RGtk2 uses the R generic `names` as a familiar alternative for this method. Auto-completion of property names is gained as a side effect. For the button just defined, we can see the first eight properties listed with:

```
head(names(button), n=8)                    # or b$getPropInfo()
```

```
[1] "related-action"        "use-action-appearance"
[3] "user-data"             "name"
[5] "parent"                "width-request"
[7] "height-request"        "visible"
```

Some common properties are: `parent`, to store the parent widget (if any); `user-data`, which allows one to store arbitrary data with the widget; and `sensitive`, to control whether a widget can receive user events.

There are a few different ways to access these properties. GTK+ provides the functions `gObjectGet` and `gObjectSet` to get and set properties of a widget. The set funtion treats the arguments names as the property names, and setting multiple properties at once is supported. Here we add an icon to the top-left corner of our window and set the title:

```
image <- gdkPixbuf(filename = imagefile("rgtk-logo.gif"))[[1]]
window$set(icon = image, title = "Hello World 1.0")
```

Additionally, most user-accessible properties have specific `get` and `set` methods defined for them. For example, to set the title of the window, we could have used the `setTitle` method and verified the change with `getTitle`.

```
window$setTitle("Hello World 1.0")
window$getTitle()
```

```
[1] "Hello World 1.0"
```

`RGtk2` provides the convenient and familiar `[` and `[<-` methods to get and access the properties. In our example, we might check the window to ensure that it is not yet visible:

```
window["visible"]
```

```
[1] FALSE
```

Finally, we can make our window visible by setting the "visible" property, although calling `gtkWidgetShow` is more conventional:

```
window["visible"] <- TRUE
window$show() # same effect
```

For ease of referencing the appropriate help pages, we tend to use the full method name in the examples, although at times the move R-like vector notation will be used for commonly accessed properties.

1.5 Events and signals

In RGtk2, a user action, such as a mouse click, key press, or drag and drop motion, triggers the widget to emit a corresponding signal. A GUI can be made interactive by specifying a callback function to be invoked upon the emission of a particular signal.

The signals provided by a class or interface are returned by the function gTypeGetSignals. For example

```
names(gTypeGetSignals("GtkButton"))
```

```
[1] "pressed"  "released" "clicked"  "enter"    "leave"
[6] "activate"
```

shows the "clicked" signal in addition to others. Note that this only lists the signals provided directly by the GtkButton. To list all inherited signals, we need to loop over the hierarchy, but it is not common to do this in practice, as the documentation includes information on the signals.

The gSignalConnect (or gSignalConnect) function is used to add a callback to a widget's signal. Its signature is

```
args(gSignalConnect)
```

```
function (obj, signal, f, data = NULL, after = FALSE, user.data.first = FALSE)
```

The basic usage is to call gSignalConnect to connect a callback function f to the signal named signal belonging to the object obj. The function returns an identifier for managing the connection. This is not usually necessary but will be discussed later.

We demonstrate this usage by adding a callback to our "Hello World" example, so that "Hello World" is printed to the console when the button is clicked:

```
gSignalConnect(button, "clicked",
               function(widget) print("Hello world!"))
```

The data argument allows arbitrary data to be passed to the callback. The user.data.first argument specifies if this data argument should be the first argument to the callback or (the default) the last.

The after argument is a logical indicating if the callback should be called after the default handlers (see ?gSignalConnect).

The signature for the callback varies for each signal. Unless `user.data.first` is `TRUE`, the first argument is the widget. Other arguments are possible depending on the signal type. For window events, the second argument is a `GdkEvent` type, which can carry with it extra information about the event that occurred. The GTK+ API lists the signature of each signal.

Is important to note that the widget, and possibly other arguments, are references, so their manipulation has side effects outside of the callback. This is obviously a critical feature, but it is one that may be surprising to the R user.

```
w <- gtkWindow(); w['title'] <- "test signals"
x <- 1;
b <- gtkButton("click me"); w$add(b)
ID <- gSignalConnect(b, signal = "clicked", f = function(widget) {
  widget$setData("x", 2)
  x <- 2
  return(TRUE)
})
```

Then after clicking, we would have

```
cat(x, b$getData("x"), "\n") # 1 and 2
```

```
1 2
```

Callbacks for signals emitted by window manager events are expected to return a logical value. Failure to do so can cause errors to be raised. For most other callbacks the return value is ignored, so it is safe to always return a logical value. For window events, a return value of `TRUE` indicates that no further callbacks should be called, whereas `FALSE` indicates that the next callback should be called. So in the following example, only the first two callbacks are executed when the user presses on the button.

```
b <- gtkButton("click")
w <- gtkWindow()
w$add(b)
id1 <- gSignalConnect(b, "button-press-event",
function(b, event, data) {
  print("hi"); return(FALSE)
})
id2 <- gSignalConnect(b, "button-press-event",
function(b, event, data) {
  print("and"); return(TRUE)
})
id3 <- gSignalConnect(b, "button-press-event",
function(b, event, data) {
  print("bye"); return(TRUE)
})
```

Multiple callbacks can be assigned to each signal. They will be processed in the order they were bound to the signal. The `gSignalConnect` function returns an ID that can be used to disconnect a callback if desired using `gSignalHandlerDisconnect` or temporarily blocked using `gSignalHandlerBlock` and `gSignalHandlerUnblock`. The man page for `gSignalConnect` gives the details on this, and much more.

1.6 Enumerated types and flags

At the beginning of our example, we constructed the window thusly:

```
window <- gtkWindow("toplevel", show = FALSE)
```

The first parameter indicates the window type. The set of possible window types is specified by what in `C` is known as an *enumeration*. A value from an enumeration can be thought of as a length one factor in R. The possible values defined by the enumeration are analogous to the factor levels. Since enumerations are foreign to `R`, `RGtk2` accepts string representations of enumeration values, like `"toplevel"`.

For every `GTK+` enumeration, `RGtk2` provides an `R` vector that maps the nicknames to the underlying numeric values. In the above case, the vector is named `GtkWindowType`.

```
GtkWindowType
```

```
An enumeration with values:
toplevel     popup
       0         1
```

The names of the vector indicate the allowed nickname for each value of the enumeration. It is rarely necessary to explicitly use the enumeration vectors; specifying the nickname will work in most cases, including all method invocations, and is preferable as it is easier for human readers to comprehend.

Flags are an extension of enumerations, where the value of each member is a unique power of two, so that the values can be combined unambiguously. An example of a flag enumeration is `GtkWidgetFlags`.

```
GtkWidgetFlags
```

```
A flag enumeration with values:
        toplevel        no-window        realized
              16               32              64
          mapped          visible       sensitive
             128              256             512
parent-sensitive        can-focus       has-focus
            1024             2048            4096
     can-default      has-default        has-grab
            8192            16384           32768
```

```
       rc-style  composite-child      no-reparent
          16384           131072           262144
   app-paintable  receives-default  double-buffered
         524288          1048576          2097152
     no-show-all
        4194304
```

`GtkWidgetFlags` represents the possible flags that can be set on a widget. We can retrieve the flags currently set on our window:

```
window$flags()
```

```
GtkWidgetFlags: toplevel, realized, mapped, visible, sensitive, parent-sensitive, double-buffe
```

Flag values can be combined using `|` the bitwise *OR*. The `&` function, the bitwise *AND*, allows one to check whether a value belongs to a combination. For example, we could check whether our window is top-level:

```
(window$flags() & GtkWidgetFlags["toplevel"]) > 0
```

```
[1] TRUE
```

1.7 The event loop

`RGtk2` integrates the GTK+ eventloop with the R event loop. A separate thread continuously iterates the GTK+ event loop, in synchronization with the main R thread. Thus, if the R thread is busy, the GTK+ event loop will not be iterated. During a long calculation, the GUI can seem unresponsive. To avoid this, the following construct should be inserted into the long running algorithm in order to ensure that GTK+ events are periodically processed.

```
while(gtkEventsPending())
  gtkMainIteration()
```

2

RGtk2: Windows, Containers, and Dialogs

This section covers some of the basic widgets and containers of GTK+. We begin with a discussion of top level containers and box containers. Then we describe many of the basic controls, and conclude with the mention of a few special-case containers.

2.1 Top-level windows

As we saw in our "Hello World" example, top-level windows are constructed by the `gtkWindow` constructor. This function has arguments `type` to specify the type of window to create. The default is a top-level window, which we will always use, as the alternative is for "popups" which are meant for internal use, e.g., for implementing menus. The second argument is `show`, which by default is `TRUE`, indicating that the window should be shown. If set to `FALSE`, the window, like other widgets, can later be shown by calling its `show` method. The `showAll` method will also show any child components. These can be reversed with `hide` and `hideAll`.

As with all objects, windows have several properties. The window title is stored in the `title` property. As usual, this property can be accessed via the "get" and "set" methods `getTitle` and `setTitle`, or using the `[` function. To illustrate, the following sets up a new window with a title.

```
w <- gtkWindow(show=FALSE)              # use default type
w$setTitle("Window title")              # set window title
w['title']                              # or w$getTitle()
```

```
[1] "Window title"
```

```
w$setDefaultSize(250,300)               # 250 wide, 300 high
w$show()                                # show window
```

Window size The initial size of the window can be set with the `setDefaultSize` method, as shown, which takes a `width` and `height` argument specified in pixels. This specification allows the window to be resized, but must be made before the window is drawn, as the window then falls under control of the window manager. The setSizeRequest method will request a minimum size, which the window manager will usually honor, as long as a maximum bound is not violated. To fix the size of a window, the resizable property may be set to FALSE.

Adding a child component to a window A window is a container. GtkWindow inherits from GtkBin, a class whose instances can contain only a single child. As before, this child is added through the add method. To display multiple widgets in a window, one simply needs to add a non-GtkBin container as the child widget.

We illustrate the basics by adding a simple label to a window.

```
w <- gtkWindow(show=FALSE); w$setTitle("Hello world")
l <- gtkLabel("Hello world")
w$add(l)
```

Destroying windows A window is normally closed by the window manager. Most often, this occurs in response to the user clicking on a close button in a title bar. It is also possible to close a window programmatically by calling its destroy method. When the user clicks on the close button, the window manager requests that the window be deleted, and the delete-event signal is emitted. As with any window manager event, the default handler is overridden if a callback connected to delete-event returns TRUE. This can be useful for confirming the intention of the user before closing the window. The contract of deletion is that the window should no longer be visible on the screen. It is not necessary for the actual window object to be removed from memory, although this is the default behavior. Calling the hideOnDelete method configures the window to hide but not destroy itself.

Transient windows New windows may be standalone top-level windows, or may be associated with some other window. For example, a dialog is usually associated with the primary document window. The setTransientFor method can be used to specify the window with which a transient (dialog) window is associated. This hints to the window manager that the transient window should be kept on top of its parent. The position relative to the parent window can be specified with setPostion, which takes a value from the GtkWindowPosition enumeration. Optionally, a dialog be can be set to be destroyed with its parent. For example:

```
w <- gtkWindow(show=FALSE); w$setTitle("Top level window")
d <- gtkWindow(show=FALSE); d$setTitle("dialog window")
d$setTransientFor(w)
```

```
d$setPosition("center-on-parent")
d$setDestroyWithParent(TRUE)
w$show()
d$show()
```

The above code produces a non-modal dialog window from scratch. Due to its transient nature, it can hide parts of the top-level window, but, unlike a modal dialog, it does not prevent that window from receiving events. GTK+ provides a number of convenient high-level dialogs, discussed later, that support modal operation.

2.2 Layout containers

Once a top-level window is constructed, it remains to fill the window with the controls that will constitute our GUI. As these controls are graphical, they must occupy a specific region on the screen. The region could be specified explicitly, as a rectangle. However, as a user interface, a GUI is dynamic and interactive. The size constraints of widgets will change, and the window will be resized. The programmer cannot afford to explicitly manage a dynamic layout. Thus, GTK+ implements automatic layout in the form of container widgets.

Basics

The method `getChildren` will return the children of a container as a list. Since in this case the list will be at most length one, the `getChild` method may be more convenient, as it directly returns the only child, if any. For instance, to retrieve the label text one could do:

```
w$getChild()['label']                    # return label property of child
```

```
[1] "Hello world"
```

The `[[` method accesses the child containers by number, as a convenient wrapper around the `getChildren` method.

In GTK+, the widget hierarchy is built when children are added to a parent container. In our example, the window is the immediate parent of the label. The `getParent` method for GTK+ widgets will return the parent container of a widget.

Every container supports removing a child with the `remove` method. The child can later be re-added using `packStart`. For instance

```
b <- g[[3]]
g$remove(b)                              # removed
g$packStart(b, expand=TRUE, fill=TRUE)
```

To remove a widget from the screen but not its container, use the `hide` method on the widget. This can be reversed with the `show` method. The `reparent` method is a convenience for moving a widget between containers.

Widget size negotiation

We have already seen perhaps the simplest automatic layout container, `GtkWindow`, which fills all of its space with its child. While simple, there is a considerable amount of logic for calculating the size of the widget on the screen. The child will first inform the parent of its desired natural size. For example, a label might ask for the dimensions necessary to display all of its text. The container then decides whether to allocate the requested size or to allocate more or less than the requested amount. The child consumes the allocated space. Consider the previous example of adding a label to a window:

```
w <- gtkWindow(show=FALSE); w$setTitle("Hello world")
l <- gtkLabel("Hello world")
w$add(l)
```

The window is shown before the label is added, and the default size is likely much larger than the space the label needs to display "Hello world". However, as the window size is now controlled by the window manager, `GtkWindow` will not adjust its size. Thus, the label is allocated more space than it requires.

```
l$getAllocation()
```

```
$retval
NULL

$allocation
$x
[1] -1

$y
[1] -1

$width
[1] 1

$height
[1] 1

attr(,"class")
[1] "GtkAllocation"
```

If, however, we avoid showing the window until the label is added, the window will size itself so that the label has its natural size:

```
w <- gtkWindow(show=FALSE); w$setTitle("Hello world")
l <- gtkLabel("Hello world")
w$add(l)
w$show()
l$getAllocation()
```

```
$retval
NULL

$allocation
$x
[1] 0

$y
[1] 0

$width
[1] 79

$height
[1] 18

attr(,"class")
[1] "GtkAllocation"
```

One might notice that it is not possible to decrease the size of the window further. This is due to `GtkLabel` asserting a minimum size request that is sufficient to display its text. The `setSizeRequest` sets a user-level minimum size request for any widget. It is obvious from the method name, however, that this is still strictly a request. It may not be satisfied, for example, if the maximum window size constraint of the window manager is violated. More importantly, setting a minimum size request is generally discouraged, as it decreases the flexibility of the layout.

Any non-trivial GUI will require a window containing multiple widgets. Let us consider the case where the child of the window is itself a container, with multiple children. Essentially the same negotiation process occurs between the container and its children (the grandchildren of the window). The container calculates its size request based on the requests of its children and communicates it to the window. The size allocated to the container is then distributed to the children according to its layout algorithm. This process is the same for every level in the container hierarchy.

Box containers

The most commonly used multi-child container in GTK+ is the box, `GtkBox`, which packs its children as if they were in a box. Instances of `GtkBox` are constructed by `gtkHBox` or `gtkVBox`. These produce horizontal or vertical "boxes", respectively. Each child widget is allocated a cell in the box. The cells are arranged in a single column (`GtkVBox`) or row (`GtkHBox`). This one dimensional stacking is usually all that a layout requires. The child widgets can be containers themselves, allowing for very flexible layouts. For special cases where some widgets need to span multiple rows or columns, GTK+ provides the `GtkTable` class, which is discussed later. Many of the principles we discuss in this section also apply to `GtkTable`.

Here we will explain and demonstrate the use of `GtkHBox`, the general horizontal box layout container. `GtkVBox` can be used exactly the same way; only the direction of stacking is different. Figure **??** illustrates a sampling of the possible layouts that are possible with a `GtkHBox`.

Figure 2.1: A screenshot demonstrating the effect of packing two buttons into `GtkHBox` instances using the `packStart` method with different combinations of the `expand` and `fill` settings. The effect of the `homogeneous` spacing setting on the `GtkHBox` is also shown.

The code for some of these layouts is presented here. We begin by creating a `GtkHBox` widget. We pass `TRUE` for the first parameter, `homogeneous`. This means that the horizontal allocation of the box will be evenly distributed between the children. The second parameter directs the box to leave 5 pixels of space between each child. The following code constructs the `GtkHBox`:

```
box <- gtkHBox(TRUE, 5)
```

The equal distribution of available space is strictly enforced; the minimum size requirement of a homogeneous box is set such that the box always satisfies this assertion, as well as the minimum size requirements of its children.

The `packStart` and `packEnd` methods pack a widget into a box with left and right justification (top and bottom for a `GtkVBox`), respectively. For this explanation, we restrict ourselves to `packStart`, since `packEnd` works the same except for the justification. Below, we pack two buttons, `button_a` and `button_b` using left justification:

```
button_a <- gtkButton("Button A")
button_b <- gtkButton("Button B")
box$packStart(button_a, fill = FALSE)
box$packStart(button_b, fill = FALSE)
```

First, `button_a` is packed against the left side of the box, and then we pack `button_b` against the right side of `button_a`. This results in the first row in Figure **??**. The space distribution is homogeneous, but making the space available to a child does not mean that the child will fill it. That depends on the minimum size requirement of the child, as well as the value of the fill parameter passed to packStart. In this case, fill is FALSE, so the extra space is not filled. When a widget is packed with the fill parameter set to TRUE, the widget is sized to consume the available space. This results in rows 2 and 3 in Figure **??**.

In many cases, it is desirable to give children unequal amounts of available space, as in rows 4–9 in Figure **??**. To create an inhomogeneously spaced GtkHBox, we pass FALSE as the first argument to the constructor, as in the following code:

```
box <- gtkHBox(FALSE, 5)
```

An inhomongeneous layout is freed of the restriction that all widgets must be given the same amount of available space; it only needs to ensure that each child has enough space to meet its minimum size requirement. After satisfying this constraint, a box is often left with extra space. The programmer may control the distribution of this extra space through the expand parameter to packStart. When a widget is packed with expand set to TRUE, we will call the widget an *expanding* widget. All expanding widgets in a box are given an equal portion of the entirety of the extra space. If no widgets in a box are expanding, as in row 5 of Figure **??**, the extra space is left undistributed.

It is common to mix expanding and non-expanding widgets in the same box. An example is given below, where button_a is expanding, while button_b is not:

```
box$packStart(button_a, expand = TRUE, fill = FALSE)
box$packStart(button_b, expand = FALSE, fill = FALSE)
```

The result is shown in row 6 of Figure **??**. The figure contains several other permutations of the homogeneous, expand and fill settings.

There are several ways to add space around widgets in a box container. The spacing argument for the constructors specifies the amount of space, in pixels, between the cells. This defaults to zero. The pack methods have a padding argument, also defaulting to zero, for specifying the padding in pixels on either side of the child. It is important to note the difference: spacing is between children and the same for every boundary, while the padding is specific to a particular child and occurs on either side, even on the ends. The spacing between widgets is the sum of the spacing value and the two padding values when the children are added. Example **??** provides an example and Figure **??** an illustration.

The reorderChild method can be used to reorder the child widgets. The new position of the child is specified using 0-based indexing. This code will move the last child to the second position.

```
b3 <- g[[3]]
g$reorderChild(b3, 2 - 1)                # second is 2 - 1
```

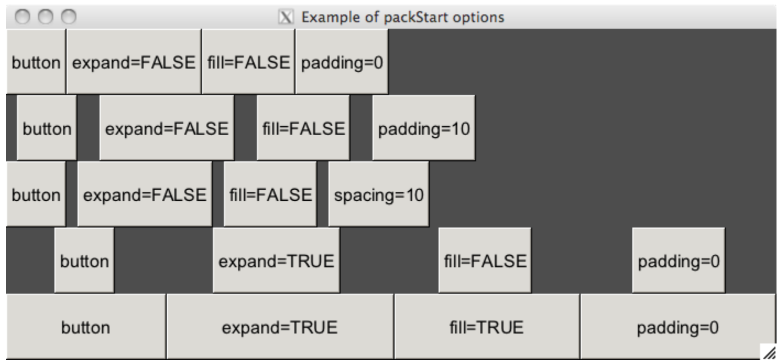

Figure 2.2: Examples of packing widgets into a box container. The top row shows no padding, whereas the 2nd and 3rd illustrate the difference between `padding` (an amount around each child) and `spacing` (an amount between each child). The last two rows show the effect of `fill` when `expand=TRUE`. This illustration follows one in original GTK+ tutorial.

