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; GIO for synchronous and asynchronous input/output for files and network resources; 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 version of 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.

1.1 Synopsis of the RGtk2 API

Constructing a GUI with RGtk2 generally proceeds by constructing a widget and then configuring it by calling methods and setting properties. Handlers are connected to signals, and the widget is combined with other widgets to form the GUI. For example:

```
button <- gtkButton("Click Me")
button['image'] <- gtkImage(stock = "gtk-apply",
                             size = "button")
gSignalConnect(button, "clicked", function(x) {
  message("Hello World!")
})
##
window <- gtkWindow(show = FALSE)
window$add(button)
window$showAll()
```

Once one understands the syntax and themes of the above example, it is only a matter of reading through the proceeding chapters and the documentation to discover all of the widgets and their features. The rest of this chapter will explain these basic components of the API.

1.2 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 through 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.

The type system supports reflection, so we can, for example, obtain a list of the ancestors for a given class:

```
gTypeGetAncestors("GtkWidget")
```

```
[1] "GtkWidget"          "GtkObject"
[3] "GInitiallyUnowned" "GObject"
```

For those familiar with object-oriented programming in R, the returned character vector could be interpreted as it were a class attribute on an S3 object.

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. For example, we list the interfaces implemented by `GtkWidget`:

```
gTypeGetInterfaces("GtkWidget")
```

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

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 1.1. A more detailed and technical explanation of `GObject` is available in Chapter 6.

1.3 Constructors

The next few sections will contribute to a unifying example that displays a button in a window. When clicked, the button will print a message to the R console. 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

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

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. We prefer these shorter, more flexible constructors, such as gtkWindow or gtkButton, but note their documentation is provided by the R package author and is in addition to the formal API. These constructors can take many arguments, and only some subsets of the arguments may be specified at once. For example, this call

```
gtkImage(stock = "gtk-apply", size = "button")
```

uses only two arguments, stock and size, which always must be specified together. The entire signature is more complex:

```
args(gtkImage)
```

```
function (size, mask = NULL, pixmap = NULL, image = NULL,
    filename, pixbuf = NULL, stock.id, icon.set, animation,
    icon, show = TRUE)
```

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. A reference object will not be copied before modification. This is different from the behavior of most R objects. For example, calling `abs` on a numeric vector does not change the value assigned to the original symbol:

```
a <- -1
abs(a)
```

```
[1] 1
```

```
a
```

```
[1] -1
```

Setting the text label on our button, however, will change the original value:

```
gtkButtonSetLabel(button, "New text")
gtkButtonGetLabel(button)
```

```
[1] "New text"
```

If this widget were displayed on the screen, the label would also be updated.

The class hierarchy of an 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.

1.4 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.[1] 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

[1] `RGtk2` uses `R`'s standard dollar-sign notation (also used with reference classes) for class-based method dispatch.

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.

1.5 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 definitions belonging to the widget is returned by its `getPropInfo` method. Calling `names` on the object returns the property names. 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. The methods `get` and `set` get and set properties of a widget, respectively. The set function treats the argument 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"))
window$set(icon = image[[1]], 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"
```

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

```
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.

1.6 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` function adds 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 to store, but uses will be discussed later.

We demonstrate `gSignalConnect` 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 the `data` argument should be the first argument to the callback or (the default) the last.

The `after` argument is a logical value indicating if the callback should be called after the default handler (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.

It 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. A return value of `TRUE` indicates that no further callbacks should be called, whereas `FALSE` indicates that the next callback should be called. In other words, the return value indicates whether the handler has consumed the event. In the following example, only the first two callbacks are executed when the user clicks 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 handler, if desired, using `gSignalHandlerDisconnect`. To temporarily block a handler, call `gSignalHandlerBlock` and then `gSignalHandlerUnblock` to unblock. The man page for `gSignalConnect` gives the details.

1.7 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-buffered
```

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
```

1.8 The event loop

`RGtk2` integrates the GTK+ and `R` event loops by treating the `R` loop as the master and iterating the GTK+ event loop whenever R is idle. 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()
```

This is often useful, for example, to update a progress bar.

If one runs an `RGtk2` script non-interactively, such as by assigning an icon to launch a GUI under Windows, R will exit after the script is finished and the GUI will disappear just after it appears. To work around this, call the function `gtkMain` to run the main loop until the function `gtkMainQuit` is called. Since there is no interactive session, `gtkMainQuit` should be called through some event handler.

1.9 Importing a GUI from Glade

This book focuses almost entirely on the direct programmatic construction of GUIs. Some developers prefer visually constructing a GUI by pointing, clicking and dragging in another GUI, which one might call a GUI builder, a type of RAD (Rapid Application Development) tool. `Glade` is the primary GUI builder for GTK+ and exports an interface as XML that is loadable by `GtkBuilder`. It is freely available for all major platforms from `http://glade.gnome.org/`. Documentation is also at that location.

We will assume that the reader has saved an interface as a `GtkBuilder` XML file named `buildable.xml` and is ready to load it with `RGtk2`:

```
g <- gtkBuildableNew()
g$addFromFile("buildable.xml")
```

The `getObject` extracts a widget by its ID, which is specified by the user through `Glade`. It normally suffices to load the top-level widget, named `dialog1` in this example, and show it:

```
d <- g$getObject("dialog1")
d$showAll()
```

In order to add behaviors to the GUI, we need to register R functions as signal handlers. In `Glade`, the user should specify the name of an R function as a handler for some signal. `RGtk2` extends `GtkBuilder` to look up the functions and connect them to the appropriate signals. Let us assume that the user has named the `ok_button_clicked` function as the handler for the `clicked` signal on a `GtkButton`. The `connectSignals` method will establish that connection and any others in the interface:

```
ok_button_clicked <- function(w, userData) {
  print("hello world")
}
g$connectSignals()
```

The GUI should now be ready for use.

2

RGtk2: Windows, Containers, and Dialogs

This chapter covers top-level windows, dialogs and the container objects provided by GTK+.

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 argument `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  use 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 above, 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`, which derives from `GtkContainer` and allows only a single child. As before, this child is added through the `add` method. 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)
```

To display multiple widgets in a window, one simply needs to add a non-`GtkBin` container as the child widget.

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. When this happens, 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. For example:

```
gSignalConnect(w, "delete-event", function(event) {
  gtkMessageDialog(parent=w, flags=0, type="question",
                   buttons=c("yes", "no"),
                   "Are you sure you want to quit?")
  dlg$run() != GtkResponseType["yes"]
})
```

(We describe the use of message dialogs in Section 2.3.) 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.

It is also possible to close a window programmatically by calling its `destroy` method:

```
w$destroy()
```

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 can be set to be destroyed with its parent. For example:

```
## create a window and a dialog window
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 in Section 2.3, 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

In GTK+, the widget hierarchy is built when children are added to a parent container. In this example, a window is made the parent of a label:

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

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']
```

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

The [[method accesses the child widgets by number, as a convenient wrapper around the getChildren method:

```
w[[1]]['label']
```

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

Conversely, 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. For instance

```
w$remove(l)
w$add(l)
```

To remove a widget from the screen but not its container, use the hide method on the widget. The reparent method is a convenience for moving a widget between containers that ensures the child is not garbage collected during the transition.[1]

Widget size negotiation

We have already seen perhaps the simplest automatic layout container, GtkWindow, which fills all of its space with its child. Despite the apparent simplicity, 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 then consumes the allocated space. Consider the previous example of adding a label to a window:

```
w <- gtkWindow(); 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$getAllocation()$allocation
```

[1] An object becomes available for garbage collection when it has no references to it, which can happen if it is removed from the parent container.

```
     x       y  width  height
    -1      -1      1       1
```

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()$allocation
```

```
     x       y  width  height
     0       0     79      18
```

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 (implemented in class `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 and align themselves in both dimensions, 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 2.1 illustrates a sampling of the possible layouts that are possible with a `GtkHBox`.

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 against the left and right side (top and bottom for a `GtkVBox`), respectively. For this explanation, we restrict ourselves to `packStart`, since `packEnd` works the same except for the direction. Below, we pack two buttons, `button_a` and `button_b` against the left side:

```
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 2.1. 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 natural size 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 and the widget is aligned in the center of its space. When a widget is packed with the `fill` parameter set to `TRUE`, the widget is resized to consume the available space. This results in rows 2 and 3 in Figure 2.1.

In many cases, it is desirable to give children unequal amounts of available space, as in rows 4–9 in Figure 2.1. To create a heterogeneously spaced `GtkHBox`, we pass `FALSE` as the first argument to the constructor, as in the following code:

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

A heterogeneous 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`

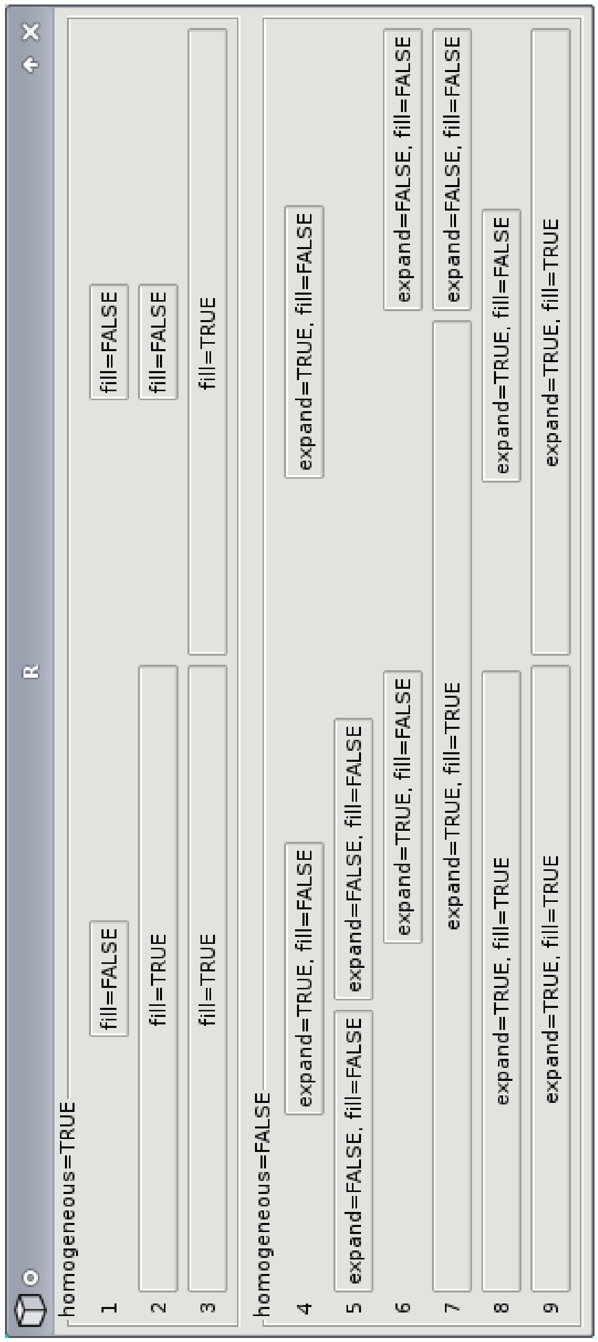


Figure 2.1: A screenshot demonstrating the effect of packing two buttons into `GtkHBox` instances using the `packStart` method with different combinations of the `expand` and `fill` settings. The effect of the `homogeneous` spacing setting on the `GtkHBox` is also shown.

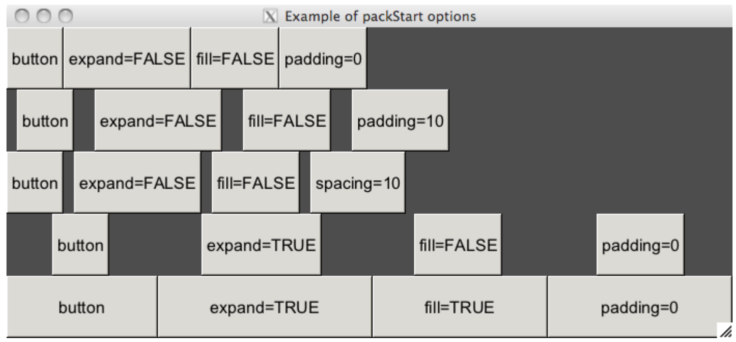


Figure 2.2: Examples of packing widgets into a box container. The top row shows no padding, whereas the 2nd and 3rd illustrate the difference between `padding` (an amount around each child) and `spacing` (an amount between each child). The last two rows show the effect of `fill` when `expand=TRUE`. This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

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 2.1, 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 2.1. The figure contains several other permutations of the `homogeneous`, `expand` and `fill` settings.

Padding 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 3.3 provides an example and Figure 2.2 an illustration.

Positioning The `reorderChild` method reorders the child widgets. The new position of the child is specified using 0-based indexing. This code will move the third child of `g` to the second position:

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

Alignment

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

```
w <- gtkWindow(); 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 a number of convenient dialogs for the most common use cases, as well as a 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. Typically, a dialog belongs to a main application window and might be modal, in which case input is blocked to other parts of the GUI. `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 their order in the call. 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.

The dialog has a content area, which is an instance of GtkVBox. To complete our dialog, we place a labeled text entry into the content area:

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

The content is placed above the button box, with a separator between them.

In the message dialog example, we called the run method to make the dialog modal. To make a non-modal dialog, do not call run but connect to the response signal of the modal dialog. The response code of the clicked button is passed to the callback:

```
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 commonly used 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 (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.

Print dialog

Rendering documents for printing is outside our scope; however, we will mention that `GtkPrintOperation` can launch the native, platform-specific print dialog for customizing a printing operation. See Example 3.11 for an example of printing R graphics using `cairoDevice`.

2.4 Special-purpose containers

In Section 2.2, 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 many methods.

Framed containers

The `gtkFrame` function constructs a container that draws a decorative, labeled frame around its single child:

```
frame <- gtkFrame("Options")
vbox <- gtkVBox()
vbox$packStart(gtkCheckButton("Option 1"), FALSE)
vbox$packStart(gtkCheckButton("Option 2"), FALSE)
frame$add(vbox)
```

A frame is useful for visually segregating a set of conceptually related widgets from the rest of the GUI. The type of decorative shadow is stored in the `shadow-type` property. The `setLabelAlign` aligns the label relative to the frame. This is to the left, by default.

