1

RGtk2: Overview

As the name implies, the `RGtk2` package provides a connection, or bindings, between GTK+ and R allowing nearly the full power of GTK+ to be available to the R programmer. In addition, `RGtk2` provides bindings to other libraries accompanying GTK+: The Pango libraries for font rendering; the Cairo libraries for vector graphics; the GdkPixbuf libraries for image manipulation; libglade for designing GUI layouts from an XML description; ATK for the accessiblity toolkit; and GDK, which provides an abstract layer between the windowing system, such as X11, and GTK+. These libraries are multi-platform and extensive and have been used for many major projects, such as the linux versions of the firefox browser and open office.

`RGtk2`, for the most part, automatically creates R functions that call into the GTK+ library. For example, the R function `gtkContainerAdd` eventually calls the C function `gtk_container_add`. The naming convention is the C name has its underscores removed and each following letter capitalized (camelback).

The full API for GTK+ is quite large, and clearly can not be documented here. However, the GTK+ documentation is converted into R format in the building 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 API of the stable versions GTK+.

1.1 How GTK+ is organized

GTK+ objects are created using constructors such as `gtkWindowNew` and `gtkButtonNewWithLabel` (these mapping to `gtk_window_new` and `gtk_button_new_with_label` respectively). `RGtk2` also provides constructors with names not ending in "New" that may, depending on the arguments given, call different, but similar, constructors. As such we prefer the shorter named constructors, such as `gtkWindow` or `gtkButton`.

1

Methods

The underlying GTK+ library is written in C, but still provides a a singly inherited, object-oriented framework that leads naturally to the use of S3 classes for the R package. In GTK+ the `GtkWindow` class inherits methods, properties, and signals from the `GtkBin`, `GtkContainer`, `GtkWidget`, `GtkObject`, `GInitiallyUnowned`, and `GObject` classes. In `RGtk2`, we can see the class heiarchy by calling `class` on a `gtkWindow` instance: [1]

```
class(gtkWindow())
```

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

The classes are identical except for the addition of the base `RGtkObject` class. When a widget is destroyed, the R object is assigned `<invalid>` class.

Methods of `RGtk2` do not use S3 dispatch, but rather an internal one. The call `obj$method(...)` resolves to a function call `f(obj,...)`. The function is found by looking for any function prefixed with with either an interface or a class from the object followed by the method name. The interfaces are checked first.

For instance, if `win` is a `gtkWindow` instance, then to resolve the call `win$add(widget)` (or `win$add(widget)`) R looks for methods with the name `gtkBuildableAdd`, `atkImplementorIfaceAdd`, `gtkWindowAdd`, `gtkBinAdd` before finding `gtkContainerAdd` and calling it as `gtkContainerAdd(win,widget)`. The $ method for `RGtk2` objects does the work. Understanding this mechanism allows us to add to the `RGtk2` API, as convenient. For instance, we can add to the button API with

```
gtkButtonPrintHello <- function(obj) print("hello")
b <- gtkButton()
b$printHello()
```

```
[1] "hello"
```

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

Properties

Also inherited are widget properties. A list of properties that a widget has is returned by its `GetPropInfo` method. `RGtk2` provides the R generic `names` as

---

[1]We use the term "instance" of a constructor to refer to the object returned by the constructor, which is an instance of some class.

a familiar alternative for this method. For the button just defined, we can see the first eight properties listed with:

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

```
[1] "user-data"     "name"           "parent"         "width-request"
[5] "height-request" "visible"       "sensitive"      "app-paintable"
```

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

There are a few different ways to access these properties. Consider the `label` property of a `gtkButton` instance. GTK+ provides the functions `gObjectGet` and `gObjectSet` to get and set properties of a widget. The set funtion using the arguments names for the property key.

```
b <- gtkButton("A button")
gObjectGet(b,"label")
```

```
[1] "A button"
```

```
gObjectSet(b,label="a new label for our button")
```

Additionally, most user-accessible properties have specific `Get` and `Set` methods defined for them. In our example, the methods `getLabel` and `setLabel` can be used.

```
b$getLabel()
```

```
[1] "a new label for our button"
```

```
b$setLabel("Again, a new label for our button")
```

`RGtk2` provides the convenient and familiar `[` and `[<-` methods to get and access the properties:

```
b['label']
```

```
[1] "Again, a new label for our button"
```

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.

Enumerated types and flags

The GTK+ libraries have a number of constants that identify different states. These enumerated types are defined in the C code. For instance, for a toolbar, there are four possible styles: with icons, just text, both text and icon, and both text and icon drawn horizontally. The flags indicating the

3

style are stored in C in an enumeration `GtkToolbarStyle` with constants `GTK_TOOLBAR_ICONS`, `GTK_TOOLBAR_TEXT`, etc. In `RGtk2` these values are conveniently stored in the vector `GtkToolbarStyle` with named integer values

```
GtkToolbarStyle
```

icons	**text**	both	both−horiz
0	1	2	3

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

A list of enumerated types for GTK+ is listed in the man page `?gtk-Standard-Enumerations` and for `Pango` in `?pango-Layout-Objects`. The `Gdk` variables are prefixed with `Gdk` and so can be found using apropos, say, using `ignore.case=TRUE`.

To use these enumerated types, one can specify them by name as

```
tb <- gtkToolbar()
tb$setStyle(GtkToolbarStyle['icons'] )
```

But `RGtk2` provides the convenience of specifying the style name only, as in

```
tb$setStyle("icons")
```

When more than one value is desired, they can be combined using c.

Events and signals

In `RGtk2` user actions, such as mouse clicks, keyboard usage, drag and drops, etc. trigger `RGtk2` widgets to signal the action. A GUI can be made interactive, by adding callbacks to respond when these signals are emitted. In addition to signals, there are a number of window manager events, such as a `button-press-event`. These events have callbacks attached in a similar manner.

The signals and events that an object adds are returned by the method `GetSignals`. For example

```
names(b$getSignals())
```

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

shows the "clicked" signal in addition to others.

To list all the inherited signals can be achieved using `gtkTypeGetSignals`. For instance, the following code will print out all the inherited signals and events.

```
types <- class(b)
lst <- sapply(head(types,n=−1), gtkTypeGetSignals)
for(i in names(lst)) { cat(i,"\n"); print(lst[[i]])}
```

4

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

```
args(gSignalConnect)
```

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

The `obj` is the widget the callback is attached to and `signal` the signal
name, for instance `"drag-drop"`. This may also be an event name.

The `f` argument is for the callback. Although, it can be specified as an
expression or a call, our examples always use a function to handle the call-
back. More detail follows. The `after` argument is a logical indicating if the
callback should be called after the default handlers (see `?gSignalConnect`).

The `data` argument allows arbitrary data to be passed to the callback.
The `user.data.first` argument specifies if this `data` argument should be
the first argument to the callback or (the default) the last. As the signature of
the callback has varying length, setting this to `TRUE` can prove useful.

The signature for the callback varies for each signal and window manager
event. Unless the default for `user.data.first` is overridden, the argument
is the widget. For signals, other arguments are possible depending on the
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 each argument.

As the callback is an R function, it is passed copies of the object. Since
`RGtk2` objects are pointers, there is no practical difference. So changes within
the body of a callback to `RGtk2` objects are reflected outside the scope of the
callback, unlike changes to most other R objects.

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

Then after clicking, we would have

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

```
1 2
```

Callbacks for signals emitted by window manager events are expected to
return a logical value. Failure to do so can cause errors to be raised. For
other callbacks the return value is ignored, so it is safe to always return a
logical value. When it is not ignored, a return value of `TRUE` indicates that
no further callbacks should be called, whereas `FALSE` indicates that the next

callback should be called. So in the following example, only the first two callbacks are executed when the user presses on the button.

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

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

The eventloop

The `RGtk2` eventloop integrates with the R event loop. In practice, such integration is tricky. In a C program, GTK+ programs call the function `gtk_main` which puts control of the GUI into the main event loop of GTK+. This sits idle until some event occurs. According to the `RGtk2` website, "The nature of the R event loop prevents the continuous execution of the GTK main loop, thus preventing things like timers and idle tasks from executing reliably. This manifests itself when using functionality such as GtkExpander and GtkEntryCompletion."

During a long calculation, the GUI can seem unresponsive. To avoid this the following construct can be used during the long calculation to process pending events.

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

1.2   RGtk2 and gWidgetsRGtk2

The widgets described above, are also available through `gWidgetsRGtk2`. The two packages can be used together, for the most part. The `add` method of

6

gWidgetsRGtk2 can be used to add an RGtk2 widget to a gWidgetsRGtk2 container. Whereas, the getToolkitWidget method will (usually) return the RGtk2 component to use within RGtk2.

RGtk2: Basic Components

This section covers some of the basic widgets and containers of GTK+. We begin with a discussion of top level containers and box containers. Then we continue with describing many of the simpler controls – essentially those without an underlying model, and then finish by describing a few more containers.

2.1 Top-level windows

Top-level windows are constructed by the `gtkWindow` constructor. This function has arguments `type` to specify the type of window to create. The default is a top-level window, which we will always use, as the alternative is for "popups" which are used, for example, with 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 vector notion. To illustrate, the following sets up a new window with some title.

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

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

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

Window size The initial size of the window can be set with the `setDefaultSize` method, as shown, which takes a `width` and `height` argument specified in

pixels. This specification allows the window to be resized, but must be made before the window is drawn. The `SetSizeRequest` method will also set the size, but does not allow for resizing smaller than the requested size. To really fix the size of a window, the `resizable` property may be set to `FALSE`.

Transient windows   New windows may be standalone top-level windows, or may be associated to some other window, such as a how a dialog is associated with some parent window. In this case, the `SetTransientFor` method can be used to specify which window. This allows the window manager to keep the transient window on top. The position on top, can be specified with `SetPostion` which takes a constant given by `GtkWindowPosition`. Finally it can be specified that the dialog 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(GtkWindowPosition["center-on-parent"])
d$setDestroyWithParent(TRUE)
w$show()
d$Show()
```

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

Destroying windows   The window can be closed through the window manager, by clicking on its close icon, or programatically by calling its `Destroy` method. When the window manager is clicked, the `delete-event` event signal is raised, and can have a callback listen for it. If that callback returns `FALSE`, then the window's `destroy` signal is emitted. It is this signal that is propogated to the windows child components. However, if a callback for the `delete-event` signal returns `TRUE`, then the `destroy` signal will not be emitted. This can be useful if a confirmation is desired before closing the window.

Adding a child component to a window   A window is a container. However, `gtkWindow` objects, inherit from the `GtkBin` class which allows only one child container. This child is added through the windows `Add` method. This child is often another container that allows for more than one component to be added.

We illustrate by adding a simple label to a window.

```
w <- gtkWindow(); w$setTitle("Hello world")
l <- gtkLabel("Hello world")
```

10

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

The method `GetChildren` will return the children of a container as a list. In this case, as the `GtkWindow` window class is a subclass of `GtkBin`, which holds only 1 child component, the `GetChild` method may be used to access the label directly. For instance, to retrieve the label's text one can do.

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

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

The `[[` method can be used to access the child containers by number, as an shorcut for list extraction from the return value of the `GetChildren`ethod.

From the label's perspective, the window is its immediate parent. The `GetParent` method for GTK+ widgets, will return a widget's parent container.

2.2   Box containers

Flexible containers for holding more than one child are the box containers constructed by `gtkHBox` or `gtkVBox`. These produce horizontal, or vertical "boxes" which allow packing of child components in a manner analagous to packing a box. These components can be subsequent box containers, allowing for very flexible layouts.

Each child component is allocated a cell in the box. The `homogeneous` argument can be set to `TRUE` to ensure all the cells have the same size allocated to them. The default, is so have non-homogeneous size allocations.

Packing child components   Adding a child component to the box is done with the methods `PackStart` or `PackEnd`. The `PackStart` method adds children from left to right when the box is horizontal, or top to bottom when vertical. the `PackEnd` method is opposite. These methods have initial argument `child` to specify the child component and `padding` to specify a padding in pixels between child components.

Once packed in the child components can be referenced through the `GetChildren` method which is conveniently called through list extraction.

Removing and reordering children   Once children are packed into a box container, they can be manipulated in various ways.

The `Hide` method of a child component will cause it not to be drawn. This can be reversed with the `Show` method.

The `Remove` method for containers can cause a child component to be removed. It can later be re-added using `PackStart`. For instance

```
b <- g[[3]]
g$remove(b)                              # removed
```
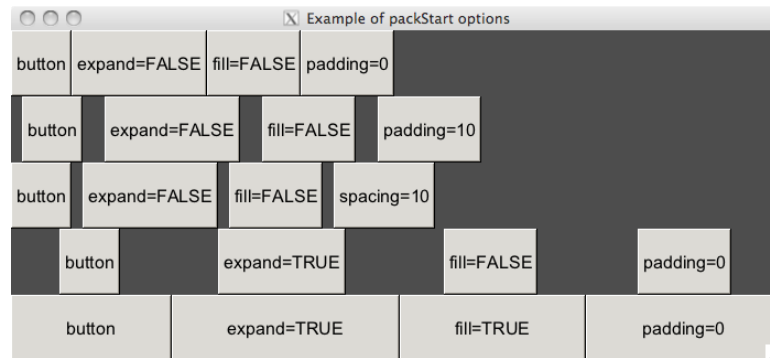
11

Figure 2.1: 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`. Illustration follows one in orignial GTK+ tutorial.