Alignment

We began this section with a simple example of a window containing a label:

```
w <- gtkWindow(show=FALSE); w$setTitle("Hello world")
l <- gtkLabel("Hello world")
w$add(l)
```

The window allocates all of its space to the label, despite the actual text consuming a much smaller region. The size of the text is fixed, according to the font size, so it could not be expanded. Thus, the label decided to center the text within itself (and thus the window). A similar problem is faced by widgets displaying images. The image cannot be expanded without distortion. Widgets that display objects of fixed size inherit from `GtkMisc`, which provides methods and properties for tweaking how the object is aligned within the space of the widget. For example, the `xalign` and `yalign` properties specify how the text is aligned in our label and take values between 0 and 1, with 0 being left and top. Their defaults are 0.5, for centered alignment. We modify them below to make our label left justified:

```
l["xalign"] <- 0
```

Unlike a block of text or an image, a widget usually does not have a fixed size. However, the user may wish to tweak how a widget fills the space allocated by its container. GTK+ provides the `GtkAlignment` container for this purpose. For example, rather than adjust the justification of the label text, we could have instructed the layout not to expand but to position itself against the left side of the window:

```
w <- gtkWindow(); w$setTitle("Hello world")
a <- gtkAlignment()
a$set(xalign = 0, yalign = 0.5, xscale = 0, yscale = 1)
w$add(a)
l <- gtkLabel("Hello world")
a$add(l)
```

2.3 Dialogs

GTK+ provides general infrastructure for constructing custom dialogs. A dialog is a window that generally consists of an icon, a content area, and an action area containing a row of buttons representing the possible user responses. A dialog is typically related to some main application window. Dialogs can be modal, blocking the rest of the application from any input, but this is optional. `GtkDialog` represents a generic dialog and serves as the base class for all special purpose dialogs in GTK+.

Message dialogs

Communicating textual messages to the user is perhaps the most common application of a dialog. GTK+ provides the `gtkMessageDialog` convenience wrapper for `GtkDialog` for creating a message dialog showing a primary and secondary message. We construct one presently:

```
w <- gtkWindow(); w['title'] <- "Parent window"
#
dlg <- gtkMessageDialog(parent=w,
                        flags="destroy-with-parent",
                        type="question",
                        buttons="ok",
                        "My message")
dlg['secondary-text'] <- "A secondary message"
```

The `flags` argument allows one to specify a combination of values from `GtkDialogFlags`. These include `destroy-with-parent` and `modal`. Here, the dialog will be destroyed upon destruction of the parent window. The `type` argument specifies the message type, using one of the 4 values from `GtkMessageType`, which determines the icon that is placed adjacent to the message text. The `buttons` argument indicates the set of response buttons with a value from `GtkButtonsType`. The remaining arguments are pasted together into the primary message. The dialog has a `secondary-text` property that can be set to give a secondary message.

Dialogs are optionally modal. Below, we enable modality by calling the `run` method, which will additionally block the R session:

```
response <- dlg$run()
```

```
if(response == GtkResponseType["cancel"] || # for other buttons
   response == GtkResponseType["close"] ||
   response == GtkResponseType["delete-event"]) {
  ## pass
} else if(response == GtkResponseType["ok"]) {
  print("Ok")
}
```

```
[1] "Ok"
```

```
dlg$Destroy()
```

The return value can then be inspected for the action, such as what button was pressed. `GtkMessageDialog` will return response codes from the `GtkResponseType` enumeration. We will see an example of asynchronous response handling in the next section.

Custom dialogs

The `gtkDialog` constructor returns a generic dialog object which can be customized, in terms of its content and response buttons. Usually, a `GtkDialog` is constructed with `gtkDialogNewWithButtons`, as a dialog almost always contains a set of response buttons, such as Ok, Yes, No and Cancel.

In this example, we will create a simple dialog showing a label and text entry.

```
dlg <- gtkDialogNewWithButtons(title="Enter a value",
                               parent=NULL, flags=0,
                               "gtk-ok", GtkResponseType["ok"],
                               "gtk-cancel", GtkResponseType["cancel"],
                               show=FALSE)
```

Buttons are added with a label and a response id, and their order is taken from the how they are entered. (There is no automatic ordering based on an operating system's conventions.) When the button label matches a stock ID, the icon and text are taken from the stock definition. We used standard responses from `GtkResponseType`, although in general the codes are simply integer values; interpretation is up to the programmer.

To complete our dialog we use the dialog as a container for the following:

```
hb <- gtkHBox()
hb['spacing'] <- 10
#
hb$packStart(gtkLabel("Enter a value:"))
entry <- gtkEntry()
hb$packStart(entry)
```

The dialog has a content area, which is an instance of `GtkVBox`. This is returned by `getContentArea`:

```
vb <- dlg$getContentArea()
vb$packStart(hb)
```

The content of the dialog is drawn above an optional horizontal separator and button box.

In the message dialog example, we called the `run` method to block the R session. An alternative that does not block the R session is to connect to the `response` signal of the modal dialog. The callback passes in the buttons' response codes assigned at construction.

```
ID <- gSignalConnect(dlg, "response",
                     f=function(dlg, resp, user.data) {
                       if(resp == GtkResponseType["ok"])
                         print(entry$getText()) # Replace this
                       dlg$Destroy()
                     })
dlg$showAll()
dlg$setModal(TRUE)
```

File chooser

A common task in a GUI is the selection of files and directories, for example to load or save a document. `GtkFileChooser` is an interface shared by widgets that choose files. GTK+ provides three such widgets. The first is `GtkFileChooserWidget`, which may be placed anywhere in a GUI. The other two are based on the first. `GtkFileChooserDialog` embeds the chooser widget in a modal dialog, while `GtkFileChooserButton` is a button that displays a file path and launches the dialog when clicked.

Example 2.1: An open file dialog
Here, we demonstrate the use of the dialog, the most popular of the three. An open file dialog can be created with:

```
dlg <- gtkFileChooserDialog(title="Open a file",
                     parent=NULL, action="open",
                     "gtk-ok", GtkResponseType["ok"],
                     "gtk-cancel", GtkResponseType["cancel"],
                     show=FALSE)
```

The dialog constructor allows one to specify a title, a parent and an action, either `open`, `save`, `select-folder` or `create-folder`. In addition, the dialog buttons must be specified, as with the last example using `gtkDialogNewWithButtons`.

We connect to the `response` signal

```
gSignalConnect(dlg, "response", f=function(dlg, resp, data) {
```

```
  if(resp == GtkResponseType["ok"]) {
    filename <- dlg$getFilename()
    print(filename)
  }
  dlg$destroy()
})
```

The file selected is returned by `getFilename`. If multiple selection is enabled with via the `select-multiple` property, one should call the plural `getFilenames`.

For the open dialog, one may wish to specify one or more filters that narrow the available files for selection:

```
fileFilter <- gtkFileFilter()
fileFilter$setName("R files")
fileFilter$addPattern("*.R")
fileFilter$addPattern("*.Rdata")
dlg$addFilter(fileFilter)
```

The `gtkFileFilter` function constructs a filter, which is given a name and a set of file name patterns, before being added to the file chooser. Filtering by mime type is also supported.

The save file dialog would be similar. The initial filename could be specified with `setFilename`, or folder with `setFolder`. The `do-overwrite-confirmation` property controls whether the user is prompted when attempting to overwrite an existing file.

Other features not discussed here, include embedding of preview and other custom widgets, and specifying shortcut folders.

Other choosers There are several other types of dialogs for making common types of selections. These include `GtkCalendar` for picking dates, `GtkColorSelectionDialog` for choosing colors, and `GtkFontSelectionDialog` for fonts. These are very high-level dialogs that are trivial to construct and manipulate, at a cost of flexibility.

2.4 Containers

In Section **??**, we presented `GtkBox` and `GtkAlignment`, the two most useful layout containers in GTK+. This section introduces some other important containers. These include the merely decorative `GtkFrame`; the interactive `GtkExpander`, `GtkPaned` and `GtkNotebook`; and the grid-style layout container `GtkTable`. All of these widgets are derived from `GtkContainer`, and so share methods like `add`, which adds a child.

Framed containers

The gtkFrame function constructs a container that draws a decorative, labeled frame around its single child. This is useful for visually segregating a set of conceptually related widgets from the rest of the GUI. The optional label argument specifies the label text, stored in the label property. The setLabelAlign aligns the label relative to the frame. Frames have a decorative shadow whose type, a value of GtkShadowType, is stored in the shadow-type property.

Expandable containers

The GtkExpander widget provides a button that hides and shows a single child upon demand. This is often an effective mechanism for managing screen space. Expandable containers are constructed by gtkExpander. Use gtkExpanderNewWithMnemonic if a mnemonic is desired. The label text can be passed to the constructor or set later with the setLabel method. The expanded property, which can be accessed with getExpanded and setExpanded, represents the visible state of the widget. When the expanded property changes, the activate signal is emitted.

Notebooks

The gtkNotebook constructor creates a notebook container, a widget that displays an array of buttons resembling notebook tabs. Each tab corresponds to a widget, and when a tab is selected, its widget is made visible, while the others are hidden. If GtkExpander is like a check button, GtkNotebook is like a radio button group.

The current page number is stored in the page property. The total number of pages is returned by getNPages. The default position of the notebook tabs is on the top, ordered from left to right. The property tab-pos represents the tab position with a value from GtkPositionType: "left", "right", "top", or "bottom".

Adding pages to a notebook New pages can be added to the notebook with the InsertPage method, which takes the widget associated with the page, the 0-based insertion position (defaults to last), as well as a widget, such as a GtkLabel instance, not a string, to label the tab. This allows for more complicated tabs, such as a box container with a label and close icon. The setTabLabelText method is a convenience for setting a label as text. To use this method, the child widget is needed, which can be retrieved with the [[method or the getNthPage method. Both are a shortcut around retrieving all of the children as a list through getChildren.

Manipulating pages Methods that manipulate pages operate on the page number. To map from the child widget to the page number, use the method `pageNum`. A given page can be raised with the `setCurrentPage` method. Incremental movements are possible through the methods `nextPage` and `prevPage`.

Pages can be reordered using the `reorderChild`, although it is usually desirable to allow the user to reorder pages. The `setTabReorderable` enables drag and drop reordering for a specific tab. It is also possible for the user to drag and drop pages between notebooks. Pages can be deleted using the method `removePage`.

Managing Many Pages By default, a notebook will request enough space to display all of its tabs. If there are many tabs, space may be wasted. `GtkNotebook` solves this with the scrolling idiom. If the property `scrollable` is set to `TRUE`, arrows will be added to allow the user to scroll through the tabs. In this case, the tabs may become difficult to navigate. Setting the `enable-popup` property to `TRUE` enables a right-click popup menu listing all of the tabs for direct navigation.

Signals The notebook widget emits signals when pages are toggled, added, removed, and reordered. The most useful is likely to be `switch-page`, which is emitted when the current page is changed.

Example 2.2: Adding a page with a close button
A familiar element of notebooks in many web browsers is a tab close button. The following defines a new method `insertPageWithCloseButton` that will use the themeable stock close icon. The callback passes both the notebook and the page through the `data` argument, so that the proper page can be deleted.

```
gtkNotebookInsertPageWithCloseButton <-
  function(object, child, label.text="", position=-1) {
    label <- gtkHBox()
    label$packStart(gtkLabel(label.text))
    icon <- gtkImage(pixbuf =
                     object$renderIcon("gtk-close", "button"))
    closeButton <- gtkButton()
    closeButton$setImage(icon)
    label$packEnd(closeButton)
    ID <- gSignalConnect(b,"clicked",
                         function(userData, b, ...) {
                           nb <- userData$nb
                           page <- userData$page
                           nb$removePage(nb$pageNum(page))
                         },
```

```
                        data = list(nb=object, page=child),
                        user.data.first=TRUE)
    object$insertPage(child, label, position)
  }
```

We now show a simple usage of a notebook.

```
w <- gtkWindow()
nb <- gtkNotebook(); w$add(nb)
nb$setScrollable(TRUE)
nb$insertPageWithCloseButton(gtkButton("hello"),
                             label.text="page 1")
nb$insertPageWithCloseButton(gtkButton("world"),
                             label.text="page 2")
```

Scrollable windows

The `GtkExpander` and `GtkNotebook` widgets support efficient use of screen real estate. However, when a widget is always too large to fit in a GUI, partial display is necessary. A `GtkScrolledWindow` supports this by providing scrollbars for the user to adjust the visible region of a single child. The range, step and position of `GtkScrollbar` are controlled by an instance of `GtkAdjustment`, just as with the slider and spin button.

The constructor `gtkScrolledWindow` creates a `GtkScrolledWindow` instance. By default, the horizontal and vertical adjustments are automatically determined, although they may be overridden by the programmer.

The widget in a scrolled window must know how to display only a part of itself, i.e., it must be scrollable. Some widgets, including `GtkTreeView` and `GtkTextView`, have native scrolling support. Other widgets must be embedded within the proxy `GtkViewport`. The `GtkScrolledWindow` convenience method `addWithViewport` allows the programmer to skip the `GtkViewport` step.

The properties `hscrollbar-policy` and `vscrollbar-policy` determine when the scrollbars are drawn. By default, they are always drawn. The `"automatic"` value from the `GtkPolicyType` enumeration draws the scrollbars only if needed, i.e, the child widget requests more space than can be allocated. The `setPolicy` method allows both to be set at once, as in the following example.

Example 2.3: Scrolled window example
This example shows how to display a long list of values with scrolled windows. The tree view widget can also do this, but here we can very easily customize the display of each value. In the example, we simply locates where a label is placed.

```
g <- gtkVBox(spacing=0)
```

```
sapply(state.name, function(i) {
  l <- gtkLabel(i)
  l['xalign'] <- 0; l['xpad'] <- 10
  g$packStart(l, expand=TRUE, fill=TRUE)
})
```

The scrolled window has just two basic steps in its construction. Here we specify never using a scrolled window for the vertical display.

```
sw <- gtkScrolledWindow()
sw$setPolicy("never","automatic")
sw$addWithViewport(g)          # just "Add" for text, tree, ...
```

```
w <- gtkWindow(show=FALSE)
w$setTitle("Scrolled window example")
w$setSizeRequest(-1, 300)
w$add(sw)
w$show()
```

Divided containers

The `gtkHPaned` and `gtkVPaned` constructors create containers that contain two widgets, arranged horizontally or vertically and separated by a handle. The user may adjust the position of the handle to apportion the allocation between the widgets.

The two children may be added two different ways. The methods `pack1` and `pack2` have arguments `resize`, whether the child expands with the parent, and `shrink`, whether the widget is allowed to shrink. The methods `add1` and `add2` add children such that both are allowed to shrink and only the second widget expands. After children are added, they can be referenced from the container through the `getChild1` and `getChild2` methods.

The screen position of the handle can be set with the `setPosition` method. The properties `min-position` and `max-position` can be used to convert a percentage into a screen position. The `move-handle` signal is emitted when the gutter position is changed.

Tabular layout

The `gtkTable` constructor produces a container for laying out objects in a tabular format. The container sets aside cells in a grid, and a child widget may occupy one or more cells. The `homogeneous` argument can be used to make all cells homogeneous in size. Otherwise, each column and row can have a different size. At the time of construction, the number rows and columns for the table many be specified with the `rows` and `columns` arguments. After construction, the `Resize` method can be used to resize these values.

Child widgets are added to this container through the `attach` method. Its first argument, `child`, is the child widget. This widget can span more than one cell. The arguments `left.attach` and `right.attach` specify the horizontal bounds of the child in terms of its left column and right column, respectively. Analogously, `top.attach` and `bottom.attach` define the vertical bounds. By default, the widgets will expand into and fill the available space, much as if `expand` and `fill` were passed as `TRUE` to `packStart` (see Section **??**). There is no padding between children by default. Both the resizing behavior and padding may be overridden by specifying additional arguments to `attach`.

The child properties `xalign` and `yalign` specify the alignment of child widgets within their allocated space. These behave as with `GtkAlignment`.

3

RGtk2: Basic Components

3.1 Buttons

The button is the very essence of a GUI. It communicates its purpose to the user and executes a command in response to a simple click or key press. In GTK+, A basic button is usually constructed using `gtkButton`, as the following example demonstrates.

Example 3.1: Button constructors

```
w <- gtkWindow(show=FALSE)
w$setTitle("Various buttons")
w$setDefaultSize(400, 25)
g <- gtkHBox(homogeneous=FALSE, spacing=5)
w$add(g)
b <- gtkButtonNew()
b$setLabel("long way")
g$packStart(b)
g$packStart(gtkButton(label="label only") )
g$packStart(gtkButton(stock.id="gtk-ok") )
g$packStart(gtkButtonNewWithMnemonic("_Mnemonic") ) # Alt-m to "click"
w$show()
```

A `GtkButton` is simply a clickable region on the screen that is decorated to appear as a button. `GtkButton` is a subclass of `GtkBin`, so it will accept any widget as an indicator of its purpose. By far the most common button

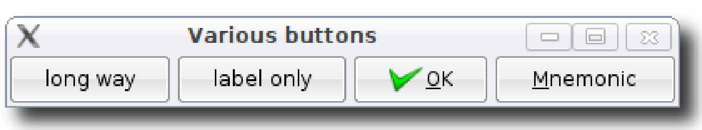

Figure 3.1: Various buttons

decoration is a label. The first argument of `gtkButton`, `label`, accepts the text for an automatically created `GtkLabel`. We have seen this usage in our "Hello World" example and others.

The alternative `stock.id` argument will use decorations associated with the stock identifier. For example, "gtk-ok" would produce a button with a theme-dependent image (such as a checkmark) and the "Ok" label, with the appropriate mnemonic and translated into the current language. The available stock identifiers are listed by `gtkStockListIds`. See `help(‘‘stock-items’’)` for more information.

The final button created in the example uses `gtkButtonNewWithMnemonic` to create a button with a mnemonic. Mnemonics are specified by prefixing the character with an underscore.

The method `setRelief` changes the relief style of the button. For example, the relief can be disable so that the button is drawn like a label.

Signals The `clicked` signal is emitted when the button is clicked on with the mouse or when the button has focus and the enter key is pressed. A callback can listen for this event to perform a command when the button is clicked.

Example 3.2: Callback example for `gtkButton`

```
w <- gtkWindow(); b <- gtkButton("click me");
w$add(b)
ID <- gSignalConnect(b,"button-press-event",   # just mouse click
                     f = function(w,e,data) {
                       print(e$getButton())    # which button
                       return(FALSE)           # propagate
                     })
ID <- gSignalConnect(b,"clicked",              # click or keyboard
                     f = function(w,...) {
                       print("clicked")
                     })
```

As buttons are intended to call an action immediately after being clicked, it is customary to make them insensitive to user input when the action is not possible. The `setSensitive` method can adjust this for the button, as with other widgets.

Windows often have a default action. For example, if a window contains a form, the default action often submits the form. If the action a button is to initiate is the default action for the window it can be set so that it is activated when the user presses enter while the parent window has the focus. To implement this, the property `can-default` must be `TRUE` and the widget method `grabDefault` must be called. (This is not specific to buttons, but any widget that can be activatable.)

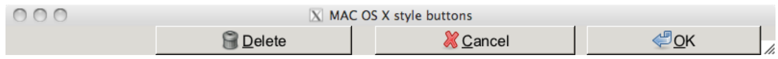

Figure 3.2: Example using stock buttons with extra spacing added between the `delete` and `cancel` buttons.

If the action that a button initiates is to be represented elsewhere in the GUI, say a menu bar, then a `GtkAction` object may be appropriate. Action objects are covered in Section **??**.

Example 3.3: Spacing between buttons
This example shows how to pack buttons into a box so that the spacing between the similar buttons is 12 pixels, but between potentially dangerous buttons is 24 pixels, as per the Mac human interface guidelines. GTK+ provides the constructor `gtkHButtonBox` for holding buttons, which provides a means to apply consistent styles, but the default styles do not allow such spacing as desired. (Had all we wanted was to right align the buttons, then that style is certainly supported.) As such, we will illustrate how this can be done through a combination of `spacing` arguments. We assume that our parent container, g, is a horizontal box container.