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`:

```
expander <- gtkExpander("Advanced")
expander$add(frame)
```

Use `gtkExpanderNewWithMnemonic` if a mnemonic is desired. 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.

We create a notebook and add some pages:

```
nb <- gtkNotebook()
nb$appendPage(gtkLabel("Page 1"), gtkLabel("Tab 1"))
```

```
[1] 0
```

```
nb$appendPage(gtkLabel("Page 2"), gtkLabel("Tab 2"))
```

```
[1] 1
```

A page specification consists of a widget for the page and a widget for the tab. Any type of widget is accepted, although a label is typically used for the tab. This flexibility allows for more complicated tabs, such as a box container with a label and close icon.

The tabs can be positioned on any of the four sides of the notebook; this depends on the `tab-pos` property, with a value from `GtkPositionType`: `"left"`, `"right"`, `"top"`, or `"bottom"`. By default, the tabs are on top. We move the current ones to the bottom:

```
nb['tab-pos'] <- "bottom"
```

Methods and properties that affect pages expect the page index, instead of the page widget. To map from the child widget to the page number, use the method `pageNum`. The `page` property holds the zero-based index of the active tab. We make the second tab active:

```
nb['page'] <- 1
nb['page']
```

```
[1] 1
```

To move sequentially through the pages, call the methods `nextPage` and `prevPage`. When the current page changes, the `switch-page` signal is emitted.

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, as long as they belong to the same group, which depends on the `group-id` property. 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.

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) {
    icon <- gtkImage(pixbuf =
      object$renderIcon("gtk-close", "button", size="menu"))
    closeButton <- gtkButton()
```

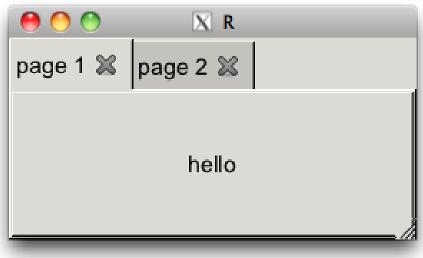


Figure 2.3: Simple illustration of customized tab in a notebook. These include a close button.

```
    closeButton$setImage(icon)
    closeButton$setRelief("none")
    ##
    label <- gtkHBox()
    label$packStart(gtkLabel(label.text))
    label$packEnd(closeButton)
    ##
    gSignalConnect(closeButton, "clicked", function(b) {
      index <- nb$pageNum(child)
      nb$removePage(index)
    })
    object$insertPage(child, label, position)
  }
```

Here is a simple demonstration of its usage:

```
w <- gtkWindow()
nb <- gtkNotebook(); w$add(nb)
nb$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. Scrolled windows are most often used with potentially large widgets like table views and when displaying images and graphics.

Our example will embed an R graphics device in a scrolled window and allow the user to zoom in and out and pull on the scroll bars to pan the view. First, we create an R graphics device using the `cairoDevice` package

```
library(cairoDevice)
device <- gtkDrawingArea()
device$setSizeRequest(600, 400)
asCairoDevice(device)
```

and then embed it within a scrolled window

```
scrolled <- gtkScrolledWindow()
scrolled$addWithViewport(device)
```

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, like our `GtkDrawingArea`, must be embedded within the proxy `GtkViewport`. The `GtkScrolledWindow` convenience method `addWithViewport` allows the programmer to skip the `GtkViewport` step.

Next, we define a function for scaling the plot:

```
zoomPlot <- function(x = 2.0) {
  allocation <- device$getAllocation()$allocation
  device$setSizeRequest(allocation$width * x,
                        allocation$height * x)
  updateAdjustment <- function(adj) {
    adj$setValue(x * adj$getValue() +
                 (x - 1) * adj$getPageSize() / 2)
  }
  updateAdjustment(scrolled$getHadjustment())
  updateAdjustment(scrolled$getVadjustment())
}
```

The function gets the current size allocation from the device, scales it by `"x"` and requests the new size. It then scrolls the window to preserve the center point. The state of each scroll bar is represented by a `GtkAdjustment`. We will update the value of the horizontal and vertical adjustments to scroll the window. The value of an adjustment corresponds to the left-/top position of the window, so we to adjust by half the page size after scaling the value.

We had key press events, so that pressing + zooms in and pressing - zooms out:

```
gSignalConnect(scrolled, "key-press-event", function(x, ev) {
  key <- ev[["keyval"]]
  if (key == GDK_plus)
    zoomPlot(2.0)
```

```
  else if (key == GDK_minus)
    zoomPlot(0.5)
  TRUE
})
```

Despite its name, the scrolled window is not a top-level window. Thus, it needs to be added to a top-level window:

```
win <- gtkWindow(show = FALSE)
win$add(scrolled)
win$showAll()
```

Finally, a basic scatterplot is displayed in the viewer:

```
plot(mpg ~ hp, data = mtcars)
```

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, if the child widget requests more space than can be allocated. The `setPolicy` method allows both to be set at once.

Divided containers

The `gtkHPaned` and `gtkVPaned` constructors create containers that contain two widgets, arranged horizontally or vertically and separated by a divider displaying a handle allowing the user to adjust the allocation of space between the child components. We will demonstrate only the horizontal pane `GtkHPaned` here, without loss of generality.

First, we construct an instance of `GtkHPaned`:

```
paned <- gtkHPaned()
```

The two children may be added two different ways. The simplest approach calls `add1` and `add2` for adding the first and second child, respectively.

```
paned$add1(gtkLabel("Left (1)"))
paned$add2(gtkLabel("Right (2)"))
```

This configures the container such that both children are allowed to shrink and only the second widget can expand. Such a configuration is appropriate for a GUI with main widget and a side pane to the left. More flexibility is afforded by the methods `pack1` and `pack2`, which have arguments for specifying whether the child should expand (`"resize"`) and/or `"shrink"`. Here we add the children such that both can expand and shrink:

```
paned$pack1(gtkLabel("Left (1)"), resize = TRUE, shrink = TRUE)
paned$pack2(gtkLabel("Right (2)"), resize = TRUE, shrink = TRUE)
```

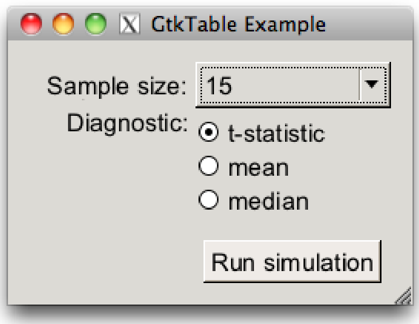


Figure 2.4: A basic dialog using a `gtkTable` container for layout.

After children are added, they can be retrieved 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` are useful for converting a percentage into a screen position. The `move-handle` signal is emitted when the gutter position is changed.

Tabular layout

`GtkTable` is a container for laying out objects in a tabular (or grid) format. It is *not* meant for displaying tabular data. The container divides its space into cells of a grid, and a child widget may occupy one or more cells. The allocation of space within a row or column follows logic similar to that of box layouts. The most common use case of a `GtkTable` is a form layout, which we will demonstrate in our example.

Example 2.3: Dialog layout
This example shows how to layout a form in a dialog with some attention paid to how the widgets are aligned and how they respond to resizing of the window.

Our form layout will require 3 rows and 2 columns:

```
tbl <- gtkTable(rows=3, columns=2, homogeneous=FALSE)
```

By default, the cells are allowed to have different sizes. This may be overridden by passing `"homogeneous = TRUE"` to the constructor, which forces all cells to have the same size.

We construct the widgets that will be placed in the form:

```
sizeLabel <- gtkLabel("Sample size:")
sizeCombo <- gtkComboBoxNewText()
sapply(c(5, 10, 15, 30), sizeCombo$appendText)
```

```
##
diagLabel <- gtkLabel("Diagnostic:")
diagRadios <- gtkVBox()
rb <- list()
rb$t <- gtkRadioButton(label="t-statistic")
rb$mean <- gtkRadioButton(rb, label="mean")
rb$median <- gtkRadioButton(rb, label="median")
sapply(rb, diagRadios$packStart)
##
submitBox <- gtkVBox()
submitBox$packEnd(gtkButton("Run simulation"), expand = FALSE)
```

We align the labels to the right, up against their corresponding entry widgets, which are left-aligned:

```
sizeLabel['xalign'] <- 1
diagLabel['xalign'] <- 1; diagLabel['yalign'] <- 0
diagAlign <- gtkAlignment(xalign = 0)
diagAlign$add(diagRadios)
```

The labels are aligned through the `GtkMisc` functionality inherited by `GtkLabel`. The `GtkVBox` with the radio buttons does not support this, so we have embedded it within a `GtkAlignment` instance. We have aligned the diagnostic label to the top of its cell; otherwise, it would have been centered vertically. The radio buttons are left aligned, up against the label (cf. Figure 2.4).

Child widgets are added to a `GtkTable` instance through its `attach` method. The child 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 2.2). There is no padding between children by default. Both the resizing behavior and padding may be overridden by specifying additional arguments to `attach`.

The following attaches the combo box, radio buttons and their labels to the table:

```
tbl$attach(sizeLabel, left.attach=0,1, top.attach=0,1,
           xoptions = c("expand", "fill"), yoptions="")
tbl$attach(sizeCombo, left.attach=1,2, top.attach=0,1,
           xoptions="fill", yoptions="")
##
tbl$attach(diagLabel, left.attach=0,1, top.attach=1,2,
           xoptions = c("expand", "fill"),
           yoptions=c("expand", "fill"))
##
tbl$attach(diagAlign, left.attach=1,2, top.attach=1,2,
```

```
            xoptions=c("expand", "fill"), yoptions = "")
##
tbl$attach(submitBox, left.attach=1,2, top.attach=2,3,
            xoptions="", yoptions=c("expand", "fill"))
```

The labels are allowed to expand and fill in the x direction, because correct alignment, to the right, requires them to have the same size. The combo box is instructed to fill its space, as it would otherwise be undesirably small, due to its short menu items.

One can add spacing to the right of cells in a particular row or column. Here we add 5 pixels of space to the right of the label column:

```
tbl$setColSpacing(0, 5)
```

We complete the example by placing the table into a window:

```
w <- gtkWindow(show=FALSE)
w['border-width'] <- 14
w$setTitle("GtkTable Example")
w$add(tbl)
```

3

RGtk2: Basic Components

In this Chapter we cover many of the basic controls of GTK+.

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") )
w$show()
```

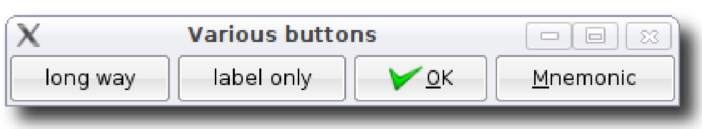


Figure 3.1: Various buttons

A `GtkButton` is simply a clickable region on the screen that is rendered 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 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.

Passing the `stock.id` argument to `gtkButton` will use decorations associated with a so-called stock identifier, see Section 3.2. 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 (see below) and language translation. The available stock identifiers are listed by `gtkStockListIds`.

The `gtkButtonNewWithMnemonic` constructor creates a button with a mnemonic. A mnemonic is a key press that will activate the button and is indicated by prefixing the character with an underscore. In our example, we pass the string "_Mnemonic", so pressing Alt-M will effectively press the button.

Signals The `clicked` signal is emitted when the button is clicked with the mouse, when the associated mnemonic is pressed 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$add(b)
ID <- gSignalConnect(b,"button-press-event",   # just mouse
                     f = function(w,e,data) {
                       print(e$getButton())    # which button
                       return(FALSE)           # propagate
                     })
ID <- gSignalConnect(b,"clicked",              # keyboard too
                     f = function(w,...) {
                       print("clicked")
                     })
```

As buttons are intended to call an action immediately after being clicked, it is advisable to make them insensitive to user input when the action is not possible. For example, we set our button to be insensitive through:

```
b$setSensitive(FALSE)
```

Windows often have a default action. For example, if a window contains a form, the default action submits the form. If a button executes

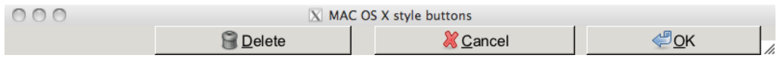


Figure 3.2: Example using stock buttons with extra spacing added between the `delete` and `cancel` buttons.

the default action for the window, the button 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.) The `GtkDialog` widget and its derivatives facilitate the use of buttons in this manner, see Section 2.3.

If the action that a button initiates is to be represented elsewhere in the GUI, say a menubar, then a `GtkAction` object may be appropriate. Action objects are covered in Section 5.5.

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, while potentially dangerous buttons are separated from the rest by 24 pixels, as per the Apple human interface guidelines.

GTK+ provides the widget `GtkHButtonBox` for organizing buttons in a manner consistent across an application. However, the default layout modes would not yield the desired spacing. As such, we will illustrate how to customize the spacing. We assume that our parent container, `hbox`, is a horizontal box container.

We include standard buttons, so we 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 specify the padding as we pack the widgets into the box, from right to left, with `packEnd`:

```
hbox$packEnd(ok, padding=0)
hbox$packEnd(cancel, padding=12)
hbox$packEnd(delete, padding=12)
hbox$packEnd(gtkLabel(""), expand=TRUE, fill=TRUE) # a spring
```

The padding occurs to the left and right of the child. The `ok` button is given no padding. The `cancel` button is packed with 12 pixels of spacing, which separates it from the `ok` button. Recognizing the `delete` button as potentially irreversible, we add 12 pixels of separation between it and the `cancel` button, for a total of 24 pixels. The blank label pushes the buttons against the right side of the box. We instruct the `ok` button to grab focus, so that it becomes the default button:

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

```
ok$grabFocus()
```

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, which takes the label text as its first argument. 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 most text in a GTK+ GUI is ultimately displayed by `GtkLabel`, there are many formatting options available. This example demonstrates a sample of these (Figure 3.3)

```
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 3.4. 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 2.2.

Pango markup 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. Hyperlinks are possible with `<a>`, as of version 2.18, and similar logic to `browseURL` is implemented for launching a web browser. Connect to the `activate_link` signal to override it. 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.

For efficiency reasons `GtkLabel` does not receive any input events. It lacks an underlying `GdkWindow`, meaning that there are no window system resources allocated for receiving the events. Thus, to make a label interactive, one must first embed it within a `GtkEventBox`, which provides the `GdkWindow`.

Images

It is often said that a picture can be worth a thousand words. Applying this to a GUI, good images can be worth a thousand pixels, as they can compactly represent ideas and actions. `GtkImage` is the widget that displays images. The constructor `gtkImage` creates 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.

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)
hbox <- gtkHBox(); w$add(hbox)
w$showAll()
```

The size of the image is taken as the size allocated to the box `hbox`. 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 <- hbox$getAllocation()$allocation
width <- theSize$width; height <- theSize$height
```

We create a `GdkPixmap` of the correct dimensions and initialize an R graphics device that targets the pixmap. A simple histogram is then plotted using base R graphics.