```
g$packStart(b, expand=TRUE, fill=TRUE)
```

The `Reparent` method of widgets, will allow a widget to specify a new parent container.

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

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

Spacing   There are several adjustments possible to add space around components in a box container. The `spacing` argument for the constructors specifies the amount of space, in pixels, between the cells with a default of 0. The `Pack` methods also have a `padding` argument to specify the padding between subsequent children, again with default 0. For horizontal packing, this space goes on both the left and right of the child component, whereas the `spacing` value is just between children. (The spacing between components is the sum of the `spacing` value and the two `padding` values when the children are added.) Child widgets also have properties `xpad` and `ypad` for setting the padding around themselves. Example 2.3 provides an example and Figure 2.1 an illustration.

Component size   Each component has properties `width` and `height` to determine the size of the component when mapped. When these are both -1, the natural size of the widget will be used. To set the requested size of a

12

component, the method `SetSizeRequest` is used to specify minimum values for the width and `height` of the widget. The methods help page warns that it is impossible to adequately hardcode a size that will always be correct.

When a parent container is resized, it queries its children for their preferred size (`GetSizeRequest`), If these children have children, they then are asked, etc. This size information is then passed back to the top-level component. It resizes itself, then passes on the available space to its children to resize themselves, etc. After resizing the `GetAllocation` method returns the new width and height, as components in a list. The space allocated to a cell, may be more than the space requested by the widget. In this case, the `expand` and `fill` arguments for the `Pack` methods are important. If `expand=TRUE` is given, then the cell will expand to fill the extra space. Furthermore, if also `fill=TRUE` then the widget will expand to fill the space allocated to the cell. Figure 2.1 illustrates.

Widgets inherit the properties `xalign` and `yalign` from the `GtkMisc` class. These properties are used to specify how the widget is aligned within the cells when the widget size request is less than that allocated to the cell. These properties take values between 0 and 1, with 0 being left and top. Their defaults are 0.5, for centered alignment.

2.3 Buttons

A basic button is constructed using `gtkButton`. This is a convenience wrapper for several constructors. With no argument, it returns a simple button. When the first argument, `label`, is used it returns a button with a label, calling `gtkButtonNewWithLabel`. The `stock.id` argument calls `gtkButtonNewFromStock`. Buttons in GTK+ are actually containers (of class `GtkBin`), By default, they have a `label` and `image` property. The image is specified using a stock id. The available stock icons are listed by `gtkStockListIds`. Finally, if a mnemonic is desired, for the button, the constructor `gtkButtonNewWithMnemonic` can be used. Mnemonics are specified by prefixing the character with an underscore, as illustrated in this example.

Example 2.1: 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") )
```
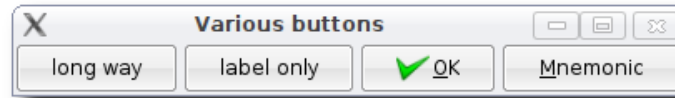
13

Figure 2.2: Various buttons

```
g$packStart(gtkButton(stock.id="gtk-ok") )
g$packStart(gtkButtonNewWithMnemonic("_Mnemonic") ) # Alt-m to "click"
w$show()
```

Buttons are essentially containers with a decoration to give them a button like appearance. The relief style of the button can be changed so that the button is drawn like a label. The method `SetRelief` is used, with the available styles found in the `GtkReliefStyle` enumeration.

A button, can be drawn with extra space all around it. The `border-width` property, with default of 0, specifies this space. One can use the method `SetBorderWidth` to make a change.

signals The `clicked` signal is emitted when the button is clicked on with the mouse or when the button has focus and the enter key is pressed. A callback can listen for this event, to initiate an action. If one wishes to filter out the mouse button that was pressed on the button, the `button-press-event` signal is also emitted. Since this is a window manager event, the second argument to the callback is an event which contains the button information. This can be retrieved using the event's `getButton` method. However, the `button-press-event` signal is not emitted when the keyboard initiates the action.

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

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

If the action that a button initiates is to be represented elsewhere in the GUI, say a menu bar, then a `GtkAction` object may be appropriate. Action objects are covered in Section 4.5.
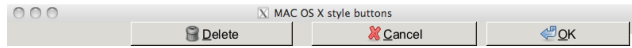
Example 2.2: Callback example for `gtkButton`

14

Figure 2.3: Example using stock buttons with extra spacing added between the `delete` and `cancel` buttons.

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

Example 2.3: Spacing between buttons

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

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

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

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

```
g$packEnd(ok, padding =0)
g$packEnd(cancel, padding =12)
g$packEnd(delete, padding =12)
```

15

```
g$packEnd(gtkLabel(""), expand=TRUE, fill=TRUE)
```

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

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

2.4 Labels

Labels are created by the `gtkLabel` constructor. Its main argument is `str` to specify the button text, through its `label` property. This text can be set with either `SetLabel` or `SetText` and retrieved with either `GetLabel` or `GetText`. The difference being the former can respect formatting marks.

The text can include line breaks, specified with "\n." Further formatting is available. Wrapping of long labels can be specified using a logical value with the method `SetLineWrap`. The line width can be specified in terms of the number of characters thorough `SetWidthChars` or by setting the size request for the label. This is not determined by the size of the parent window. Long labels can also have ellipsis inserted into them to shorten when there is not enough space. By default this is turned off. The variable `PangoEllipsizeMode` contains the constants, and the method `SetEllipsize` is used to set this. The property `justify`, with values taken from `GtkJustification`, controls the justification.

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

By default, text in a label can not be copied and pasted into another widget or application. To allow this, the `selectable` property can be set to `TRUE` with `SetSelectable`.

Labels can hold mnemonics for other widgets. The constructor is `gtkLabelNewWithMnemonic`. The label needs to idenfy the widget it is holding a mnemonic for, this is done with the `SetMnemonicWidget` method.

Example 2.4: Label formatting

```
w <- gtkWindow(); w$setTitle("Label formatting")
w$setSizeRequest(250,100)                    # narrow
```

16

```
g <- gtkVBox(spacing=2); g$setBorderWidth(5); w$add(g)
string = "the quick brown fox jumped over the lazy dog"
## wrap by setting number of characters
basicLabel <- gtkLabel(string)
basicLabel$setLineWrap(TRUE);
basicLabel$setWidthChars(35)                # specify number of characters
## Set ellipsis to shorten long text
ellipsized <- gtkLabel(string)
ellipsized$setEllipsize(PangoEllipsizeMode["middle"])
## Right justify text
## use xalign property for label in cell
rightJustified <- gtkLabel("right justify");
rightJustified$setJustify(GtkJustification["right"])
rightJustified['xalign'] <- 1
## PANGO markup
pangoLabel <- gtkLabel();
pangoLabel$setMarkup(paste("<span foreground='blue' size='x-large'>",
                           string,"</span>"));
QT <- sapply(list(basicLabel, ellipsized, rightJustified, pangoLabel),
             function(i) g$packStart(i, expand=TRUE, fill=TRUE ))
w$showAll()
```

Signals Unlike buttons, labels do not emit any signals. Labels are intended to hold static text. However, if one wishes to define callbacks to react to events, then the label can be placed within an instance of gtkEventBox. This creates a non-visible parent window for the label that does signal events. Example 2.11 will illustrate the use of an event box. Alternatively, one could use an instance of gtkButton with its relief property assigned to GtkReliefStyle['none'].

Link Buttons

A link button is a special label which shows an underlined link, such as is done by a web browser. (Newer versions of GTK+ allow the label of a button to contain HTML links.) The uri is specified to the gtkLinkButton constructor with an optional label argument. If none is specified, the uri is used to provide the value. This uri is stored in the uri property and the label in the label value. These may be adjusted later.

As the link button inherits from the gtkButton class, the clicked signal is emitted when a user clicks a mouse on the link.

Example 2.5: Basic link button usage

```
w <- gtkWindow()
g <- gtkVBox(); w$add(g)
```

```
lb <- gtkLinkButton(uri="http://www.r-project.org")
lb1<- gtkLinkButton(uri="http://www.r-project.org", label="R Home")
g$packStart(lb)
g$packStart(lb1)
f <- function(w,...) browseURL(w['uri'])
ID <- gSignalConnect(lb,"clicked",f = f)
ID <- gSignalConnect(lb1,"clicked",f = f)
```

2.5 Images

Images in `RGtk2` are constructed with `gtkImage`. This is a front end for several constructors: `gtkImageNewFromIconSet`, `gtkImageNewFromPixmap`, `gtkImageNewFromImage`, `gtkImageNewFromFile`, `gtkImageNewFromPixbuf`, `gtkImageNewFromStock`, `gtkImageNewFromAnimation`. We only discuss loading an image from a file, and so use the `gtkImageNewFromFile` constructor. To add an image after construction of the main widget, the `gtkImageNew` constructor can be used along with methods such as `SetFromFile`.

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

Example 2.6: Using a pixbuf to present graphs

This example shows how to use a `gtkImage` object to embed a graphic within `RGtk2`, as an alternative to using the `cairoDevice` package. The basic idea is to use the `Cairo` device to create a file containing the graphic, and then use `gtkImageNewFromFile` to construct a widget to show the graphic.

We begin by creating a window of a certain size.

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

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

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

Now we draw a basic graphic as a png file (`Cairo`'s default) stored in a temporary file.

```
require(Cairo)
filename <- tempfile()
```

18

```
Cairo(file = filename, width = width, height = height)
hist(rnorm(100))
QT <- dev.off()
```

The constructor may be called as `gtkImage(filename=filename)` or as follows:

```
image <- gtkImageNewFromFile(filename)
g$packStart(image, expand=TRUE, fill = TRUE)
unlink(filename)                           # tidy up
```

```
## Work this into an example ###
makeIconRGtk2 <- function(w, giffile) {
  if(checkPtrType(w, "GtkWindow")) {
    img <- gdkPixbufNewFromFile(giffile)
    if(!is.null(img$retval))
      w$setIcon(img$retval)
  }
}
```

2.6   Stock icons

GTK+ comes with several "stock" icons. These are used by the `gtkButton` constructor when its `stock.id` argument is specified, and will be used for menubars, and toolbars. The size of the icon used is one of the values returned by `GtkIconSize`.

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

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

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

To load a stock icon into an image widget, the `gtkImageNewFromStock` can be used. The `stock.id` contains the icon name and `size` the size.

The example below, we use the method `RenderIcon` to return a pixbuf containing the icon that can be used with the constructor `gtkImageNewFromPixbuf` to display the icon. Here the stock id and size are specified to the `RenderIcon` method.

Example 2.7: `gtkButtonNewFromStock` **– the hard way**
The following example, shows how to do the work of `gtkButtonNewFromStock` by hand using an image and label together.

```
b <- gtkButton()
g <- gtkHBox()
pbuf <- b$renderIcon("gtk-ok", size=GtkIconSize["button"])
```

19

```
i <- gtkImageNewFromPixbuf(pbuf)
i['xalign'] <- 1; i['xpad'] <- 5           # right align with padding
g$packStart(i, expand=FALSE)
l <- gtkLabel(gettext("ok"));
l['xalign'] <- 0 # left align
g$packStart(l, expand=TRUE, fill=TRUE)
b$add(g)
## show it
w <- gtkWindow(); w$add(b)
```

Example 2.8: Adding to the stock icons
This example shows, without much explanation the steps to add images to
the list of stock icons. To generate some sample icons, we use those provided
by objects in the ggplot2 package.

First we create the icons using the fact that the objects have a function
icon to draw an image.

```
require(ggplot2)
require(Cairo)
iconNames <- c("GeomBar","GeomHistogram")    # 2 of many ggplot functions
icon.size <- 16
iconDir <- tempdir()
fileNames <- sapply(iconNames, function(name) {
  nm <- paste(iconDir, "/", name, ".png", sep="", collapse="")
  Cairo(file=nm, width=icon.size, height=icon.size, type="png")
  val <- try(get(name))
  grid.newpage()
  try(grid.draw(val$icon()), silent=TRUE)
  dev.off()
  nm
})
```

The following function works through the steps to add a new icon. The basic
ideas are sketched out in the API for GtkIconsSet.

```
addToStockIcons <- function(iconNames, fileNames, stock.prefix="new") {
  iconfactory <- gtkIconFactoryNew()

  for(i in seq_along(iconNames)) {

    iconsource = gtkIconSourceNew()
    iconsource$setFilename(fileNames[i])

    iconset = gtkIconSetNew()
    iconset$addSource(iconsource)

    stockName = paste(stock.prefix, "-", iconNames[i], sep="")
    iconfactory$add(stockName, iconset)
```
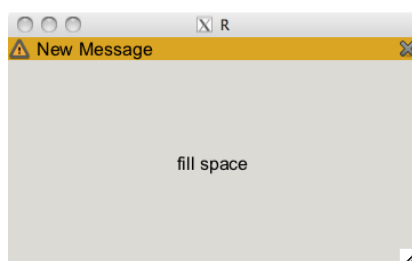
20

Figure 2.4: The alert panel showing a message.

```
    items = list(test=list(stockName, iconNames[i],"","",""))
    gtkStockAdd(items)
  }
  iconfactory$AddDefault()
  invisible(TRUE)
}
```

We call this function and then check that the values are added:

```
addToStockIcons(iconNames, fileNames)
nms <- gtkStockListIds()
unlist(nms[grep("^new", nms)])
```

```
[1] "new-GeomHistogram" "new-GeomBar"
```

Example 2.9: An alert panel

This example puts together images, buttons, labels and box containers to create an alert panel, or information bar. This is an area that seems to drop down from the menu bar to give users feedback about an action that is less disruptive than a modal dialog. A similar widget is used by Mozilla with its popup blocker. Although, as of version 2.18, a similar feature is available in GTK+ through the `GtkInfoBar` widget, this example is given, as it shows how several useful things in GTK+ can be combined to customize the user experience.