We include standard buttons, so use the stock names and icons.

```
cancel <- gtkButton(stock.id="gtk-cancel")
ok <- gtkButton(stock.id="gtk-ok")
delete <- gtkButton(stock.id="gtk-delete")
```

We will right align our buttons, so use the parent container's `PackEnd` method. The `ok` button has no padding, the 12-pixel gap between it and the `cancel` button is ensured by the `padding` argument when the `cancel` button is added. Treating the `delete` button as potentially irreversible, we aim to have 24 pixels of seperation between it and the `cancel` button. This is given by adding 12 pixels of padding when this button is packed in, giving 24 in total. The blank label is there to fill out space if the parent container expands.

```
g$packEnd(ok, padding=0)
g$packEnd(cancel, padding=12)
g$packEnd(delete, padding=12)
g$packEnd(gtkLabel(""), expand=TRUE, fill=TRUE)
```

We make `ok` the default button, so have it grab the focus and add a simple callback when the button is either clicked or the enter key is pressed when the button has the focus.

```
ok$grabFocus()
QT <- gSignalConnect(ok, "clicked", function(...) print("ok"))
```

Figure 3.3: Various formatting for a label: wrapping, alignment, ellipsizing, PANGO markup

3.2 Static Text and Images

Labels

The primary purpose of a label is to communicate the role of another widget, as we showed for the button. Labels are created by the `gtkLabel` constructor. Its main argument is `str` to specify the button text, stored in the `label` property. This text can be set with either `setLabel` or `setText` and retrieved with either `getLabel` or `getText`. The difference being the former respects formatting marks.

Example 3.4: Label formatting
As all text in a GTK+ GUI is ultimately displayed by `GtkLabel`, there are many formatting options available. This example demonstrates a sample of these (Figure **??**)

```
w <- gtkWindow(show=FALSE); w$setTitle("Label formatting")
w$setSizeRequest(250,300)                # narrow
g <- gtkVBox(spacing=2); g$setBorderWidth(5); w$add(g)
string <- "the quick brown fox jumped over the lazy dog"
## wrap by setting number of characters
basicLabel <- gtkLabel(string)
basicLabel$setLineWrap(TRUE)
basicLabel$setWidthChars(35)              # no. characters
## Set ellipsis to shorten long text
ellipsized <- gtkLabel(string)
ellipsized$setEllipsize("middle")
## Right justify text lines
## use xalign property for aligning entire block
```

```
rightJustified <- gtkLabel("right justify");
rightJustified$setJustify("right")
rightJustified['xalign'] <- 1
## PANGO markup
pangoLabel <- gtkLabel()
tmpl <- "<span foreground='blue' size='x-small'>%s</span>"
pangoLabel$setMarkup(sprintf(tmpl, string))
#
sapply(list(basicLabel, ellipsized, rightJustified, pangoLabel),
       g$packStart, expand = TRUE, fill = TRUE)
w$showAll()
```

Many of the text formatting options are demonstrated in Example **??**. Line wrapping is enabled with `setLineWrap`. Labels also support explicit line breaks, specified with "`\n`." The `setWidthChars` method is a convenience for instructing the label to request enough space to show a specified number of characters in a line. When space is at a premium, long labels can be ellipsized, i.e., have some of their text replaced with an ellipsis, "...". By default this is turned off; to enable, call `setEllipsize`. The property `justify`, with values taken from `GtkJustification`, controls the alignment of multiple lines within a label. To align the entire block of text within the space allocated to the label, modify the `xalign` property, as described in Section **??**.

GTK+ allows markup of text elements using the Pango text attribute markup language, an XML-based format that resembles basic HTML. The method `setMarkup` accepts text in the format. Text is marked using tags to indicate the style. Some convenient tags are `<b>` for bold, `<i>` for italics, `<ul>` for underline, and `<tt>` for monospace text. More complicated markup involves the `<span>` tag markup, such as `<span color='red'>some text</span>`. As with HTML, the text may need to be escaped first so that designated entities replace reserved characters.

Although mostly meant for static text display, `GtkLabel` has some interactive features. If the `selectable` property is set to `TRUE`, the text can be selected and copied into the clipboard. Labels can hold mnemonics for other widgets; this is useful for navigating forms. The mnemonic is specified at construction time with `gtkLabelNewWithMnemonic`. The `setMnemonicWidget` method identifies the widget to which the mnemonic refers.

Signals Unlike buttons, labels do not emit any specific signals, as they are intended to hold static text. Although a label is a `GtkWidget`, it does not receive any system events. A work-around is to place the label within an instance of `GtkEventBox`. This creates a non-visible parent window for the label that listens to the windowing system. Example **??** will illustrate the use of an event box. Alternatively, if a clickable label is desired, one could use an instance of `gtkButton` with its `relief` property assigned to `"none"`.

Statusbars

In GTK+, a statusbar is constructed through the `gtkStatusbar` function. Statusbars must be placed at the bottom of a top-level window by the programmer. In GTK+, a statusbar keeps various stacks of messages for display. One adds a message to display for given stack through the `Push` method by specifying first an integer value for `context.id` and a message. To pop the top message on a stack and display the next, the method `Pop` method is available.

Information bars

An information bar is similar in purpose to a message dialog, only is intended to be less obtrusive. GTK+ provides the `GtkInfoBar` class for managing information bars. The use is similar to a dialog: one places widgets into their content area, and listens to the `response` signal of the button. However, the presentation of the message in this case is a simple call to its `show` method (or `hide`). The placement of the bar is left to the user, it does not carry its own top-level window.

A simple use might look like:

```
ib <- gtkInfoBar(show=FALSE)
ib$setNoShowAll(TRUE)
```

We call `setNoShowAll`o that the widget isn't displayed when its parent container is. To this information bar we add a simple label and specify the message type using one of the enumerated values in `GtkMessageType`.

```
l <- gtkLabel("Warning, Warning ....")
ib$setMessageType("warning")
ib$getContentArea()$add(l)
```

A button to allow the user to hide the bar can be added as follows:

```
ib$addButton(button.text="gtk-ok",
             response.id=GtkResponseType['ok']) # not just "ok"
```

```
<pointer: 0x102a320c0>
attr(,"interfaces")
[1] "GtkActivatable"      "AtkImplementorIface"
[3] "GtkBuildable"
attr(,"class")
[1] "GtkButton"         "GtkBin"            "GtkContainer"
[4] "GtkWidget"         "GtkObject"         "GInitiallyUnowned"
[7] "GObject"           "RGtkObject"
```

As with dialogs, we connect to the `response` signal which is emitted when the button is activated.

```
gSignalConnect(ib, "response", function(w, ...) w$hide())
```

Our configuration is done. We now add the bar to a top-level window:

```
w <- gtkWindow(); w['title'] <- "Info bar example"
g <- gtkVBox()
w$add(g)
g$packStart(ib, expand=FALSE)
#
contentArea <- gtkHBox()
g$packStart(contentArea, expand=TRUE)
l <- gtkLabel("Lorem ipsum so dolor ...")
contentArea$packStart(l)
#
ib$show()                              # show
```

Progress bars

It is common to use a progress bar to indicate to the user the amount of time remaining during a long running computation. The `gtkProgressBar` creates the widget. The typical use involves placing a text label to notify the user and periodically updating the bar to reflect the percentage of time spent.

The following will do so:

```
w <- gtkWindow(); w$setTitle("Progress bar example")
pb <- gtkProgressBar()
w$add(pb)
#
pb$setText("Please be patient...")
for(i in 1:100) {
  pb$setFraction(i/100)
  Sys.sleep(0.05) ## replace with a step in the process
}
pb$setText("All done.")
```

Progress bars can also show activity but not an indication of duration. Activity is displayed by periodically pulsing the bar through the `pulse` method.

`GtkSpinner` Related to a progress bar is the `GtkSpinner` widget, which is also used to show an indefinite amount of activity, and is commonly used in web browser. The basic usage is straightforward: the object is created by `gtkSpinner` and the animation is begun and ended with teh respective methods `start` and `stop`.

Images

It is often said that a picture can be worth a thousand words. Applying this to GUIs, images can often provide a more space efficient alternative to labels.

GtkImage is the widget that displays images. The constructor gtkImage supports creating images from various in-memory image representations, files, and other sources. Images can be loaded after construction, as well. For example, the setFromFile method loads an image from a file.

The image widget, like the label widget, does not have a parent GdkWindow, which means it does not receive window events. As with the label widget, the image widget can be placed inside a gtkEventBox container if one wishes to connect to such events.

Example 3.5: Using a pixmap to present graphs

This example shows how to use a GtkImage object to embed a graphic within RGtk2, using the cairoDevice package. The basic idea is to draw onto an off-screen pixmap using cairoDevice and then to construct a GtkImage from the pixmap.

We begin by creating a window of a certain size.

```
w <- gtkWindow(show=FALSE); w$setTitle("Graphic window");
w$setSizeRequest(400,400)
g <- gtkHBox(); w$add(g)
w$showAll()
```

The size of the image is retrieved from the size allocated to the box g. This allows the window to be resized prior to drawing the graphic. Unlike an interactive device, after drawing, this graphic does not resize itself when the window resizes.

```
theSize <- g$getAllocation()$allocation
width <- theSize$width; height <- theSize$height
```

Now we create a GdkPixmap of the correct dimensions and initialize an R graphics device that targets the pixmap. We then draw a simple histogram using base R graphics.

```
require(cairoDevice)
pixmap <- gdkPixmap(drawable = NULL, width = width, height = height,
                    depth = 24)
asCairoDevice(pixmap)
```

```
[1] TRUE
```

```
hist(rnorm(100))
```

The final step is to create the GtkImage widget to display the pixmap:

```
image <- gtkImage(pixmap = pixmap)
g$packStart(image, expand=TRUE, fill = TRUE)
```

Stock icons

In GTK+, standard icons, like the one on the "OK" button, can be customized by themes. This is implemented by a database that maps a *stock* identifier to an icon image. The stock identifier corresponds to a commonly performed type of action, such as the "OK" response or the "Save" operation. There is no hard-coded set of stock identifiers, however GTK+ provides a default set for the most common operations. These identifiers are all prefixed with "gtk-". Users may register new types of stock icons.

As mentioned previously, the full list of stock icons are returned in a list by `gtkStockListIds`. The first 4 are:

```
head(unlist(gtkStockListIds()), n=4)
```

```
[1] "gtk-zoom-out" "gtk-zoom-in"  "gtk-zoom-fit" "gtk-zoom-100"
```

The use of stock identifiers over specific images is encouraged, as it allows an application to be customized through themes. The `gtkButton` and `gtkImage` constructors accept a stock identifier passed as `stock.id` argument, and the icons in toolbars and menus are most conveniently specified by stock identifier.

Example 3.6: An alert panel

This example shows how to create an alert panel, a hideable space at the top of a window to place messages. The widget we create is similar to that now provided by `GtkInfoBar`. Our point here is not to replicate that functionality, but rather to show how the `hide` and `show` methods can be used and how an event box is utilized to capture events and to allow for background painting.

We will employ R's reference methods to define our widget. We break up the method definitions for formatting purpose, the longest of which is the initialization method which creates the widgets. An event box is used so that we can color the background, as this isn't possible for a box container due to its lack of a gdk window. To this event box we add a box container that will hold an icon indicating this is an alert, a label for the message, and another icon to indicate to the user how to close the alert. Since we wish to receive mouse clicks on close icon, we place this inside another event box. To this, we bind a callback to the `button-press-event` signal.

```
initialize <-
  function(message="", icon="gtk-dialog-warning", panel.color="goldenrod") {
    "Initialize widgets"
    widget <<- gtkEventBox(show=FALSE)
    widget$ModifyBg(state="normal", color=panel.color)
    ##
    g <- gtkHBox(homogeneous=FALSE, spacing=5)
```

```
    widget$add(g)
    ##
    image <- gtkImageNewFromStock(icon, size="button")
    image['yalign'] <- .5
    g$packStart(image, expand=FALSE)
    ##
    label <<- gtkLabel(message)
    label['xalign'] <- 0; label['yalign'] <- .5
    label$setLineWrap(TRUE)
    g$packStart(label, expand=TRUE, fill=TRUE)
    ##
    xbutton <- gtkEventBox()
    xbutton$modifyBg(state="normal", color=panel.color)
    xbutton$add(gtkImageNewFromStock("gtk-close", size="menu"))
    g$packEnd(xbutton, expand=FALSE, fill=FALSE)
    ##
    callback <- function(...) {hide(); FALSE}
    sapply(list(xbutton, widget), function(i)
           gSignalConnect(i, "button-press-event", f= callback))
    ##
    message <<- message                 # set message
    ##
    .self
  }
```

Now we can define our main class.

```
setRefClass("AlertPanel",
            fields = list(
              widget="ANY",             # main event box
              image="ANY", label="ANY",
              icon="character", message="character"
              ),
            methods=list(
              initialize=initialize,
              set_message=function(msg) {
                "Set message, show box"
                message <<- msg
                show()
              },
              show=function() {
                "Show message"
                label$setText(message)
                widget$show()
              },
              hide=function() {
                "hide message"
                widget$hide()
```

```
               }))
```

```
Generator object for class "AlertPanel":

Class fields:

Name:      widget     image      label       icon    message
Class:        ANY       ANY        ANY character character

 Class Methods:
    "callSuper", "export", "hide", "import", "initFields",
"initialize", "set_message", "show"


 Reference Superclasses:
    "envRefClass"
```

To test it out we have the following example. Our widget needs to be placed at the top of the window (or bottom).

```
w <- gtkWindow()
g <- gtkVBox(); w$add(g)
#
ap <- getRefClass("AlertPanel")$new()
g$packStart(ap$widget, expand=FALSE)
#
g$packStart(gtkLabel("fill space"), expand=TRUE, fill=TRUE)
ap$set_message("New Message")
```

To improve this, one could also add a time to close the panel after some delay. The `gTimeoutAdd` function is used to specify a function to call periodically until the function returns `FALSE`.

3.3 Input Controls

Text entry

The widgets explained thus far are largely static. For example, GTK+ does not yet support editable labels. Text editing is handled by other widgets that are rendered in the familiar depressed box form. We will discuss complex multi-line text editing in Section **??**. For entering a single line of text, the `GtkEntry` widget is appropriate. It is constructed by `gtkEntry`. Specifying the `max` argument calls a deprecated function, instead use the method `setMaxLength` after construction.

The `text` property stores the text. This can be set with the method `setText` and retrieved with `getText`. Editing text programmatically relies on the `GtkEditable` interface, which `GtkEntry` implements. The method `insertText` inserts text. Its argument `new.text` contains the text and `position` specifies the position of the text to be added. The return value is a list with the component `position` indicating the position *after* the new text. The `deleteText` method deletes text. This takes two integers indicating the start and finish location of the text to delete.

Example 3.7: Insert and Delete text
The example will show how to add then delete text.

```
e <- gtkEntry()
e$setText("Where did that guy go?")
add.pos <- regexpr("guy", e['text']) - 1 # before "guy"
ret <- e$insertText("@$#%! ", position = add.pos)
e$getText()                              # or e['text']
```

```
[1] "Where did that @$#%! guy go?"
```

```
e$deleteText(start = add.pos, end= ret$position)
e$getText()
```

```
[1] "Where did that guy go?"
```

The `GtkEditable` interface supports three signals: `changed` when text is changed, `delete-text` for delete events, and `insert-text` for insert events. It is possible to prevent the insertion or deletion of text by connecting to the corresponding signal and stopping the signal propagation with `gSignalStopEmission`.

`GtkEntry` defines a number of its own signals, including the `activate` signal, which is emitted when the enter key is pressed.

Check button

Very often, the action performed by a button simply changes the value of a state variable in the application. GTK+ defines several types of buttons that explicitly manage and display the value of a state variable. The simplest type of state variable is binary (boolean) and is usually proxied by a `GtkCheckButton`.

A `GtkCheckButton` is constructed by `gtkCheckButton`. The optional argument `label` places a label next to the button. The alternative constructor `gtkCheckButtonNewWithMnemonic` gives the label a mnemonic.

As with any `GtkButton`, the `label` property stores the label. The state of the binary variable is represented by the `active` property. It can be set or retrieved with the methods `setActive` and `getActive`.

When the state is changed the `toggle` signal is emitted. The callback should check the `active` property to determine if the button has been enabled or disabled.

An alternative to GtkCheckButton is the lesser used GtkToggleButton, which is actually the parent class of GtkCheckButton. A toggle button is drawn as an ordinary button. It remains depressed while the state variable is TRUE, instead of relying on a check box to communicate the binary value.

Radio button group

GTK+ provides two button types for discrete state variables that accept more than two possible values: combo boxes, discussed in the next section, and radio buttons. The gtkRadioButton constructor creates an instance of GtkRadioButton, an extension of GtkCheckButton. Each radio button belongs to a group. There is no explicit group object; rather, the buttons are chained together as a linked list. By default, a newly constructed button is added to its own group. If group is a list of radio buttons, the newly created button is added to the group. The constructor returns a single radio button widget.

Like other types derived from GtkToggleButton, each radio button in the group has an active property. Only one button in the group can have active set to TRUE at a time. To determine which button is active, each button needs to be queried individually. Setting active to TRUE activates the corresponding button and ensures that the other buttons are disabled.

Example 3.8: Radio group construction
Creating a new radio button group with the basic gtkRadioButton constructor follows this pattern:

```
vals <- c("two.sided", "less", "greater")
radiogp <- list()                                  # list for group
radiogp[[vals[1]]] <- gtkRadioButton(label=vals[1]) # group = NULL
for(i in vals[-1])
  radiogp[[i]] <- gtkRadioButton(radiogp, label=i)  # group is a list
```

Each button needs to be managed. Here we illustrate a simple GUI doing so.

```
w <- gtkWindow(); w$setTitle("Radio group example")
g <- gtkVBox(FALSE, 5); w$add(g)
sapply(radiogp, gtkBoxPackStart, object = g)
```

We can set and query which button is active, as follows:

```
g[[3]]$setActive(TRUE)
sapply(radiogp, '[', "active")
```

```
two.sided      less   greater
    FALSE     FALSE      TRUE
```

Here is how we might register a callback for the toggled signal.

```
sapply(radiogp, gSignalConnect, "toggled",     # connect each
```

```
        f = function(w, data) {
          if(w$getActive()) # set before callback
            cat("clicked", w$getLabel(),"\n")
        })
```

The getGroup method returns a list containing the radio buttons in the same group. However, it is in the reverse order of construction (newest first). This results from an internal optimization that prepends, rather than appends, the buttons to a linked list.

As a convenience, there are constructor functions ending with FromWidget that determine the group from a radio button belonging to the group. As we will see in our second example, this allows for a more natural sapply idiom that avoids the need to allocate a list and populate it in a for loop.

Example 3.9: Radio group using getGroup

In this example below, we illustrate two things: using the gtkRadioButtonNewWithLabelFromWidget function to add new buttons to the group and the GetGroup method to reference the buttons. The rev function is used to pack the widgets, to get them to display first to last.

```
radiogp <- gtkRadioButton(label=vals[1])
sapply(vals[-1], gtkRadioButtonNewWithLabelFromWidget, group = radiogp)
w <- gtkWindow();
w['title'] <- "Radio group example"
g <- gtkVBox(); w$add(g)
sapply(rev(radiogp$getGroup()), gtkBoxPackStart, object = g)
```

Combo boxes

The combo box is a more space efficient alternative to radio buttons and is better suited for when there are a large number of options. A basic, text-only GtkComboBox is constructed by gtkComboBoxNewText. Later we will discuss more complicated combo boxes, where an underlying data model is manipulated.

For the basic combo box, items may be added in a few different ways. The methods appendText and prependText add a text item to the end or beginning, respectively. A text item is inserted at an arbitrary position in the list with the insertText.

The currently selected value is specified by index with the method setActive and returned by getActive. The index, as usual, is 0-based, and in this case, a value of -1 indicates that no value is selected. The getActiveText method can be used to retrieve the text shown by the basic combo box.

Although combo boxes are much more space efficient than radio buttons, it can be difficult to use a combo box when there are a large number of selections. The setWrapWidth method specifies the preferred number of columns for displaying the items.

The main signal to connect to is `changed` which is emitted when the active item is changed either by the user or the programmer through the `setActive` method.