```
require(cairoDevice)
pixmap <- gdkPixmap(drawable = NULL,
                    width = width, height = height, depth = 24)
asCairoDevice(pixmap)
hist(rnorm(100))
```

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

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

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.

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 3 are:

```
head(unlist(gtkStockListIds()), n=3)
```

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

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 a stock identifier.

3.3 Input controls

Text entry

The widgets explained thus far are largely static. For example, GTK+ does not yet support editable labels. GTK+ has two different widgets for editing text. One is optimized for multi-line text documents, the other for single line entry. We will discuss complex multi-line text editing in Section 4.6. For entering a single line of text, the `GtkEntry` widget is appropriate:

```
e <- gtkEntry()
```

The `text` property stores the text. This can be set with the method `setText` and retrieved with `getText`. When the user has committed an entry, e.g. by pressing the enter key, the `activate` signal is emitted. Here we connect to this signal to obtain the entered text upon activation:

```
gSignalConnect(e, "activate", function() {
  message("Text entered: ", e$getText())
})
```

Sometimes the length of the text needs to be constrained to some number of characters. The `max` argument to `gtkEntry` specifies this, but that usage is deprecated. Instead, one should call `setMaxLength`.

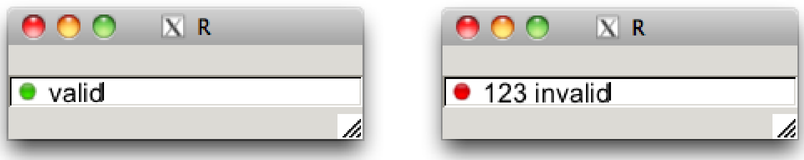


Figure 3.4: Illustration of adding an icon to a `GtkEntry` instance to indicate if the text entered is valid or not

The `GtkEditable` interface Editing text programmatically relies on the `GtkEditable` interface, which `GtkEntry` implements. The method `insertText` inserts text before a position specified by a 0-based index. The return value is a list with the component `position` indicating the position *after* the new text. The `deleteText` method deletes text between two positions.

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`.

Advanced `GtkEntry` features `GtkEntry` has a number of features beyond basic text entry, including: completion, buffer sharing, icons, and progress reporting. We discuss completion in Section 4.4 and shared buffers in Section 4.5. The progress reporting API, introduced with version 2.16, is virtually identical to that of `GtkProgressBar`, introduced in Section 3.4. We treat icons here. This feature has been present since version 2.16.

One can set an icon on an entry from a `GdkPixbuf`, stock ID, icon name, or `GIcon` (Figure 3.4). Two icons are possible, one at the beginning (`primary`) and one at the end (`secondary`). A common use would be to place a search icon in an entry widget, were it used for searching. In our example below, an entry might listen to its input and update its icon to indicate whether the entered text is valid (in this case, consisting only of letters):

```
validatedEntry <- gtkEntry()
gSignalConnect(validatedEntry, "changed", function(entry) {
  text <- entry$getText()
  if (nzchar(gsub("[a-zA-Z]", "", text))) {
    entry$setIconFromStock("primary", "gtk-no")
    validatedEntry$setIconTooltipText("primary",
                                      "Only letters are allowed")
  } else {
    entry$setIconFromStock("primary", "gtk-yes")
    validatedEntry$setIconTooltipText("primary", NULL)
```

```
  }
})
validatedEntry$setIconFromStock("primary", "gtk-yes")
```

We add a tooltip on the error icon to indicate the nature of the problem to the user. Icons can also be made clickable and used as a source for drag and drop operations.

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 one aspect of the application state. The simplest type of state variable is binary (boolean/logical) and is usually proxied by a `GtkCheckButton`.

A `GtkCheckButton` is constructed by `gtkCheckButton`:

```
cb <- gtkCheckButton("Option")
```

The state of the binary variable is represented by the `active` property. We check our button:

```
cb['active']
```

```
[1] FALSE
```

```
cb['active'] <- TRUE
```

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:

```
gSignalConnect(cb, "toggled", function(x) {
  message("Button is ", ifelse(x$active, "active","inactive") )
})
```

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 groups

GTK+ provides two widgets 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 and only one button in a group may be active at once.

Example 3.6: Basic radio button usage
When we construct a radio button, we need to add it 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 the group list is passed to the constructor, the newly created button is added to the group:

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

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.

We add each button to a vertical box:

```
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:

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

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

The `toggle` signal is emitted when a button is toggled. We need to connect a handler to each button:

```
sapply(radiogp, gSignalConnect, "toggled",      # connect each
       f = function(w, data) {
         if(w['active']) # set before callback
           message("clicked", w$getLabel(),"\n")
       })
```

Example 3.7: Radio group via a `FromWidget` constructor
In this example, we illustrate using the `gtkRadioButtonNewWithLabelFromWidget` function to add new buttons to the group:

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

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. Thus, we need to call `rev` to reverse the list before packing the widgets into the box.

Combo boxes

The combo box is a more space efficient alternative to radio buttons and is better suited when there are a large number of options. A basic, text-only `GtkComboBox` is constructed by `gtkComboBoxNewText`. In Section 4.3 we will discuss combo boxes that are based on an external data model.

We can construct and populate a simple combo box with:

```
combo <- gtkComboBoxNewText()
sapply(c("two.sided", "less", "greater"), combo$appendText)
```

The index of the currently active item is stored in the `active` property. The index, as usual, is 0-based, and a value of -1 indicates that no value is selected (the default):

```
combo['active']
```

```
[1] -1
```

The `getActiveText` method retrieves the text shown by the basic combo box.

When the active index changes, the `changed` signal is emitted. The handler then needs to retrieve the active index:

```
gSignalConnect(combo, "changed",
               f = function(w, ...) {
                 if(w$getActive() < 0)
                   message("No value selected")
                 else
                   message("Value is", w$getActiveText())
               })
```

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

Example 3.8: Using one combo box to populate another

The goal of this example is to populate a combo box of variables whenever a data frame is selected in another. We use two convenience functions from the `ProgGUIInR` package to find the possible data frames, and for a data frame to find its variables.

We create the two combo boxes and the enclosing window:

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

Our layout uses boxes. To add a twist, we will hide our variable combo box 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 combo boxes. When a data frame is selected, we first clear out the variable combo box and then populate it:

```
sapply(avail_dfs(), df_combo$appendText)
df_combo$setActive(-1)
#
gSignalConnect(df_combo, "changed", function(w, ...) {
  var_combo$getModel()$clear()
  sapply(find_vars(w$getActiveText()),  var_combo$appendText)
  g2$show()
})
```

An extension of `GtkComboBox`, `GtkComboBoxEntry`, replaces the main button with a text entry. This supports the entry of arbitrary values, in addition to those present in the menu.

Sliders and Spin buttons

The slider widget and spin button widget allow selection from a regularly spaced, semi-continuous list of values. Both have their possible values for selection determined by an instance of `GtkAdjustment`, which is used to represent ranges that have an upper and lower bound with step and page increments. This adjustment may be specified to the constructor, or more frequently will be created by the widget after an appropriate specification of the range.

Sliders Sliders are implemented by `GtkScale` with constructors `gtkHScale` and `gtkVScale`, the difference being the orientation.

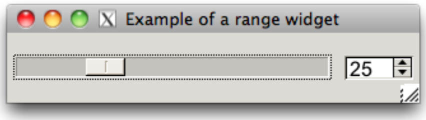


Figure 3.5: A range widget with coordinated slider and spin box sharing the same `GtkAdjustment` instance

These constructors have arguments `min`, `max` and `step` to specify the range, if an adjustment is not specified.

The `value` property stores the currently selected value. When this is changed, the `value-changed` signal is emitted.

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`.

In Example 3.12 we show how a slider can be used to update a 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 through the `min`, `max`, and `step` arguments. The argument `digits` is used to configure how many digits are displayed and `climb.rate` can adjust how fast the display changes when the button is held depressed.

As with `GtkScale` the `value` property holds the state and the `value-changed` signal is emitted when this changes.

A spin button has a few additional features. 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 whether the sequence will "wrap" around at the bounds

Example 3.9: A range widget

This example shows how to make a range widget that combines both the slider and spinbutton to choose a single number (Figure 3.5). Such a widget is useful, 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 propagate without effort to the other.

We name our scale parameters according to the corresponding arguments to the `seq` function:

```
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 then 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, but not the spin button, set to expand into the available space.

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

.

3.4 Progress reporting

Progress bars

It is common to use a progress bar to indicate the progress of a long running computation. This implemented by GtkProgressBar. A text label describes the current operation, and the progress bar communicates the fraction completed:

```
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 indefinite activity by periodically pulsing the bar:

```
pb$pulse()
```

Spinners

Related to a progress bar is the GtkSpinner widget, which is a graphical heartbeat to assure the user that the application is still alive during long-running operations. Spinners are commonly found in web browsers. The basic usage is straightforward:

```
spinner <- gtkSpinner()
spinner$start()
spinner$stop()
```

3.5 Wizards

The `GtkAssistant` class provides a wizard widget for GTK+. The simplest setup is that one adds pages to the assistant object and they are navigated in a linear manner. In our example, we override this.

Wizard pages have a certain type which must be declared. These are enumerated in `GtkAssistantPageType` and set by `setPageType`. The last page must be of type `"confirm"`, `"summary"`, or `"progress"`. Each wizard page has a content area and buttons. As well, each page in the assistant object has an optional side image, header image and/or page title that may be customized. The buttons allow the user to navigate through the wizard. The content area of a wizard page is simply an instance of class `GtkWidget` (e.g., some container) and are added to the assistant through the `appendPage`, `insertPage`, or `prependPage` methods. Pages are referred to by the `GtkWidget` object or their page index, 0-based. The `forward` button on a page must be made sensitive by calling `setPageComplete` with the widget and logical value.

Signals The `cancel` button emits a `cancel` signal that can be connected to for destroying the wizard widget. The `apply` signal is emitted on a page change. The `prepare` signal is emitted just before a page is made visible, which is needed to create the dynamically generated pages in our example.

Example 3.10: An `install.packages` wizard

This example wraps the `install.packages` function into a wizard with different pages for the (optional) selection of a CRAN mirror, the selection of the package to install, the configuration options provided and feedback. In general, wizards are quite common for software installation.

We begin by defining our assistant and connecting to its `cancel` signal.

```
asst <- gtkAssistant(show=FALSE)
asst$setSizeRequest(500, 500)
gSignalConnect(asst, "cancel", function(asst) asst$destroy())
```

Our pages will be computed dynamically. here we populate the pages using box containers and specify their respective types.

```
pages <- lapply(1:5, gtkVBox, spacing=5, homogeneous=FALSE)
```

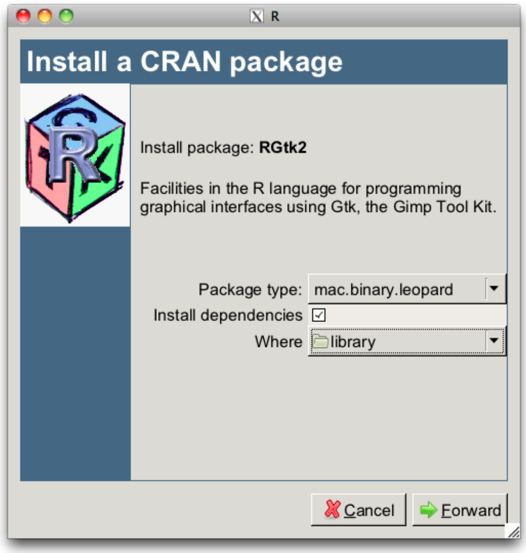


Figure 3.6: An installation wizard programmed using `GtkAssistant`. This is page 4 which allows options for a call to `install.packages` to be configured.

```
page_types <- c("intro", rep("confirm",3), "summary")
sapply(pages, gtkAssistantAppendPage, object=asst)
sapply(pages, gtkAssistantSetPageType, object=asst,
       type=page_types)
```

We customize each page with a side logo.

```
image <- gdkPixbuf(filename = imagefile("rgtk-logo.gif"))[[1]]
sapply(pages, gtkAssistantSetPageSideImage, object=asst,
       pixbuf=image)
```

When a page is about to be called we check and see if it has any children, if not we call a function to create the page. These functions are stored in a list so that they can be called by page index.

```
populatePage <- list()
gSignalConnect(asst, "prepare", function(a, w, data) {
  page_no <- which(sapply(pages, identical, w))
  if(!length(w$getChildren()))
    populatePage[[page_no]]()
})
```

Although we don't show how to create the CRAN selection page (cf. Example 4.5 for a similar construction) we call `setForwardPageFunc` to set a function that will skip this page if it is not needed. This function simply returns an integer with the next page number based on the last one.

```
asst$setForwardPageFunc(function(i, data) {
  ifelse(i == 0 && have_CRAN(), 2L, as.integer(i + 1))
}, data=NULL)
```

We have a few script globals that allow us to pass data between pages.

```
CRAN_package <- NA
install_options <- list() #type, dependencies, lib
```

We now show how some of the pages are populated. The initial screen is just a welcome and simply shows a label.

```
populatePage[[1]] <- function() {
  asst$setPageTitle(pages[[1]], "Install a CRAN package")
  pages[[1]]$packStart(l <- gtkLabel())
  pages[[1]]$packStart(gtkLabel(), expand=TRUE) # a spring

  l$setMarkup(paste(
       "<span font='x-large'>Install a CRAN package</span>",
       "This wizard will help install a package from",
       "<b>CRAN</b>. If you have not already specified a",
       "CRAN repository, you will be prompted to do so.",
       sep="\n"))
  asst$setPageComplete(pages[[1]], TRUE)
}
```