This constructor for the widget specifies some properties and returns an environment to store these properties, as our function calls will need to update these properties and have be persistent.

```
newAlertPanel <- function(wrap=35,
                          icon="gtk-dialog-warning",
                          message="",
                          panel.color="goldenrod",
                          evb=NULL,
```

21

```
                               image=NULL,
                               label=NULL # info
                          ) {
  x <- c("wrap","icon","message","panel.color","evb","image","label")
  e <- new.env()
  sapply(x, function(i) assign(i, envir=e, get(i)))
  return(e)
}
```

An alert panel needs just a few methods: one to create the widget, one to show the widget and one to hide the widget. We create a function `getAlertPanelBlock` to return a component that can be added to a container. An event box is used so that we can color the background, as this isn't possible for a box container due to its lack of a gdk window. To this event box we add a box container that will hold an icon indicating this is an alert, a label for the message, and another icon to indicate to the user how to close the alert. Since we wish to receive mouse clicks on close icon, we place this inside another event box. To this, we bind a callback to the `button-press-event` signal.

```
getAlertPanelBlock <- function(obj) {

  obj$evb <- gtkEventBox(show=FALSE)
  obj$evb$ModifyBg(state="normal",color=obj$panel.color)

  g <- gtkHBox(homogeneous=FALSE, spacing=5)
  obj$evb$add(g)

  obj$image <- gtkImageNewFromStock(obj$icon, size="button",show=TRUE)
  obj$image['yalign'] <- .5
  g$packStart(obj$image, expand=FALSE)

  obj$label <- gtkLabel(obj$message)
  obj$label['xalign'] <- 0; obj$label['yalign'] <- .5
  obj$label$setLineWrap(TRUE)
  obj$label$setWidthChars(obj$wrap)
  g$packStart(obj$label, expand=TRUE, fill= TRUE)

  xbutton <- gtkEventBox()
  xbutton$modifyBg(state="normal",color=obj$panel.color) # same color
  xbutton$add(gtkImageNewFromStock("gtk-close",size="menu"))
  xbuttonCallback <- function(data,widget,...) {
    hideAlertPanel(data)
    return(FALSE)
  }
  ID <- gSignalConnect(xbutton,"button-press-event",
                  f = xbuttonCallback,
                  data=obj, user.data.first=TRUE)
  g$packEnd(xbutton, expand=FALSE, fill=FALSE)
```

22

```
  ## also close when event box is clicked
  obj$motionID <-
    gSignalConnect(obj$evb,signal="button-press-event",
                   f = xbuttonCallback,
                   data=obj, user.data.first=TRUE)
  return(obj$evb)
}
```

The `showAlertPanel` function updates the message and then calls the `Show` method of the event box.

```
showAlertPanel <- function(obj) {
  obj$label$setText(obj$message)
  obj$evb$show()
}
```

Our `hideAlertPanel` function simply calls the `hide` method the event box.

```
hideAlertPanel <- function(obj) obj$evb$hide()
```

To test it out, we create a simple GUI

```
w <- gtkWindow()
g <- gtkVBox(); w$add(g)
ap <- newAlertPanel()
g$packStart(getAlertPanelBlock(ap), expand=FALSE)
g$packStart(gtkLabel("fill space"), expand=TRUE, fill=TRUE)
ap$message <- "New Message"              # add message
showAlertPanel(ap)
```

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

2.7   Text entry

A one-line text entry widget is constructed by `gtkEntry`. An argument `max` specifies the maximum number of characters if positive, but this calls a deprecated function, so this restriction should be set using the method `SetMaxLength`.

The `text` property stores the text. This can be set with the method `SetText` and retrieved with `GetText`. To insert text, the method `InsertText` is used. Its argument `new.text` contains the text and `position` specifies the position of the next text. The return value is a list with components `position` indicating the position *after* the new text. The `DeleteText` method can be used to delete text. This takes two integers indicating the start and finish location of the text.

23

Example 2.10: Insert and Delete text
The example will show how to add then delete text.

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

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

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

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

The `GtkEntry` class adds three signals `changed` when text is changed, `delete-text` for delete events, and `insert-text` for insert events. The `changed` signal will be emitted each time there is a keypress, while the widget has focus. When the enter key is pressed the `activate` signal is also emitted.

Example 2.11: Editable label
We use the `activate` signal in the following example where we make an editable label. When the label is clicked, a text entry replaces the label allowing its content to be edited. The user indicates the editing is over by pressing the enter key. This example also illustrates the `gtkEventBox`, which emits a `button-press-event` signal, when the label is clicked on.

```
w <- gtkWindow(); w$setTitle("Editable label")
evb <- gtkEventBox();
w$add(evb);
e <- gtkEntry()
l <- gtkLabel("Click me to edit")
evb$setData("entry", e); evb$setData("label", l)
evb$add(l)
ID <- gSignalConnect(evb, "button-press-event", function(w, e, ...) {
  label <- w$getData("label"); entry <- w$getData("entry")
  entry$setText(label$getText())
  w$remove(label);  w$add(entry) # swap
})
ID <- gSignalConnect(e,"activate", function(userData, w, ...) {
  evb = userData$evb; label <- evb$getData("label")
  label$setText(w$getText())
  evb$remove(w); evb$add(label) #swap
},
                      data=list(evb=evb),
                      user.data.first=TRUE)
```

24

2.8   Check button

A check button widget is constructed by `gtkCheckButton`. The optional argument `label` places a label next to the button. The label can have a mnemonic, but then the constructor is `gtkCheckButtonewWithMnemonic`.

The `label` property stores the label. This can be set or retrieved with the methods `SetLabel` and `GetLabel`.

A check button's state is stored as a logical variable in its `active` property. It can be set or retrieved with the methods `SetActive` and `GetLabel`.

When the state is changed the `toggle` signal is emitted.

Toggle buttons

A toggle button, is a useful way to set configuration values in an obvious way to the user. A toggle button has a depressed look when in an active state. The `gtkToggleButton` constructor is used to create toggle buttons. The `label` argument sets the `label` property. This can also be set or retrieved with the methods `SetLabel` and `GetLabel`.

The `active` property is `TRUE` when the button is depressed, and `FALSE` otherwise. This can be queried with the `GetActive` method.

As with other buttons, the `clicked` signal is emitted when the user clicks on the button.

2.9   Radio groups

The `gtkRadioButton` constructor is used to create linked radio buttons. The argument group is missing or `NULL` will create a new radio button group. If specified as a list of radio buttons, will create a new button for the group. The constructor returns a single radio button widget. The labels for each individual button are determined by their `label` property, This can be set at construction time through the `label`, or can be modified through the `SetLabel` method.

Each radio button in the group has its `active` property either `TRUE` or `FALSE`, although only one can be `TRUE` at a time. The methods `GetActive` and `SetActive` may be used to manipulate the state of an individual button. To determine which button is active, they can be queried individually. The same property can be set to make a given button active.

When the state of a radio button is changed, it emits the `toggled` signal. To assign a callback to this event, each button in the group must register a callback for this signal The `active` property can be queried to decide if the toggle is from being selected, or deselected.

Example 2.12: Radio group construction
Creating a new radio button group follows this pattern:

25

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

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

```
w <- gtkWindow(); w$setTitle("Radio group example")
g <- gtkVBox(FALSE, 5); w$add(g)
QT <- sapply(l, function(i) g$packStart(i))
```

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

```
l[[3]]$setActive(TRUE)
sapply(l, function(i) i$getActive())
```

two.sided	less	greater
FALSE	FALSE	TRUE

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

```
QT <- sapply(l, function(i)
      gSignalConnect(i, "toggled",      # attach each to "toggled"
                     f = function(w, data) {
                       if(w$getActive()) # set before callback
                         cat("clicked", w$getLabel(),"\n")
                     }))
```

The `RGtk2` package coverts a list in R to the appropriate list for GTK+. However, you may wisth to refer to this list within a callback, but only the current radio button is passed through. Rather than passing the list through the `data` argument or using a global, The `GetGroup` method can be used to reference the buttons stored within a radio group. This method returns a list containing the radio button. However, it is in the reverse order of how they were added (newest first). (As GLib list uses prepend to add elements, not append, as it is more efficient.)

Example 2.13: Radio group using `GetGroup`
In this example below, we illustrate two things: using the `NewWithLabelFromWidget` method to add new buttons to the group and the `GetGroup` method to reference the buttons. The `rev` function is used to pack the widgets, to get them to display first to last.

```
radiogp <- gtkRadioButton(label=vals[1])
for(i in vals[-1])
  radiogp$newWithLabelFromWidget(i)
w <- gtkWindow(); w['title'] <- "Radio group example"
g <- gtkVBox(); w$add(g)
```

26

```
QT <- sapply(rev(radiogp$getGroup()),          # reverse list
                function(i) g$packStart(i))
```

2.10 Combo boxes

A basic combobox is constructed by `gtkComboBoxNewText`. Later we will
discuss more complicated comboboxes. Unlike others, as of writing, this
widget must have its `Show` method called to be mapped.

For the basic combobox, items may be added to the combobox in a few
manners: to add to the end or beginning we have `AppendText` and `PrependText`;
to insert within the list the `InsertText` method is used with the argument
`position` specified in addition to the argument `text` to indicate the index
where the values should added. (The prepend method would be index 0,
the append method would be with an index equal to the number of existing
items.)

The currently selected value is specified by index with the method `SetActive`
and returned by `GetActive`. The index, as usual, is 0-based, and in this case
uses a value of -1 to specify that no value is selected. The `GetActiveText`
method can be used to retrieve the text shown by the basic combo box

It can be difficult to use a combobox when there are a large number of
selections. The `SetWrapWidth` method allows the user to specify the preferred
number of columns to be used to display the data.

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

Example 2.14: Combo box
A simple combobox may be produced as follows:

```
vals <- c("two.sided", "less", "greater")
cb <- gtkComboBoxNewText()
for(i in vals) cb$appendText(i)
cb$setActive(0)                                 # first one
ID <- gSignalConnect(cb, "changed",
                f = function(w, ...) {
                  i <- w$getActive() + 1 # shift index
                  if(i == 0)
                     cat("No value selected\n")
                  else
                     cat("Value is", w$getActiveText(), "\n")
                })
```

A simple GUI is shown, the call to `ShowAll` is use here, as this widget does
not get mapped otherwise.

```
## show it
```

27

```
w <- gtkWindow(show=FALSE); w['title'] <- "Combobox example"
w$add(cb)
w$showAll()     # propogate down to cb
```

Sliders

The slider widget and spinbutton widget allow selection from a regulary
spaced list of values. In GTK+ these values are stored in an adjustment
object, whose details are mostly hidden in normal use.

The slider widget in GTK+ may be oriented either horizontally or ver-
tically. The decision is made through the choice of constructor: `gtkHScale`
or `gtkVScale`. For these widgets, the adjustment can be specified – if de-
sired, or for convenience, will be created if the arguments `min`, `max`, and
`step` are given. These arguments take numeric values. As the first argument
(`adjustment`) is used to specify an adjustment, these values are best specified
by name. Alternatively, the `gtkHScaleNewWithRange` constructor can be used
with positional arguments for `min`, `max` and `step`.

The methods `getValue` and `getValue` can be used to return and set the
value of the widget. When assigning a value, values outside the bounds will
be set to the minimum or maximum value.

A few properties can be used to adjust the appearance of the slider wid-
get. The `digits` property controls the number of digits after the decimal
point that are displayed. The property `draw-value` can be used to turn off the
drawing of the selected value near the slider. Finally, the property `value-pos`
specifies where this value will be drawn using values from `GtkPositionType`.
The default is `top`.

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

Example 2.15: A slider controlling histogram bin selection
A simple mechanism to make a graph interactive, is to have the graph redraw
wheneven a slider has its value changed. The following shows how this can
be achieved.

```
library(lattice)
w <- gtkWindow(); w$setTitle("Histogram bin selection")
slider <- gtkHScaleNewWithRange(1, 100, 1) # min, max, step
slider['value-pos'] <- "bottom"
w$add(slider)
## initial graphic
x <- rnorm(100)
print(histogram(x), nint = slider$getValue())
ID <- gSignalConnect(slider,"value-changed",
                f = function(w, ...) {
                    val <- w$getValue()
```

```
                    print(histogram(x, nint=val))
                  })
```

Spinbuttons

The spinbutton widget is very similar to the slider widget in GTK+. Spinbuttons are constructed with `gtkSpinButton`. As with sliders, this constructor allows a specification of the adjustment with an actual adjustment, or through the arguments `min`, `max`, and `step`.

As with sliders, the methods `GetValue` and `SetValue` are used to get and set the widgets value. The property `snap-to-ticks` can be set to `TRUE` to force the new value to be one of sequence of values in the adjustment. The `wrap` property indicates if the widget will "wrap" around when at the bounds of the adjustment.

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

Example 2.16: A range widget
This example shows how to make a range widget that combines both the slider and spinbutton to choose a single number. Such a widget is popular, as the slider is easy to make large changes and the spinbutton better at finer changes. In GTK+ we use the same adjustment, so changes to one widget propogate without effort to the other.

```
## make a range widget combining both a slider and spinbutton to choose a number
library(RGtk2)
```

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

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

The slider is drawn without a value, so that the user sees only that in the spinbutton. This spinbutton is created by specifying the adjustement created when the slider widget is.

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

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

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

29

The cairoDevice package

The package `cairoDevice` describes itself as a "Cairo-based cross-platform antialiased graphics device driver." It can be embedded in a `RGtk2` GUI as with any other widget. Its basic usage involves a few steps. First a new drawing area is made with `gtkDrawingArea`. This drawing area can be used by various drawing functions, that we do not describe. (In fact, arbitrary widgets, such as pixbufs, can be used here.) The `cairoDevice` package provides the function `asCairoDevice` to coerce the drawing area to a graphics device. This function has standard argument `pointsize` and for some underlying widgets `width` and `height` arguments.

2.11 Containers

In addition to boxes, there are a number of useful containers detailed next. Each of the following, except for the table widget, inherit the `Add` method and allow for only one child component.

Framed containers

The `gtkFrame` function constructs a container with a decorative frame to set off the containers components. The optional `label` argument can be used to specify the `label` property. This can be subsequently retrieved and set using the `GetLabel` and `SetLabel` methods. The label can be aligned using the `SetLabelAlign` method. This has arguments `xalign` and `yalign`, with values in $[0,1]$, to specify the position of the label relative to the frame.

Frames have a decorative shadow whose type is stored in the `shadow-type` property. This type is a value from `GtkShadowType`.