Example 3.10: Using one combo box to populate another
We use two convenience functions below from the `ProgGUIInR` package to find the possible data frames, and for a data frame to find the variables.

```
source("~/GUI/ProgGUIInR/R/misc.R")
```

We have a basic set of widgets:

```
w <- gtkWindow(show=FALSE)
w$setTitle("gtkComboBox example")
df_combo <- gtkComboBoxNewText()
var_combo <- gtkComboBoxNewText()
```

Our layout uses boxes, a table is better but that comes later. To add a twist, we will hide our variable combobox until after a data frame has been initially selected.

```
g <- gtkVBox(); w$add(g)
#
g1 <- gtkHBox(); g$packStart(g1)
g1$packStart(gtkLabel("Data frames:"))
g1$PackStart(df_combo)
#
g2 <- gtkHBox(); g$packStart(g2)
g2$packStart(gtkLabel("Variable:"))
g2$PackStart(var_combo)
g2$hide()
```

Finally we configure the comboboxes. When a data frame is selected we first clear out the variable combobox (by calling it's model's `Clear` method) the populate.

```
sapply(avail_dfs(), gtkComboBoxAppendText, object=df_combo)
df_combo$setActive(-1)
#
gSignalConnect(df_combo, "changed", function(w, ...) {
  var_combo$getModel()$clear()          # how to clear, or removeText.
  sapply(find_vars(w$getActiveText()), gtkComboBoxAppendText, object=var_combo)
  g2$show()
})
```

Example 3.11: Dialog layout
This example shows how to layout some controls for a dialog with some attention paid to how the widgets are aligned and how they respond to resizing of the window.

Our basic GUI is a table with 4 rows and 2 columns.

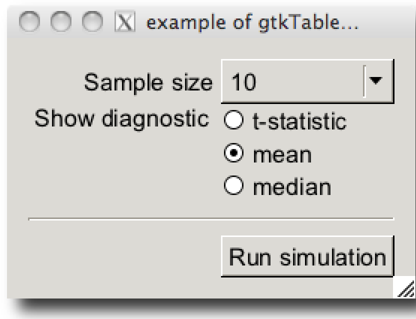

Figure 3.4: A basic dialog using a `gtkTable` container for layout.

```
w <- gtkWindow(show=FALSE)
w['border-width'] <- 14
w$setTitle("example of gtkTable and attaching")
tbl <- gtkTable(rows=4, columns=2, homogeneous=FALSE)
w$add(tbl)
```

We define our widgets first then do their layout.

```
l1 <- gtkLabel("Sample size")
w1 <- gtkComboBoxNewText()
sapply(c(5, 10, 15, 30), gtkComboBoxAppendText, object=w1)
#
l2 <- gtkLabel("Show diagnostic ")
w2 <- gtkVBox()
rb <- list()
rb[["t"]] <- gtkRadioButton(label="t-statistic")
for(i in c("mean","median")) rb[[i]] <- gtkRadioButton(rb, label=i)
sapply(rb, gtkBoxPackStart, object=w2)
#
w3 <- gtkButton("Run simulation")
```

The basic `AttachDeafults` method will cause the widgets to expand when resized, which we want to control here. As such we use `Attach`. To get the control's label to center align yet still have some breathing room we set its `xalign` and `xpad` properties. For the combobox we avoid using `"expand"` as otherwise it resizes to fill the space allocated to the cell in the `y` direction.

```
tbl$attach(l1, left.attach=0,1, top.attach=0,1, yoptions="fill")
l1["xalign"] <- 1; l1["xpad"] <- 5
tbl$attach(w1, left.attach=1,2, top.attach=0,1, xoptions="fill", yoptions="fill")
```

We use `"expand"` here to attach the radio group, so that it expands to fill the space. The label has its `yalign` proporty set, so that it stays at the top of the cell, not the middle.

```
tbl$attach(l2, left.attach=0,1, top.attach=1,2, yoptions="fill")
l2["xalign"] <- 1; l2['yalign'] <- 0; l2["xpad"] <- 4
tbl$attach(w2, left.attach=1,2, top.attach=1,2, xoptions=c("expand", "fill"))
```

A separator with a bit of padding provides a visual distinction between the controls and the button to initiate an action.

```
tbl$attach(gtkHSeparator(),left.attach=0,2, top.attach=2,3, ypadding=10, yoptions="fill")
tbl$attach(w3, left.attach=1,2, top.attach=3,4, xoptions="fill", yoptions="fill")
```

To finish, we call the windows `show` method.

```
w$show()
```

Sliders

The slider widget and spin button widget allow selection from a regularly spaced, semi-continuous list of values.

The slider widget is called `GtkScale` and may be oriented either horizontally or vertically. This depends on the constructor: `gtkHScale` or `gtkVScale`. The user must specify the minimum, maximum and step values for the scale. This set of values is formally represented by the `GtkAdjustment` structure. Ordinarily, it is not necessary to construct a `GtkAdjustment` explicitly. Instead, the constructors accept the the numeric arguments `min`, `max`, and `step`.

The underlying `GtkAdjustment` serves as the data model for the slider. Multiple sliders can be synchronized by attaching to the same adjustment object.

The methods `getValue` and `setValue` can be used to get and set the value of the widget. Values are clamped to the bounds defined by the adjustment.

A few properties define the appearance of the slider widget. The `digits` property controls the number of digits after the decimal point. The property `draw-value` toggles the drawing of the selected value near the slider. Finally, `value-pos` specifies where this value will be drawn using values from `GtkPositionType`. The default is `top`.

Callbacks can be assigned to the `value-changed` signal, which is emitted when the slider is moved.

Example 3.12: A slider controlling histogram bin selection
A simple mechanism to make a graph interactive is to redraw the graph whenever a slider, controlling a plot parameter, is changed. The following shows how this can be achieved.

```
library(lattice)
x <- rnorm(100)                          # the data
drawHistogram <- function(val) print(histogram(x, nint = val))
#
w <- gtkWindow(); w$setTitle("Histogram bin selection")
```

```
#
slider <- gtkHScale(min = 1, max = 100, step = 1)
slider$setValue(10)                       # initial value
slider['value-pos'] <- "bottom"
w$add(slider)
#
gSignalConnect(slider, "value-changed",
               f = function(w, ...) {
                 val <- w$getValue()
                 drawHistogram(val)
               })
#
drawHistogram(slider$getValue())          # initial graphic
```

Spin buttons

The spin button widget is very similar to the slider widget, conceptually and in terms of the GTK+ API. Spin buttons are constructed with `gtkSpinButton`. As with sliders, this constructor requires specifying adjustment values, either as a `GtkAdjustment` or individually.

As with sliders, the methods `getValue` and `setValue` get and set the widgets value. The property `snap-to-ticks` can be set to `TRUE` to force the new value to belong to the sequence of values in the adjustment. The `wrap` property indicates if the sequence will "wrap" around at the bounds.

The `value-changed` signal is emitted when the spin button is changed, as with sliders.

Example 3.13: A range widget

This example shows how to make a range widget that combines both the slider and spinbutton to choose a single number. Such a widget is popular, as the slider is better at large changes and the spin button better at finer changes. In GTK+ we use the same `GtkAdjustment` model, so changes to one widget propogate without effort to the other.

Were this written as a function, an R user might expect the arguments to match those of `seq`:

```
from <- 0; to <- 100; by <- 1
```

The slider is drawn without a value, as the value is already displayed by the spin button. The call to `gtkHScale` implicitly creates an adjustment for the slider. The spin button is created with the same adjustment.

```
slider <- gtkHScale(min=from, max=to, step=by)
slider['draw-value'] <- FALSE
adjustment <- slider$getAdjustment()
spinbutton <- gtkSpinButton(adjustment = adjustment)
```

Our layout places the two widgets in a horizontal box container with the slider set to expand into the available space, but not the spinbutton.

```
g <- gtkHBox()
g$packStart(slider, expand=TRUE, fill=TRUE, padding=5)
g$packStart(spinbutton, expand=FALSE, padding=5)
```

3.4 Graphics

The cairoDevice package

The package `cairoDevice` is an R graphics device based on the Cairo graphics library. It is cross-platform and supports alpha-blending and antialiasing. Through its support for the `getGraphicsEvent` function, it is currently the most interactive cross-platform graphics device.

`RGtk2` and `cairoDevice` are integrated through the `asCairoDevice` function.If a `GtkDrawingArea`, `GdkDrawable`, `Cairo` context, or `GtkPrintContext` is passed to `asCairoDevice`, an R graphics device will be initialized that targets its drawing to the object. Example **??** employed this function to paint a histogram on a pixmap. For simply displaying graphics in a GUI, the `GtkDrawingArea` is the best choice. For more complex use cases, such as compositing a layer above or below the R graphic, one should pass an off-screen `GdkDrawable`, like a `GdkPixmap`, or a `Cairo` context. The off-screen drawing can then be composited with other images when displayed. Finally, passing a `GtkPrintContext` to `asCairoDevice` allows printing R graphics through the GTK+ printing dialogs.

3.5 Drag and drop

GTK+ has mechanisms to provide drag and drop facilities for widgets. To setup drag and drop actions requires setting a widget to be a source for a drag request, and setting a widget to be a target for a drop action, and assigning callbacks to respond to certain signals. Only widgets which can receive signals will work for drag and drop, so to drag or drop on a label, say, an event box must be used.

We illustrate how to set up the dragging of a text value from one widget to another. Much more complicated examples are possible, but we do not pursue it here.

When a drag and drop is initiated, different types of data may be transferred. GTK+ allows the user to specify a target type. Below, we define target types for text and pixmap objects. These give numeric IDs for lookup purposes.

```
TARGET.TYPE.TEXT   <- 80
TARGET.TYPE.PIXMAP <- 81
```

We use these to make different types of objects that can be dragged.

```
widgetTargetTypes <- list(
## target -- string representing the drag type. MIME type used.
## flag delimiting drag scope. 0 -- no limit
## info -- application assigned value to identify
text = gtkTargetEntry("text/plain", 0, TARGET.TYPE.TEXT),
pixmap = gtkTargetEntry("image/x-pixmap", 0, TARGET.TYPE.PIXMAP)
)
```

A drag source A widget that can have a value dragged from it is a drag source. It is specified by calling `gtkDragSourceSet`. This function has arguments `object` for the widget we are making a source, `start.button.mask` to specify which mouse buttons can initiate the drag, `targets` to specify the target type, and `actions` to indicate which of the `GdkDragAction` types is in effect, for instance `copy` or `move`.

When a widget is a drag source, it sends the data being dragged in response to the `drag-data-get` signal using a callback. The signature of this callback is important, although we only use the `selection` argument, as this is assigned the text that will be the data passed to the target widget. (Text, as we are passing text information.)

```
w <- gtkWindow(); w['title'] <- "Drag Source"
dragSourceWidget <-  gtkButton("Drag me")
w$add(dragSourceWidget)
gtkDragSourceSet(dragSourceWidget,
                 start.button.mask=c("button1-mask", "button3-mask"),
                 targets=widgetTargetTypes[["text"]],
                 actions="copy") ## can also be any of GdkDragAction
ID <-
  gSignalConnect(dragSourceWidget, "drag-data-get",
                 f=function(widget, context,
                   selection, targetType, eventTime) {
                   ## customize this to set the text
                   selection$setText(str="some value")
                 })
```

Drop target To make a widget a drop target, we call `gtkDragDestSet` on the object with the argument `flags` for specifying the actions GTK+ will perform when the widget is dropped on. We use the value `"all"` for `"motion"`, `"highlight"`, and `"drop"`. The `targets` argument matches the type of data being allowed, in this case text. Finally, the value of `action` specifies what `GdkDragAction` should be sent back to the drop source widget. If the action was `"move"` then the source widget emits the `drag-data-delete` signal, so that a callback can be defined to handle the deletion of the data.

```
w <- gtkWindow(); w['title'] <- "Drop Target"
dropTargetWidget <- gtkButton("Drop here")
w$add(dropTargetWidget)
gtkDragDestSet(dropTargetWidget,
               flags="all",
               targets=widgetTargetTypes[["text"]],
               actions="copy"
               )
```

When data is dropped, the widget emits the `drag-data-received`. The data is passed to the callback through the `selection` argument. The `context` argument is a `gdkDragContext`, containing information about the drag event. The `x` and `y` arguments are integer valued and pass in the position in the widget where the drop occurred. In the example below, we see that text data is passed to this function in `raw` format, so it is converted with `rawToChar`.

```
ID <-
  gSignalConnect(dropTargetWidget, "drag-data-received",
                 f=function(dropTargetWidget,
                   context, x, y,
                   selection, targetType, eventTime) {
                   dropdata <- selection$getText()
                   if(class(dropdata)[1] == "raw")
                     val <- paste(rawToChar(dropdata), sep="")
                   else
                     val <- paste(dropdata, sep="")
                   print(val) ## some action
                 })
```

4

RGtk2: Widgets Using Data Models

Many widgets in GTK+ use the model, view, controller (MVC) paradigm. For most, like the button, the MVC pattern is implicit; however, widgets that primarily display data explicitly incorporate the MVC pattern into their design. The data model is factored out as a separate object, while the widget plays the role of the view and controller. The MVC approach adds a layer of complexity but facilitates the display of the dynamic data in multiple, coordinated views.

4.1 Display of tabular data

Widgets that display lists, tables and trees are all based on the same basic data model, `GtkTreeModel`. Although its name suggests a hierarchical structure, `GtkTreeModel` is also tabular. We first describe the display of an R data frame in a list or table view. The display of hierarchical data, as well as further details of the `GtkTreeModel` framework, are treated subsequently.

Loading a data frame

As an interface, `GtkTreeModel` may be implemented in any number of ways. GTK+ provides simple in-memory implementations for hierarchical and non-hierarchical data. For improved speed, convenience and familiarity, `RGtk2` includes a custom `GtkTreeModel` implementation called `RGtkDataFrame`, which is based on an R data frame. For non-hierarchical data, this is usually the model of choice, so we discuss it first.

R uses data frames to hold tabular data, where each column is of a certain class, and each row is related to some observational unit. This fits the structure of `GtkTreeModel` when there is no hierarchy. As such it is natural to have a means to map a data frame into a store for a tree view. `RGtkDataFrame` implements `GtkTreeModel` to perform this role and is constructed with the `rGtkDataFrame` function. Populating a `RGtkDataFrame` is far faster than for a GTK+ model, because data is retrieved from the data frame on demand.

There is no need to copy the data row by row into a separate data structure. Such an approach would be especially slow if implemented as a loop in R. The constructor takes a data frame as an argument. The column classes are important, so even if this data frame is empty, the user should specify the desired column classes upon construction.

An object of class `RGtkDataFrame` supports the familiar S3 methods `[`, `[<-`, `dim`, and `as.data.frame`. The `[<-` method does not have quite the same functionality as it does for a data frame. Columns can not be removed by assigning values to `NULL`, and column types should not be changed. These limitations are inherit in the design of GTK+: columns may not be removed from `GtkTreeModel`, and views expect the data type to remain the same.

Example 4.1: Defining and manipulating a `RGtkDataFrame`
The basic data frame methods are similar.

```
data(Cars93, package="MASS")            # mix of classes
model <- rGtkDataFrame(Cars93)
model[1, 4] <- 12
model[1, 4]                              # get value
```

```
[1] 12
```

Factors are treated differently from character values, as is done with data frames, so assignment to a factor must be from one of the possible levels.

The data frame combination functions `rbind` and `cbind` are unsupported, as they would create a new data model, rather than modify the model in place. Thus, one should add rows with `appendRows` and add columns with `appendColumns` (or sub-assignment, `[<-`).

The `setFrame` method replaces the underlying data frame.

```
model$setFrame(Cars93[1:5, 1:5])
```

Replacing the data frame is the only way to remove rows, as this is not possible with the conventional data frame sub-assignment interface. Removing columns or changing their types remains impossible. The new data frame cannot contain more columns and rows than the current one. If the new data frame has more rows or columns, then the appropriate `append` method should be used first.

Displaying data as a list or table

`GtkTreeView` is the primary view of `GtkTreeModel`. It serves as the list, table and tree widget in GTK+. A tree view is essentially a container of columns, where every column has the same number of rows. If the view has a single column, it is essentially a list. If there are multiple columns, it is a table. If the rows are nested, it is a tree table, where every node has values on the same columns.

A tree view is constructed by `gtkTreeView`.

```
view <- gtkTreeView(model)
```

Usually, as in the above, the model is passed to the constructor. Otherwise, the model may be accessed with `setModel` and `getModel`.

A newly created tree view displays zero columns, regardless of the number of columns in the model. Each column, an instance of `GtkTreeViewColumn`, must be constructed, inserted into the view and instructed to render content based on one or more columns in the data model:

```
vc <- gtkTreeViewColumn()
vc$setTitle("Manufacturer")
cr <- gtkCellRendererText()
vc$packStart(cr)
vc$addAttribute(cr, "text", 0)
view$insertColumn(vc, 0)
```

A column with the title "Manufacturer" is inserted at the first, 0-based, position. For displaying a simple data frame, we only need to render text. Each row in a column consists of one or more cells, managed in a layout. The number of cells and how each cell is rendered is uniform down a column. As an implementation of `GtkCellLayout`, `GtkTreeViewColumn` delegates the responsibility of rendering to one or more `GtkCellRenderer` objects. The cell renderers are packed into the column, which behaves much like a box container. Rendering of text cells is the role of `GtkCellRendererText`; we create an instance with `gtkCellRendererText`. There are several properties that control how the text is rendered. A so-called *attribute* links a model column to a renderer property. The most important property is `text`, the text itself. In the example, we bind the `text` property to the first (0-indexed) column in the model.

`GtkTreeView` provides the `insertColumnWithAttributes` convenience method to perform all of these steps with a single call. We invoke it to add a second column in our view:

```
view$insertColumnWithAttributes(position = -1,
     title = "Model", cell = gtkCellRendererText(), text = 1)
```

The -1 passed as the first argument indicates that the column should be appended. Next, we specify the column title, a cell renderer, and an attribute that links the `text` renderer property to the second column in the model. In general, any number of attributes may be defined after the third argument. We will use the above idiom in all of the following examples, as it is much more concise than performing each step separately.

To display the entire Cars93 data frame, we insert a view column for every column in the data frame. Here, we reconstruct the view, inserting a view column for every column in the data frame, i.e., the model.

```
view <- gtkTreeView(model)
mapply(view$insertColumnWithAttributes,  -1, colnames(model),
       list(gtkCellRendererText()),
       text = seq_len(ncol(model)) - 1)
```

Although it was relatively easy to create a GtkTreeModel for the data frame using RGtkDataFrame, the complexity of GtkTreeView complicates the task of displaying the data frame in a simple, textual table. When this is all that is necessary, one might consider gtable from gWidgets. For those who wish to render text in each row differently (e.g., in a different color) or fill cells with images, check boxes, progress bars and the like, direct use of the GtkTreeView API is required.

Manipulating view columns The GtkTreeView widget is essentially a collection of columns. Columns are added to the tree view with the methods insertColumn or, as shown above, insertColumnWithAttributes. A column can be moved with the moveColumnAfter method, and removed with the removeColumn method. The getColumns method returns a list containing all of the tree view columns.

There are several properties for controlling the behavior and dimensions of a GtkTreeViewColumn instance. The property "resizable" determines whether the user can resize a column, by dragging with the mouse. The size properties "width", "min-width", and "fixed-width" control the size. The visibility of the column can be adjusted through the setVisible method.

Additional Features Tree views have several special features, including sorting, incremental search and drag-n-drop reordering. Sorting is discussed in Section **??**. To turn on searching, enable-search should be TRUE (the default) and the search-column property should be set to the column to be searched. The tree view will popup a search box when the user types control-f. To designate an arbitrary text entry widget as the search box, call setSearchEntry. The entry can be placed anywhere in the GUI. Columns are always reorderable by drag and drop. Reordering rows through drag-and-drop is enabled by the reorderable property.

Aesthetic properties GtkTreeView is capable of rendering some visual guides. The rules-hint, if TRUE, will instruct the theme to draw rows in alternating colors. To show grid lines, set enable-grid-lines to TRUE.

Accessing GtkTreeModel

Although RGtkDataFrame provides a familiar interface for manipulating the data in a GtkTreeModel, it is often necessary to directly interact with the GTK+ API, such as when using another type of data model or interpreting user selections. There are two primary ways to index into the rows of a tree model: paths and iterators.

To index directly into an arbitrary row, a GtkTreePath is appropriate. For a list, a tree path is essentially the row number, 0-based; for a tree it is a sequence of integers referring to the offspring index at each level. The sequence of integers may be expressed as either a numeric/integer vector or a string, using gtkTreePathNewFromIndices or gtkTreePathNewFromString, respectively. For a flat list model, there is only one integer in the sequence:

```
secondRow <- gtkTreePathNewFromIndices(2)
```

Referring to a row in a hierarchy is slightly more complex:

```
abcPath <- gtkTreePathNewFromIndices(c(1, 3, 2))
abcPath <- gtkTreePathNewFromString("1:3:2")
```

In the above, both paths refer to the second child of the third child of the first top-level node. To recover the integer or string representation of the path, use getIndices or toString, respectively.