We skip showing the pages to select a CRAN site and a package, as they are based on the forthcoming `GtkTreeView` class. On the fourth page (cf. Figure 3.6 for a realization) is a summary of the package taken from CRAN and a chance for the user to configure a few options for the `install.packages` function.

```
populatePage[[4]] <- function() {
  asst$setPageTitle(pages[[4]], "Install a CRAN package")
  ##
  get_desc <- function(pkgname) {
    o <- "http://cran.r-project.org/web/packages/%s/DESCRIPTION"
    x <- readLines(sprintf(o, pkgname))
    f <- tempfile(); cat(paste(x, collapse="\n"), file=f)
    read.dcf(f)
  }
  desc <- get_desc(CRAN_package)
  #
  l <- gtkLabel()
  l$setLineWrap(TRUE)
  l$setWidthChars(40)
```

```
l$setMarkup(paste(
  sprintf("Install package: <b>%s</b>", desc[1,'Package']),
  "\n",
  sprintf("%s", gsub("\\n", " ", desc[1,'Description'])),
  sep="\n"))

pages[[4]]$packStart(l)
##
tbl <- gtkTable()
pages[[4]]$packStart(tbl, expand=FALSE)
pages[[4]]$packStart(gtkLabel(), expand=TRUE)

##
combo <- gtkComboBoxNewText()
pkg_types <- c("source", "mac.binary", "mac.binary.leopard",
               "win.binary", "win64.binary")
sapply(pkg_types, combo$appendText)
combo$setActive(which(getOption("pkgType") == pkg_types) - 1)
gSignalConnect(combo, "changed", function(w, ...) {
  cur <- 1L + w$getActive()
  install_options[['type']] <<- pkg_types[cur]
})
tbl$attachDefaults(gtkLabel("Package type:"), 0, 1, 0, 1)
tbl$attachDefaults(combo, 1, 2, 0, 1)

##
cb <- gtkCheckButton()
cb$setActive(TRUE)
gSignalConnect(cb, "toggled", function(w) {
  install_options[['dependencies']] <<- w$getActive()
})
tbl$attachDefaults(gtkLabel("Install dependencies"),
                   0, 1, 1, 2)
tbl$attachDefaults(cb, 1, 2, 1, 2)

##
fc <- gtkFileChooserButton("Select a directory...",
                           "select-folder")
fc$setFilename(.libPaths()[1])
gSignalConnect(fc, "selection-changed", function(w) {
  dir <- w$getFilename()
  install_options[['lib']] <<- dir
})
tbl$attachDefaults(gtkLabel("Where"), 0, 1, 2, 3)
tbl$attachDefaults(fc, 1, 2, 2, 3)
## align labels to right and set spacing
sapply(tbl$getChildren(), function(i) {
```

```
    widget <- i$getWidget()
    if(is(widget, "GtkLabel"))  widget['xalign'] <- 1
  })
  tbl$setColSpacing(0L, 5L)
  ##
  asst$setPageComplete(pages[[4]], TRUE)
}
```

Our last page, where the selected package is installed, would naturally be of type `progress`, but there is no means to interrupt the flow of `install.packages` to update the page. A real application would reimplement that. Instead we just set a message once the package install attempt is done.

```
populatePage[[5]] <- function() {
  asst$setPageTitle(pages[[5]], "Done")
  install_options$pkgs <- CRAN_package
  out <- try(do.call("install.packages", install_options),
             silent=TRUE)

  l <- gtkLabel(); pages[[5]]$packStart(l)
  if(!inherits(out, "try-error")) {
    l$setMarkup(sprintf("Package %s installed successfully",
                        CRAN_package))
  } else {
    l$setMarkup(paste(sprintf("Package %s failed to install",
                              CRAN_package),
                      paste(out, collapse="\n"),
                      sep="\n"))
  }

  asst$setPageComplete(pages[[5]], FALSE)
}
```

To finish we simply need to populate the first page and call the assistant's `show` method.

```
populatePage[[1]]()
asst$show()
```

3.6 Embedding R graphics

The package `cairoDevice` is an R graphics device based on the Cairo graphics library. It supports alpha-blending and antialiasing and reports user events through the `getGraphicsEvent` function. `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. For simply displaying graphics in a GUI, the `GtkDrawingArea` is the best choice.

This is the simplest usage:

```
library(cairoDevice)
device <- gtkDrawingArea()
asCairoDevice(device)
##
win <- gtkWindow(show=FALSE)
win$add(device)
win$showAll()
plot(mpg ~ hp, data = mtcars)
```

In the above, we create the `GtkDrawingArea`, coerce it to a Cairo-based graphics device, and then place it in a window. Example 2.4 goes further by embedding the drawing area into a scrolled window to support zooming and panning.

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 could then be composited with other images when displayed. Example 3.5 generates an icon by pointing the device to a pixmap. Finally, passing a `GtkPrintContext` to `asCairoDevice` allows printing R graphics through the GTK+ printing dialogs.

Example 3.11: Printing R graphics

This example will show how to use the printing support in GTK+ for printing an R plot.

A print operation is encapsulated by `GtkPrintOperation`:

```
printOp <- gtkPrintOperation()
```

A print operation may perform several different actions: print directly, print through a dialog, show a print preview and export to a file. Before performing any such action, we need to implement the rendering of our document into printed form. This is accomplished by connecting to the `draw-page` signal. The handler is passed a `GtkPrintContext`, which contains the target Cairo context. In general, one would call Cairo functions to render the document, which is beyond our scope. In this case, though, we can pass the context directly to `cairoDevice` for rendering the R plot:

```
gSignalConnect(printOp, "draw-page",
               function(x, context, page_nr) {
                 asCairoDevice(context)
                 plot(mpg ~ wt, data = mtcars)
               })
```

The final step is to run the operation to perform one of the available actions. In this example, we launch a print dialog:

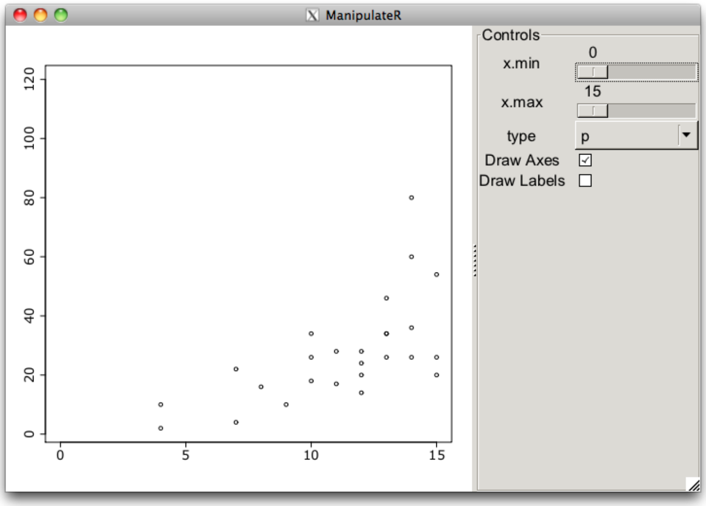


Figure 3.7: An implementation of RStudio's `manipulate` package in `RGtk2`

```
printOp$run(action = "print-dialog", parent = NULL)
```

When the user confirms the dialog, the `draw-page` handler is invoked, and the rendered page is sent to the printer.

Example 3.12: The `manipulate` package in `RGtk2`

The RStudio™ Workbench is an IDE for R that provides a similar interface whether run on any of its supported operating systems or through a web browser. Accompanying the workbench is an `R` package `manipulate` that provides a convenient means to create simple graphical interfaces for plotting. As RStudio leverages web technologies to render its widgets and there is no public interface, the package is not available for non-RStudio users. Too bad. This example shows how one can use `RGtk2` to provide a similar interface. In the example, we borrow liberally from the `manipulate` code, which is released under an AGPL license. Although we don't show the entire code here, the `ProgGUIinR` package contains it all.

The `manipulate` package uses environments to store state etc. Here we use reference classes, as they allow for a more structured programming interface.

A typical use of `manipulate` is along the lines of (Figure 3.7) the following example from the `manipulate` help pages:

```
manipulate(## expression
           plot(cars, xlim = c(x.min, x.max), type = type,
                axes = axes, ann = label),
```

```
           ## controls
           x.min = slider(0,15),
           x.max = slider(15,30, initial = 25),
           type = picker("p", "l", "b", "c", "o", "h", "s"),
           axes = checkbox(TRUE, label="Draw Axes"),
           label = checkbox(FALSE, label="Draw Labels")
           )
```

The first argument is an expression, possibly containing parameters, that produces a plot. The other arguments create widgets that control the parameter values in the plotting expression. There are three basic controls: a slider, a picker (combo box), and a check box. As one can glean, the constructors have a terse but simple set of arguments. A main task ahead will be mapping these controls to one of GTK+'s widgets.

For now, we begin by defining our `Manipulate` class to have two properties, one to hold the expression and the other to hold a list of controls.

```
Manipulate <- setRefClass("Manipulate",
                          fields=list(
                            .code="ANY",
                            .controls="list"
                            ))
```

When one of the controls is changed, the entire plot will be redrawn. The following handler will be assigned to each control. Its environment contains the main properties so the evaluation can be done as expected. Note that each control is expected to provide a `get_value` method.

```
Manipulate$methods(
           get_values=function() {
             "Get widget values as list"
             sapply(.controls, function(i) i$get_value(),
                    simplify=FALSE)
           },
           change_handler=function(...) {
             "Evaluate code with current values"
             values <- get_values()
             result <- withVisible(eval(.code, envir=values))
             if (result$visible) {
               eval(print(result$value))
             }
           })
```

The `execute` method is called after initialization to setup the GUI. We use a `GtkHPaned` instance to allow the user to adjust the space between the graphic device and the controls frame. Each control is expected to have a `make_gui` interface.

```
Manipulate$methods(
```

```
            execute=function() {
              "Make the GUI"
              w <- gtkWindow(show=FALSE)
              w$setTitle("ManipulateR")
              ## Set up graphic device
              paned <- gtkHPaned()
              w$add(paned)
              device <- gtkDrawingArea()
              device$setSizeRequest(480, 480)
              asCairoDevice(device)
              paned$add(device)
              ## Controls frame
              f <- gtkFrame("Controls")
              control_table <- gtkTableNew()
              control_table$setHomogeneous(FALSE)
              control_table['column-spacing'] <- 10
              ## insert horizontal strut
              control_table$attach(strut <- gtkHBox(), 1,2,0,1,
                            xoptions="", yoptions="shrink")
              strut$setSizeRequest(75, -1)
              f$add(control_table)
              paned$add(f)
              ## add each control
              sapply(.controls, function(i) {
                i$make_gui(cont=control_table,
                           handler=.self$change_handler)
              })
              w$show()
              change_handler()                    # initial
            })
```

The `control_table` is used hold the respective controls. We added a strut to request a minimum width for the second column, as otherwise the slider controls can render too narrowly.

The initialize method calls a function provided by the `manipulate` package to to pick the controls out of the ... argument. The `validate_controls` method is not shown, but simply borrows code from the package to do some error checking, ensuring the controls are defined properly.

```
Manipulate$methods(
            initialize=function(code, ...) {
              controls <- resolveVariableArguments(list(...))
              initFields(.code=code,
                         .controls=controls)
              validate_controls()
              .self
            })
```

We now provide a constructor allowing access to our class.

```
manipulate <- function('_expr',...) {
  obj <- Manipulate$new(substitute('_expr'),...)
  obj$execute()
  }
```

There are three main controls, but perhaps more could be added. We give ourselves the flexibility to expand by creating a base class for a control that can be subclassed. We define the class below. The properties are `l`, to store a list of arguments (a legacy of the original code); `widget`, to store the widget; `label` to hold the label for the control; and `initial`.

```
ManipulateControls <- setRefClass("ManipulateControls",
                        fields=list(
                          l="list",
                          widget = "ANY",
                          label="character",
                          initial="ANY"
                          ))
```

There main interface includes three methods: `validate_inputs` (to ensure the control is defined properly) and the previously noted `get_value` and `make_gui` (defined separately).

```
ManipulateControls$methods(
            validate_inputs = function(...) {
              "Validate input code"
            },
            get_value=function(...) {
              "Get value of widget"
            })
```

The `make_gui` method has two tasks: to define the widget instance and to add the widget to the GUI. This is done in the base class. The label and widget are added as a row to a `GtkTable` instance.

```
ManipulateControls$methods(make_gui=function(cont) {
            "Create widget, then add to table"
            ## cont a GtkTable instance
            nrows <- cont['n-rows']
            label_widget <- gtkLabel(label)
            label_widget['xalign'] <- 1
            cont$attach(label_widget, 0, 1, nrows, nrows+1,
                        xoptions="shrink", yoptions="shrink"
                        )
            cont$attach(widget, 1, 2, nrows, nrows+1,
                        xoptions=c("expand", "fill"),
                        yoptions="")
          })
```

The `slider` constructor just creates an instance of a soon to be defined sub-class of the `ManipulateControls` class. The arguments follow RStudio's.

```
slider <- function(min, max, initial=min, label=NULL,
                   step=-1, ticks=TRUE) {
  Slider$new(min, max, initial=initial, label=label,
             step=step, ticks=ticks)
}
```

The `Slider` class has no new properties:

```
Slider <- setRefClass("Slider",
                      contains="ManipulateControls")
```

The `initialize` method simply creates a list and sets some properties. This follows the setup of the original package.

```
Slider$methods(
       initialize=function(min, max, initial=min, label=NULL,
         step=-1, ticks=TRUE) {
           validate_inputs(min, max, initial, step, ticks)
           ## create slider and return it
           slider <- list(type = 0,
                          min = min,
                          max = max,
                          step = step,
                          ticks = ticks)
           initFields(l=slider, label=label, initial=initial)
           .self
         })
```

Our `make_gui` method basically defines the widget, turning the arguments of the constructor into those for the GTK+ widget. It then calls the same method from the superclass to lay it the widget. Here we define a slider and initialize it using the values in the list, `l`. The `handler` is the change handler passed in from a `Manipulate` instance.

```
Slider$methods(
       make_gui=function(cont, handler, ...) {
         widget <<- gtkHScale(min=l$min, max=l$max, step=l$step)
         widget$setValue(initial)
         gSignalConnect(widget, "value-changed", handler)
         callSuper(cont)
       },
       get_value=function() {
         as.numeric(widget$getValue())
       })
```

The `picker` and `checkbox` functions (and their classes) are similarly defined. For example, the `Checkbox` class, the three main methods are given by:

```
Checkbox$methods(
          initialize=function(initial=FALSE, label=NULL) {
            validate_inputs(initial, label)
            checkbox <- list(type = 2)
            initFields(l=checkbox, label=label, initial=initial)
            .self
          },
          make_gui=function(cont, handler, ...) {
            widget <<- gtkCheckButton() # no label
            widget$setActive(initial)
            gSignalConnect(widget, "toggled", handler)
            callSuper(cont)
          },
          get_value=function() widget['active']
          )
```

We don't provide a label to the check button, as one is provided in the table.