Expandable containers

Although they are a little unresponsive due to eventloop issues, an expandable container proves quite useful to manage screen space. Expandable containers are constructed by `gtkExpander`. Use `gtkExpanderNewWithMnemonic` if a mnemonic is desired. The containers are drawn with a trigger button and optional label, which can be clicked on to hide or show the containers child.

The label can be given as an optional argument to the constructor, or assigned later with the `SetLabel` method, The label can use Pango markup. This is indicated by setting the `use-markup` property through `SetUseMarkup`.

The state of the widget is stored in the `expanded` property, which can be accessed with `GetExpanded` and `SetExpander`. When the state changes, the `activate` signal is emitted.

30

2.12 Divided containers

The `gtkHPaned` and `gtkVPaned` create containers with a "gutter" to allocate the space between its two children. Like the `gtkBin` containers, the two spaces allow only one child component. The two children may be added two different ways. The methods `Add1` and `Add2` simply add the child, whereas the methods `Pack1` and `Pack2` have arguments `resize` and `shrink` which specify how the child will resize if the paned container is resized. These two arguments take logical values. After children are added, they can be referenced from the container through its `GetChild1` and `GetChild2` methods.

The position of the gutter can be set with the `SetPosition` method. This is given in terms of screen position. The properties `min-position` and `max-position` can be used to convert a percentage into a screen position.

The `move-handle` signal is emitted when the gutter position is changed.

2.13 Notebooks

The `gtkNotebook` constructor creates a notebook container. The default position of the notebook tabs is on the top, starting on the left. The property `tab-pos` property (`SetTabPos`) uses the `GtkPositionType` values of `"left"`,`"right"`,`"top"`, or `"bottom"` to adjust this. The property `scrollable` should be set to `TRUE` to have the widget gracefully handle the case when there are more page tabs than can be shown at once. If the same size tab for each page is desired, the method `SetHomogeneousTabs` can be called with a value of `TRUE`.

Adding pages to a notebook   New pages can be added to the notebook with the `InsertPage` method. Each page of a notebook holds one child component. This is specified with the `child` argument. The tab label can be specified with the `tab.label` argument, but can also be set later with `SetTabLabel` and retrieved with `getTabLabel`. The label is specified using a widget, such as a `gtkLabel` instance, but this allows for more complicated tabs, such as a box container with a close icon. The `SetTabLabelText` can be used if just a text label is desired. To use this method, the child widget is needed. the `[[` method can be used to retrieve this. So can the `GetNthPage` method. Both are an alternative to getting all the children returned as a list through `GetChildren`. By default, the new page will be at the last position (the same as `AppendPage`). This can be changed by supplying the desired position to the argument `position` using 0-based indexing. The default value is −1, indictating the last page.

Rearranging pages   Pages can be reordered using the `ReorderChild` method. The arguments are the `child` for the child widget, and `position`, again 0-based with −1 indicating appending at the end. Pages can be deleted using

31

the method `RemovePage`. The `page.num` argument specifies the page by its position. If the child is known, but not the number the method `PageNum` returns the page number. Its argument is `child`.

The current page number is stored in the `page` property. The number of pages can be found by inspecting the length of the return value of `GetChildren`, but more directly is done with the method `GetNPages`. A given page can be raised with the `SetCurrentPage` method. The argument `page.num` specifies which page number to raise. If the child container should not be hidden, or the page won't change. Incremental movements are possible through the methods `NextPage` and `PrevPage`.

Signals   The notebook widget emits various signals when its state is changed. Among these: the signal `focus-tab` is emitted when a tab receives the focus, `select-page` is similar and the `switch-page` is emitted when the current page is changed.

Example 2.17: Adding a page with a close button
A familiar element of notebook tabs from web browsing is a close button. The following defines a new method `InsertPageWithCloseButton` that will use an "x" to indicate a close button. An icon would be prettier, of course. The callback passes both the notebook and the page through the `data` argument, so that the proper page can be deleted. One caveat, the simpler command `nb$getCurrentPage()` will return the page of the focused tab prior to clicking the "x" button, which may not be the correct page to close.

```
gtkNotebookInsertPageWithCloseButton <-
  function(object, child, label.text="", position=-1) {
    label <- gtkHBox()
    label$packStart(gtkLabel(label.text))
    label$packEnd(b <- gtkButton("x"))    # prettier with icon
    ID <- gSignalConnect(b,"clicked",
                    function(userData, b, ...) {
                      nb <- userData$nb
                      page <- userData$page
                      nb$removePage(nb$pageNum(page))
                    },
                    data = list(nb=object, page=child),
                    user.data.first=TRUE)
    object$insertPage(child, label, position)
  }
```

We now show a simple usage of a notebook.

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

32

```
QT <- nb$insertPageWithCloseButton(gtkButton("world"), label.text="page 2")
```

Scrollable windows

Scrollbars allow components that are larger than the space allotted to be interacted with, by allowing the user to ajdust the visible portion of the componented. Scrollbars in GTK+ use adjustments (like the spin button widget) to control the x and y position of the displayed portion of the component.

The convenience function `gtkScrolledWindow` creates a window that allows the user to scroll around its child component. By default, the horizontal and vertical adjustments are generated, although, if desired, these may be specified by the programmer.

Like a top-level window, a scrolled window is a `GtkBin` object and has only one immediate child component. If this child is a tree view, text view (discussed in the following), icon view, layout or viewport the `add` method is used. Otherwise, the method `addWithViewport` can be used to create an intermediate viewport around the child.

The properties `hscrollbar-policy` and `vscrollbar-policy` determine if the scrollbars are drawn. By default, they are always drawn. The `GtkPolicyType` enumeration, allows a specification of `"automatic"` so that the scrollbars are drawn if needed, i.e, the child component requests more space than can be allotted. The `setPolicy` method allows both to be set at once, as in the following example.

Example 2.18: Scrolled window example
This example shows how a scrolled window can be used to display a long list of values. The tree view widget can also do this, but here we can very easily customize the display of each value. In the example, we simply locates where a label is placed.

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

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

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

```
w <- gtkWindow(show=FALSE)
w$setTitle("Scrolled window example")
```

33

```
w$setSizeRequest(-1, 300)
w$add(sw)
w$show()
```

2.14 Tabular layout

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

Child components are added to this container through the `AttachDefaults` method. Its first argument, `child`, is the child component. This component can span more than one cell. To specify which cells, the arguments `left.attach` and `right.attach` specify the columns through the column number to attach the left (or right) side of the child to, and `top.attach` and `bottom.attach` to specify the rows.

The `Attach` method is similar, but allows the programmer more control over the placement of the child component. This method has the arguments `xoptions` and `yoptions` to specify how the widget responds to resize events. These arguments use the values of `GtkAttachOptions` to specify either `"expand"`, `"shrink"` and/or `"fill"`. Just `"fill"` will cause the widget to remain the same size if the window is enlarged, the `"expand"` and `"fill"` combination will cause the component to fill the available space, and the `shrink` option instructs the widget to shrink if the table is made smaller through resizing. Finally, the `xpadding` and `xpadding` arguments allow the specification of padding around the cell in pixels.

Anchoring of widgets within a cell can be done by setting the `xalign` and `yalign` properties of the child widgets.

Example 2.19: Layout of calculator buttons
The following example shows how to layout calculator buttons. In this example we want each button click to call the same callback, passing in user data to parameterize the call. To do this efficiently, we define a new method `GetLabelsFromDataFrame` which will map the entries in a data frame to the button labels. This method is defined as,

```
gtkTableGetLabelsFromDataFrame <-
   function(object, dataFrame, callBack, userData=NULL) {
     checkPtrType(object, "GtkTable")
     dataFrame <- as.data.frame(dataFrame)
```

34

```
    d <- dim(dataFrame)
    object$resize(d[1], d[2])              # resize dynamically
    for(i in 1:d[1]) {
      for(j in 1:d[2]) {
        child <-  gtkButton(as.character(dataFrame[i,j]))
        id <- connectSignal(child,"clicked", f=callBack,
                              data=list(i=i, j=j, data=userData),
                              user.data.first=TRUE)
        ## 0-based for attaching
        object$attach(child,
                      left.attach=j-1, j, top.attach=i-1, i)
      }}}
```

The callback function should have signature (`userData`,`button`). We use the following matrix to store our calculator layout.

```
m = rbind(
  c("^", "(", ")", " / ", " == "),
  c(7  ,  8,   9, " * ", " > "),
  c(4  ,  5,   6, " - ", " >= "),
  c(1  ,  2,   3, " + ", " < "),
  c(0  , ".", " ", " ! ", " <= ")
  )
```

Finally, we create our GUI using the method above to do the bulk of the work.

```
tab <- gtkTable(homogeneous=TRUE)        # same size buttons
tab$getLabelsFromDataFrame(m, function(userData, b)
                                cat(b$getLabel()))
w <- gtkWindow(show=FALSE)
w$setTitle("Show matrix")
w$add(tab)
w$show()
```

Example 2.20: Dialog layout
This example shows how to layout some controls for a dialog with some attention paid to how the widgets are aligned and how they respond to resizing of the window.

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

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

We define our widgets first then deal with their layout.

```
l1 <- gtkLabel("Sample size")
w1 <- gtkComboBoxNewText()
```
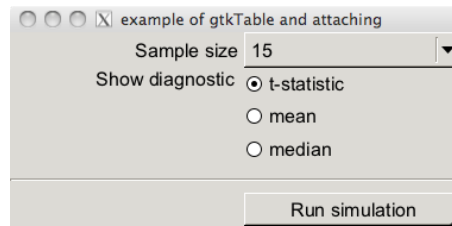
35

Figure 2.5: A basic dialog using a `gtkTable` container for layout.

```
QT <- sapply(c(5, 10, 15, 30), function(i) w1$appendText(i))
l2 <- gtkLabel("Show diagnostic ")
w2 <- gtkVBox()
rb <- list()
rb[["t"]] <- gtkRadioButton(label="t-statistic")
for(i in c("mean","median")) rb[[i]] <- gtkRadioButton(rb, label=i)
QT <- sapply(rb, function(i) w2$packStart(i))
w3 <- gtkButton("Run simulation")
```

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

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

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

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

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

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

Finally, we use the `ShowAll` method so that it propogates to the combobox.

```
w$showAll()                              # propogate to combo
```

2.15 Drag and drop

GTK+ has mechanisms to provide drag and drop features to widgets. Only widgets which can receive signals will work for drag and drop, so to drag or drop on a label an event box must be used. To setup drag and drop actions requires setting a widget to be a source for a drag request, and setting a widget to be a target for a drop action. We illustrate how to set up the dragging of a text value from one widget to another. Much more complicated examples are possible, but we do not pursue it here.

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

```
## info arguments -- application assigned
TARGET.TYPE.TEXT   = 80                    #
TARGET.TYPE.PIXMAP = 81                    #
TARGET.TYPE.OBJECT = 82
widgetTargetTypes = list(
## target -- string representing the drag type
## flag delimiting drag scope. 0 -- no limit
## info -- application assigned value to identify
text = gtkTargetEntry("text/plain", 0, TARGET.TYPE.TEXT),
pixmap = gtkTargetEntry("image/x-pixmap", 0, TARGET.TYPE.PIXMAP),
object = gtkTargetEntry("text/plain", 0, TARGET.TYPE.OBJECT)
)
```

To set a widget to be a drag source requires calling `gtkDragSourceSet`. This function has arguments `object` for the widget we are making a source, `start.button.mask` to specify which mouse buttons can initiate the drag, `targets` to specify the target type, and `actions` to indicate which of the `GdkDragAction` types is in effect, for instance `copy` or `move`.

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

```
widget <-  gtkButton("Drag me")
w <- gtkWindow(); w$add(widget)
QT <-
  gtkDragSourceSet(widget,
                   start.button.mask=c("button1-mask", "button3-mask"),
                   targets=widgetTargetTypes[["text"]],
                   actions="copy") ## can also be any of GdkDragAction
ID <-
  gSignalConnect(widget, "drag-data-get",
                 f=function(userData, widget, context,
```

```
                          selection, targetType, eventTime) {
                          selection$setText(str="some value")
                          return(TRUE)
                        },
                        user.data.first=TRUE,
                        data="")
```

To make an object a drop target, we call `gtkDragDestSet` on the object with new argument `flags` for specifying the actions GTK+ will perform when the widget is dropped on. We use the value `"all"` for `"motion"`, `"highlight"`, and `"drop"`. Additionally, values for `targets` and `actions` are given.

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

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

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

If the action was `"move"` then the source widget emits the `drag-data-delete` signal, so that a callback can be defined to handle the deletion of the data.

RGtk2: Widgets Using Models

Many widgets in GTK+ use the model, view, controller paradigm. While many times this is in the background, for the widgets in this chapter one needs to be aware of the usage. This framework adds a layer of complexity, in exchange for more flexibility.

3.1   Text views and text buffers

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

   A text view is created with `gtkTextView`. The `buffer` argument is used to specify a text buffer, otherwise one will be created. This buffer is returned by the method `getBuffer` and may be set for a view with the `setBuffer` method. Text views are typically placed inside a scrolled window, and since a viewport is established, this is done with the `add` method for scrolled windows.

   Text may be added programmatically through various methods of the text buffer. The easiest to use are `setText` which simply replaces the current text with that specified by `text`. The method `insertAtCursor` will add the text to the buffer at the current position of the cursor. Other means are described after the first example.

Properties   Key properties of the text view include `editable`, which if assigned a value of `FALSE` will prevent users from editing the text. If the view is not editable, the cursor may be hidden by setting the `cursor-visible` property to `FALSE`. The text in a buffer may be wrapped or not. The method `setWrapMode` takes values from `GtkWrapMode` with default of `"none"`, but options for `"char"`, `"word"`, or `"word_char"`. The justification for the entire

buffer is controlled by the `justification` property which takes values of `"left"`, `"right"`, `"center"`, or `"fill"` from `GtkJustification`. The global value may be overwridden for parts of the text buffer through the use of text tags. The left and right margins are adjusted through the `left-margin` and `right-margin` properties.