The second means of row indexing is through an iterator, GtkTreeIter, which is better suited for traversing a model. While a tree path is an intuitive, transparent row index, an iterator, by contrast, is an opaque index that is efficiently incremented. It is probably most common for a model to be accessed in an iterative manner, so all of the data accessor methods for GtkTreeModel expect GtkTreeIter, not GtkTreePath. The GTK+ designers imagined that the typical user would obtain an iterator for the first row and to visit each row in sequence:

```
iter <- model$getIterFirst()
manufacturer <- character()
while(iter$retval) {
  manufacturer <- c(manufacturer, model$get(iter$iter, 0)[[1]])
  model$iterNext(iter$iter)
}
```

In the above, we recover the manufacturer column from the Cars93 data frame. Whenever a GtkTreeIter is returned by a GtkTreeModel, the return value in R is a list of two components: retval, a logical indicating whether the iterator is valid, and iter, the pointer to the underlying C data structure. The call to get also returns a list, with an element for each column index passed as an argument. The method iterNext updates the passed iterator in place, i.e., by reference, to point to the next row. Thus, no new iterator is returned. This is unfamiliar behavior in R. Instead, the method returns a logical value indicating whether the iterator is still valid, i.e. FALSE is returned if no next row exists.

It is clear that the above usage is designed for languages like C, where multiple return values are conveniently passed by reference parameters. The iterator design also prevents the use of the apply functions, which are generally preferred over the `while` loop for reasons of performance and clarity. An improvement would be to obtain the number of children, generate the sequence of row indices and access the row for each index:

```
nrows <- model$iterNChildren(NULL)
manufacturer <- sapply(seq(nrows), function(i) {
  iter <- model$iterNthChild(NULL, i)
  model$get(iter$iter, 0)[[1]]
})
```

Here we use `NULL` to refer to the virtual root node that sits above the rows in our table. Unfortunately, this usage too is unintuitive and slow, so the benefits of `RGtkDataFrame` should be obvious.

One can convert between the two representations. The method `getIter` on `GtkTreeModel` returns an iterator for a path. A shortcut from the string representation of the path to an iterator is `getIterFromString`. The path pointed to by an iterator is returned by `getPath`.

One might note that `GtkTreeIter` is created and managed by the model, while `GtkTreePath` is model independent. It is not possible to use iterators across models or even across modifications to a model. After a model changes, an iterator is invalid. A tree path may still point to a valid row, though it will not in general be the same row from before the change. To refer to the same row across tree model changes, use a `GtkTreeRowReference`.

Selection

There are multiple modes of user interaction with a tree view: if the cells are not editable, then selection is the primary mode. A single click selects the value, and a double click is often used to initiate an action. If the cells are editable, then a double click or a click on an already selected row will initiate editing of the content. Editing of cell values is a complex topic and is handled by derivatives of `GtkCellRenderer`, see Section **??**. Here, we limit our discussion to selection of rows.

GTK+ provides the class `GtkTreeSelection` to manage row selection. Every tree view has a single instance of `GtkTreeSelection`, returned by the `getSelection` method.

The usage of the selection object depends on the selection mode, i.e., whether multiple rows may be selected. The mode is configured with the `setMode` method, with values from `GtkSelectionModel`, including `"multiple"` for allowing more than one row to be selected and `"single"` for limiting selections to a single row, or none.

When only a single selection is possible, the method `getSelected` returns the selected row as a list, with components `retval` to indicate success, `model` pointing to the tree model and `iter` representing an iterator to the selected row in the model.

```
model <- rGtkDataFrame(mtcars)
view <- gtkTreeView(model)
selection <- view$getSelection()
selection$setMode("single")
```

```
[1] 1
```

If this tree view is shown and a selection made, this code will return the value in the first column:

```
curSel <- selection$getSelected()     # retrieve selection
with(curSel, model$getValue(iter, 0)$value) # model, iter
```

```
[1] 21.4
```

When multiple selection is permitted, then the method `getSelectedRows` returns a list with component `model` pointing to the model, and `retval`, a list of tree paths.

We can change the selection mode as follows.

```
selection$setMode("multiple")
```

If a selection is made this code will print the selected values in the first column (we have selected the first three rows):

```
curSel <- selection$getSelectedRows()
if(length(curSel$retval)) {
  rows <- sapply(curSel$retval, gtkTreePathGetIndices) + 1L
  curSel$model[rows, 1]
}
```

```
[1] 21.0 22.8 21.4
```

To respond to a selection, connect to the `changed` signal on `GtkTreeSelection`. The signal itself does not contain any selection information; the selection object should be queried instead.

When a row is not editable, then the double-click event or a keyboard command triggers the `row-activated` signal for the tree view. The callback has arguments `tree.view` pointing to the widget that emits the signal, `path` storing a tree path of the selected row, and `column` containing the tree view column. The column number is not returned. If that is of interest, it can be passed in via the user data argument, or matched against the children of the tree view through a command like

```
sapply(view$getColumns(), function(i) i == column)
```

Sorting

A common GUI feature is sorting a table widget by column. By convention, the user clicks on the column header to toggle sorting. `GtkTreeView` supports this interaction, although the actual sorting occurs in the model. Any model that implements the `GtkTreeSortable` interface supports sorting. `RGtkDataFrame` falls into this category. When `GtkTreeView` is directly attached to a sortable model, it is only necessary to inform each view column of the model column to use for sorting when the header is clicked:

```
vc <- view$getColumn(0)
vc$setSortColumnId(0)
```

In the above, clicking on the header of the first view column, `vc`, will sort by the first model column. Behind the scenes, `GtkTreeViewColumn` will set its sort column as the sort column on the model, i.e.:

```
model$setSortColumnId(0, "ascending")
```

Some models, however, do not implement `GtkTreeSortable`, such as `GtkTreeModelFilter`, introduced in the next section. Also, sorting a model permanently changes the order of its rows, which may be undesirable in some cases. The solution is to proxy the original model with a sortable model. The proxy obtains all of its data from the original model and reorders the rows according to the order of the sort column. GTK+ provides `GtkTreeModelSort` for this:

```
store <- rGtkDataFrame(mtcars)
sorted <- gtkTreeModelSortNewWithModel(store)
view <- gtkTreeView(sorted)
view$insertColumnWithAttributes(0, "Click to sort", gtkCellRendererText(),
                                text = 0)
view$getColumn(0)$setSortColumnId(0)
```

When the user sorts the table, the underlying `store` will not be modified.

The default sorting function can be changed by calling the method `setSortFunc` on a sortable model. The following function shows how the default sorting might be implemented.

```
f <- function(model, iter1, iter2, user.data) {
  column <- user.data
  val1 <- model$getValue(iter1, column)$value
  val2 <- model$getValue(iter2, column)$value
  as.integer(val1 - val2)
}
sorted$setSortFunc(sort.column.id=0, sort.func=f, user.data=0)
```

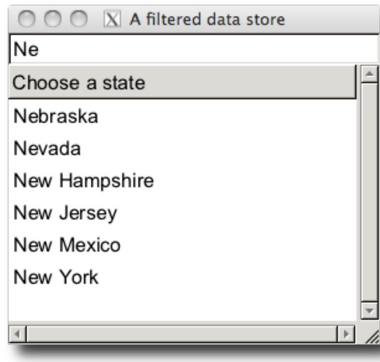

Figure 4.1: Example of a data store filtered by values typed into a text-entry widget.

Filtering

The previous section introduced the concept of a proxy model in `GtkTreeModelSort`. Another common application of proxying is filtering. For filtering via a proxy model, GTK+ provides the `GtkTreeModelFilter` class. The basic idea is that an extra column in the base model stores logical values to indicate if a row should be visible. The index of that column is passed to the filter model, which provides only those rows where the filter column is `TRUE`.

```
df <- data.frame(col=letters[1:3], vis=c(TRUE, TRUE, FALSE))
store <- rGtkDataFrame(df)
filtered <- store$filter()
filtered$setVisibleColumn(1)            # 0-based
view <- gtkTreeView(filtered)
```

The constructor of the filter model is `gtkTreeModelFilter`, which, somewhat coincidentally, also works as a method on the base model, i.e., `model$filter()`. To retrieve the original model from the filter, call its `getModel` method. The method `setVisibleColumn` specifies which column in the model holds the logical values. The configured filter model may now be treated as any other tree model, including attachment to a `GtkTreeView`.

Example 4.2: Using filtering
This example shows how to use `GtkTreeModelFilter` to filter rows according to whether they match a value entered into a text entry box. The end result is similar to an entry widget with completion.

First, we create a data frame. The `visible` column will be added to the `rGtkDataFrame` instance to adjust the visible rows.

```
df <- data.frame(state.name)
df$visible <- rep(TRUE, nrow(df))
```

```
store <- rGtkDataFrame(df)
```

The filtered store needs to have the column specified that contains the logical values; in this example, it is the last column.

```
filteredStore <- store$filter()
filteredStore$setVisibleColumn(ncol(df)-1)      # offset
view <- gtkTreeView(filteredStore)
```

Next, we create a basic view of a single column:

```
view$insertColumnWithAttributes(0, "Col",
                   gtkCellRendererText(), text = 0)
```

```
[1] 1
```

An entry widget will be used to control the filtering. In the callback, we adjust the `visible` column of the `rGtkDataFrame` instance to reflect the rows to be shown. When `val` is an empty string, the result of `grepl` is `TRUE`, so all rows will be shown.

```
e <- gtkEntry()
gSignalConnect(e, "changed", function(w, data) {
  pattern <- w$getText()
  df <- data$getModel()
  values <- df[, "state.name"]
  df[, "visible"] <- grepl(pattern, values)
}, data=filteredStore)
```

Figure **??** shows the two widgets placed within a simple GUI.

Cell renderer details

The values in a tree model are rendered in a rectangular cell by the derivatives of `GtkCellRenderer`. Cell renderers are interactive, in that they also manage editing and activation of cells.

A cell renderer is independent of any data model. Its rendering role is limited to drawing into a specified rectangular region according to its current property values. An object that implements the `GtkCellLayout` interface, like `GtkTreeViewColumn` and `GtkComboBox` (see Section **??**), associates a set of *attributes* with a cell renderer. An attribute is a link between an aesthetic property of a cell renderer and a column in the data model. When the `GtkCellLayout` object needs to render a particular cell, it configures the properties of the renderer with the values from the current model row, according to the attributes. Thus, the mapping from data to visualization depends on the class of the renderer instance, its explicit property settings, and the attributes associated with the renderer in the cell layout.

For example, to render text, a `GtkCellRendererText` is appropriate. The `text` property is usually linked via an attribute to a text column in the model, as the text would vary from row to row. However, the background color (the background property) might be common to all rows in the column and thus is set explicitly, without use of an attribute.

The base class GtkCellRenderer defines a number of properties that are common to all rendering tasks. The xalign and yalign properties specify the alignment, i.e., how to position the rendered region when it does not fill the entire cell. The cell-background property indicates the color for the entire cell background.

The rest of this section describes each type of cell renderer, as well as some advanced features.

Text cell renderers The gtkCellRendererText constructor is used to display text and numeric values. Numeric values in the model are shown as strings. The most important property is text, the actual text that is displayed. Other properties control the display of the text, such as the font family and size, the foreground and background colors, and whether to ellipsize or wrap the text if there is not enough space for display.

To display right-aligned text in a Helvetica font, the following could be used:

```
cr <- gtkCellRendererText()
cr['xalign'] <- 1                    # default 0.5 = centered
cr['family'] <- "Helvetica"
```

The wrap attribute can be specified as TRUE, if the entries are expected to be long. There are several other attributes that can changed.

When an attribute links the text property to a numeric column in the model, the property system automatically converts the number to its string representation. This occurs according to the same logic that R follows to print numeric values, so options like scipen and digits are considered. See the "Overriding attribute mappings" paragraph below for further customization.

Editable text renderers GtkCellRendererCombo and GtkCellRendererSpin allow editing a text cell with a combo box or spin button, respectively. Populating the combo box menu requires specifying two properties: model and text-column. The menu items are retrieved from the GtkTreeModel given by model at the column index given by text-column. If has-entry is TRUE, a combo box entry is displayed.

```
cr <- gtkCellRendererCombo()
store <- rGtkDataFrame(state.name)
cr['model'] <- store
cr['text-column'] <- 0
cr['editable'] <- TRUE               # needed
```

The spin button editor is configured by passing a GtkAdjustment to the adjustment property.

Pixbuf cell renderers To display an image in a cell, `GtkCellRendererPixbuf` is appropriate. The image is specified through one of its properties: `stock-id`, a stock identifier; `icon-name`, the name of a themed icon; or `pixbuf`, an actual `GdkPixbuf` object. It is possible to store a `GdkPixbuf` in a `data.frame`, and thus an `RGtkDataFrame`, using a `list`, as in the next example.

Example 4.3: A variable selection widget

This example will show how to create a means to select variables from a data frame using two tables. The left one indicating the variables that can be selected, the right the variables that have been selected. A similar mechanism is part of the SPSS model specification GUI of Figure **??**. For illustration purposes we use the `Cars93` data set.

```
df <- Cars93
```

To indicate to the user the type of variables we use an icon. The following function will create a pixmap holding an icon that depends on the class of the variable. The `make_icon` function from the `ProgGUIinR` package will be used to draw the icon.

```
source("~/GUI/ProgGUIInR/R/misc.R")     # for make_icon
make_icon_pixmap <- function(x, ...) {
  require(grid)
  require(cairoDevice)
  pixmap <- gdkPixmap(drawable=NULL, width=16, height=16, depth=24)
  asCairoDevice(pixmap)
  grid.newpage()
  grid.draw(make_icon(x))
  dev.off()
  gdkPixbufGetFromDrawable(NULL, pixmap, NULL, 0,0,0,0,-1,-1)
}
```

We will use a filtered tree model for our two tables. Here we define a data frame, a corresponding `rGtkDataFrame` object and from that a filtered tree model. The filtering is done at the model level – not the view level – by specifying a column in the model to indicate if a row should be shown. As such, we need two similar models.

```
mdf <- data.frame(Variables=sort(names(df)),
                  visible = rep(TRUE, ncol(df)),
                  stringsAsFactors=FALSE
                 )
mdf$pixmap <- sapply(names(df), function(i) make_icon_pixmap(df[,i, drop=TRUE]))
#
models <- list()
models[[1]] <- rGtkDataFrame(mdf)
mdf$visible <- !mdf$visible
models[[2]] <- rGtkDataFrame(mdf)
```

Here we create our filtered models and set the second column as the visible column.

```
filterModels <- sapply(models, function(i)
                       gtkTreeModelFilterNew(i))
sapply(filterModels, function(i) i$setVisibleColumn(1))
```

```
[[1]]
NULL

[[2]]
NULL
```

We now layout our GUI using a horizontal box. We pack in the views and a box container to hold the selecion buttons. The views will be scrollable, so we construct `GtkScrolledWindow` instances.

```
w <- gtkWindow(show=FALSE)
g <- gtkHBox()
w$add(g)
## scrollbars
scrollbars <- sapply(1:2, function(i) gtkScrolledWindow())
sapply(1:2, function(i) {
  scrollbars[[i]]['hscrollbar-policy'] <- "automatic"
  scrollbars[[i]]['vscrollbar-policy'] <- "automatic"
})
```

```
[1] "automatic" "automatic"
```

```
buttonBox <- gtkVBox();
#
g$packStart(scrollbars[[1]], expand=TRUE)
g$packStart(buttonBox, expand=FALSE)
g$packStart(scrollbars[[2]], expand=TRUE)
#
views <- sapply(1:2, function(i) gtkTreeView())
sapply(1:2, function(i) scrollbars[[i]]$add(views[[i]]))
```

```
[[1]]
NULL

[[2]]
NULL
```

Now we configure the views, first by setting the model and then the selection mode.

```
sapply(1:2, function(i) views[[i]]$setModel(filterModels[[i]]))
sapply(1:2, function(i)
       views[[i]]$getSelection()$setMode('multiple'))
```

We use a single column to display our data (a list box) but that column will hold both an icon and the text label. This is done by packing in two cell renderers below:

```
make_view_column <- function() {
  vc <- gtkTreeViewColumn()
  vc$setTitle("Variable")
  cr <- gtkCellRendererPixbuf()
  vc$packStart(cr)
  vc$addAttribute(cr, "pixbuf", 2)
  cr <- gtkCellRendererText()
  vc$packStart(cr)
  vc$addAttribute(cr, "text", 0)
  vc
}
sapply(views, function(view)
  view$insertColumn(make_view_column(), 0))
```

For later use we extend the API for a tree view – one method to find the selected indices (1-based) and one to indicate if there is a selection.

```
gtkTreeViewSelectedIndices <- function(object) {
  paths <- object$getSelection()$getSelectedRows()$retval
  out <- sapply(paths, function(i) {
    model <- object$getModel()          # Filtered!
    model$ConvertPathToChildPath(i)$toString()
  })
  if(length(out) == 0)
    integer(0)
  else
    as.numeric(out) + 1                               # 1-based
}
#
gtkTreeViewHasSelection <- function(obj) length(obj$selectedIndices()) > 0
```

Now we create the buttons and add in a callback for the `clicked` signal that moves the selected values to the other list. This is simply by adjusting the Boolean value that instructs the filtered view to show a row or not.

```
buttons <- lapply(c(">", "<"), gtkButton)
#
move <- function(i, ...) {
  ind <- views[[i]]$selectedIndices()
  models[[3-i]][ind,2] <- TRUE
  models[[i]][ind,2] <- FALSE
}
sapply(1:2, function(i)
  gSignalConnect(buttons[[i]], "clicked", move, data=i, user.data.first=TRUE))
```

We only want our buttons sensitive if there is a possible move. This is determined by the presence of a selection.

```
sapply(buttons, gtkWidgetSetSensitive, FALSE)
sapply(1:2, function(i)
       gSignalConnect(views[[i]]$getSelection(), "changed", function(...)
                      buttons[[i]]['sensitive'] <- views[[i]]$hasSelection()))
```

Finally we add the buttons using some extra boxes to center them vertically and call the `show` method of the toplevel window.

```
buttonBox$packStart(gtkVBox(), expand=TRUE) # align in center
sapply(buttons, gtkBoxPackStart, object=buttonBox, expand=FALSE, padding=6)
buttonBox$packStart(gtkVBox(), expand=TRUE)
#
w$show()
```

Toggle cell renderers Binary data can be represented by a toggle. The `gtkCellRendererToggle` will create a check box in the cell that will appear checked if the `active` property is `TRUE`. If an attribute is defined for the property, then changes in the model will be reflected in the view. More work is required to modify the model in response to user interaction with the view. The `activatable` attribute for the cell must be `TRUE` in order for it to receive user input. The programmer then needs to connect to the `toggled` to update the model in response to changes in the active state.

```
cr <- gtkCellRendererToggle()
cr['activatable'] <- TRUE              # cell can be activated
cr['active'] <- TRUE
gSignalConnect(cr, "toggled", function(w, path) {
  model$active[as.numeric(path) + 1] <- w['active']
})
```

To render the toggle as a radio button instead of a check box, set the `radio` property to `TRUE`. Again, the programmer is responsible for implementing the radio button logic via the `toggled` signal.

Example 4.4: Displaying a check box column in a tree view
This example demonstrates the construction of a GUI for selecting one or more rows from a data frame. We will display a list of the installed packages that can be upgraded from CRAN, although this code is trivially generalizable to any list of choices. The user selects a row by clicking on a check box produced by a toggle cell renderer.