3.7 Drag and drop

A drag and drop operation is the movement of data from a source widget to a target widget. In GTK+ the source widget serializes the selected item as MIME data, and the destination interprets that data to perform some operation, often creating an item of its own. Our task is to configure the source and destination widgets, so that they listen for the appropriate events and understand each other. As a trivial example, we allow the user to drag the text from one button to another.

Initiating a drag

When a drag and drop is initiated, different types of data may be transferred. We need to define a target type for each type of data, as a `GtkTargetEntry` structure:

```
TARGET.TYPE.TEXT   <- 80                # our enumeration
TARGET.TYPE.PIXMAP <- 81
widgetTargetTypes <-
  list(text = gtkTargetEntry("text/plain", 0,
         TARGET.TYPE.TEXT),
       pixmap = gtkTargetEntry("image/x-pixmap", 0,
         TARGET.TYPE.PIXMAP))
```

The first component of GtkTargetEntry is the name, which is often a MIME type. The flags come next, which are usually left at 0, and finally we specify an arbitrary identifier for the target. We will only use the "text" target in this example.

We construct a button and call gtkDragSourceSet to instruct it to act as a drag source:

```
w <- gtkWindow(); w['title'] <- "Drag Source"
dragSourceWidget <-  gtkButton("Text to drag")
w$add(dragSourceWidget)
gtkDragSourceSet(dragSourceWidget,
       start.button.mask=c("button1-mask", "button3-mask"),
       targets=widgetTargetTypes[["text"]],
       actions="copy")
```

The start.button.mask, with values from GdkModifierType, indicates the modifier buttons that need to be pressed to initiate the drag. The allowed target is "text" in this case. The actions argument lists the supported actions, such as copy or move, from the GdkDragAction enumeration.

When a drag is initiated, we will receive the drag-data-get signal, which needs to place some data into the passed GtkSelectionData object:

```
gSignalConnect(dragSourceWidget, "drag-data-get",
               function(widget, context, sel, tType, eTime) {
                 sel$setText(widget$getLabel())
               })
```

If we had allowed the move action, we would also need to connect to drag-data-delete, in order to delete the data that was moved away.

Handling drops

In a separate window from the drag source button, we construct another button and call gtkDragDestSet to mark it as a drag target:

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

The signature is similar to that of gtkDragSourceSet, except for the flags argument, which indicates which operations, of the set motion, highlight and drop, GTK+ will handle with reasonable default behavior. Specifying all is the most convenient course, in which case we only need to implement the extraction of the data from the GtkSelectionData object. For a drop to occur, there must be a non-empty intersection between the targets passed to `gtkDragSourceSet` and those passed to `gtkDragDestSet`.

When data is dropped, the destination widget emits the `drag-data-received` signal. The handler is responsible for extracting the dragged data from `selection` and performing some operation with it. In this case, we set the text on the button:

```
gSignalConnect(dropTargetWidget, "drag-data-received",
       function(widget, context, x, y, sel, tType, eTime) {
         dropdata <- sel$getText()
         widget$setLabel(rawToChar(dropdata))
       })
```

The `context` argument is a `GdkDragContext`, containing information about the drag event. The `x` and `y` arguments are integer valued and represent the position in the widget where the drop occurred. The text data is returned by `getText` as a `raw` vector, so it is converted with `rawToChar`.

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 instances are constructed through `rGtkDataFrame`. 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.[1] 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 inherent 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
```

As with a data frame, 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

[1] As is proved with `tcltk`, where this is needed.

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 the `GtkCellRendererText` class. 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 = 2 - 1) # second column
```

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

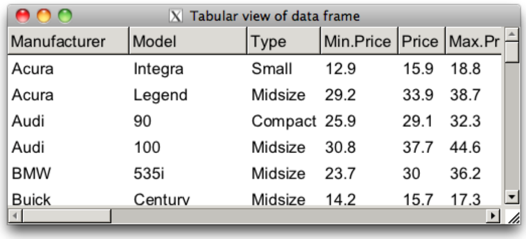


Figure 4.1: A `GtkTreeView` instance shown with a scrolled window

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, is not much different. 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,
       position=-1,
       title=colnames(model),
       cell=list(gtkCellRendererText()),
       text = seq_len(ncol(model)) - 1
       )
```

Figure 4.1 shows the view within a scrollable window:

```
w <- gtkWindow()
w$setTitle("Tabular view of data frame")
sw <- gtkScrolledWindow()
w$add(sw)
sw$add(view)
```

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 4.1. 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 table, 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 vector or a string, using `gtkTreePathNewFromIndices` or `gtkTreePathNewFromString`, respectively. For a flat table 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.

Iterators The second means of row indexing is through an iterator, `GtkTreeIter`, which is better suited for traversing a model. 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, in the sense that they are invalidated when their source is modified. An iterator is often updated by reference, behavior that is atypical in `R` programming.

While a tree path is an intuitive, transparent row index, an iterator 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 visit each row in sequence:

```
iter <- model$getIterFirst()
manufacturer <- character()
while(iter$retval) {
  manufacturer <- c(manufacturer, model$get(iter$iter, 0)[[1]])
  iter$retval <- model$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. This iterator design also prevents the use of the apply functions (`R`'s iterators), 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.

Converting between paths and iterators One can convert between paths and iterators. 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 final point: `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 4.1. 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. For example, we create a view and limit it to single selection:

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

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

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

```
[1] 21.4
```

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

To respond to a selection, connect to the changed signal on GtkTreeSelection. Upon a selection, this handler will print the selected values in the first column:

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

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

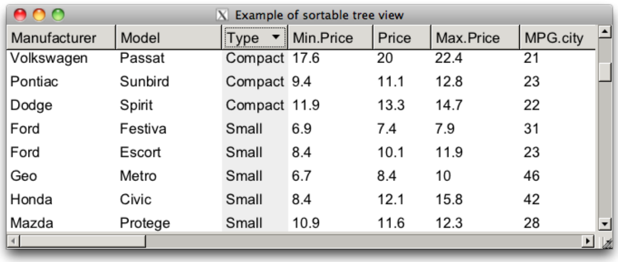
A common GUI feature is sorting a table widget by column. By convention, the user clicks on the column header to toggle sorting. GtkTreeView supports this interaction, although the actual sorting occurs in the model. Any model that implements the GtkTreeSortable interface supports sorting. RGtkDataFrame 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:



Manufacturer	Model	Type	Min.Price	Price	Max.Price	MPG.city
Volkswagen	Passat	Compact	17.6	20	22.4	21
Pontiac	Sunbird	Compact	9.4	11.1	12.8	23
Dodge	Spirit	Compact	11.9	13.3	14.7	22
Ford	Festiva	Small	6.9	7.4	7.9	31
Ford	Escort	Small	8.4	10.1	11.9	23
Geo	Metro	Small	6.7	8.4	10	46
Honda	Civic	Small	8.4	12.1	15.8	42
Mazda	Protege	Small	10.9	11.6	12.3	28

Figure 4.2: When a sortable model is passed to the treeview, one can click on the column headers to sort the data. The "Type" column has a custom sort function applied.

```
store <- rGtkDataFrame(Cars93)
sorted <- gtkTreeModelSortNewWithModel(store)
view <- gtkTreeView(sorted)
mapply(view$insertColumnWithAttributes,
       position=-1,
       title=colnames(store),
       cell=list(gtkCellRendererText()),
       text=seq_len(ncol(store)) - 1)
sapply(seq_len(ncol(store)), function(i)
       view$getColumn(i-1)$setSortColumnId(i-1))
```

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 a special sort for the `Type` of car can be implemented (Figure 4.2).

```
f <- function(model, iter1, iter2, user.data) {
  types <- c("Compact", "Small", "Sporty", "Midsize",
             "Large", "Van")
  column <- user.data
  val1 <- model$getValue(iter1, column)$value
  val2 <- model$getValue(iter2, column)$value
  as.integer(match(val1, types) - match(val2, types))
}
sorted$setSortFunc(sort.column.id=3-1, sort.func=f,
                   user.data=3 - 1)
```

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`.

This is the basic usage:

```
df <- Cars93
model <- rGtkDataFrame(cbind(df, .visible=rep(TRUE, nrow(df))))
filtered <- model$filter()
filtered$setVisibleColumn(length(df))          # 0-based
view <- gtkTreeView(filtered)
## Adjust filter
model[,".visible"] <- df$MPG.highway >= 30
```

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. To customize filtering, one can register a function with `setVisibleFunc`. The callback, given a row pointer, should return `TRUE` if the row passes the filter, see Example 4.4. A filter model may be treated as any other tree model, including attachment to a `GtkTreeView`.

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
[19] 19 20 21 22 23 24 25 26 27
```

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)
```

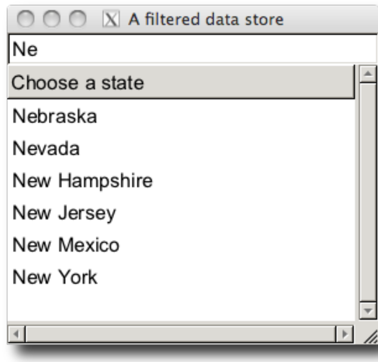


Figure 4.3: Example of a data store filtered by values typed into a text-entry widget.

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 4.3 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 4.3), 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 `cell-background` property) might be common to all rows in the column and thus is set explicitly, without use of an attribute:

```
renderer <- gtkCellRendererText()
renderer['cell-background'] <- "gray"
```

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 `GtkCellRendererText` displays 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. There are several other attributes that can changed. For example, we display right-aligned text in a Helvetica font:

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

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 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 the column index are not passed to the callback, unless one uses a closure or user data. 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) {
  i <- as.numeric(path) + 1
  j <- user.data$column
  model <- user.data$model
  model[i, j] <- newtext
}, data=list(model=store, column=1))
```

Before using editable cells to create a data frame editor, one should see if the editor provided by the gtkDfEdit in the RGtk2Extras package satisfies the requirements.

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 implemented by the setCursor method. The path argument takes a tree path instance, the column argument should be a tree view column object, and the flag start.editing indicates whether to initiate editing.

Example 4.3: Using a table to gather arguments

This example shows one way to gather arguments or options using an editable cell in a table, rather than a separate text entry widget. Tables can provide compact entry areas in a familiar interface.

For this example we collect values for arguments to the title function. We first create a data frame with the argument name and default value, along with some additional values:

```
opts <- c("main","sub","xlab","ylab", "line","outer")
df <- data.frame(option=opts,
           value=c("","","","","0","FALSE"),
           class=c(rep("character", 4), "integer", "logical"),
           edit_color=rep("gray95", 6),
           dirty = rep(FALSE, 6),
           stringsAsFactors=FALSE)
```

Unfortunately, we need to coerce the default values to character, in order to store them in a single column. We preserve the class in the class column, for coercion later. The edit_color and dirty columns are related to editing and explained later.

Now we create our model and configure the first column:

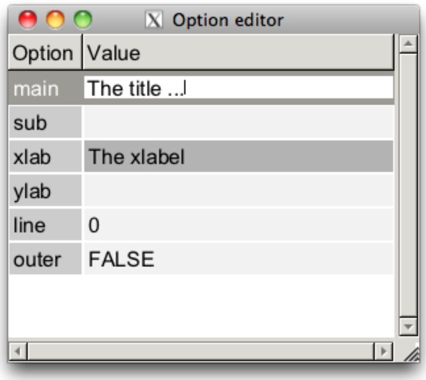


Figure 4.4: A tree view used to gather arguments for a call to `title`.

```
m <- rGtkDataFrame(df)
v <- gtkTreeView(m)
##
cr <- gtkCellRendererText()
cr['background'] <- 'gray80'
v$insertColumnWithAttributes(position=-1,
                             title="Option",
                             cell=cr,
                             text=1 - 1)
```

The first column has a special background color which we specify below, which indicates that the cells are not editable. The second column is editable and has a background color that is state dependent and indicates if a cell has been edited (The `xlab` column in Figure 4.4):

```
cr <- gtkCellRendererText()
cr['editable'] <- TRUE
v$insertColumnWithAttributes(position = -1,
                             title = "Value",
                             cell = cr,
                             text = 2 - 1,
                             background = 4 - 1
                             )
```

To attach the view to the model, we connect the cell renderer to the `edited` signal. Here we use the `class` value to format the text and then set the background color and `dirty` flag of the entry. The latter allows one to easily find the values which were edited.

```
gSignalConnect(cr, "edited", function(cr, path, new.text,
                                      user.data) {
  m <- user.data$model
```

```
  i <- as.numeric(path) + 1; j <- user.data$column
  m[i,j] <- as(as(new.text, m[i, 'class']), "character")
  m[i, 'dirty'] <- TRUE                        # mark dirty
  m[i, 'edit_color'] <- 'gray70'               # change color
}, data=list(model=m, column=2))
```

A simple window displays our GUI.

```
w <- gtkWindow(show=FALSE)
w['title'] <- "Option editor"
w$setSizeRequest(300,500)
sw <- gtkScrolledWindow()
w$add(sw)
sw$add(v)
w$show()
```

Implementing this into a GUI requires writing a function to map the model values into the appropriate call to the `title` function. The `dirty` flag makes this easy, but this is a task we do not pursue here. Instead we add a bit of extra detail by providing a tooltip.

Tooltips For this example, our function has built-in documentation. Below we use an internal function from the `helpr` package[2] to extract the description for each of the arguments. We leave this in a list, `descs`, for later lookup.