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

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

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

Example 3.1: Simple textview usage
We illustrate the basics of using a text view, including setting some of the view's properties.

```
tv <- gtkTextView ()
sw <- gtkScrolledWindow ()
sw$setPolicy ("automatic","automatic")
sw$add (tv)
w <- gtkWindow (); w$add (sw)
tv['editable'] <- TRUE
tv['cursor-visible'] <- TRUE
tv['wrap-mode'] <- "word"              # GtkWrapMode value
tv['justification'] <- "left"          # GtkJustification value
tv['left-margin'] <- 20                # 0 is default
tb <- tv$getBuffer ()
```

40

```
tb$setText("the quick brown fox jumped over the lazy dog")
font <- pangoFontDescriptionFromString("Serif,monospace bold italic 8")
tv$modifyFont(font)
```

Tags, iterators, marks

In order to do more with a text buffer, such as retrieve the text, retrieve a selection, or modify attributes of just some of the text, one needs to become familiar with how pieces of the buffer are referred to within Rgtk2.

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

Iterators   A text iterator is used to specify a position in a buffer in a transient manner. Iterators become invalid as soon as a buffer changes. In RGtk2, iterators are stored as lists with components iter to hold a pointer to the underlying iterator and component retval to indicate whether the iterator when it was returned is valid. Many methods of the text buffer will update the iterator. This can happen inside a function call where the iterator is passed as an argument, so is unexpected to the R user. The copy method will create a copy of an iterator, in case one is to be modifed yet the original is desired.

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

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

There are several methods for iterators that allow one to refer to positions in the buffer relative to the iterator, for example, these with obvious names to move a character or characters: forwardChar, forwardChars, backwardChar, and backwordChars. As well, there are methods to move to the end or be-

41

ginning of the word the iterator is in or the end or beginning of the sentence (`forwardWordEnd`, `backwardWordStart`, `backwardSentenceStart`, and `forwardSentenceEnd`). There are also various methods, such as `insideWord`, returning logical values indicating if the condition is met. To use these methods, the iterator in the `iter` component is used, not the value returned as a list. Example 3.2 shows how some of the above are used, in particular how these methods update the iterator rather than return a new one.

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

Example 3.2: Finding the word one clicks on
This example shows how one can find the iterator corresponding to a mouse-button-press event. The callback has an event argument which is a `GdkEventButton` object with methods `getX` and `getY` to extract the x and y components of the event object. These give the position relative to the widget. The methods `getXRoot` and `getYRoot` give the position relative to the parent window the widget resides in.

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

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

42

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

There are many text buffer methods to work with marks. The `getMark` method will return the mark object for a given name. (There are functions which refer to the name of a mark, and others requiring the mark object.) The method `getIterAtMark` will return an iterator for the given mark to be used when an iterator is needed.
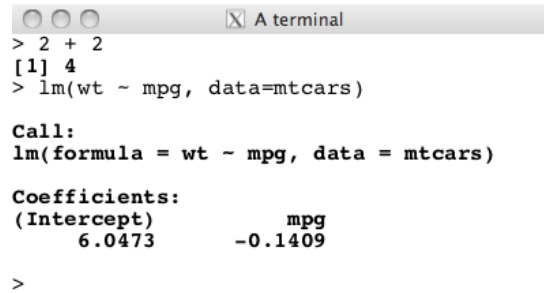
Tags Marks and iterators can be used to specify different properties for different parts of the text buffer. GTK+ uses tags to specify how pieces of text will differ from those of the textview overall. To create a tag, the `createTag` method is used. This has optional argument `tag.name`gtkTextBufferCreateTag which can be used to refer to the tag later, and otherwise uses named arguments so specify a properties names and the corresponding values. These tags may be applied to the text between two iters using the methods `applyTag` or `applyTagByName`.

Example 3.3: Using text tags
We define two text tags to make text bold or italic and illustrate how to apply them.

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

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

43

Figure 3.1: A basic R terminal implemented using a `gtkTextView` widget.

Example 3.4: A simple command line interface
This example shows how the text view widget can be used to make a simple command line. While programming a command line isn't likely to be the most common task in designing a GUI for a statistics application (presumably you are already using a good one), 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
```

Our main object will be the text buffer which will have only one view. As there is no built-in method to return a corresponding view from the buffer, we use the `setData` method to associate the view with the buffer.

```
tb$setData("textview", tv)
```

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

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

We define one new mark to mark the prompt for a new line. We need to be able to identify a new command, and this marks the beginning of this command.

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

44

We define several functions, which we think of as methods of the text buffer (not the text view). This first shows how to move the viewport so that the command line is visibile.

```
moveViewport <- function(obj) {
  tv <- obj$getData("textview")
  endIter <- obj$getEndIter()
  QT <- tv$scrollToIter(endIter$iter, 0)
}
```

There are two types of prompts needed. This function adds a new one or a continuation one. An argument allows one to specify that the `startCmd` mark is set.

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

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

This helper method is used to write the output of a command to the text buffer. We arrange to truncate large outputs. By passing in the tag name, we could reuse this function. If we were to streamline the code for this example, we might use this function to also write out the error messages, but leave that to the similarly defined function `addErrorMessage` (not shown).

```
addOutput <- function(obj, output, tagName="cmdOutput") {
  if(length(output) > 100)                    # shorten if needed
    out <- c(output[1:100], "...")

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

  addPrompt(obj, "prompt", setMark=TRUE)
  obj$applyTagByName("uneditable", obj$getStartIter()$iter,
                       obj$getEndIter()$iter)
  moveViewport(obj)
}
```

45

This next function uses the `startCmd` mark and the end of the buffer to extract the current command. Multi-line commands are handled through a regular expression which should not be hard-coded to the standard continue prompt, but for sake of simplicity is.

```
findCMD <- function(obj) {
  endIter <- obj$getEndIter()
  startIter <- obj$getIterAtMark(startCmd)
  cmd <- obj$getText(startIter$iter, endIter$iter, TRUE)

  cmd <- unlist(strsplit(cmd, "\n[+] ")) # hardcoded "+"
  cmd
}
```

The following function takes the current command and does the appropriate thing. It uses a hack (involving `grep`) to distinguish between an incomplete command and a true syntax error. The `addHistory` call refers to a function that is not shown, but is left to illustrate where one would add to a history stack if desired.

```
evalCMD <- function(obj, cmd) {
  cmd <- paste(cmd, sep="\n")
  out <- try(parse(text=cmd), silent=TRUE)
  if(inherits(out, "try-error")) {
    if(length(grep("end", out))) {        # unexpected end of input
      ## continue
      addPrompt(obj, "continue", setMark=FALSE)
      moveViewport(obj)
    } else {
      ## error
      addErrorMessage(obj, out)
    }
    return()
  }
  addHistory(obj, cmd)  ## if keeping track of history

  out <- capture.output(eval(parse(text = cmd), envir=.GlobalEnv))
  addOutput(obj, out)
}
```

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

```
ID <- gSignalConnect(tv, "key-release-event", f=function(w, e, data) {
  obj <- w$getBuffer()                      # w is textview
  keyval <- e$getKeyval()
  if(keyval == GDK_Return) {
```

46

```
    cmd <- findCMD(obj)
    if(nchar(cmd) > 0)
      evalCMD(obj, cmd)
  }
  return(FALSE)                            # events need return value
})
```

Figure 3.1 shows the widget placed into a very simple GUI.

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

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

3.2   Views of tabular and heirarchical data

Widgets to create comboboxes, display tabular data values, and to display tree-like data are treated similarly in GTK+. Each uses the MVC paradigm and for these, the models are defined similarly. We begin by discussing the models, then present the various views. Each view is described by its column, which in turn have their cells specifed by cell renderers.

Tabular stores and tree stores

GTK+ provides list stores and tree stores as models to hold tabular and heirarchical data to be viewed through various widgets, such as the combo box or tree view. Like a data frame, each row in these stores contains data of varying types. The main difference between the two is that tree stores also have information about about whether a row has any offspring. The list store is just a tree store where there are no children of the top-level offspring.

For speed, much greater convenience and familiarity purposes, `RGtk2` provides a third store through `rGtkDataFrame` for storing data frames.

rGtkDataFrame   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 is also the way tree views are organized when no heirarchical structure is needed. As such it is natural to have a means to map a data frame into a store for a tree view. The `rGtkDataFrame` constructor does this, producing an object that can be used as the model for a view. This R-specific addition

47

to GTK+ not only is more convenient, it has the added bonus of being especially fast. The constructor takes a data frame as an argument. The column classes are important, so even if this data frame is empty, it should specify the desired column classes.

The constructor produces an object of class `RGtkDataFrame` which has a few methods defined for it. The familiar S3 methods `[`, `[<-`, `dim`, and `as.data.frame` are defined. The `dimnames` attributes are kept, but have no well-defined meaning for this model. 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`, column types should not be changed which can be an issue with coercion to character from numeric say, rows can not be dropped. To add a new column or row, the methods `appendColumns` and `appendRows` may be used, where the new column or row may be given as the argument.

To remove rows from this model, the `setFrame` method can be used to specify the new data. This method can also be used to replace the existing data in the model with a new data frame. There are few issues though. If the new data frame has more rows or columns, then the appropriate `append` method should be used first. As well, one should not change the column classes with the new frame, as views of the model may be expecting a certain class of data.

Example 3.5: Defining and manipulating a data store
The basic data frame methods are similar.

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

```
[1] 12
```

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

To change the backend data, we can use the `SetFrame` method:

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

List stores and tree stores   Although the `rGtkDataFrame` model is very useful, there are times when it can't be employed. List stores can be used when the underlying data contains values that can not be stored in a data frame (such a images) and tree stores are used for heirarchical data.

A tree store or list store is constructed using `gtkTreeStore` or `gtkListStore`. Both are interfaces for the abstract `GtkTreeModel` class. 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

48

logical data, `gint` for integer data, `gdouble` for numeric data, and `GObject` for GTK+ objects, such as pixbufs.

Iterators and tree paths   Similar to a text buffer, a list store uses transient iterators to refer to position – in this case the row – within a store. One can also refer to position through a path, which for a list store is essentially the row number, 0-based, as a character; and for a tree is a colon-separated set of values referring to the offspring (`"a:b:c"` indicates the cth child of the bth child of a). A third way, through a row reference, is not discussed here.

A `GtkTreePath` object is created by the constructor `gtkTreePathNewFromString` which takes a string specifying the position. To retrieve this string from a path object, the `toString` method can be used.

Paths are convenient, as they are human readable, but iterators are empolyed by the various methods and more easily allow the programmer to traverse the store. One can flip between the two representations. The iterator referring to the path can be returned by the method `getIterFromString`. The method `getStringFromIter` will return the string. The tree path object itself is returned by the method `getPath`. In `RGtk2` iterators are lists with component `retval` indicating if this is a valid iterator and a component `iter` holding the object of the `GtkTreeIter` class.

Adding values to a store   Values are added to and returned from a store by specifying the row and column for the value. The row is specified by an iterator and the columns by its index, 0-based. The method `setValue` is used to specify value by value, whereas an entire row can be assigned through the `set` method. The former has arguments `iter`, `column`, `value`, in that order; the latter has no `column` or `value` argument. Instead, `Set` 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 $2k + 1$). When calling `setValue` or `set` the iterator updates to the next row Values are returned by the `getValue` method, a list with component `value` storing the value.

Finding iterators   For a list or tree model, an iterator for the first child is returned by `getIterFirst`. This iterator corresponds to the path `"0"`. The `append` method for the store returns an iterator indicating the next value at the end of the store (this is slightly different from the GTK+ function which modifies an iterator passed as an argument). The `prepend` method is similar, only returning an iterator pointing to the initial row. Other methods allow for specifying postion relative to some row. The `insert` method is used to return an iterator that allows one to insert a row at a position specified to its `position` argument. (The `Prepend` method is similar to using `postion=0`). To avoid the two-step approach of getting the iterator, then assigning the value,

the method `insertWithValues` can be used, where values are specified as with the `Set` method. The `insertBefore`, and `insertAfter`, methods take an iterator, `sibling` and will return an iterator indicating the postion just before or after the sibling.

Example 3.6: Appending to a list store
To illustrate, to create a simple list store to hold a column of text we have:

```
lstore <- gtkListStore("gchararray")
for(i in 1:nrow(Cars93)) {
  iter <- lstore$append()
  if(is.null(iter$retval))
    lstore$setValue(iter$iter, 0, Cars93[i,1])
}
```

To retrieve a value, we have this example to get the first one in the store:

```
iter <- lstore$getIterFirst()              # first row
lstore$getValue(iter$iter, column = 0)
```

```
$retval
NULL


$value
[1] "Acura"
```

Adding heirarchical information For a tree store, the methods `append`, `prepend` etc. are similar to that for a list store with the difference being that a `parent` argument is used for tree stores to specify in iterator for the parent of the new row, thereby creating the heirarchical structure of a tree.

Example 3.7: Defining a tree
As an application, we can create a tree with parents the car manufacturers in the `Cars93` data set, and children the makes of their cars, as follows:

```
tstore <- gtkTreeStore("gchararray")
Manufacturers <- Cars93$Manufacturer
Makes <- split(Cars93[,"Model"], Manufacturers)
for(i in unique(Manufacturers)) {
  piter <- tstore$append()              # parent
  tstore$setValue(piter$iter, column=0, value=i)
  for(j in Makes[[i]]) {
    sibiter <- tstore$append(parent=piter$iter) # child
    if(is.null(sibiter$retval))
      tstore$setValue(sibiter$iter,column=0, value=j)
  }
}
```

50

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

```
 iter <- tstore$getIterFromString("0:0") #  the 1st child of root
 tstore$getValue(iter$iter,column=0)$value
```

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

Manipulating rows   Rows within a store can be rearranged using the methods `swap` to swap rows referenced by their iterators; `moveAfter` to move one row after another, both referenced by iterators, although if the last is blank, the end of the store is assumed; and `moveBefore`, where if the second iterator is blank the first position is assumed. To totally reorder the store, the `reorder` method is available. Its `new.order` argument specifies the new order as row indices. For tree stores, these rows are the children of the `parent` argument.

Once added, rows may be removed using the `remove` method. The iterator for the row to delete is given as an argument. The store's entire contents can be removed by its `clear` method.

Traversing the store   An iterator points to a row in a tree or list store. For both lists and trees, an interator pointing to the next row (at the same level for trees) is produced by the method `iterNext`. This method returns `FALSE` if no next row exists. Otherwise, it updates the iterator in place. (That is, calling `store$iterNext(iter$iter)` updates `iter`, despite it not being assigned to.) The path method `prev` will point to the previous child at the same depth in a tree, but no such method is defined for iterators. One could be, for example the following will do so for both list and tree stores.