To get the installed packages that can be upgraded, we use some of the functions provided by the `utils` package.

```
d <- old.packages()[,c("Package", "Installed", "ReposVer")]
d <- as.data.frame(d)
```

This function will be called on the selected rows. Here, we simply call `install.packages` to update the selected packages.

```
doUpdate <- function(d)  install.packages(d$Package)
```

To display the data frame, we first append a column to the data frame to store the selection information and then create a corresponding `RGtkDataFrame`.

```
n <- ncol(d)
nms <- colnames(d)
d$.toggle <- rep(FALSE, nrow(d))
store <- rGtkDataFrame(d)
```

Our tree view shows each text column using a simple text cell renderer, except for the first column that contains the check boxes for selection.

```
view <- gtkTreeView()
# add toggle
cr <- gtkCellRendererToggle()
view$insertColumnWithAttributes(0, "", cr, active = n)
cr['activatable'] <- TRUE
gSignalConnect(cr, "toggled", function(cr, path, user.data) {
  view <- user.data
  row <- as.numeric(path) + 1
  model <- view$getModel()
  n <- dim(model)[2]
  model[row, n] <- !model[row, n]
}, data=view)
```

The text columns are added one-by-one in a similar manner:

```
mapply(view$insertColumnWithAttributes, -1, nms,
       list(gtkCellRendererText()), text = 1:n-1)
```

Finally, we connect the store to the model.

```
view$setModel(store)
```

To allow the user to initiate the action, we create a button and assign a callback. We pass in the view, rather than the model, in case the model would be recreated by the `doUpdate` call. In a real application, once a package is upgraded it would be removed from the display.

```
b <- gtkButton("Update packages")
gSignalConnect(b, "clicked", function(w, data) {
  view <- data
  model <- view$getModel()
  n <- dim(model)[2]
  vals <- model[model[, n], -n, drop=FALSE]
  doUpdate(vals)
}, data=view)
```

Our basic GUI places the view into a box container that also holds the button to initiate the action.

```
w <- gtkWindow(show=FALSE)
w$setTitle("Installed packages that need upgrading")
w$setSizeRequest(300, 300)
g <- gtkVBox(); w$add(g)
sw <- gtkScrolledWindow()
g$packStart(sw, expand=TRUE, fill=TRUE)
sw$add(view)
sw$setPolicy("automatic", "automatic")
g$packStart(b, expand=FALSE)
w$show()
```

Rendering progress in cells To visually communicate progress within a cell, both progress bars and spinner animations are supported. These modes correspond to `GtkCellRendererProgress` and `GtkCellRendererSpinner`, respectively.

In the case of `GtkCellRendererProgress`, its `value` property takes a value between 0 and 100 indicating the amount finished, with a default value of 0. Values out of this range will be signaled by an error message. The `orientation` property, with values from `GtkProgressBarOrientation`, can adjust the direction that the bar grows. For example,

```
cr <- gtkCellRendererProgress()
cr["value"] <- 50                          # fixed 50%
cr['orientation'] <- "right-to-left"
```

For indicating progress in the absence of a definite end point, `GtkCellRendererSpinner` is more appropriate. The spinner is displayed when the `active` property is `TRUE`. Increment the `pulse` property to drive the animation.

Overriding attribute mappings The default behavior for mapping model values to a renderer property is simple: values are extracted from the model and passed directly to the cell renderer property. If the data types are different, such as a numeric value for a string property, the value is converted using low-level routines defined by the property system. It is sometimes desirable to override this mapping with more complex logic.

For example, conversion of numbers to strings is a non-trivial task. Although the logic in the R print system often performs acceptably, there is certainly room for customization, for example aligning floating point numbers by fixing the number of decimal places. This could be done in the model (e.g., using `sprintf` to format and coerce to character data). However, performing the conversion during rendering requires one to to intercept the model value before it is passed to the cell renderer. In the specific case `GtkTreeView`, it is possible to specify a callback that overrides this step.

The callback, of type GtkTreeCellDataFunc, accepts arguments for the tree view column, the cell renderer, the model, an iterator pointing to the row in the model and an argument for user data. The function is tasked with setting the appropriate attributes of the cell renderer. For example, this function could be used to format floating point numbers:

```
func <- function(viewCol, cellRend, model, iter, data) {
  curVal <- model$getValue(iter, 0)$value
  fVal <- sprintf("%.3f", curVal)
  cellRend['text'] <- fVal
  cellRend['xalign'] <- 1
}
```

The function then needs to be registered with a GtkTreeViewColumn that is rendering a numeric column from the model:

```
view <- gtkTreeView(rGtkDataFrame(data.frame(rnorm(100))))
cr <- gtkCellRendererText()
view$insertColumnWithAttributes(0, "numbers", cr, text = 0)
vc <- view$getColumn(0)
vc$setCellDataFunc(cr, func)
```

The last line is the key: calling setCellDataFunc registers our custom formatting function with the view column.

One drawback with the use of such functions is that R code is executed every time a cell is rendered. If performance matters, consider pre-converting the data in the model or tweaking the options in R for printing real numbers, namely scipen and digits.

For customizing rendering further and outside the scope of GtkTreeView, one could implement a new type of GtkCellRenderer. See Section **??** for more details on this advanced concept.

Editable cells When the editable property of a text cell (or activatable property of a toggle cell) is set to TRUE, then the cell contents can be changed. This allows the user to make changes to the underlying model through the GUI. Although the view automatically reflects changes made to the model, the reverse is not true. A callback must be assigned to the editable (toggled) signal for the cell renderer to implement the change. The callback for the "edited" signal has arguments renderer, path for the path of the selected row (as a string), and new.text containing the value of the edited text as a string. The tree view object and which column was edited are not passed in by default. These can be passed through the user data argument, set as user data on the renderer, or accessed from an enclosing environment if needed within the callback.

For example, here is how one can update an RGtkDataFrame model from within the callback.

```
cr['editable'] <- TRUE
ID <- gSignalConnect(cr, "edited",
f=function(cr, path, newtext, user.data) {
  curRow <- as.numeric(path) + 1
  curCol <- user.data$column
  model <- user.data$model
  model[curRow, curCol] <- newtext
}, data=list(model=store, column=1))
```

Moving the cursor Users may expect that once a cell is edited, the next cell is then set up to be edited. In order to do this, one must advance the cursor and activate editing of the next cell. For `GtkTreeView`, this is done through the `setCursor` method. The `path` argument takes a tree path instance, the `column` argument a tree view column object, and the flag `start.editing` indicates whether to initiate editing.

4.2 Display of hierarchical data

Although the `RGtkDataFrame` model is a convenient implementation of `GtkTreeModel`, it has its limitations. Primary among them is its lack of support for hierarchical data. GTK+ implements `GtkTreeModel` with `GtkListStore` and `GtkTreeStore`, which respectively store non-hierarchical and hierarchical tabular data. `GtkListStore` is a flat table, while `GtkTreeStore` organizes the table into a hierarchy. Here, we discuss `GtkTreeStore`.

Loading hierarchical data

A tree store is constructed using `gtkTreeStore`. The column types are specified through a character vector at the time of construction. The specification uses "GTypes" such as `gchararray` for character data, `gboolean` for logical data, `gint` for integer data, `gdouble` for numeric data, and `GObject` for GTK+ objects, such as pixbufs.

Example 4.5: Defining a tree
Below, we create a tree based on the `Cars93` dataset, where the car models are organized by manufacturer, i.e., each model row is the child of its manufacturer row:

```
tstore <- gtkTreeStore("gchararray")
by(Cars93, Cars93$Manufacturer, function(df) {
  piter <- tstore$append()              # parent
  tstore$setValue(piter$iter, column = 0, value = df$Manufacturer[1])
  sapply(df$Model, function(model) {
    sibiter <- tstore$append(parent = piter$iter) # child
    if (is.null(sibiter$retval))
```

```
      tstore$setValue(sibiter$iter, column = 0, value = model)
  })
})
```

To retrieve a value from the tree store using its path we have:

```
iter <- tstore$getIterFromString("0:0")
tstore$getValue(iter$iter, column = 0)$value
```

```
[1] "Integra"
```

This obtains the first model from the first manufacturer.

As shown in this example, populating a tree store relies on two functions: `append`, for appending rows, and `setValue`, for setting row values. The iterator to the parent row is passed to `append`. A parent of `NULL`, the default, indicates that the row should be at the top level. It would also be possible to insert rows using `insert`, `insertBefore`, or `insertAfter`. The `setValue` method expects the row iterator and the column index, 0-based.

An entire row can be assigned through the `set` method. The method uses positional arguments to specify the column and the value. The column index appears as an even argument (say $2k$) and the corresponding value in the odd argument (say $2k + 1$). Values are returned by the `getValue` method, in a list with component `value` storing the value.

Traversing a tree store is most easily achieved through the use of `GtkTreeIter`, introduced previously in the context of flat tables. Here we perform a depth-first traversal of our `Cars93` model to obtain the model values:

```
iter <- tstore$getIterFirst()
models <- NULL
while(iter$retval) {
  child <- tstore$iterChildren(iter$iter)
  while(child$retval) {
    models <- c(models, tstore$get(child$iter, 0)[[1]])
  }
  tstore$iterNext(iter$iter)
}
```

The hierarchical structure introduces the method `iterChildren` for obtaining an iterator to the first child of a row. As with other methods returning iterators, the return value is a list, with the `retval` component indicating the validity of the iterator, stored in the `iter` component. The method `iterParent` performs the reverse, iterating from child to parent.

Rows within a store can be rearranged using several methods. Call `swap` to swap rows referenced by their iterators. The methods `moveAfter` and `moveBefore` move one row after or before another, respectively. The `reorder` method totally reorders the rows under a specified parent given a vector of row indices, like that returned by `order`.

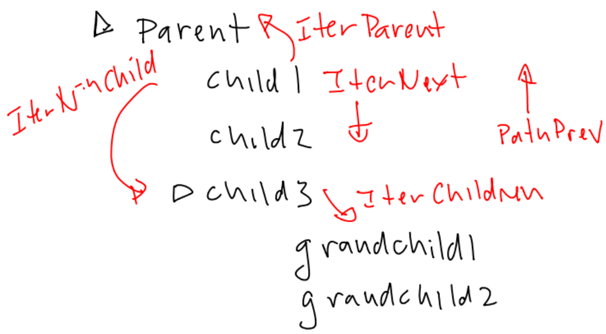

Figure 4.2: [REPLACEME!] Graphical illustration of the functions used by iters to traverse a tree store.

Once added, rows may be removed using the `remove` method. To remove every row, call the `clear` method.

Displaying data as a tree

Once a hierarchical dataset has been loaded into a `GtkTreeModel` implementation like `GtkTreeStore`, it can be passed to a `GtkTreeView` widget for display as a tree. Indeed, this is the same widget that displayed our flat data frame in the previous section. As before, `GtkTreeView` displays the `GtkTreeModel` as a table; however, it now adds controls for expanding and collapsing nodes where rows are nested.

The user can click to expand or collapse a part of the tree. These actions correspond to the signals `row-expanded` and `row-collapsed`, respectively.

Example 4.6: A simple tree display
Here, we demonstrate the application of `GtkTreeView` to the display of hierarchical data. We will use the model constructed in Example **??** from the `Cars93` dataset. In that example we defined a simple tree store from a data frame, with a level for manufacturer and make for different cars. We refer to that model by `tstore` below.

Now, we make a simple rectangular store for the make information with the following:

```
store <- rGtkDataFrame(Cars93[,"Model", drop=FALSE])
```

The basic view is similar to that for rectangular data already presented.

```
view <- gtkTreeView()
view$insertColumnWithAttributes(0, "Make",
           gtkCellRendererText(), text = 0)
```

```
[1] 1
```

Finally, we illustrate that the same view can be used with either model:

```
view$setModel(store)                    # the rectangular store
view$setModel(tstore)                   # or the tree store
```

Example 4.7: Dynamically growing a tree

This example uses a tree to explore an R list object, such as that returned by one of R's modelling functions. As the depth of these lists is not specified in advance, we use a dynamic approach to creating the tree store, modifying the tree store when the tree view is expanded or collapsed.

We begin by defining our tree store and an accompanying tree view. This example allows sorting, and so calls the `gtkTreeModelSort` function.

```
store <- gtkTreeStore(rep("gchararray", 2))
sstore <- gtkTreeModelSort(store)
```

We create a root row:

```
iter <- store$append(parent=NULL)$iter
store$setValue(iter, column=0, "GlobalEnv")
store$setValue(iter, column=1, "environment")
iter <- store$append(parent=iter)
```

It is necessary to append an empty row to the root so that root becomes expandable.

We now define the tree view and allow for multiple selection:

```
view <- gtkTreeView(sstore)
view$getSelection()$setMode("multiple")
```

The basic idea is to create child nodes when the parent is expanded and to delete the children when the parent is collapsed. This relies on the `row-expanded` and `row-collapsed` signals, respectively. First, we define the expansion handler:

```
gSignalConnect(view, signal = "row-expanded",
               f = function(view, iter, tpath, user.data) {
                 sortedModel <- view$getModel()
                 iter <- pathToIter(sortedModel, tpath)
                 path <- iterToRPath(sortedModel, iter)
                 children <- getChildren(path)
                 addChildren(store, children, parentIter=iter)
                 ## remove errant 1st offspring. See addChildren
                 ci <- store$iterChildren(iter)
                 if(ci$retval) store$remove(ci$iter)
               })
```

The callback calls several helper functions to map the tree path to an R object, get the child components of the object and add them to the tree. The details are in the definitions of the helper functions.

The pathToIter function finds the iterator in the base tree model for a tree path in the sorted proxy.

```
pathToIter <- function(sstore, tpath) {
  store <- sstore$getModel()
  uspath <- sstore$convertPathToChildPath(tpath)
  store$getIter(uspath)$iter
}
```

We now need to convert the iterator to an "R path," which is made up of the names of each component that makes up an element in the list. This function returns the path for a component specified by its iterator.

```
iterToRPath <- function(sstore, iter) {
  store <- sstore$getModel()
  indices <- store$getPath(iter)$getIndices()
  iter <- NULL
  path <- sapply(indices, function(i) {
    iter <<- store$iterNthChild(iter, i)$iter
    store$getValue(iter, 0)$value
  })
  return(path[-1])
}
```

The getChildren function obtains the child components of a given R object path. If the path is empty, the children are the objects in the global environment, the root. The return value is a data.frame with three columns: object name, object class and whether the object is recursive.

```
getChildren <- function(path=character(0)) {
  hasChildren <- function(obj)
    (is.list(obj) || is.environment(obj)) && !is.null(names(as.list(obj)))

  getType <- function(obj) head(class(obj), n=1)

  obj <-
    if (!length(path)) {
      .GlobalEnv
    } else {
      x <- get(path[1], envir=.GlobalEnv)
      if(length(path) > 1)
        get(path[1], envir=.GlobalEnv)[[path[-1]]]
      else
        x
    }
```

```
  children <- as.list(obj)

  d <- data.frame(children = names(children),
                  class = sapply(children, getType),
                  offspring = sapply(children, hasChildren))

  ## filter out Gtk ones
  d[!grepl("^Gtk", d$class), ]
}
```

The final step in the expansion handler is to add the children to the tree store with the `addChildren` function. Its one quirk is the addition of a dummy child row when the item has children. This makes the node expandable, i.e., the tree view draws an icon for the user to click to request the expansion.

```
addChildren <- function(store, children, parentIter = NULL) {
  if(nrow(children) == 0)
    return(NULL)
  for(i in 1:nrow(children)) {
    iter <- store$append(parent=parentIter)$iter
    sapply(1:(ncol(children) - 1), function(j)
           store$setValue(iter, column = j - 1, children[i, j]))
    ## Add a branch if there are children
    if(children[i, "offspring"])
      store$append(parent=iter)
  }
}
```

Next, we define a handler for the `row-collapsed` signal, which has a similar signature as the `row-expanded` signal. The handler removes the children of the newly collapsed node, so that we can add them again when the node is expanded.

```
gSignalConnect(view, signal = "row-collapsed",
               f = function(view, iter, tpath, user.data) {
                 sortedModel <- view$getModel()
                 iter <- pathToIter(sortedModel, tpath)
                 n = store$iterNChildren(iter)
                 if(n > 1) { ## n=1 gets removed when expanded
                   for(i in 1:(n-1)) {
                     child.iter <- store$iterChildren(iter)
                     if(child.iter$retval)
                       store$remove(child.iter$iter)
                   }
                 }
               })
```

Our last handler simply demonstrates the retrieval of an object when its row is activated, i.e., double-clicked:

```
gSignalConnect(view, signal = "row-activated",
               f = function(view, tpath, tcol) {
                 sortedModel <- view$getModel()
                 iter <- pathToIter(sortedModel, tpath)
                 path <- iterToRPath(sortedModel, iter)
                 sel <- view$getSelection()
                 out <- sel$getSelectedRows()
                 if(length(out) == 0) return(c()) # nothing
                 vals <- c()
                 for(i in out$retval) {  # multiple selections
                   iter <- sortedModel$getIter(i)$iter
                   newValue <- sortedModel$getValue(iter, 0)$value
                   vals <- c(vals, newValue)
                 }
                 print(vals)  # [Replace this]
               })
```

To finish this example, we would need to populate the tree view with columns and display the view in a window.

4.3 Model-based combo boxes

Basic combo box usage was discussed in Section **??**; here we discuss the more flexible and complex approach of using an explicit data model for storing the menu items. The item data is tabular, although it is limited to a single column. Thus, `GtkTreeModel` is again the appropriate model, and `RGtkDataFrame` is usually the implementation of choice.

To construct a `GtkComboBox` based on a user-created model, one should pass the model to the constructor `gtkComboBox`. This model may be changed or set through the `setModel` method and is returned by `getModel`. It remains to instruct the combo box how to display one or more data columns in the menu. Like `GtkTreeViewColumn`, `GtkComboBox` implements the `GtkCellLayout` interface and thus delegates the rendering of model values to `GtkCellRenderer` instances that are packed into the combo box.

As introduced in the previous chapter, the `GtkComboBoxEntry` widget extends `GtkComboBox` to provide an entry widget for the user to enter arbitrary values. To construct a combo box entry on top of a tree model, one should pass the model, as well as the column index that holds the textual item labels, to the `gtkComboBoxEntry` constructor. It is not necessary to create a cell renderer for displaying the text, as the entry depends on having text labels and thus enforces their display. It is still possible, of course, to add cell renderers for other model columns.

The `getActiveIter` returns a list containing the iterator pointing to the currently selected row in the model. If no row has been selected, the `retval` component of the list is `FALSE`. The `setActiveIter` sets the currently selected item by iterator. As discussed previously, the getActive and setActive behave analogously with 0-based indices.

Signals When a user selects a value with the mouse, the changed signal is emitted. For combo box entry widgets, the changed signal will also be emitted when a new value has been entered. also make changes by typing in the new value. To detect when the user has finished entering text, one needs to retrieve the underlying GtkEntry widget with getChild and connect to its activate signal.

Example 4.8: A combo box with memory

This example uses an editable combobox as an simple interface to R's help system. We record the number of times a page is searched for.

Our model fo the combobox will be a rGtkDataFrame instance where 3 variables are kept a function name, a string describing the number of visits and an integer to record the number of visits. We create the combobox with this model using the first column for the text.

```
m <- rGtkDataFrame(data.frame(fname="", visits="", novisits=0,
                              stringsAsFactors=FALSE))
cb <- gtkComboBoxEntryNewWithModel(m, text.column=0)
```

It isn't possible to put tooltip information on the dropdown elements of a combobox, rather here we borrow from popular web browser interfaces and add to the standard appearance of the drop down elements by adding in textual information about the number of visits. To do so we pack in a new cell renderer to accompany that provided by the gtkComboBoxEntry widget.

```
cr <- gtkCellRendererText()
cb$packStart(cr)
cb$addAttribute(cr, "text", 1)
cr['foreground'] <- "gray50"
cr['ellipsize'] <- "end"
cr['style'] <- "italic"
cr['alignment'] <- "right"
```

We place this in a minimal GUI with just a label to inicate what is to be done.

```
w <- gtkWindow(show=FALSE)
w['border-width'] <- 15
g <- gtkHBox(); w$add(g)
g$packStart(gtkLabel("Help on:"))
g$packStart(cb, expand=TRUE, fill=TRUE)
#
w$show()
```

Now we configure the combobox. When a new value is typed in we need to add it to the model. There are a few things to watch out for here. First, if we define the model to have 0 rows, then the `appendRows` call will coerce the characters to factors. To avoid that, we check to see if we are in the initial state in which case we replace the first row. If not, then we update the model. In GTK+ there is no means to notify the views of a model of a change of dimension, and in this case appending rows will cause the combobox to complain. Here we disconnect the model, update it, then reconnect it to the view.

```
addValue <- function(cb, value) {
  model <- cb$getModel()
  if(nrow(m) == 1 && m[1,1] == "") {
    m[1,1] <- value; m[1,2] <- ""; m[1,3] <- 0
  } else {                              # not first time
    cb$setModel(NULL)
    model$AppendRows(list(fname=value, visits="", novisits=0))
    cb$setModel(model)
  }
}
```

This helper function will be called each time a help page is requested. It first updates the visit information, selects the text for easier editing the next time round, then calls `help`.

```
callHelpFunction <- function(cb, value) {
  model <- cb$getModel()
  ind <- match(value,model[,1, drop=TRUE])
  n <- model[ind, "novisits"] <- model[ind, "novisits"] + 1
  model[ind, "visits"] <- sprintf(ngettext(n, "%s visit", "%s visits"), n)
  ## select for easier editing
  cb$getChild()$selectRegion(start=0,end=-1)
  help(value)
}
```

We connect to two signals emitted by the combobox. First when a selection is made (and `getActive` returns a non-negative value) we call the help function.

```
gSignalConnect(cb, "changed", f=function(w, ...) {
  if(cb$getActive() >= 0) {
    val <- w$getActiveText()
    callHelpFunction(w, val)
  }
})
```

The `activate` callback checks to see if the requested value has not been visited previosly. If not, this calls our `addValue` function before calling the help function.

```
gSignalConnect(cb$getChild(), "activate", f = function(cb, entry, ...) {
  val <- entry$getText()
  if(!any(val == cb$getModel()[,1, drop=TRUE])) {
    addValue(cb, val)                    # a new one
  }
  callHelpFunction(cb, val)
}, data=cb, user.data.first=TRUE)
```

4.4 Text entry widgets with completion

Often, the number of possible choices is too large to list in a combo box. One example is a web-based search engine: the possible search terms, while known and finite in number, are too numerous to list. The auto-completing text entry has emerged as an alternative to a combo box and might be described as a sort of dynamic combo box entry widget. When a user enters a string, partial matches to the string are displayed in a menu that drops down from the entry.