```
require(helpr, quietly=TRUE)
package <- "graphics"; topic <- "title"
rd <- helpr:::parse_help(helpr:::pkg_topic(package, topic),
                         package = package)
descs <- rd$params$args
names(descs) <- sapply(descs, function(i) i$param)
```

For many widgets, adding a tooltip is as easy as calling `setTooltipText`. However, it is more complicated in a tree view, as each cell should get a different tip. To add tooltips to the tree view we first indicate that we want tooltips, then connect to the `query-tooltip` signal to implement the tooltip:

```
v["has-tooltip"] <- TRUE
gSignalConnect(v, "query-tooltip",
       function(w, x, y, key_mode, tooltip, user.data) {
         out <- w$getTooltipContext(x, y, key_mode)
         if(out$retval) {
           m <- w$getModel()
```

[2] It is important to note that we are calling internal routines of a package still under active development, which in turn relies on volatile features of R. In general, such practice can lead to maintenance headaches. The purpose of this example is only to provide a natural demonstration of tooltips on a tree view.

```
            i <- as.numeric(out$path$toString()) + 1
            val <- m[i, "option"]
            txt <- descs[[val]]$desc
            txt <- gsub("code>","b>", txt)  # no code in Pango
            tooltip$setMarkup(txt)
            TRUE
          } else {
            FALSE                          # no tooltip
          }
        })
```

Within this callback we check if we have the appropriate context (we are in a row), then, if so, use the path to find the description to set in the tooltip. The descriptions use HTML for markup, but the tooltip only uses Pango. As the `code` tag is not PANGO, we change to a bold tag using `gsub`.

Combo and spin cell 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 setting a `GtkAdjustment` on the `adjustment` property.

The `changed` signal is emitted when an items is selected in the combo box. The spin cell renderer inherits the `edited` signal from `GtkCellRendererText`.

Pixbuf cell renderers To display an image in a cell, `GtkCellRendererPixbuf` is appropriate. The image is specified through one of these properties: `stock-id`, a stock identifier; `icon-name`, the name of a themed icon; or `pixbuf`, an actual `GdkPixbuf` object, holding an image in memory. Using a `list`, one can store a `GdkPixbuf` in a `data.frame`, and thus an `RGtkDataFrame`. This is demonstrated in the next example.

Example 4.4: A variable selection widget
This example shows how to create a GUI for selecting variables from a data frame. The GUI is based on two lists. The left one indicates the

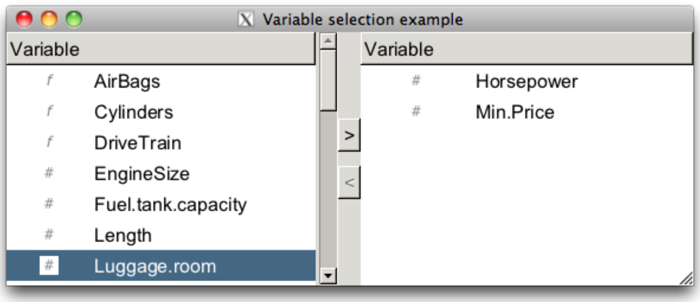


Figure 4.5: Illustration of an interface to select one or more variables. An icon is used in the table view to indicate the variable type.

variables that can be selected, and the right shows the variables that have been selected. An icon, indicating the variable type, is placed next to the variable name (Figure 4.5.) A similar mechanism is part of the SPSS model specification GUI of Figure **??**. For illustration purposes we use the `Cars93` data set.

```
df <- get(data(Cars93, package="MASS"))
```

First, we render an icon for each variable. The `make_icon` function from the `ProgGUIinR` package creates an icon as a `grid` object, which we render with `cairoDevice`:

```
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)
}
```

The two list views are based on the same underlying data model, which contains three columns: the variable name, the icon, and whether the variable has been selected. We construct the corresponding data frame and wrap it in a `RGtkDataFrame` instance:

```
mdf <- data.frame(Variables = I(sort(names(df))),
                  icon = I(sapply(df, make_icon_pixmap)),
                  selected = rep(FALSE, ncol(df)))
model <- rGtkDataFrame(mdf)
```

The first view shows only unselected variables, while the other is limited to selected variables. Thus, each view will be based on a different filter:

```
selectedFilter <- model$filter()
selectedFilter$setVisibleColumn(2)
unselectedFilter <- model$filter()
unselectedFilter$setVisibleFunc(function(model, iter) {
  !model$get(iter, 2)[[1]]
})
```

The selected filter is relatively easy to define, using `selected` as the visible column. For the unselected filter, we need to define a custom visible function that inverts the `selected` column.

Next, we create a view for each filter:

```
views <- list()
views$unselectedView <- gtkTreeView(unselectedFilter)
views$selectedView <- gtkTreeView(selectedFilter)
##
sapply(views, function(i) i$getSelection()$setMode('multiple'))
```

Each cell needs to display both an icon and a label. This is achieved by packing two cell renderers into the column:

```
make_view_column <- function() {
  vc <- gtkTreeViewColumn()
  vc$setTitle("Variable")
  cr <- gtkCellRendererPixbuf()
  vc$packStart(cr)
  vc$addAttribute(cr, "pixbuf", 1)
  cr <- gtkCellRendererText()
  vc$packStart(cr)
  vc$addAttribute(cr, "text", 0)
  vc
}
sapply(views, function(i) i$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:

```
## add to the gtkTreeView API for convenience
gtkTreeViewSelectedIndices <- function(object) {
  model <- object$getModel()          # Filtered!
  paths <- object$getSelection()$getSelectedRows()$retval
  out <- sapply(paths, function(i) {
    model$convertPathToChildPath(i)$toString()
  })
  if(length(out) == 0)
    integer(0)
```

```
  else
    as.numeric(out) + 1                                # 1-based
}
## does object have selection?
gtkTreeViewHasSelection <-
  function(obj) length(obj$selectedIndices()) > 0
```

Now we create the buttons and connect to the `clicked` signal. The handler moves the selected values to the other list by toggling the `selected` variable:

```
buttons <- list()
buttons$unselectButton <- gtkButton("<")
buttons$selectButton <- gtkButton(">")
toggleSelectionOnClick <- function(button, view) {
  gSignalConnect(button, "clicked", function (x) {
    print("clicked")
    ind <- view$selectedIndices()
    model[ind, "selected"] <- !model[ind, "selected"]
  })
}
mapply(toggleSelectionOnClick,
       button=buttons,
       view=rev(views))
```

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)
trackSelection <- function(button, view)
  gSignalConnect(view$getSelection(), "changed",
    function(x) button['sensitive'] <- view$hasSelection())
mapply(trackSelection, buttons, rev(views))
```

We now layout our GUI using a horizontal box, into which is packed the views and a box holding the selection buttons. The views will be scrollable, so we place them in scrolled windows:

```
w <- gtkWindow(show=FALSE)
w$setTitle("Select variables example")
w$setDefaultSize(600, 400)
g <- gtkHBox()
w$add(g)
## scrollwindows
scrolls <- list()
scrolls$unselectedScroll <- gtkScrolledWindow()
scrolls$selectedScroll <- gtkScrolledWindow()
mapply(gtkContainerAdd, object=scrolls, widget=views)
mapply(gtkScrolledWindowSetPolicy, scrolls,
       "automatic", "automatic")
```

```
## buttons
buttonBox <- gtkVBox()
centeredBox <- gtkVBox()
buttonBox$packStart(centeredBox, expand = TRUE, fill = FALSE)
centeredBox$setSpacing(12)
sapply(buttons, centeredBox$packStart, expand=FALSE)
##
g$packStart(scrolls$unselectedScroll, expand=TRUE)
g$packStart(buttonBox, expand=FALSE)
g$packStart(scrolls$selectedScroll, expand=TRUE)
```

Finally, we show the top-level window:

```
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 4.5: 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)
```

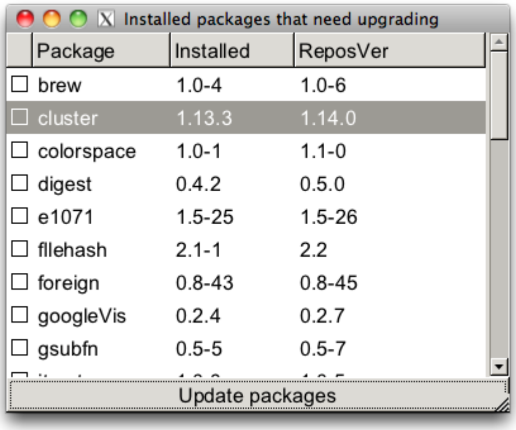


Figure 4.6: A GUI to select packages using checkboxes rendered with a `GtkCellRenererToggle` instance.

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 in one go:

```
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()
```

Progress cell renderers 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. For example,

```
cr <- gtkCellRendererProgress()
cr["value"] <- 50
```

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. One example is 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). Alternatively, one could preserve the integrity of the data and perform the conversion during rendering. This requires intercepting the model value before it is passed to the cell renderer.

In the specific case of `GtkTreeView`, it is possible to specify a callback that overrides this step. The callback, of type `GtkTreeCellDataFunc`, is passed arguments for the tree view column, the cell renderer, the model, an iterator pointing to the row in the model and, optionally, an argument for user data. The function is tasked with setting the appropriate attributes of the cell renderer. For example, this callback would 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 in the general case beyond `GtkTreeView`, one could implement a new type of `GtkCellRenderer`. See Section 6 for more details on extending GTK+ classes.

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 4.6: Defining a tree
Below, we create a tree based on the `Cars93` dataset, where the car models (`Model`) are organized by manufacturer (`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(car_model) {
    sibiter <- tstore$append(parent = piter$iter) # child
    if (is.null(sibiter$retval))
      tstore$setValue(sibiter$iter, column=0, value=car_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 (path `"0:0"`).

As shown in the above 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 a 0-based, column index.

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$iter, 0)[[1]])
    child$retval <- tstore$iterNext(child$iter)
  }
  iter$retval <- tstore$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.

Row manipulations Rows within a store can be rearranged using several methods. Call the swap method 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.

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 trigger the emission of the signals row-expanded and row-collapsed, respectively.

Example 4.7: 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 4.6 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 model information with the following:

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

Creating a 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
```

4.3 Model-based combo boxes

Basic combo box usage was discussed in Section 3.3; 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.

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.

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.

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. 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 combo box as an simple interface to the R help system. Along the way, we record the number of times the user searches for a page.

Our model for the combo box will be an RGtkDataFrame instance with three columns: a function name, a string describing the number of visits and an integer to record the number of visits. We create the combo box with this model using the first column for the text:

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

It is not currently possible to put tooltip information on the drop down elements of a combo box, as was done with a tree view. Instead, we borrow from popular web browser interfaces and add textual information about the number of visits to the drop down menu. This requires us to pack in a new cell renderer to accompany the original label 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"
```

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()$selectRegion(start=0,end=-1)
  help(value)
}
gSignalConnect(cb, "changed", f=function(w, ...) {
  if(cb$getActive() >= 0) {
    val <- w$getActiveText()
    callHelpFunction(w, val)
  }
})
```

When the user enters a new value in the entry, we need to check if we have previously accessed the item. If not, we add the value to our model.

```
gSignalConnect(cb$getChild(), "activate",
               f = function(cb, entry, ...) {
  val <- entry$getText()
  if(!any(val == cb$getModel()[,1, drop=TRUE])) {
    model <- cb$getModel()
    tmp <- data.frame(fname=val, visits="", novisits=0,
                      stringsAsFactors = FALSE)
    model$appendRows(tmp)
  }
  callHelpFunction(cb, val)
}, data=cb, user.data.first=TRUE)
```

We place this in a minimal GUI with a label:

```
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()
```

An alternative approach would be to use the completion support of `GtkEntry`, presented next, but we leave that as an exercise to the reader.

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.

We create an entry with a completion database:

```
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 we define our own match function:

```
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)
```

We get the string from the entry widget, not the passed value, as the latter has been standardized to lower case.

4.5 Sharing buffers between text entries

As of GTK+ version 2.18, multiple instances of GtkEntry can synchronize their text through a shared buffer. Each entry obtains its text from the same underlying model, a GtkEntryBuffer. Here, we construct two entries, with a shared buffer:

```
buffer <- gtkEntryBuffer()
entry1 <- gtkEntry(buffer = buffer)
entry2 <- gtkEntry(buffer = buffer)
entry1$setText("echo")
entry2$getText()
```

The change of text in "entry1" has been reflected in "entry2".

4.6 Text views

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 2.4).

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

```
tv <- gtkTextView()
sw <- gtkScrolledWindow()
sw$add(tv)
sw$setPolicy("automatic", "automatic")
##
w <- gtkWindow()
w['border-width'] <- 15
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 ..."
```

Adding text 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.

Properties By default, the text in a view is editable. This can be disabled through the `editable` property. Typically, one then sets the `cursor-visible` property to `"FALSE"` so that the cursor is hidden:

```
tv['editable'] <- FALSE
tv['cursor-visible'] <- FALSE
```

Formatting The text view supports several general formatting options. Automatic line wrapping is enabled through `setWrapMode`, which takes values from `GtkWrapMode`: `"none"`, `"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, see Section 4.7. 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. To set the font for specific regions, use text tags (see Section 4.7). The font is specified as a Pango font description, which may be generated from a string through `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.

4.7 Text buffers

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 `getModified` method, which returns `TRUE` if the buffer has changed. To clear this state, such as when a buffer has been saved to disk, one can pass `FALSE` to `setModified`.

In order to do more with a text buffer, such as retrieve a selection, or modify text attributes, one needs to become familiar with the two mechanism for referencing text in a buffer: iterators and marks. A text iterator is an opaque, transient pointer to a region of text, whereas a text mark specifies a location that remains valid across buffer modifications.

Iterators

In GTK+ a *text iterator* is the primary means of specifying a position in a buffer. As mentioned in Section 4.1, iterators are typically transient, in the sense that they are invalidated or updated by reference when their source is modified.

Several methods of the text buffer return iterators marking positions in the buffer. Iterators are returned as lists with two components: `iter`, which represents the actual C iterator object, and `retval`, a logical value indicating whether the iterator is valid. The beginning and end of the buffer are returned by the methods `getStartIter` and `getEndIter`. Both of these iterators are returned together in a list by the method `getBounds`. For example:

```
bounds <- buffer$getBounds()
bounds
```

```
$retval
NULL

$start
<pointer: 0x125b06de0>
attr(,"interfaces")
character(0)
attr(,"class")
[1] "GtkTextIter" "GBoxed"      "RGtkObject"

$end
<pointer: 0x125b06d90>
attr(,"interfaces")
character(0)
attr(,"class")
[1] "GtkTextIter" "GBoxed"      "RGtkObject"
```

The current selection is returned by the method `getSelectionBounds`, as a list of the same structure. If there is no selection, then the component `retval` will be `FALSE`, otherwise it is `TRUE`.

One can also obtain an iterator for a specific position in a document. 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. For example, we might request the seventh character of the first line:

```
iter <- buffer$getIterAtLineOffset(0, 6)
iter$iter$getChar()                    # unicode, not text
```

```
[1] 105
```

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.

Once we obtain an iterator, we typically enter a loop which performs some operation on the text at the iterator position and updates the iterator with each iteration. This requires retrieving the text to which an iterator refers. The character at the iterator position is returned by `getChar`. We obtain the first character in the buffer:

```
bounds$start$getChar()                    # unicode
```

```
[1] 76
```

To obtain the text between two text iterators, call the `getText` method on the left iterator, passing the right iterator as an argument:

```
bounds$start$getText(bounds$end)
```

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

The `insert` method will insert text at a specified iterator:

```
buffer$insert(bounds$start, "prefix")
```

The `delete` method will delete the text between two iterators. An important observation is that we always pass the actual iterator, i.e., the `iter` component of the list, to the above methods. Passing the original list would not work.

Next, we introduce the methods for updating an iterator. One can move an iterator forward or backward, stopping a certain type of landmark. Supported landmarks include characters (`forwardChar`, `forwardChars`, `backwardChar`, and `backwordChars`), words (`forwardWordEnd`, `backwardWordStart`), and sentences (`backwardSentenceStart` and `forwardSentenceEnd`). There are also various methods, such as `insideWord`, for determining the textual context of the iterator. Example 4.11 shows how some of the above are used, in particular how these methods update the iterator rather than return a new one.

Example 4.11: Finding the word that is clicked by the user

This example shows how one can find the iterator corresponding to a mouse click. In the callback we obtain the X and Y coordinates of the mouse button press event, find the corresponding iterator, and retrieve the surrounding word:

```
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$forwardWordEnd()
  val <- siter$getText(niter)
  print(val)                              # replace
  return(FALSE)                           # call next handler
})
```

Marks

A text mark tracks a position in the document that is relative to other text and is preserved across buffer modifications. One can think of a mark as an invisible object stuck between two characters. A mark corresponds to a specific position, like an iterator, except its gravity sets it either to the left or right of the character. An example is the text cursor, the position of which is represented by a mark.

Marks are identified by name. We retrieve the mark for the cursor, which is called `"insert"`:

```
insert <- buffer$getMark("insert")
```

To access the text at a mark, we need to find the corresponding iterator:

```
insertIter <- buffer$getIterAtMark(insert)$iter
bounds$start$getText(insertIter)
```

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

Marks have a gravity of `"left"` or `"right"`, with `"right"` being the default. If text is inserted at a mark with right gravity, then the mark is moved to the end of the insertion. A mark with left gravity would not be moved. This is intuitive if one relates it to the behavior of the text cursor, which has right gravity:

```
insertIter$getOffset()
```

```
[1] 36
```

```
buffer$insert(insertIter, "at insertion point")
buffer$getIterAtMark(insert)$iter$getOffset()
```

```
[1] 54
```

A custom mark is created with its name, gravity and position. We create one for the start of the document:

```
mark <- buffer$createMark(mark.name = "start",
                          where = buffer$getStartIter()$iter,
                          left.gravity = TRUE)
```

By setting `left.gravity` to `"TRUE"`, the iterator will not move when text is inserted.

Tags

Tags are annotations placed on specific regions of a text buffer. To create a tag, we call the `createTag` method, which takes an argument for each attribute to apply to the text. Here, we create three tags: one for bold text, one for italicized text and one for large text:

```
tag.b <- buffer$createTag(tag.name="bold",
                          weight=PangoWeight["bold"])
tag.em <- buffer$createTag(tag.name="em",
                           style=PangoStyle["italic"])
tag.large <- buffer$createTag(tag.name="large",
                              font="Serif normal 18")
```

Next, we associate the tags with one or more regions of text:

```
iter <- buffer$getBounds()
buffer$applyTag(tag.b, iter$start, iter$end)  # updates iters
buffer$applyTagByName("em", iter$start, iter$end)
```

Selection and the clipboard

The selection is defined by the text buffer as the region between the `"insert"` and `"selection_bound"` marks. While we could directly move the marks around, calling `selectRange` is more efficient and convenient. Here, we select the first word:

```
siter <- buffer$getStartIter()$iter
eiter <- siter$copy(); eiter$forwardWordEnd()
buffer$selectRange(siter, eiter)
```

`GtkTextBuffer` provides some convenience methods for interaction with the clipboard: `copyCliboard`, `cutClipboard` and `pasteClipboard`. To use these, we first need a clipboard object:

```
cb <- gtkClipboardGet()
```

We can then, for example, copy the selected text (the first word) and paste it at the end:

```
buffer$copyClipboard(cb)
buffer$pasteClipboard(cb,
            override.location = buffer$getEndIter()$iter,
            default.editable = TRUE)
```

The `default.editable` indicates that the pasted text should be editable. If we had passed `"NULL"`, to the `override.location` argument, the insertion would have occurred at the cursor.

Inserting non-text items

If desired, one can insert images and/or widgets into a text buffer. The method `insertPixbuf` will insert a `GdkPixbuf` object. The buffer will count the image as a character, although `getText` will obviously not return the image.

Arbitrary child widgets, like a button, can also be inserted. First, one must create an anchor in the text buffer with `createChildAnchor`:

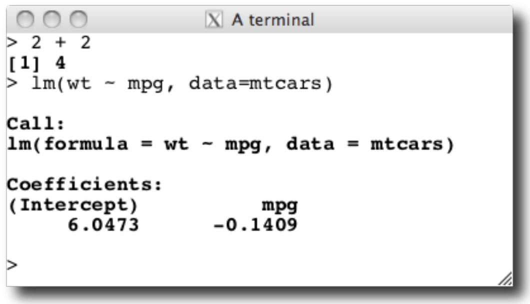


Figure 4.7: A basic `R` terminal implemented using a `gtkTextView` widget.

```
anchor <- buffer$createChildAnchor(buffer$getEndIter()$iter)
```

To add the widget, we call the text view method `addChildAtAnchor`:

```
b <- gtkButton("click me")
tv$addChildAtAnchor(b, anchor)
```

Example 4.12: A simple command line interface
This example shows how to create a simple command line interface with the text view widget (Figure 4.7). While few statistical applications will include a command line widget, 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$getBuffer()
font <- pangoFontDescriptionFromString("Monospace")
tv$modifyFont(font)                         # widget wide
```

We will use a few formatting tags, defined next. We do not need the tag objects as variables in the workspace, 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 a mark to indicate the beginning of a newly entered command, and another mark tracks the end of the buffer:

```
startCmd <- tb$createMark("startCmd", tb$getStartIter()$iter,
```

```
                            left.gravity=TRUE)
bufferEnd <- tb$createMark("bufferEnd", tb$getEndIter()$iter)
```

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:

```
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)
    })
}
```

We did not arrange to truncate large outputs, but that would be a nice addition. By passing in the tag name, we can specify whether this is normal output or an error message.

This next function uses the `startCmd` mark and the end of the buffer to extract the current command. The `"regex"` is used to parse multi-line commands.

```
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 using the `evaluate` package. It uses a hack (involving `grepl`) to distinguish between an incomplete command and a true syntax error.

```
require(evaluate)
evalCMD <- function(tv, cmd) {
  tb <- tv$getBuffer()
  out <- try(evaluate:::evaluate(cmd, .GlobalEnv), silent=TRUE)

  if(inherits(out, "try-error")) {
    ## parse error
    addOutput(tb, out, "cmdError")
  } else if(inherits(out[[2]], "error")) {
    if(grepl("end", out[[2]])) {        # a hack here
      addPrompt(tb, "continue", setMark=FALSE)
      return()
    } else {
      addOutput(tb, out[[2]]$message, "cmdError")
    }
  } else {
    addOutput(tb, out[[2]], "cmdOutput")
  }
  addPrompt(tb, "prompt", setMark=TRUE)
}
```

We arrange that the `evalCMD` command is called when the return key is pressed next. Other key bindings might also be of interest, such as one for tab completion.

```
gSignalConnect(tv, "key-release-event", f=function(w, e) {
  obj <- w$getBuffer()                  # w is textview
  keyval <- e$getKeyval()
  if(keyval == GDK_Return) {
    cmd <- findCMD(obj)                 # poss. character(0)
    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:

```
scrollViewport <- function(view, ...) {
  iter <- view$getBuffer()$getEndIter()$iter
  view$scrollToMark(bufferEnd, within.margin=0)
  return(FALSE)
}
gSignalConnect(tb, "changed", scrollViewport, data=tv,
               after = TRUE, user.data.first = TRUE)
```

5 RGtk2: Application Windows

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. Menu Bars and toolbars are the two most common widgets for organizing application-wide actions. An application also needs to report its status in an unobtrusive way. Thus, a typical application window contains, from top to bottom, a menubar, a toolbar, a large application-specific region, and a status bar. In this chapter, we will introduce actions, menus, toolbars and status bars 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 `GtkActi-`

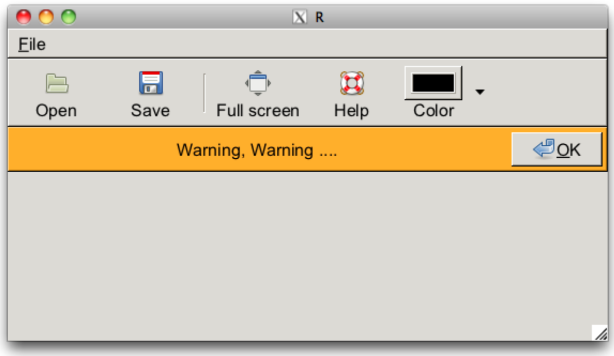


Figure 5.1: An application window mock up showing a menubar, toolbar, and info bar

vatable 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. By default, aesthetic properties like the label and stock-id are also inherited.

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",
                              stock.id="gtk-fullscreen")
```

```
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 menubar 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 5.5.

Menubars

We will first demonstrate the menubar, leaving the popup menu for later. Figure 5.1 will show a realization. The first step is to construct the menubar itself:

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

A menubar 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 menubar consists of 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 menubar:

```
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. We demonstrate with a "Save" 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$createMenuItem()
fileMenu$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 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:

```
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 `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`).

The above will popup a menu, but until we bind a callback to the `activate` signal on each item, nothing will happen when a menu item is selected. Below we supply a stub for sake of illustration:

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

We iterate over the children, avoiding the separator.

5.3 Toolbars

Toolbars are like menubars 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 menubar, toolbars are containers for tool items. Technically, an instance of `GtkToolItem` could contain any type of widget, yet toolbars typically represent actions with buttons. The `GtkToolButton` widget 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 icons. 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 menubar, 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$add(expander)
toolbar$childSet(expander, expand = TRUE)
```

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 $20x20$ 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$setImage(da)
  item
}
colorItems <- sapply(palette(), colorMenuItem)
colorMenu <- gtkMenu()
for (item in colorItems)
  colorMenu$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 Status reporting

Statusbars

In GTK+, a status bar is constructed through the `gtkStatusbar` function. Statusbars must be placed at the bottom of a top-level window by the programmer. In GTK+, a status bar 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 it is intended to be less obtrusive. Typically, an information bar raises from the bottom of the window, displaying a message, possibly with response buttons. It then fades away after a number of seconds. The focus is not affected, nor is the user interrupted. GTK+ provides the `GtkInfoBar` class for this purpose. The use is similar to a dialog: one places widgets into a content area, and listens to the `response` signal.

We create our info bar:

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

We call `setNoShowAll` to prevent the widget from being shown when `showAll` is called on the parent. Normally, an information bar is not shown until it has a message.

We will emit a warning message by adding a simple label with the text and specifying the message type as `warning`, from `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'])
```

This is similar to the dialog API: the appearance of the "Ok" button is defined by the stock ID `gtk-ok`, and the response ID will be passed to the `response` signal when the button is clicked. Our handle simply closes the bar:

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

Finally, we add the info bar to our main window and show it:

```
vbox$packStart(ib, expand = FALSE)
ib$show()
```

5.5 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. We illustrate through an example.

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

1. construct the UI manager,
2. specify in XML the layout of the menubars and toolbars,
3. define the actions in groups,
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

In this example, we show how to use a UI manager to create the menu and toolbars for a data frame editor, similar to, but with enhanced functionality, the one produced in some platforms by the `data.entry` function.

Our menubar and toolbar layout is expressed in 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.xml <- readLines(out <- textConnection('
<ui>
```

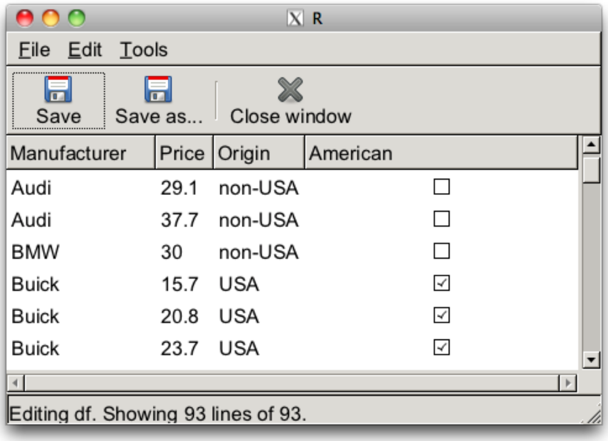


Figure 5.2: An instance of an editable data frame with menu and tool bars specified using an instance of `GtkUIManager`. This example, implements the command pattern to provide simple undo and redo functionality.