```
 gtkTreeModelIterPrev <- function(object, iter) {
   path <- object$getPath(iter)
   ret <- path$prev()
   if(ret)
     return(list(retval=NULL, iter=object$getIter(path)$iter))
   else
     return(list(retval=FALSE,iter=NA))
 }
```

For trees, the method `iterParent` returns an iterator to point to the parent row, if no parent is found. The `retval` component is `FALSE`. There are several methods when a row has children. The method `iterHasChild` returns a logical indicating if a row has children. The method `iterChildren` returns an iterator to point to the first child of parent. If no child exists, the `retval` component is `FALSE`. If an iterator for the nth child is desired, the method `iterNthChild` can be used. Again, it returns an iterator referring to the nth child, or has a `retval` of `FALSE` if none exists. To find the number of children, the method `iterNChildren` is provided. This method returns 0 if there are no children.
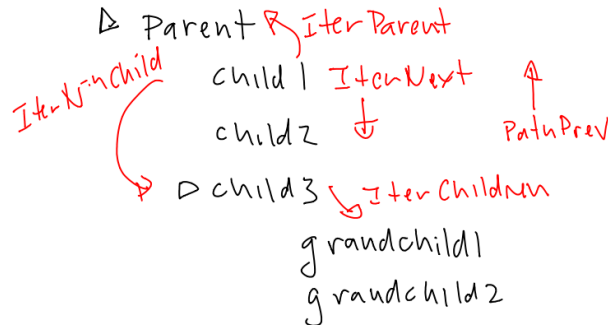
51

Figure 3.2: [REPLACEME!] Graphical illustration of the functions used by iters to traverse a tree store.

Cell renderers

The various views ultimately display the information in the model column by column (a combobox having one column). Within each column, the display is controlled by cell renderers, which are used to specify how each cell is layed out. Cell renderers are used whenever the `gtkCellLayout` interface is implemented, such as with comboboxes and tree views, but also other widgets not discussed here.

A cell renderer is customized by adjusting its attributes. These attributes are documented in the help pages for the corresponding constructor. These attributes can be set to one value for all rows, or can be set to depend on a corresponding row in the model. The latter allows them to change from cell to cell. For example, the `text` attribute of the text cell renderer would usually get its values from the model, as that would vary from cell to cell, but a background color (`background`) might be common to the column. The `addAttribute` method is used to associate a column in the store with a cell renderer's attribute.

There are many different cell renderers, we mention first the text and pixbuf renderer, as they are commonly used in comboboxes. With the discussion of tree views, we mention others.

Text cell renderers The `gtkCellRendererText` constructor is used to display text and numeric values. Numeric values are shown as strings, but are not converted in the model. For the text renderer, important properties are `text` to indicate the column in the data store that the text for the cell is to come from, `font` to specify the font from a string, `size` for the font size, `background` for the background color and `foreground` for the text color (as strings).

52

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

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

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

Pixbuf cell renderers Graphics can be added to the cell with the renderer `gtkCellRendererPixbuf`. The graphic can be spepcified by its `stock-id` attribute as a character string, or `icon-name` for a themed icon. It can also be specifed as an image object, through the `pixbuf` attribute. Pixbuf objects can be placed in a list store using the `GObject` type. A simple use, might be the following:

```
cr <- gtkCellRendererPixbuf ()
cr['stock.id'] <- "gtk-ok" ## or from a column in a model
```

Combo boxes

The basic combo box usage was discussed in Section 2.10, here we discuss more complicated comboboxes that use a model for the backend This data is tabular and may be kept in a `rGtkDataFrame` instance or a list store.

The basic `gtkComboBoxEntryNewWithModel` constructor allows one to specify the model, and a column where the values are found. For this, the cell renderers (below) are not needed.

If some layout of the values in the combobox is desired, such as adding an image, then the constructor `gtkComboBox` is used. The model may be specified at the time of construction through the optional `model` argument. This model may be changed or set through the `setModel` method and is returned by `getModel`.

The constructor `gtkComboBoxEntry` returns a combobox widget that allows the user to add their own values. This constructor does not allow the model to be specified, so the `SetModel` method must be used. The editable combobox uses a `gtkEntry` object, which can be accessed directly through the `getChild` method of the combobox.

Cellrenderers Comboboxes display rows of data, each row referred to as a cell. In GTK+ each cell is like a box container and can show different bits of information, like an image or text. Each bit of information is presented by a cellrenderer. Cellrenderers are added to the combo box by its `packStart` method. As with box containers, more than one cell renderer can be added per row.

53

To specify the data from the model to be displayed, the `addAttribute` method maps columns of the model to attributes of the cellrenderer.

Retrieving the selected value   For a non-editable combobox, the selected value may be retrieved by index or by iterator. The `getActive` method returns the index of the current selection, 0-based. The value is -1 if no selection has been made. The `getActiveIter` method returns an iterator pointing to the row in the data store. If no row has been selected, the `retval` component of the iterator is `FALSE`. These may be used with the data store to retrieve the value. The data store itself is returned by the `getModel` method.

To set the combobox to a certain index is done through the `setActive` method, using a 0-based index to specify the row.

For editable comboboxes, one can first get the entry widget then call its `GetText` method. The `SetText` method of the entry widget would be used to specify the text.

Signals   When a user selects a value with the mouse, the `changed` signal is emitted. For editable combo boxes, the user may also make changes by typing in the new value. The underlying widget is a `gtkEntry` widget, so the signal `changed` is emitted each time the text is changed and the signal `activate` is emitted by the `gtkEntry` widget when the enter key is pressed. One binds to the signal of the entry widget, not the combobox widget, to have a callback for that event.

Example 3.8: Modifying the values in a combobox
This example shows how to use two comboboxes to achieve a useful task. That being, allowing the user a means to select from the available variables in a data frame. We use a `rGtkDataFrame` model for each, but for one use the basic constructor and the other the more involved, as an illustration.

```
data("Cars93", package="MASS")
dfNames <- c("mtcars", "Cars93")
dfModel <- rGtkDataFrame(dfNames)
dfCb <- gtkComboBoxEntryNewWithModel(dfModel, text.column=0)
```

The variable names are initially just an empty string. We use an `rGtkDataFrame` as the model and also specify a cell renderer to view the data.

```
variableNames <- character(0)
varModel <- rGtkDataFrame(variableNames)
varCb <- gtkComboBoxNewWithModel(varModel)
cr <- gtkCellRendererText()
varCb$packStart(cr)
varCb$addAttribute(cr, "text", 0)
```

Our basic GUI uses a table for layout. Comboboxes fill and expand to fill the cell.

54

```
tbl <- gtkTableNew(rows=2, columns=2, homogeneous=FALSE)
tbl$attach(gtkLabel("Data frame"), left.attach=0,1, top.attach=0,1)
tbl$attach(dfCb, left.attach=1,2, top.attach=0,1)
tbl$attach(gtkLabel("Variables"), left.attach=0,1, top.attach=1,2)
tbl$attach(varCb, left.attach=1,2, top.attach=1,2)
```

This callback will be used for both the entry widget and the combobox, so we first check which it is and if it is the combobox, we get the entry widget from it. To update the display we replace the model. The option of replacing the frame within the current model requires us to be careful when adding additional rows.

```
newDfSelected <- function(varCb, w, ...) {
  if(inherits(w, "GtkComboBox"))         # get entry widget
    w <- w$getChild()
  val <- w$getText()
  df <- try(get(val, envir=.GlobalEnv), silent=TRUE)
  if(!inherits(df, "try-error") && is.data.frame(df)) {
    nms <- names(df)
    ## update model
    newModel <- rGtkDataFrame(nms)
    varCb$setModel(newModel)
    varCb$setActive(-1)
  }
}
```

Our callbacks for the data frame combobox simply call the above function. As for the variable combobox, we show how to get the selected value, but for no real purpose.

```
QT <- gSignalConnect(dfCb, "changed", f=newDfSelected,
                     user.data.first=TRUE,
                     data=varCb)
QT <- gSignalConnect(dfCb$getChild(), "activate", f=newDfSelected,
                     user.data.first=TRUE,
                     data=varCb)
QT <- gSignalConnect(varCb, "changed", f=function(w, ...) {
  model <- w$getModel()
  iter <- w$getActiveIter()
  val <- model$getValue(iter$iter, column=0)
  print(val$value)                        # add real purpose
})
```

Example 3.9: A color selection widget
This examples shows how a combobox can be used as an alternative to `gtkColorButton` to select a color. We use two cellrenderers for each row, one to hold an image and the other a text label.

This function uses the `grid` package to produce a graphic that will read into the pixbuf.

```
require(Cairo)
makePixbufFromColor <- function(color) {
  filename <- tempfile()
  Cairo(file=filename, width=25,height=10)
  grid.newpage()
  grid.draw(rectGrob(gp = gpar(fill = color)))
  dev.off()
  image <- gdkPixbufNewFromFile(filename)
  unlink(filename)
  return(image$retval)
}
```

Our data store has one column for the pixbuf and one for the color text. The pixbuf is stored using the `GObject` class.

```
store <- gtkListStore(c("GObject","gchararray"))
```

This loop adds the colors and their name to the data store.

```
theColors <- palette()                      # some colors
for(i in theColors) {
  iter <- store$append()
  store$setValue(iter$iter, 0, makePixbufFromColor(i))
  store$setValue(iter$iter, 1, i)
}
```

Next we define the combobox using the store as the model. There are two cell renderers to add.

```
combobox <- gtkComboBox(model=store)
## pixbuf
crp <- gtkCellRendererPixbuf(); crp['xalign'] <- 0
combobox$packStart(crp, expand=FALSE)
combobox$addAttribute(crp, "pixbuf", 0)
## text
crt <- gtkCellRendererText();
crt['xpad'] <- 5                              # give some space
combobox$packStart(crt)
combobox$addAttribute(crt, "text", 1)
```

Text entry widgets with completion

A common alternative to a combobox, implemented on many websites, is to add the completion features to a `gtkEntry` instance. When a user types a partial match, all available matches are offered to select from. To implement completion, one creates a completion object with the constructor `gtkEntryCompletion`. The values to complete from are stored in a model, the

example uses an `rGtkDataFrame` instance, which is assigned to the completion object through its `setModel` method. To set the completion for the entry widget, the entry widget's `setCompletion` method is used. The `text-column` property is used to specify which column in the model is used to find the matches, we use the method `SetTextColumn` in the example to set this.

There are several properties that can be adjusted to tailor the completion feature, we metion some of them. Setting the property `inline-selection` to `TRUE` will place the 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 function through the `setMatchFunc` method. The signature of this function is the completion object, the string from the entry widget (lower case), an interator pointing to a row in the model and optionally user data that is passed through the `func.data` argument of the `SetMatchFunc` method. This method should return `TRUE` or `FALSE` depending on whether that row should be displayed in the set of completions.

Example 3.10: Text entry with completion
This example illustrates the basic steps to add completion to a text entry.

The two basic widgets are defined as follows:

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

We will use a `rGtkDataFrame` instance for our completion model, taking a convenient list of names for our example. We set the completion objects's model and text column using the similarly named methods, and then set some properties to customize how the completion is handled.

```
store <- rGtkDataFrame ( data.frame ( name=I ( state.name )))
completion$setModel ( store )
completion$setTextColumn (0)              # which column in model
completion [ 'inline - completion '] <- TRUE # inline with text edit
completion [ 'popup - single - match '] <- FALSE
```

If we wanted to set a different matching function, one would do something along the lines of the following where `grepl` is used to indicate any match, not just the initial part of the string. We get the string from the entry widget, not the value passed in, as that has been standardized to lower case.

```
f <- 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)
}
QT <- completion$setMatchFunc(func=f)
```

```
## Our basic GUI is basic:
w <- gtkWindow(show=FALSE)
w$setTitle("Test of entry with completion")
w$add(entry)
w$showAll()
```

Tree Views

Both tabular data and tree-like data are displayed through tree views. The visual difference is that a trigger icon appears in rows which represent parents with children. When these are expanded, the children are indicated by indentation. The children are all displayed in a consistent tabular format.

A tree view is constructed by `gtkTreeView`. The `model` can be used to specify the underlying model. If not specified at the time of construction, the `setModel` can be used. The accompanying `getModel` model returns the model from the view.

Tree views have several properties. The `headers-clickable` property, when set to `TRUE`, allows the column headers to receive mouse clicks. This is used for sorting, when the underlying data store allows for that. The tree view widget can popup a search box when the user types control-f if the property `enable-search` is `TRUE` (the default). To turn on searching, a column needs to be specified through the `search-column` property. Rows may be rearranged through drag-and-drop if the `reorderable` property is set to `TRUE`. The `rules-hint`, if `TRUE`, will instruct the theme that the rows hold associated data. Themes will typically use this information to stripe alternating rows.

Tree view columns   For speed purposes, the rendering of a tree view centers around the display of its columns. Each column is displayed through a tree view column, given by the `gtkTreeViewColumn`.

One can set basic properties of the column. Each column has an optional header that can contain a title or even an arbitrary widget. The `setTitle` method is used to set the title. This area can be `"clickable"`, in which case this area receives mouse clicks. This is most commonly used to allow sorting of the column by clicking on the headers, but can also be used to add popup menus (with a bit of wizardry).

58

The property "`resizable`" determines wheter the user can resize the 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.

Tree view columns are added to the tree view with the method `insertColumn`. The `column` argument specifies the tree view column, and the `position` argument the column to insert into (0-based). A column can be moved with the `moveColumnAfter` method, and removed with the `removeColumn` method. The tree view's `getChildren` method returns a list containing all of the tree view columns.

More on cell renderers In addition to the text and pixbuf cell renderers discussed in Section 3.2, there are cell renderers that allow one to display other types of data available for the tree view widget. Some only make sense if the underlying data is to be edited, for example the combobox renderer, for these, the `editable` attribute for the cell renderer should be set to `TRUE`.

As with comboboxes, the mapping of values in the data store to the attributes of a cell render is done by the `addAttribute` method of the tree view column.

Toggle cell renderers Binary data can be represented by a toggle. The `gtkCellRendererToggle` will create a check box in the cell, that will look checked or not depending on the value of its `active` attribute. If this value is found in a boolean column of the model, then changes to the model will be reflected in the state of the GUI. However, the programmer must propogate changes to the GUI (the view) back to the model. The `toggled` signal is emitted when the state is changed. The `activatable` attribute for the cell must be `TRUE` in order for it to receive user input.