The `GtkEntryCompletion` object implements text completion in GTK+. An instance is constructed with `gtkEntryCompletion`. The underlying database is a `GtkTreeModel`, like `RGtkDataFrame`, set via the `setModel` method. To connect a `GtkEntryCompletion` to an actual `GtkEntry` widget, call the `setCompletion` method on `GtkEntry`. The `text-column` property specifies the column containing the completion candidates.

There are several properties that can be adjusted to tailor the completion feature; we mention some of them. Setting the property `inline-selection` to `TRUE` will place the top completion suggestion to the entry inline as the completions are scrolled through; `inline-completion` will add the common prefix automatically to the entry widget; `popup-single-match` is a logical indicating if a popup is displayed on a single match; `minimum-key-length` takes an integer specifying the number of characters needed in the entry before completion is checked (the default is 1).

By default, the rows in the data model that match the current value of the entry widget in a case insensitive manner are displayed. This matching function can be overridden by setting a new R function through the `setMatchFunc` method. The signature of this function is the completion object, the string from the entry widget (lower case), an iterator pointing to a row in the model and optionally user data that is passed through the `func.data` argument of the `setMatchFunc` method. This callback should return `TRUE` or `FALSE` depending on whether that row should be displayed in the set of completions.

Example 4.9: Text entry with completion
This example illustrates the steps to add completion to a text entry.

The two basic widgets are defined as follows:

```
entry <- gtkEntry()
completion <- gtkEntryCompletion()
entry$setCompletion(completion)
```

We will use an `RGtkDataFrame` instance for our completion model, taking a convenient list of names for our example. We set the model and text column index on the completion object and then set some properties to customize how the completion is handled.

```
store <- rGtkDataFrame(state.name)
completion$setModel(store)
completion$setTextColumn(0)
completion['inline-completion'] <- TRUE
completion['popup-single-match'] <- FALSE
```

We wish for the text search to match against any part of a string, not only the beginning, so define our own match function. We get the string from the entry widget, not the passed value, as that has been standardized to lower case.

```
matchAnywhere <- function(comp, str, iter, user.data) {
  model <- comp$getModel()
  rowVal <- model$getValue(iter, 0)$value   # column 0 in model

  str <- comp$getEntry()$getText()      # case sensitive
  grepl(str, rowVal)
}
completion$setMatchFunc(matchAnywhere)
```

4.5 Text views and text buffers

Multiline text areas are displayed through `GtkTextView` instances. These provide a view of an accompanying `GtkTextBuffer`, which is the model that stores the text and other objects to be rendered. The view is responsible for the display of the text in the buffer and has methods for adjusting tabs, margins, indenting, etc. The text buffer stores the actual text, and its methods are for adding and manipulating the text.

A text view is created with `gtkTextView`. The underlying text buffer can be passed to the constructor. Otherwise, a buffer is automatically created. This buffer is returned by the method `getBuffer` and may be set with the `setBuffer` method. Text views provide native scrolling support and thus are easily added to a scrolled window (Section **??**).

Example 4.10: Basic `gtkTextView` usage
The steps to construct a text view consist of:

```
w <- gtkWindow()
```

```
w['border-width'] <- 15
#
tv <- gtkTextView()
sw <- gtkScrolledWindow()
sw['hscrollbar-policy'] <- "automatic"
sw['vscrollbar-policy'] <- "automatic"
#
w$add(sw)
```

To set all the text in the buffer requires accessing the underlying buffer:

```
buffer <- tv$getBuffer()
buffer$setText("Lorem ipsum dolor sit amet ...")
```

Manipulating the text requires an understanding of how positions are referred to within the buffer (iterators or marks). As an indicator, to get the contents of the buffer may be done as follows:

```
start <- buffer$getStartIter()$iter
end <- buffer$getEndIter()$iter
buffer$getText(start, end)
```

```
[1] "Lorem ipsum dolor sit amet ..."
```

Text may be added programmatically through various methods of the text buffer. The most basic `setText`, which simply replaces the current text, is shown in the example above. The method `insertAtCursor` will add the text to the buffer at the current position of the cursor. Other means are described in the following sections.

`GtkTextBuffer` Details

Text buffer properties include `text` for the stored text and `has-selection` to indicate if text is currently selected in a view. The buffer also tracks if it has been modified. This information is available through the buffer's `getModified` method, which returns `TRUE` if the buffer has changes. The method `setModifed`, if given a value of `FALSE`, allows the programmer to change this state, say after saving a buffer's contents.

In order to do more with a text buffer, such as retrieve a selection, or modify attributes of just some of the text, one needs to become familiar with how pieces of the buffer are referred to within GTK+.

There are two methods: text iterators (iters) are a transient means to mark begin and end boundaries within a buffer, whereas text marks specify a location that remains when a buffer is modified. One can use these with tags to modify attributes of pieces of the buffer.

Iterators An *iterator* is a programming object used to traverse through some data, such as a text buffer or table of values. Iterators are typically transient.

They have methods to indicate what they point to and often update these values without an explicit function call. Such behavior is unusual for typical R programming.

In GTK+ a *text iterator* is used to specify a position in a buffer. In RGtk2, iterators are stored as lists with components `iter` to hold a pointer to the underlying iterator and component `retval` to indicate whether the iterator when it was returned is valid. In general, iterators become invalid as soon as a buffer changes, say through the addition of text, however many methods of the text buffer will update the iterator in place. This can happen inside a function call where the iterator is passed as an argument. The `copy` method will create a copy of an iterator, in case one is to be modified but it is important to keep the original.

Several methods of the text buffer return iterators marking positions in the buffer. The beginning and end of the buffer are returned by the methods `getStartIter` and `getEndIter`. Both of these iterators are returned at once by the method `getBounds` again as components of a list, in this case `start` and `end`. The current selection is returned by the method method `getSelectionBounds`. Again, as a list of iterators specifying the start and end positions of the current selection. If there is no selection, then the component `retval` will be `FALSE`, otherwise it is `TRUE`.

The method `getIterAtLine` will return an iterator pointing to the start of the line, which is specified by 0-based line number. The method `getIterAtLineOffset` has an additional argument to specify the offset for a given line. An offset counts the number of individual characters and keeps track of the fact that the text encoding, UTF-8, may use more than one byte per character. In addition to the text buffer, a text view also has the method `getIterAtLocation` to return the iterator indicating the between-word space in the buffer closest to the point specified in x-y coordinates.

There are several methods for iterators that allow one to refer to positions in the buffer relative to the iterator, for example, some with obvious names move by character or characters (`forwardChar`, `forwardChars`, `backwardChar`, and `backwordChars`), by word (`forwardWordEnd`, `backwardWordStart`), or by sentence (`backwardSentenceStart` and `forwardSentenceEnd`). There are also various methods, such as `insideWord`, returning logical values indicating if the condition is met. To use these methods, the iterator in the `iter` component is used, not the value returned as a list. Example **??** shows how some of the above are used, in particular how these methods update the iterator rather than return a new one.

Modifying the buffer Iterators are specified as arguments to several methods to set and retrieve text. The `insert` method will insert text at a specified iterator. The argument `len` specifies how many bytes of the `text` argument are to be inserted. The default value of -1 will insert the entire text. This method, by default, will also update the iterator to indicate the end of where the text is inserted. The `delete` method will delete the text between the iterators specified to the arguments `start` and `start`. The `getText` method will get the text between the specified `start` and `end` iters. A similar method `getSlice` will also do this, only it includes offsets to indicate the presence of images and widgets in the text buffer.

Example 4.11: Finding the word one clicks on
This example shows how one can find the iterator corresponding to a mouse-button-press event. The callback has an event argument which is a `GdkEventButton` object with methods `getX` and `getY` to extract the `x` and `y` components of the event object. These give the position relative to the widget. [1]

```
ID <- gSignalConnect(tv, "button-press-event", f=function(w, e, ...) {
  siter <- w$getIterAtLocation(e$getX(), e$getY())$iter
  niter <- siter$copy()                    # need copy
  siter$backwardWordStart()
  niter$forwardWordEnd()
  val <- w$getBuffer()$getText(siter, niter)
  print(val)                               # replace
  return(FALSE)                            # call next handler
})
```

Marks In addition to iterators, GTK+ provides marks to indicate positions in the buffer that persist through changes. For instance, the mark `"insert"` always refers to the position of the cursor. Marks have a gravity of `"left"` or `"right"`, with `"right"` being the default. When the text surrounding a mark is deleted, if the gravity is `"right"` the mark will remain to the right af any added text.

Marks can be defined in two steps by calling `gtkTextMark`, specifying a name and a value for the gravity, and then positioned within a buffer, specified by an iterator, through the buffer's `addMark` method. The `createMark` method combines the two steps.

There are many text buffer methods to work with marks. The `getMark` method will return the mark object for a given name. (There are functions which refer to the name of a mark, and others requiring the mark object.) The method `getIterAtMark` will return an iterator for the given mark to be used when an iterator is needed.

Tags Marks and iterators can be used to specify different properties for different parts of the text buffer. GTK+ uses tags to specify how pieces of text will differ from those of the textview overall. To create a tag, the `createTag`

[1]The methods `getXRoot` and `getYRoot` give the position relative to the parent window the widget resides in.

method is used. This has optional argument `tag.name` which can be used to refer to the tag later, and otherwise uses named arguments so specify a properties names and the corresponding values. These tags may be applied to the text between two iters using the methods `applyTag` or `applyTagByName`.

Example 4.12: Using text tags
We define two text tags to make text bold or italic and illustrate how to apply them.

```
tv <- gtkTextView()
tb <- tv$getBuffer()
tb$setText("The quick brown fox jumped over the lazy dog.")
##
tag.b <- tb$createTag(tag.name="bold",
                      weight=PangoWeight["bold"])
tag.em <- tb$createTag(tag.name="em",
                       style=PangoStyle["italic"])
tag.large <- tb$createTag(tag.name="large",
                          font="Serif normal 18")
##
iter <- tb$getBounds()         # or get iters another way
tb$applyTag(tag.b, iter$start, iter$end)  # updates iters
tb$applyTagByName("em", iter$start, iter$end)
```

Interacting with the clipboard GTK+ can create clipboards and provides convenient access to the default clipboard so that the standard cut, copy and paste actions can be implemented. The function `gtkClipboardGet` returns the default clipboard if given no arguments. The clipboard is the lone argument for the method `copyCliboard` to copy the current selection to the clipboard. The method `cutClipboard` has an extra argument, `default.editable`, which is typically `TRUE`. The `pasteClipboard` method is used paste the clipboard contents into the buffer, the second argument is `NULL` to paste at the insert are, or an iterator specifying otherwise where the text should be inserted. The third argument is `TRUE` if the pasted text is to be editable.

Example 4.13: A simple command line interface
This example shows how the text view widget can be used to make a simple command line. While programming a command line is unlikely to be a common task in designing a GUI for a statistics application, the example is familiar and shows several different, but useful, aspects of the widget.

We begin by defining our text view widget and retrieving its buffer. We also specify a fixed-width font for the buffer.

```
tv <- gtkTextView()
tb <- tv$getBuffer()
font <- pangoFontDescriptionFromString("Monospace")
```

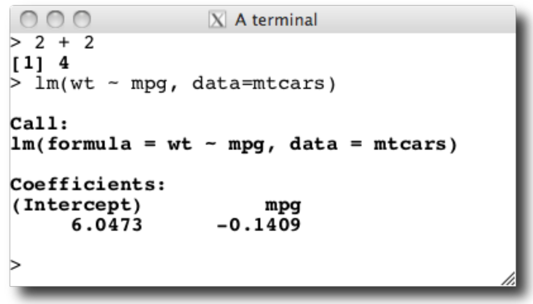

Figure 4.3: A basic R terminal implemented using a `gtkTextView` widget.

```
tv$modifyFont(font)                          # widget wide
```

We will use a few formatting tags, defined next. We don't need the tag objects, as we refer to them later by name.

```
tb$createTag(tag.name="cmdInput")
tb$createTag(tag.name="cmdOutput",
             weight=PangoWeight["bold"])
tb$createTag(tag.name="cmdError",
             weight=PangoStyle["italic"], foreground="red")
tb$createTag(tag.name="uneditable", editable=FALSE)
```

We define one new mark to indicate the prompt for a new line (we need to be able to identify a new command, and this marks its beginning) and another for the end of the buffer.

```
startCmd <- gtkTextMark("startCmd", left.gravity=TRUE)
tb$addMark(startCmd, tb$getStartIter()$iter)
bufferEnd <- tb$CreateMark("bufferEnd", tb$getEndIter()$iter)
```

We define several helper functions. This first shows how to move the viewport to the end of the buffer so that the command line is visibile.

```
scrollViewport <- function(view, ...) {
  iter <- view$getBuffer()$getEndIter()$iter
  view$scrollToMark(bufferEnd, within.margin=0)
  return(FALSE)
}
```

There are two types of prompts needed: one for entering a new command and one for a continuation. This function adds either, depending on its argument.

```
addPrompt <- function(obj, prompt=c("prompt", "continue"),
                      setMark=TRUE)
{
```

```
  prompt <- match.arg(prompt)
  prompt <- getOption(prompt)

  endIter <- obj$getEndIter()
  obj$insert(endIter$iter, prompt)
  if(setMark)
    obj$moveMarkByName("startCmd", endIter$iter)
  obj$applyTagByName("uneditable", obj$getStartIter()$iter,
                     obj$getEndIter()$iter)
}
addPrompt(tb) ## place an initial prompt
```

This helper method writes the output of a command to the text buffer. We arrange to truncate large outputs. By passing in the tag name, we can specify whether this is normal output or an error message.

```
addOutput <- function(obj, output, tagName="cmdOutput") {
  endIter <- obj$getEndIter()
  if(length(output) > 0)
    sapply(output, function(i)  {
      obj$insertWithTagsByName(endIter$iter, i, tagName)
      obj$insert(endIter$iter, "\n", len=-1)
    })
}
```

This next function uses the `startCmd` mark and the end of the buffer to extract the current command. Multi-line commands are parsed with a regular expression.

```
findCMD <- function(obj) {
  endIter <- obj$getEndIter()
  startIter <- obj$getIterAtMark(startCmd)
  cmd <- obj$getText(startIter$iter, endIter$iter, TRUE)
  regex <- paste("\n[", getOption("continue"), "] ", sep = "")
  cmd <- unlist(strsplit(cmd, regex))
  cmd
}
```

The following function takes the current command and evaluates it. It uses a hack (involving `grep`) to distinguish between an incomplete command and a true syntax error.

```
evalCMD <- function(tv, cmd) {
  tb <- tv$getBuffer()
  cmd <- paste(cmd, sep="\n")
  out <- try(parse(text=cmd), silent=TRUE)
  if(inherits(out, "try-error")) {
    if(length(grep("end", out))) {      # unexpected end of input
      ## continue
      addPrompt(tb, "continue", setMark=FALSE)
```

```
    } else {
      ## error
      addOutput(tb, out, tagName = "cmdError")
    }
    scrollViewport(tv)
    return()
  }

  e <- parse(text = cmd)
  out <- capture.output(vis <- withVisible(try(eval(e, .GlobalEnv), TRUE)))

  addOutput(tb, out)
  if (inherits(vis$value, "try-error"))
    addOutput(tb, vis$value, "cmdError")
  else if (vis$visible)
    addOutput(tb, capture.output(print(vis$value)))

  addPrompt(tb, "prompt", setMark=TRUE)
  scrollViewport(tv)
}
```

The `evalCMD` command is called when the return key is pressed. The `key-release-event` signal passes the event information through to the second argument. We inspect the key value and compare to that of the return key.

```
gSignalConnect(tv, "key-release-event", f=function(w, e, data) {
  obj <- w$getBuffer()                    # w is textview
  keyval <- e$getKeyval()
  if(keyval == GDK_Return) {
    cmd <- findCMD(obj)                   # character(0) if nothing
    if(length(cmd) && nchar(cmd) > 0)
      evalCMD(w, cmd)
  }
})
```

Finally, We connect `moveViewport` to the `changed` signal of the text buffer, so that the view always scrolls to the bottom when the contents of the buffer are modified:

Figure **??** shows the widget placed into a very simple GUI.

Inserting non-text items If desired, one can insert images and/or widgets into a text buffer, although this isn't a common use within statistical GUIs. The method `insertPixbuf` will insert into a position specified by an iter a `GdkPixbuf` object. In the buffer, this will take up one character, but will not be returned by `getText`.

Arbitrary child components can also be inserted. To do so an anchor must first be created in the text buffer. The method `createChildAnchor` will return such an anchor, and then the text view method addChildAtAnchor can be used to add the child.

GtkTextView Details

Properties Key properties of the text view include editable, which if assigned a value of FALSE will prevent users from editing the text. If the view is not editable, the cursor may be hidden by setting the cursor-visible property to FALSE. The text in a view may be wrapped or not. The method setWrapMode takes values from GtkWrapMode with default of "none", but options for "char", "word", or "word_char". The justification for the entire buffer is controlled by the justification property which takes values of "left", "right", "center", or "fill" from GtkJustification. The global value may be overridden for parts of the text buffer through the use of text tags. The left and right margins are adjusted through the left-margin and right-margin properties.

Fonts The size and font can be globally set for a text view using the modifyFont method. (Specifying fonts for parts of the buffer requires the use of tags, described later.) The argument font.desc specifies the new font using a Pango font description, which may be generated from a string specifying the font through the function pangoFontDescriptionFromString. These strings may contain up to 3 parts: the first is a comma-separated list of font families, the second a white-space separated list of style options, and the third a size in points or pixels if the units "px" are included. A typical value might look like "serif, monospace bold italic condensed 16". The various style options are enumerated in PangoStyle, PangoVariant, PangoWeight, PangoStretch, and PangoGravity. The help page for PangoFontDescription contains more information.

Signals The text buffer emits many different types of signals detailed in the help page for gtkTextBuffer. Most importantly, the changed signal is emitted when the content of the buffer changes. The callback for a changed signal has signature that returns the text buffer and any user data.

5

RGtk2: Menus and Dialogs

In the traditional WIMP-style GUI, the user executes commands by selecting items from a menu. In GUI terminology, such a command is known as an *action*. A GUI may provide more than one control for executing a particular action. Menubars and toolbars are the two most common widgets for performing application-wide actions. In this chapter, we will introduce actions, menus and toolbars and conclude by explaining the mechanisms in GTK+ for conveniently defining and managing actions and associated widgets in a large application.