```
<menubar name="menubar">
  <menu name="FileMenu" action="File">
    <menuitem action="Save"/>
    <menuitem action="SaveAs" />
    <menu name="Export" action="Export">
      <menuitem action="ExportToCSV" />
      <menuitem action="ExportToSaveFile" />
    </menu>
    <separator />
    <menuitem name="FileQuit" action="CloseWindow" />
  </menu>
  <menu action="Edit">
    <menuitem name="EditUndo" action="Undo" />
    <menuitem name="EditRedo" action="Redo" />
    <menuitem action="ChangeColumnName" />
  </menu>
  <menu action="Tools">
    <menuitem action="Filter" />
    <menuitem action="Sort" />
  </menu>
</menubar>
<toolbar name="toolbar">
  <toolitem action="Save"/>
  <toolitem action="SaveAs"/>
  <separator />
  <toolitem action="CloseWindow"/>
```

```
  </toolbar>
</ui>'), warn=FALSE)
close(out)
```

We used indenting to show the nesting of the menus. For menus we see the use of menubars, menus and menu items. The menu and menu items have a corresponding action associated with them, which can provide a callback.

If `uimanager` is our `GtkUIManager` instance, then we can add this through the command:

```
id <- uimanager$addUiFromString(ui.xml)
```

Alternately, we could load the code from a file. The return value is an id that can be used to unmerge this part of the UI. The ability to merge and unmerge parts of the UI is one main attraction for using this framework, although we do not illustrate that here.

To define the actions, we can use lists. This list defines the file menu:

```
fileL <- list(## name, ID, label, accelerator, tooltip, callback
              list("File",NULL,"_File",NULL,NULL,NULL),
              list("Save", "gtk-save", "Save", "<ctrl>S",
                   "Save data to variable", fun),
              list("SaveAs", "gtk-save", "Save as...", NULL,
                   "Save data to variable", fun),
              list("Export", NULL, "Export", NULL, NULL, NULL),
              list("ExportToCSV", "gtk-export", "Export to CSV",
                   NULL, "Save data to CSV file", fun),
              list("ExportToSaveFile", "gtk-export",
                   "Export to save file", NULL,
                   "Save data to save() file", fun),
              list("CloseWindow", "gtk-close", "Close window",
                   "<ctrl>W", "Close current window", fun)
              )
```

Each item contains 6 pieces of information: a name (which we use in `fun` to call the appropriate method), a stock-id, a label, a keyboard accelerator a tooltip, and finally a callback for when the action is invoked.

We can add these items to an action group, along the lines of

```
ag <- gtkActionGroup("FileGroup")
ag$addActions(fileL)
```

We can then insert the action group into the UI manager:

```
uimanager$insertActionGroup(ag, 0)
```

The position (`0`) is used to determine which action will be called, when there is more than one with the same name.

We now place the UI manager into the GUI. The UI manager specification create widgets which we can retrieve through its `getWidget` method. The following code sketches the layout of the GUI:

```
w <- gtkWindow(show=FALSE)
##
vbox <- gtkVBox()
w$add(vbox)
##
menubar <- uimanager$getWidget("/menubar")
vbox$packStart(menubar, FALSE)
toolbar <- uimanager$getWidget("/toolbar")
vbox$packStart(toolbar, FALSE)
## ...
```

The `menubar` and `toolbar` widgets are referred to by their path which come from the names specified in the XML description separated by "/". So, in the definition. the line

```
<ui>
  <menubar name="menubar">
...
```

define the path `"/menubar"`, where `<ui>` is always the root element, and may be omitted from the path.

Finally, to connect the UI manager to the window, we add the keyboard accelerator group:

```
w$addAccelGroup(uimanager$getAccelGroup())
```

Figure 5.2 shows an illustration of the finished application. The full details are found in the code in our accompanying package `ProgGUIinR`.

Command pattern Now we discuss how the *command pattern* was implemented to provide a simple undo and redo feature to our editing. According to [1], the command pattern is used to encapsulate a request (method call) as an object. A basic command object has just one method, `execute`. Any needed parameters are stored in the object as properties. The command pattern has GUI-related applications beyond the undo/redo stack, including the action objects (i.e., instances of `GtkAction`) that are managed by `GtkUIManager`.

For our implementation of the undo/redo stack, we use a reference class with fields:

```
Command <- setRefClass("Command",
```

[1] Eric T Freeman; Elisabeth Robson; Bert Bates; Kathy Sierra. *Head First Design Patterns*. O'Reilly Media, Inc, October 25, 2004.

```
                    fields=list(
                      receiver="ANY",
                      meth="character",
                      params="list",
                      old_params="list"
                      ))
```

The receiver is the object that receives the method call. For a simple function call, this could be the environment enclosing the function. The `meth` property is the name of the method, and `params` is a list of parameters. With these we define the main methods:

```
Command$methods(
                initialize=function(receiver, meth, ...) {
                  l <- list(...)
                  initFields(receiver=receiver, meth=meth,
                             params=l, old_params=l)
                  callSuper()
                },
                execute=function(args) {
                  do.call(call_meth(meth, receiver), args)
                })
```

Notice we pass in the arguments to our `execute` method, rather than use those in the property `params`. This allows us to implement the `do` and `undo` methods in a similar manner:

```
Command$methods(
                do=function() {
                  out <- execute(params)
                  old_params$value <<- out
                },
                undo=function() execute(old_params)
                )
```

This assumes the method executed can return a value which can be used to reverse the call. If a method call is not so straightforward to reverse, one needs only subclass the `Command` call and provide a new `undo` method.

A simple illustration might be:

```
x <- 1
set_x <- function(value) {
  old <- x
  x <<- value
  old
}
cmd <- Command$new(.GlobalEnv, "set_x", value=2)
cmd$do(); x
```

```
[1] 2
```

```
cmd$undo();
```

```
x
```

```
[1] 1
```

In our example, we create a stack of commands to keep track of what was done. This stack has methods `add`, `undo` and `redo`, each calling the `do` or `undo` method of the appropriate command in the stack.

The first command we add to the stack is the setting of a column name on a data frame:

```
cmd <- Command$new(df_model, "set_col_name", j=j, value=value)
command_stack$add(cmd)
```

To explain, `df_model` is an instance of a yet-to-be-defined reference class defining a data model for the data frame being edited, and `j` and `value` are determined by a dialog called before the command is created. The point is, the method call for the `df_model` object is encapsulated along with the needed parameters (a column number and new name) and then added to the command stack. The `add` method calls the `do` method of the command to invoke the changing of the name.

The data frame model (defined in our reference class `DfModel`) is a wrapper around an `RGtkDataFrame` object that holds the data. The method call above is implemented by:

```
DfModel$methods(
                get_col_name=function(j) varnames[j,1],
                get_col_names=function() varnames[ ,1],
                set_col_name=function(j, value) {
                  "Set name, return old"
                  old <- get_col_name(j)
                  varnames[j,1] <<- value
                  old
                })
```

We return the old value, as that is required by the implementation of the `do` method for the commands. An instance of `RGtkDataFrame` stores the variable (column) names, hence the double index. This allows us to listen for changes through the `row-changed` signal on the model. The details, and more, are in the accompanying package.

6

Extending GObject Classes

GTK+, as well as several of its dependencies, with the notable exception of `Cairo`, is based on the `GObject` library for object-oriented programming in C. `GObject` forms the basis of many other open-source projects, including the `GNOME` and `XFCE` desktops and the `GStreamer` multimedia framework.

Given the broad use of signals in the GTK+ API, it is very rarely necessary to extend a widget class when developing a typical GUI. However, it is generally good practice to encapsulate the behavior of a widget in a formal class. Although there are several such formalisms in R, `RGtk2` provides one that is congruent with the rest of GTK+. It interfaces with parts of `GObject` and permits the R programmer to create new `GObject` classes in R. A subclass can override certain methods inherited from its parent and define new methods, properties and signals. If a method is declared by a C class, it can only be overridden if it is a so-called *virtual* method, and there is no documentation as to which methods are virtual. There is a loose convention that every signal has a corresponding virtual method. The ultimate resource is the C header files. A bug in a method override could very easily crash R, so use of this feature takes some commitment from the programmer. Any method declared by an R class may be overridden by an R subclass.

Our example will be a GUI that displays a scatterplot along with a slider for adjusting the alpha level of the points (Figure 6.1). Usually, a slider operates in linear fashion. When there are a large number of points, on the order of tens of thousands or more, changing the alpha level does not have a strong visual effect until it approaches its lower limit. We desire greater control in the lower part of the alpha scale, without limiting the range of the slider. To achieve this, we need to perform a non-linear transformation from the slider value to the alpha of the plot and reflect that transformation in the label on the slider. One solution is to connect to the `format-value`GtkScale signal to override the text in the label. We present an alternative that involves extending `GtkHScale` and overriding its `format_value` virtual method.

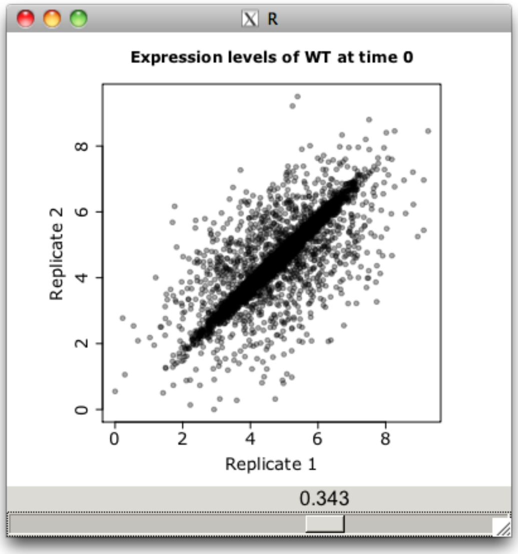


Figure 6.1: An interface using a custom slider to adjust alpha levels in a non-linear manner

A class is defined by calling gClass, which is passed the class name, the name of the parent class and a number of list arguments that define the properties, signals and methods of the class. For the sake of cleanliness, the everything is defined as part of the gClass call:

```
tform_scale_type <-
  gClass("RTransformedHScale", "GtkHScale",
         .props = list(
           gParamSpec(type = "R", name = "expr", nick = "e",
                      blurb = "Transformation of scale value",
                      default.value = expression(x))
           ),
         GtkScale = list(
           format_value = function(self, x)
             as.character(self$transformValue(x))
           ),
         .public = list(
           getExpr = function(self) self["expr"],
           getTransformedValue = function(self)
             self$transformValue(self$value)
           ),
         .private = list(
           transformValue = function(self, x)
             eval(self$expr, list(x = x))
```

```
    )
  )
```

The class definition for `RTransformedHScale` starts with a property for the R expression that transforms the value from the slider to the alpha level. A property is defined by a `GParamSpec` structure that specifies a name, nickname, descriptive blurb, value type, and other options. There are subclasses of `GParamSpec` for particular types that permit specification of further constraints. For example, `GParamSpecInt` is specific to integers and can be configured to restrict its valid range of integer values between a minimum and maximum. Many `GParamSpec` subclasses also permit default values. The `type` argument may refer to any C type by name. The names of R types, like "integer" and "character" are mapped to the corresponding scalar C type, if available. An "R" property, like our expression, stores any native R value. The actual R type, as returned by `typeof`, may be specified as the `s.type` argument; otherwise, it is taken from the default value.

We turn our attention to the methods in the class definition. The class overrides the `format_value`irtual from `GtkScale` and defines two public methods, `getExpr` and `getTransformedValue`, for retrieving the transformation expression and the transformed value, respectively. There is one private method, `transformValue` that is a utility for evaluating the expression on the current value.

Methods are implemented with R functions that are grouped into lists. The names of the list identify the methods. An override is placed into the list corresponding to the class in which the original method is declared. For new methods, the division is by the access level: public, protected or private. Public members may be accessed by any code, while protected members are restricted to methods belonging to the same class or a subclass. Access to private members is the most restricted as they are only available to methods in the same class.

A function implementing a virtual method may delegate to the method that it overrides. This is achieved by calling the `parentHandler` function and passing it the name of the method and the arguments to forward to the method. This is similar to the `super` function in `qtbase`. For example, in the override of `format_value` in the `RGtkTransformedHScale` class, we could call `parentHandler("format_value", self, x)` to delegate to the implementation of `format_value` in `GtkScale`.

If a non-function, like a vector, is placed in the `.public`, `.protected` or `.private` list, it represents a field, which is initialized to the given value.

Two elements of the class definition that are not in the example above are the list of signal definitions and the initialization function. The signal definition list is passed as a parameter named `.signals` and contains a list for each signal. Each list includes the name, return type, and parameter types of the signal. The types may be specified in the same format as used for property definitions. The initialization function, passed as the `.initialize` parameter, is invoked whenever an instance of the class is created, before any properties are set. It takes the newly created instance of the class as its only parameter.

The next step in our example is to create an instance of `RGtkTransformedHScale` and to register a handler on the `value-changed` signal that will draw the plot using the transformed value as the alpha setting:

```
adj <- gtkAdjustment(0.5, 0.15, 1.00, 0.05, 0.5, 0)
s <- gObject(tform_scale_type, adjustment = adj,
             expr = expression(x^3))
gSignalConnect(s, "value_changed", function(scale) {
  plot(ma_data, col = rgb(0, 0, 0, scale$getTransformedValue()),
       xlab = "Replicate 1", ylab = "Replicate 2",
       main = "Expression levels of WT at time 0", pch = 19)
})
```

Instances of any `GObject` class may be created using the `gObject` function. The value of the `expr` property is set to the `R` expression x^3 when the object is created. The signal handler now calls the new `getTransformedValue` method, instead of `getValue` as in the original version. The `ma_data` object is a matrix of points that is meant to resemble expression values from two replicates of a microarray experiment.

We complete the example by placing the slider and a graphics device in a window:

```
win <- gtkWindow(show = FALSE)
da <- gtkDrawingArea()
vbox <- gtkVBox()
vbox$packStart(da)
vbox$packStart(s, FALSE)
win$add(vbox)
win$setDefaultSize(400, 400)
#
require(cairoDevice)
asCairoDevice(da)
#
win$showAll()
par(pty = "s")
s$setValue(0.7)
```

Concept index

Class and method index