```
cr <- gtkCellRendererToggle ()
cr['activatable'] <- TRUE                    # cell can be edited
cr['active'] <- TRUE
QT <- gSignalConnect(cr, "toggled", function(w, path) {
  print(as.numeric(path) + 1) ## the row
})
```

Combobox cell renderers A cell can show a combobox for selection. The `gtkCellRendererCombo` produces the object. Its `model` attribute is set to give the values to choose from. The attribute `has-entry` can be set to `TRUE` to allow a user to enter values, if `FALSE` they can only select from the available ones.

```
cr <- gtkCellRendererCombo ()
store <- rGtkDataFrame (state.name)
```

59

```
cr['model'] <- store
cr['text-column'] <- 0
cr['editable'] <- TRUE                  # needed
```

Progress bar cell renderers   A progress bar can be used to display the percentage of some task. The `gtkCellRendererProgress` function returns the cell renderer. Its `value` attribute takes a value between 0 and 100 indicating the amount finished, with a default value of 0. Values out of this range will be signaled by an error message. The `orientation` property, with values from `GtkProgressBarOrientation`, can adjust the direction that the bar grows. For example,

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

Cell data functions   Formatting numbers is a bit trickier, as the cell renderer properties are geared around text values. For example, to align floating point numbers one can do so in the model and then display as text. However, to do so through the cell rendererer requires one to get the value from the model and modify it before the cell rendererer gets it. For this, a cell data function is used, and not with comboboxes, but only tree views. A cell data function has arguments pointing to a tree view column, the cell renderer, the model, an iterator pointing to the row in the model and a data argument for user data. The function should set the appropriate attributes of the cell renderer. For example, this function could be used to format floating point numbers:

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

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 imlement the change. The callback for the `"editable"` 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 and which column was edited are not passed in by default. These can be passed through the user data arugment, or set as data for the widget if needed within the callback.

60

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

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 set the cursor to the appropriate place and set the state to editing. This is done through the tree view's `setCursor` method. The `path` argument takes a tree path instance, the `column` argument is for a tree view column object, and the flag `start.editing` should be set to `TRUE` to initiate editing. The tree view method `getColumn` can be used to get the tree view column by index (0-based) and the path object can be found from a string through `gtkTreePathNewFromString`.

Example 3.11: Displaying text columns in a tree view
This example shows how to select one or more rows from a data frame that contains some information. We write it so any data frame could be used, although in the specific case we show a list of the installed packages that can be upgraded from CRAN.

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

```
avail <- available.packages()
installed <- installed.packages()
tmp <- merge(avail, installed, by="Package")
need.upgrade <- with(tmp, as.character(Version.x) != as.character(Version.y))
d <- tmp[need.upgrade, c(1, 2, 16, 6)]
names(d) <- c("Package", "Available", "Installed", "Depends")
```

This function will be called on the selected rows. The `print` call would be replaced with something more reasonable, such as a call to `install.packages`.

```
doThis <- function(d) print(d)
```

The rest of this code is independent of the details of `d`. We first append a column to the data frame to store the selection information.

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

Our tree view shows each column using a simple text cell renderer, except for an initial one where the user can select the packages they want to call `doThis` on.

61

```
view <- gtkTreeView()
# add toggle
togglevc <- gtkTreeViewColumn()
view$insertColumn(togglevc, 0)
```

```
[1] 1
```

```
cr <- gtkCellRendererToggle()
togglevc$packStart(cr)
cr['activatable'] <- TRUE
togglevc$addAttribute(cr, "active", n)
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)
```

```
toggled
    726
attr(,"class")
[1] "CallbackID"
```

The text columns are added all in a similar manner.

```
sapply(1:n, function(i) {
  vc <- gtkTreeViewColumn()
  vc$setTitle(nms[i])
  view$insertColumn(vc, i)

  cr <- gtkCellRendererText()
  vc$packStart(cr)
  vc$addAttribute(cr, "text", i-1)
})
```

```
[[1]]
NULL

[[2]]
NULL

[[3]]
NULL

[[4]]
NULL
```

62

Our basic GUI places the view into a box container that also holds a 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")
```

We add the button and its callback. We pass in the view, rather than the model, in case the model would be modified by the `doThis` call. In a real application, once a package is upgraded it would be removed from the display.

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

```
clicked
    768
attr(,"class")
[1] "CallbackID"
```

```
g$packStart(b, expand=FALSE)
```

Finally, we connect the store to the model and show the top-level window

```
view$setModel(store)
w$show()
```

Using filtered models to restrict the displayed rows GTK+ provides a means to show a filtered selection of rows. The basic idea is that an extra column in the store stores logical values to indicate if a row should be visible. To implement this, a filtered store must be made from the original store. The `filterNew` method of a data store returns a filtered data store. The original model is found from the filtered one through its `getModel` method. The method `setVisibleColumn` specifies which column in the model holds the logical values. Finally, to use the filtered store, it is simply set as the model for a tree view.

63

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

[1] 1

Sorting the display One can implement sorting of the display by clicking
on the column headers. This is done by creating a model that can be sorted
from the original store. The function `gtkTreeModelSortNewWithModel` will
produce a new store that is assigned as the model for the tree view. Then to
allow a column to be sorted, one specifies first the `clickable` property of the
view column, and then specifies a column to sort by when the column header
is clicked (it can be different if desired). The following shows the basic steps:

```
store <- rGtkDataFrame(mtcars)
sorted <- gtkTreeModelSortNewWithModel(store)
#
view <- gtkTreeView(sorted)
vc <- gtkTreeViewColumn()
view$insertColumn(vc, 0)                  # first column
```

[1] 1

```
vc$setTitle("Click to sort")
vc$setClickable(TRUE)
vc$setSortColumnId(0)
#
cr <- gtkCellRendererText()
vc$packStart(cr)
vc$addAttribute(cr, "text", 0)
```

The default sorting function can be changed. The sortable stores method
`setSortFunc` is used for this. The following function – which can easily be
modified to taste – shows how the default sorting might be implemented.

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

64

Selection   GTK+ provides a class to handle the selection of rows that the user makes. The selection object is returned from the tree view, through its `getSelection` method. To modify the selection possibilities, the selection's method `setMode` method is used, with values from `GtkSelectionMode`, including `"multiple"` for allowing more than one row to be selected and `"single"` for no more than one row. Additionally, the selection object has various methods to interact with the selection.

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

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

```
[1] 1
```

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

```
selection$selectPath(gtkTreePathNewFromString("3")) # set for example
#
curSel <- selection$getSelected()        # retrieve selection
with(curSel, model$getValue(iter, 0)$value) # both model and iter in curSel
```

```
[1] 21.4
```

When multiple selection is permitted, then the method `getSelectedRows` returns a list with componets `model` pointing to the model, and `retval` a list of tree paths. No column information is passed back by this method.

For example, if we change the selection mode then select a few rows

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

This code will print the selected values in the first column:

```
curSel <- selection$getSelectedRows()
if(length(curSel$retval)) {
  rows <- sapply(curSel$retval, function(path) {
    as.numeric(path$toString()) + 1
  })
  curSel$model[rows, 1]
}
```
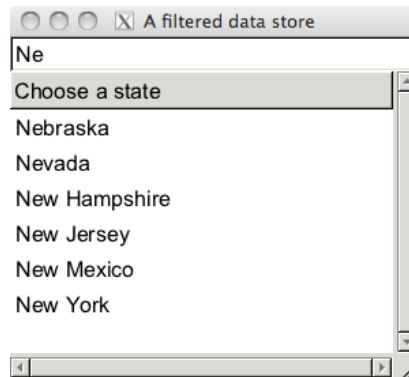
```
[1] 21.4
```

Figure 3.3: Example of a data store filtered by values typed into a text-entry widget.

Signals   Tree views can be used different ways: if the cells are not editable, then they are basically list boxes which allow the user to select one of several rows. A single click selects the value, and a double click is often used to initiate an action. If the cells are editable, then this action is to edit the content.

When a selection is made or changed, the signal `changed` is emitted by the selection.

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(tree.view$getColumns(), function(i) i == column)
```

For tree stores, the user can click to expand or collapse a part of the tree. The signals `row-expanded` and `row-collapsed` are emitted respectively by the tree view. The signature of the callback is similar to above with the view, a tree path and a view column.

Example 3.12: Using filtering
This example shows how to use GTK+'s filtering feature to restrict the rows of the model shown by matching against the values entered into a text entry box. The end result is similar to an entry widget with completion.

We use a convenient set of names and 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$filterNew()
filteredStore$setVisibleColumn(ncol(df)-1)        # offset
view <- gtkTreeView(filteredStore)
```

This example uses just one column, we create a basic view of it below.

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

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 `grep` is just `TRUE`, so all rows will be shown. The `getModel` method of the filtered store is used, although we could have passed in that store itself.

```
e <- gtkEntry()
ID <- gSignalConnect(e, "changed", function(w, data) {
  val <- w$getText()
  df <- data$getModel()
  values <- df[,1]
  df[, dim(df)[2]] <- sapply(values, function(i)
                             as.logical(length(grep(val,i))))
},
                     data=filteredStore)
```

Figure 3.3 shows the two widgets placed within a simple GUI.

Example 3.13: A widget for variable selection
This example shows a combination widget that is familiar from other statistics GUIs. It provides two tree views listing variable names and has arrows to move variable names from one side to the other. Often such widgets are used for specifying statistical models.

We will use Example 2.8, in particular its function `addToStockIcons`, to add some custom stock icons to identify the variable type.

```
nms <- c("factor","numeric")
fileNms <- c(system.file("images","factor.gif", package="gWidgets"),
             system.file("images","numeric.gif", package="gWidgets"))
QT <- addToStockIcons(nms, fileNms)
```
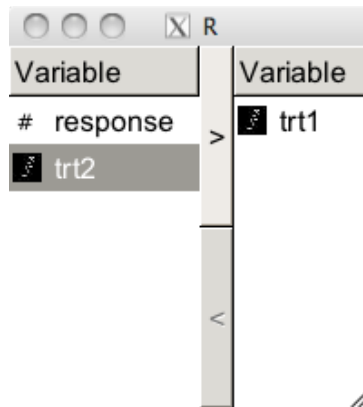
67

Figure 3.4: An example showing to tree views with buttons to move entries from one to the other. This is a common method for variable selection.

To keep track of the variables in the two tree views we use a single model. It has a column for all the variable names, a column for the icon, and two columns to keep track of which variable names are to be displayed in the respective tree views.

```
d <- data.frame(varNames=c("response", "trt1", "trt2"),
                stock.id=c("new-numeric", "new-factor", "new-factor"),
                leftView  = rep(TRUE, 3),
                rightView = rep(FALSE, 3),
                stringsAsFactors=FALSE)
model <- rGtkDataFrame(d)
```

We will use a filtered data store to show each tree view. As the two tree views are identical, except for the rows that are displayed, we use a function to generate them. The `vis.col` indicates which column in the `rGtkDataFrame` object contains the visibility information. Our tree view packs in both a pixbuf cell rendererer and a text one.

```
## make a view
makeView <- function(model, vis.col) {
  filteredModel <- model$filterNew()
  filteredModel$setVisibleColumn(vis.col - 1)
  tv <- gtkTreeView(filteredModel)
  tv$getSelection()$setMode("multiple")
  ##
  vc <- gtkTreeViewColumn()
  vc$setTitle("Variable")
  tv$insertColumn(vc, 0)
  ##
  cr <- gtkCellRendererPixbuf()
```

```
  vc$PackStart(cr, expand=FALSE)
  cr['xalign'] <- 1
  vc$addAttribute(cr, "stock-id", 1)
  ##
  cr <- gtkCellRendererText()
  vc$PackStart(cr, expand=TRUE)
  cr['xalign'] <- 0
  cr['xpad'] <- 5
  vc$addAttribute(cr, "text", 0)

  return(tv)
}
```

We know create the tree views and store the selections associated to each.

```
views <- list()
views[["left"]] <- makeView(model,3)
views[["right"]] <- makeView(model,4)
selections <- lapply(views, gtkTreeViewGetSelection)
```

We need buttons to move the values left and right, these are stored in a list for convenience later on.

```
buttons <- list()
buttons[["fromLeft"]] <- gtkButton(">")
buttons[["fromRight"]] <- gtkButton("<")
```

Our basic GUI is shown in Figure 3.4 where the two tree views are placed side-by-side.