5.1 Actions

GTK+ represents actions with the `GtkAction` class. A `GtkAction` can be proxied by widgets like buttons in a `GtkMenubar` or `GtkToolbar`. The `gtkAction` function is the constructor:

```
a <- gtkAction(name="ok", label="_Ok",
               tooltip="An OK button", stock.id="gtk-ok")
```

The constructor takes arguments `name` (to programmatically refer to the action), `label` (the displayed text), `tooltip`, and `stock.id` (identifying a stock icon). The command associated with an action is implemented by a callback connected to the `activate` signal:

```
gSignalConnect(a, "activate", f = function(w, data) {
  print(a$getName())                    # or some useful thing
})
```

An action plays the role of a data model describing a command, while widgets that implement the `GtkActivatable` interface are the views and controllers. All buttons, menu items and tool items implement `GtkActivatable` and thus may serve as action proxies. Actions are connected to widgets through the method `setRelatedAction`:

```
b <- gtkButton()
b$setRelatedAction(a)
```

Certain aspects of a proxy widget are coordinated through the action. This includes sensitivity and visibility, corresponding to the `sensitive` and `visible` properties. To synchronize with aesthetic properties like the `label` and icon (e.g., `stock-id`), the `use-action-appearance` property must be set on the activatable widget:

```
b["use-action-appearance"] <- TRUE
```

Often, the commands in an application have a natural grouping. It can be convenient to coordinate the sensitivity and visibility of entire groups of actions. `GtkActionGroup` represents a group of actions. By convention, keyboard accelerators are organized by group, and the accelerator for an action is usually specified upon insertion:

```
group <- gtkActionGroup()
group$addActionWithAccel(a, "<control>O")
```

In addition to the properties already introduced, an action may have a shorter label for display in a toolbar (`short_label`), and hints for when to display its label (`is_important`) and image (`always_show_image`).

There is a special type of action that has a toggled state: `GtkToggleAction`. The `active` property represents the toggle. A further extension is `GtkRadioAction`, where the toggled state is shared across a list of radio actions, via the `group` property. Proxy widgets represent toggle and radio actions with controls resembling check boxes and radio buttons, respectively. Here, we create a toggle action for fullscreen mode:

```
fullScreen <- gtkToggleAction("fullscreen", "Full screen", "Toggle full screen")
gSignalConnect(fullScreen, "toggled", function(action) {
  if(fullScreen['active'])
    window$fullscreen()
  else
    window$unfullscreen()
})
```

We connect to the `toggled` signal to respond to a change in the action state.

5.2 Menus

A menu is a compact, hierarchically organized collection of buttons, each of which may proxy an action. Menus listing window-level actions are usually contained within a menu bar at the top of the window or screen. Menus with options specific to a particular GUI element may "popup" when the user interacts with the element, such as by clicking the right mouse button. Menubars and popup menus may be constructed by appending each menu item and submenu separately, as illustrated below. For menus with more than a few items, we recommend the strategies described in Section **??**.

Menubars

We will first demonstrate the menu bar, leaving the popup menu for later. The first step towards populating a menu bar is to construct the menu bar itself:

```
menubar <- gtkMenuBar()
```

A menu bar is a special type of container called a menu shell. An instance of `GtkMenuShell` contains one or more menu items. `GtkMenuItem` is an implementation of `GtkActivatable`, so each menu item can proxy an action. Usually, a menu bar consists multiple instances of the other type of menu shell: the menu, `GtkMenu`. Here, we create a menu object for our "File" menu:

```
fileMenu <- gtkMenu()
```

As a menu is not itself a menu item, we first must embed the menu into a menu item, which is labeled with the menu title:

```
fileItem <- gtkMenuItemNewWithMnemonic(label="_File")
fileItem$setSubmenu(fileMenu)
```

The underscore in the label indicates the key associated with the mnemonic for use when navigating the menu with a keyboard. Finally, we append the item containing the file menu to the menu bar:

```
menubar$append(fileItem)
```

In addition to `append`, it is also possible to `prepend` and `insert` menu items into a menu shell. As with any container, one can `remove` a child menu item, although the convention is to desensitize an item, through the `sensitive` property, when it is not currently relevant.

Next, we populate our file menu with menu items that perform some command. For example, we may desire an open item:

```
open <- gtkMenuItemNewWithMnemonic("_Open")
```

This item does not have an associated `GtkAction`, so we need to implement its activate signal directly:

```
gSignalConnect(open, "activate", function(item) {
  f <- file.choose()
  file.show(f)
})
```

The item is now ready to be added to the file menu:

```
fileMenu$append(open)
```

It is recommended, however, to create menu items that proxy an action. This will facilitate, for example, adding an equivalent toolbar item later. First, we create the action:

```
saveAction <- gtkAction("save", "Save", "Save object", "gtk-save")
```

Then the appropriate menu item is generated from the action and added to the file menu:

```
save <- saveAction$createMenuItem()
## unnecessary?
##save["use-action-appearance"] <- TRUE
fileMenu$append(save)
```

A simple way to organize menu items, besides grouping into menus, is to insert separators between logical groups of items. Here, we insert a separator item, rendered as a line, to group the open and save commands apart from the rest of the menu:

```
fileMenu$append(gtkSeparatorMenuItem())
```

Toggle menu items, i.e., a label next to a check box, are also supported. A toggle action will create one implicitly:

```
autoSaveAction <- gtkToggleAction("autosave", "Autosave",
                                  "Enable autosave")
autoSave <- autoSaveAction$createMenuItem()
fileMenu$append(autoSave)
```

Finally, we add our menubar to the top of a window:

```
mainWindow <- gtkWindow()
vbox <- gtkVBox()
mainWindow$add(vbox)
vbox$packStart(menubar, FALSE, FALSE)
```

Popup Menus

Example 5.1: Popup menus
To illustrate popup menus, we show how construct one and display it in response to a mouse click. We start with a `gtkMenu` instance, to which we add some items.

```
popup <- gtkMenu()                          # top level
popup$append(gtkMenuItem("cut"))
popup$append(gtkMenuItem("copy"))
popup$append(gtkSeparatorMenuItem())
popup$append(gtkMenuItem("paste"))
```

Let us assume that we have a button that will popup a menu when clicked with the third (right) mouse button:

```
b <- gtkButton("Click me with right mouse button")
w <- gtkWindow(); w$setTitle("Popup menu example")
w$add(b)
```

This menu will be shown by calling gtkMenuPopup in response to the button-press-event signal on the button. The gtkMenuPopup function is called with the menu, some optional arguments for placement, and some values describing the event: the mouse button and time. The event values can be retrieved from the second argument of the callback (a GdkEvent).

```
gSignalConnect(b,"button-press-event",
       f = function(w, e, menu) {
         if(e$getButton() == 3 ||
            (e$getButton() == 1 && # a mac
             e$getState() == GdkModifierType['control-mask']))
           gtkMenuPopup(menu,
                        button = e$getButton(),
                        activate.time = e$getTime())
         return(FALSE)
       }, data=popup)
```

The above will popup a menu, but until we bind a callback to the activate signal, nothing will happen when a menu item is selected. Below we supply a stub for sake of illustration. The children of a popup menu are the menu items, including the separator which we avoid.

```
IDs <- sapply(popup$getChildren(), function(i) {
  if(!inherits(i, "GtkSeparatorMenuItem")) # skip these
    gSignalConnect(i, "activate",
                   f = function(w, data) print("replace me"))
})
```

5.3 Toolbars

Toolbars are like menu bars in that they are containers for activatable items, but toolbars are not hierarchical. Also, their items are usually visible for the life-time of the application, not upon user click. Thus, toolbars are not appropriate for storing a large number of items, only those that are activated most often.

We begin by constructing an instance of GtkToolbar:

```
toolbar <- gtkToolbar()
```

In analogous fashion to the menu bar, toolbars are containers for tool items. Technically, an instance of GtkToolItem could contain any type of widget, though toolbars typically represent actions with buttons. The GtkToolButton class implements this common case. Here, we create a tool button for opening a file:

```
openButton <- gtkToolButton(stock.id = "gtk-open")
```

Tool buttons have a number of properties, including label and several for specifying an icon. Above, we specify a stock identifier, for which there is a predefined translated label and theme-specific icon. As with any other container, the button may be added to the toolbar with the `add` method:

```
toolbar$add(openButton)
```

This appends the open button to the end of the toolbar. To insert into a specific position, we would call the `insert` method.

Usually, any application with a toolbar also has a menu bar, in which case many actions are shared between the two containers. Thus, it is often beneficial to construct a tool button directly from its corresponding action:

```
saveButton <- saveAction$createToolItem()
toolbar$add(saveButton)
```

A tool button is created for `saveAction`, created in the previous section.

Like menus, related buttons may be grouped using separators:

```
toolbar$add(gtkSeparatorToolItem())
```

Any toggle action will create a toggle tool button as its proxy:

```
fullScreenButton <- fullScreen$createToolItem()
toolbar$add(fullScreenButton)
```

A `GtkToggleToolButton` embeds a `GtkToggleButton`, which is depressed whenever its `active` property is `TRUE`.

As mentioned above, toolbars, unlike menus, are usually visible for the duration of the application. This is desirable, as the actions in a toolbar are among those most commonly performed. However, care must be taken to conserve screen space. The toolbar *style* controls whether the tool items display their icons, their text, or both. The possible settings are in the `GtkToolbarStyle` enumeration. The default value is specified by the global GTK+ style (theme). Here, we override the default to only display images:

```
toolbar$setStyle("icon")
```

For canonical actions like *open* and *save*, icons are usually sufficient. Some actions, however, may require textual explanation. The `is-important` property on the action will request display of the label in a particular tool item, in addition to the icon:

```
fullScreen["is-important"] <- TRUE
```

Normally, tool items are tightly packed against the left side of the toolbar. Sometimes, a more complex layout is desired. For example, we may wish to place a *help* item against the right side. We can achieve this with an invisible item that expands against its siblings:

```
expander <- gtkSeparatorToolItem()
expander["draw"] <- FALSE
toolbar$add(expander)
```

The dummy item is a separator with its `draw` property set to `FALSE`, and its `expand` child property set to `TRUE`. Now we can add the *help* item:

```
helpAction <- gtkAction("help", "Help", "Get help", "gtk-help")
toolbar$add(helpAction$createToolItem())
```

It is now our responsibility to place the toolbar at the top of the window, under the menu created in the previous section:

```
vbox$packStart(toolbar, FALSE, FALSE)
```

Example 5.2: Color menu tool button
Space in a toolbar is limited, and sometimes there are several actions that differ only by a single parameter. A good example is the color tool button found in many word processors. Including a button for every color in the palette would consume an excessive amount of space. A common idiom is to embed a drop-down menu next to the button, much like a combo box, for specifying the color, or, in general, any discrete parameter.

We demonstrate how one might construct a color-selecting tool button. Our menu will list the colors in the R palette. The associated button is a `GtkColorButton`. When the user clicks on the button, a more complex color selection dialog will appear, allowing total customization.

```
gdkColor <- gdkColorParse(palette()[1])$color
colorButton <- gtkColorButton(gdkColor)
```

`gtkColorButton` requires the initial color to be specified as a `GdkColor`, which we parse from the R color name.

The next step is to build the menu. Each menu item will display a 20*x*20 rectangle, filled with the color, next to the color name:

```
colorMenuItem <- function(color) {
  da <- gtkDrawingArea()
  da$setSizeRequest(20, 20)
  da$modifyBg("normal", color)
  item <- gtkImageMenuItem(color)
  item$setImage(da)
  item
}
colorItems <- sapply(palette(), colorMenuItem)
colorMenu <- gtkMenu()
for (item in colorItems)
  ## JV: fixed this
##  menu$append(item)
  colorMenu$append(item)
```

An important realization is that the image in a `GtkImageMenuItem` may be any widget that presumably draws an icon; it need not be an actual `GtkImage`. In this case, we use a drawing area with its background set to the color. When an item is selected, its color will be set on the color button:

```
colorMenuItemActivated <- function(item) {
  color <- gdkColorParse(item$getLabel())$color
  colorButton$setColor(color)
}
sapply(colorItems, gSignalConnect, "activate", colorMenuItemActivated)
```

Finally, we place the color button and menu together in the menu tool button:

```
menuButton <- gtkMenuToolButton(colorButton, "Color")
menuButton$setMenu(colorMenu)
toolbar$add(menuButton)
```

Some applications may offer a large number of actions, where there is no clear subset of actions that are more commonly performed than the rest. It would be impractical to place a tool item for each action in a static toolbar. GTK+ provides a *tool palette* widget as one solution, which leaves the configuration of a multi-row toolbar to the user. The tool items are organized into collapsible groups, and the grouping is customizable through drag and drop.

`GtkToolPalette` is a container of `GtkToolItemGroup` widgets, each of which is a container of tool items and implements `GtkToolShell`, like `GtkToolbar`. We begin our brief example by creating a two groups of tool items:

```
fileGroup <- gtkToolItemGroup("File")
fileGroup$add(gtkToolButton(stock.id = "gtk-open"))
fileGroup$add(saveAction$createToolItem())
helpGroup <- gtkToolItemGroup("Help")
helpGroup$add(helpAction$createToolItem())
```

The groups are then added to an instance of `GtkToolPalette`:

```
palette <- gtkToolPalette()
palette$add(fileGroup)
palette$add(helpGroup)
```

Finally, we can programmatically collapse a group:

```
helpGroup$setCollapsed(TRUE)
```

5.4 Managing a complex user interface

Complex applications implement a large number of actions and operate in a number of different modes. Within a given mode, only a subset of actions are applicable. For example, a word processor may have an editing mode and a print preview mode. GTK+ provides a *user interface manager*, `GtkUIManager`, to manage the layout of the toolbars and menubars across multiple user interface modes.

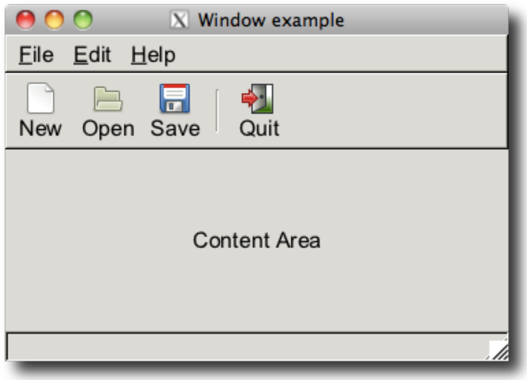

Figure 5.1: A GUI made using a UI manager to layout the menubar and toolbar.

The steps required to use GTK+'s UI manager are

1. construct the UI manager,
2. define the actions in groups,
3. specify the layout of the menubars and toolbars,
4. connect the action group to the UI manager,
5. set up an accelerator group for keyboard shortcuts, and finally
6. display the widgets.

Example 5.3: UI Manager example

We begin by constructing the UI manager:

```
uimanager = gtkUIManager()
```

Next, we define the action groups. For demonstration purposes, our actions simply push the action name onto the status bar at the bottom of the window. Below, we define the callback that will implement every action:

```
someAction <- function(action,...)
  statusbar$push(statusbar$getContextId("message"), action$getName())
Quit <- function(...) win$destroy()
```

We break up our action group definitions into one for "File" and "Edit" and another one for "Help." Every action is defined by a `GtkActionEntry` structure, which is represented by a list in R. The components (in order) are the name; the icon; the label, with _ specifying the mnemonic; the keyboard accelerator, with `<control>`, `<alt>`, `<shift>` as possible prefixes, a tooltip, and finally the callback. Empty values can be defined as `NULL` or, except for the callback, an empty string.

```
firstActionGroup <- gtkActionGroup("firstActionGroup")
firstActionEntries <- list(
  ## name,ID,label,accelerator,tooltip,callback
  file = list("File",NULL,"_File",NULL,NULL,NULL),
  new = list("New", "gtk-new", "_New", "<control>N",
    "New document", someAction),
  sub = list("Submenu", NULL, "S_ub", NULL, NULL, NULL),
  open = list("Open", "gtk-open", "_Open", "<ctrl>O",
    "Open document", someAction),
  save = list("Save", "gtk-save", "_Save", "<alt>S",
    "Save document", someAction),
  quit = list("Quit", "gtk-quit", "_Quit", "<ctrl>Q",
    "Quit", Quit),
  edit = list("Edit", NULL, "_Edit", NULL, NULL, NULL),
  undo = list("Undo", "gtk-undo", "_Undo", "<ctrl>Z",
    "Undo change", someAction),
  redo = list("Redo", "gtk-redo", "_Redo", "<ctrl>U",
    "Redo change", someAction)
)
```

In the above, we define the dummy actions "File" and "Edit" that perform no function. They are necessary for specifying the menu layout later. We now add the actions to the action group, then add this action group to the first spot in the UI manager.

```
firstActionGroup$addActions(firstActionEntries)
uimanager$insertActionGroup(firstActionGroup, 0) # 0-based
```

It is also possible to define toggle actions, as we demonstrate presently for the "Help" group. First, we define the ordinary actions:

```
helpActionGroup <- gtkActionGroup("helpActionGroup")
helpActionEntries <- list(
  help = list("Help", "", "_Help", "", "", NULL),
  about = list("About", "gtk-about", "_About", "", "", someAction)
)
helpActionGroup$addActions(helpActionEntries)
```

Next, we define a "Use tooltips" toggle action.

A toggle is defined with `gtkToggleAction` which has signature in a different order than the action entry. Notice, we don't have an icon, as the toggled icons is used. To add a callback, we connect to the `toggled` signal of the action element. This callback allows for user data, as illustrated.

```
toggleActions <- list(
        ## name, label, tooltip, stock.id
        tooltips = gtkToggleAction("UseTooltips", "Use tooltips",
          "Use tooltips to show additional information", NULL)
)
gSignalConnect(toggleActions[[1]], "toggled", function(...) print("toggled"))
```

```
helpActionGroup$addAction(toggleActions$tooltips)
```

The list structure for toggle actions is identical to that of the ordinary actions, except for the last element which indicates whether the action is initially active. We insert the help action group in the second position.

```
uimanager$insertActionGroup(helpActionGroup, 1)
```

One can also incorporate radio actions, although this is not shown.

Our menubar and toolbar layout is specified as XML according to a schema specified by the UI manager framework. The XML can be stored in a file or an R character vector. The structure of the file can be grasped quickly from this example:

```
<ui>
  <menubar name="menubar">
    <menu name="FileMenu" action="File">
      <menuitem name="FileNew" action="New"/>
      <menu action="Submenu">
<menuitem name="FileOpen" action="Open" />
      </menu>
      <menuitem name="FileSave" action="Save"/>
      <separator />
      <menuitem name="FileQuit" action="Quit"/>
    </menu>
    <menu action="Edit">
      <menuitem name="EditUndo" action="Undo" />
      <menuitem name="EditRedo" action="Redo" />
    </menu>
    <menu action="Help">
      <menuitem action="UseTooltips"/>
      <menuitem action="About"/>
    </menu>
  </menubar>
  <toolbar name="toolbar">
    <toolitem action="New"/>
    <toolitem action="Open"/>
    <toolitem action="Save"/>
    <separator />
    <toolitem action="Quit"/>
  </toolbar>
</ui>
```

The top-level element is named `ui`, only one of which is allowed in a UI definition. The children of `ui` represent a top-level action container: `menubar`, `toolbar`, or `popup`. The `name` attributes are used to refer to the widgets later on. The menubar element contains menu elements, which in turn contain menuitem and separator elements, as well as additional menu elements for nesting. The toolbars are populated with toolitem elements. The item elements have an action attribute that refers to an action in one of our action groups and an optional name (defaulting to the action value).

This file is loaded into the UI manager as follows

```
id <- uimanager$addUiFromFile("ex-menus.xml")
```

The id value can be used to merge and delete UI components according to the mode of the UI, but this is not illustrated here.

Now we can setup a basic window template with menubar, toolbar, and status bar. We first construct the three main widgets. The UI manager will construct our toolbar and menubar, as identified from the names specified in the UI definition:

```
menubar <- uimanager$getWidget("/menubar")
toolbar <- uimanager$getWidget("/toolbar")
```

The statusbar is constructed with

```
statusbar <- gtkStatusbar()
```

Now we create a top-level window and attach a keyboard accelerator group to the window so that when the window has the focus, the keyboard shortcuts defined for our actions are active.

```
win <- gtkWindow(show=TRUE)
win$setTitle("Window example")
accelgroup <- uimanager$getAccelGroup()
win$addAccelGroup(accelgroup)
```

Now it is a simple matter of packing the widgets into a box.

```
box <- gtkVBox()
win$add(box)
box$packStart(menubar, expand=FALSE, fill=FALSE,0)
box$packStart(toolbar, expand=FALSE, fill= FALSE,0)
contentArea <- gtkVBox()
box$packStart(contentArea, expand=TRUE, fill=TRUE,0)
contentArea$packStart(gtkLabel("Content Area"))
box$packStart(statusbar, expand=FALSE, fill=FALSE, 0)
```

The redo feature should only be sensitive to mouse events after a user has undone an action. If we wanted to alter the sensitivity of the redo action, we would need to retrieve it from the action group:

```
redo <- firstActionGroup$getAction("Redo")
redo['sensitive'] <- FALSE
```