The key handler moves the selected value from one side to the other. The issue here is that when the view is using filtering the selection returns values relative to the child model (the filtered one). In general the methods of the filtered model `convertChildPathToPath` and `convertChildIterToIter` will translate between the two models, but in this case we pass in the `rGtkDataFrame` instance, not the filtered model. So we use the columns indicating visibility to identify which index is being referred to. This handler assumes the model and a value indicating the view (`from`) is passed in through the user data.

```
moveSelected <- function(b, data) {
  model <- data$model

  selection <- selections[[data$from]]
  selected <- selection$getSelectedRows()
  if(length(selected$retval)) {
    childRows <- sapply(selected$retval, function(childPath) {
      childRow <- as.numeric(childPath$toString()) + 1
    })
    shownIndices <- which(model[, 2 + data$from])
    rows <- shownIndices[childRows]
```

Draft version, do not circulate – April 17, 2010

```
    model[rows, 2 + data$from] <- FALSE
    model[rows, 2 + (3-data$from)] <- !model[rows, 2 + data$from]
  }
}
```

We connect the handler to the `"clicked"` signal for the buttons.

```
IDs <- sapply(1:2, function(i)
              gSignalConnect(buttons[[i]], signal="clicked",
                             f=moveSelected,
                             data=list(from=i, model=model)))
```

We add one flourish, namely ensuring that the arrows are not sensitive when the corresponding selection is not set. This handler for the selections is used.

```
disableButton <- function(sel, data) {
  selected <- sel$getSelectedRows()
  buttons[[data]]$setSensitive(length(selected$retval) != 0)
}
IDs <- sapply(1:2, function(i)
              gSignalConnect(selections[[i]], signal="changed",
                             f=disableButton,
                             data=i))
```

As the initial state has no selection, we set the buttons sensitivity accordingly.

```
QT <- sapply(buttons, function(i) i$setSensitive(FALSE))
```

The `gtkTreeView` widget displays either list stores or tree stores. The difference for the programmer is in the creation of the data store, not the tree view.

Example 3.14: A simple tree display
This example illustrates that the same tree view can display both rectangular and heirarchical data. The data we use will come from the `Cars93` data set used in Example 3.7. In that example we defined a simple tree store from a data frame, with a level for manufacturer and make for different cars. We refer to that model by `tstore` below.

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

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

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

```
view <- gtkTreeView()
vc <- gtkTreeViewColumn()
vc$setTitle("Make")
```

70

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

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

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

Example 3.15: Dynamically growing a tree
This example uses a tree to explore an R list object, such as what is returned
by one of R's modelling functions. As the depth of these lists is not specified
in advance, we use a dynamical approach to creating the tree store, modify-
ing the tree store when the tree view is expanded or collapsed.

We begin by defining a function that gets the "children" of a list object.
For a level of the list, this function returns the named components, their class
and a logical indicating if the component is recursive.

```
getChildren <- function(path=character(0)) {

  pathToObject <- function(path) {
    x <- try(eval(parse(text=paste(path,collapse="$")),
                  envir=.GlobalEnv),silent=TRUE)
    if(inherits(x,"try-error")) {
      cat(sprintf("Error with %s",path))
      return(NA)
    }
    return(x)
  }

  theChildren <- function(path) {
    if(length(path) == 0)
      ls(envir=.GlobalEnv)
    else
      names(pathToObject(path))
  }
  hasChildren <- function(obj) is.recursive(obj) && !is.null(names(obj))

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

  children <- theChildren(path)
  objs <- sapply(children,function(i) pathToObject(c(path,i)))
  d <- data.frame(children=children,
                  class=sapply(objs, getType),
                  offspring=sapply(objs, hasChildren))
  ## filter out Gtk ones
  ind = grep("^Gtk", d$class)
```

71

```
    if(length(ind) == 0) return(d) else return(d[-ind ,])
}
```

This function is used to add the children to a tree store.

```
addChildren <- function(store , children , parentIter=NULL) {
  if(nrow(children) == 0)
    return(NULL)
  for(i in 1:nrow(children )) {
    iter <- store$append(parent=parentIter)$iter
    ## use last column to indicate logical
    sapply (1:(ncol(children) - 1), function(j)
            store$setValue(iter , column=j-1, children[i,j]))
    ## Add a branch if there are children
    ## no better way, as this adds an extra blank line
    ## we remove ir later.
    if(children[i, "offspring"])
      store$append(parent=iter)
  }
}
```

The various callbacks for the tree view pass back the view and a tree path. We define some functions to relate these values with iterators.

```
tpathToPIter <- function(view , tpath) {
  ## view$getModel --- sstore , again store
  sstore <- view$getModel ()
  store <- sstore$getModel ()
  uspath <- sstore$convertPathToChildPath(tpath)
  p.iter <- store$getIter(uspath)$iter
  return(p.iter )
}
```

A "path" is made up of the names of each component that makes up an element in the list. This function returns the path for a component specified by its iterator.

```
iterToPath <- function(view , iter) {
  sstore <- view$getModel ()
  store <- sstore$getModel ()
  string <- store$getPath(iter)$toString ()
  indices <- unlist(strsplit(string ,":"))
  thePath <- c()
  for(i in seq_along(indices )) {
    path <- paste(indices [1:i],collapse=":")
    iter <- store$getIterFromString(path)$iter
    thePath[i] <- store$getValue(iter ,0)$value
  }
  return(thePath[-1])
}
```

72

Now we can begin defining our tree store. This example allows sorting, so calls the `gtkTreeModelSortNewWithModel` function.

```
store = gtkTreeStore(rep("gchararray",2))
sstore = gtkTreeModelSortNewWithModel(store)
```

We set an initial root.

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

The call of append is used to allow the object to have an expandable icon.

Now to define the tree view. We allow multiple selection, as an illustration.

```
view = gtkTreeViewNewWithModel(sstore)
sel = view$getSelection()
sel$setMode(GtkSelectionMode["multiple"])
```

The view will have two similar columns.

```
## add two cell renderers — 1 for name, 1 for type
nms <- c("Variable name","type")
for(i in 1:2) {
  cr <- gtkCellRendererText()
  vc <- gtkTreeViewColumn()
  vc$setSortColumnId(i-1) # allow sorting
  vc$setResizable(TRUE)
  vc$setTitle(nms[i])
  vc$packStart(cr,TRUE)
  vc$addAttribute(cr,"text",i-1)
  view$insertColumn(vc, i-1)
}
```

We put the tree view widget into a basic GUI.

```
sw <- gtkScrolledWindow()
sw$setPolicy("automatic","automatic")
sw$add(view)
w <- gtkWindow()
w$setTitle("Tree view")
w$add(sw)
```

At this point, we can see the first level, but nothing will happen if we click on the trigger icon. We need to specify a handler for the `row-expanded` signal. The odd thing here, is that we appended an fake child, so that an expand icon would appear. In this, we remove it.

```
ID <- gSignalConnect(view,signal="row-expanded",
                     f = function(view, iter, tpath, user.data) {
```

73

```
                           store <- user.data
                           p.iter <- tpathToPIter(view, tpath)
                           path <- iterToPath(view, p.iter)
                           children = getChildren(path)
                           addChildren(store, children, parentIter=p.iter)
                           ## remove errant 1st offspring. See addChildren
                           ci <- store$iterChildren(p.iter)
                           if(ci$retval) store$remove(ci$iter)
                         },
                        data=store)
```

Since the new data is generated when the row is expanded, we need to remove the old data when the row is closed.

```
ID <- gSignalConnect(view,signal="row-collapsed",
                     f = function(view, iter, tpath, user.data) {
                         store <- user.data
                         p.iter <- tpathToPIter(view,tpath)

                         n = store$iterNChildren(p.iter)
                         if(n > 1) { ## n=1 gets removed when expanded
                           for(i in 1:(n-1)) {
                             child.iter = store$iterChildren(p.iter)
                             if(child.iter$retval)
                               store$remove(child.iter$iter)
                           }
                         }
                     }, data=store)
```

Finally, this handler simply shows how to get the value (or values if multiple selection is okay).

```
ID <- gSignalConnect(view,signal="row-activated",
                       f = function(view, tpath, tcol) {
                           p.iter <- tpathToPIter(view, tpath)
                           path <- iterToPath(view, p.iter)
                           sel <- view$getSelection()
                           out <- sel$getSelectedRows()
                           if(length(out) == 0) return(c()) # nothing
                           vals <- c()
                           for(i in out$retval) {  # multiple selections
                             iter <- out$model$getIter(i)$iter
                             vals <- c(vals, out$model$getValue(iter,0)$value)
                           }
                           print(vals)       # Insert Real Function Here
                       })
```

4

RGtk2: Menus and Dialogs

4.1 Actions

Actions are a means to create reusable representations for some action to be initiated. The `gtkAction` constructor creates actions, taking arguments `name`, `label` (what gets shown), `tooltip`, and `stock.id`. The act associated with an action is specified by adding a callback to the `activate` signal.

Actions are connected to widgets, through the method `connectProxy`. For buttons, the stock id information must be added to the button through the button's `setImage` method. The action's `createIcon` method, with argument coming from a value of `GtkIconSize`, will return the needed image.

Actions can have their `sensitivity` property adjusted through their `SetSensitive` method. This will propogate to all the widgets the action has a proxy connection with.

Example 4.1: An action object
A basic action can be defined as follows:

```
a <- gtkAction(name="ok", label="_Ok", tooltip="An OK button", stock.id="gtk-ok")
ID <- gSignalConnect(a, "activate", f = function(w, data) {
  print(a$GetName())                        # of some useful thing
})
```

To connect the action to a button, is straightforward.

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

The image must be manually placed, which is facilitated by methods for the button and the action object.

```
b$setImage(a$createIcon('button')) # GtkIconSize value
```

75

4.2 Menus

A menu allows access to the GUI's actions in an organized way. This organization relies on a choice of top-level menu items, their possible submenus, and grouping within the same level of a menu. Menubars are typically nested. Toolbars allow access more quickly to common actions, but do not allow for nesting.

Menubars and popup menus may be constructed by appending each menuitem and submenu separately, as illustrated below. An alternative, using a UI manager, is described in a subsequent section,

To specify a menubar step-by-step consists of defining a top-level menu bar (`gtkMenuBar`). To a menu bar we append menu items. Menu items may have sub menus (`gtkMenu`) appended, which gives the heirarchical nature of a menu. Popup menus are similar, although begin with a `gtkMenu` instance.

Building the the menu   Submenus and added to a menu item through the `SetSubMenu` method. Menu items are added to a menu through the methods `Append`; `Prepend`; and `Insert`, the latter requiring an index where the insertion is to take place, with 0 being the same as `Prepend`, in addition to the child. After a child is added, the method `ReorderChild` can be used to move it to a new position (0-based). Menuitems are not typically removed, rather they are disabled through their `SetSensitive` method, but if desired their `Show` and `Hide` methods can be used to stop them from being drawn.

Menu items   Menu items represent actions to be taken and are created by several different constructors. A basic menu item is created by `gtkMenuItem`. The argument `label` allows one to specify the label at construction time. The related constructor, `gtkMenuItemNewWithMnemonic` also allows the specification of a label, only underscores within the string specify the mnemonic for the menu item. To group menu items, one use separators (`gtkSeparatorMenuItem`).

A menu item for a `gtkAction` object can be created by its `createMenuItem` method. For window managers that display them, any icon specified the action's `stock.id` argument will be displayed.

To add a different image in the menu bar, the `gtkImageMenuItem` can be used. Although, the `stock.id` argument can be used to specify the icon, we don't use this, as then the argument `accel.group` must be specified. An accelerator group defines a set of keyboard shortcuts to initiate actions, such as a Ctrl+Q to quit. (A mnemonic is a keyboard shortcut to indicate a GUI element). We don't discuss creating those here (they are given in the UI manager example). If instead of the `stock.id` argument, just the `label` is specified for the image menu item, the image can be added later through the `SetImage` method. This takes an image object for an argument.

A check button menu item can be created by `gtkCheckMenuItem`. This

76

menu item shows a check box when the `active` state for the menu is set. The default is not active. Use the `GetActive` to test if the state is active or not. It may be best to set this state to active, so the user can identify that the item is a toggle (use the method `SetActive(TRUE)`. When the user clicks on the menu item, the `toggled` signal is emitted.

Example 4.2: A basic menu bar

We illustrate how to make a basic menu bar with a plain item, an item with an icon, and check item. Our GUI, just adds a menubar to a top-level window.

We create top-level menubar and a menu item for our top level File entry with a mnemonic.

```
mb <- gtkMenuBar()
fileMi <- gtkMenuItemNewWithMnemonic(label="_File")
mb$append(fileMi)
```

For the menu item we attach a submenu.

```
fileMb <- gtkMenu()
fileMi$setSubmenu(fileMb)
```

We now define some menu items. First a basic one:

```
open <- gtkMenuItem(label="open")
```

Next we show how an `gtkAction` item can define a menuitem.

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

This illustrates how to add an image to the menu bar using a stock icon. The size specification is important to get the correct look.

```
quit <- gtkImageMenuItem(label="quit")
quit$setImage(gtkImageNewFromStock("gtk-quit", size=GtkIconSize["menu"]))
```

A simple check menu item can be created, as follows:

```
happy <- gtkCheckMenuItem(label="happy")
happy$setActive(TRUE)
```

These items are appended in the desired order, by

```
Qt <- sapply(list(open, save, happy, gtkSeparatorMenuItem(), quit), function(i) {
      fileMb$append(i)
    })
```

We specify a handler for the toggle button

```
ID <- gSignalConnect(happy, "toggled", function(b,data) {
  if(b$getActive())
    print("User is now happy")
})
```

77

For the other items, we specify a generic action for the `activate` signal.

```
QT <- sapply(list(open, quit, saveAction), function(i)
        gSignalConnect(i, "activate", f=function(mi, data) {
          cat("item selected\n")
        })
        )
```

We make as simple GUI for the menubar.

```
w <- gtkWindow(show=FALSE)
w['title'] <- "Menubar example"
w$add(mb)
w$ShowAll()
```

Example 4.3: Popup menus
To illustrate popup menus, we show how define a one and connect it to a third-mouse click. We start with a `gtkMenu` instance, to which we add some items.

```
popup <- gtkMenu()                          # top level
for(i in c("cut","copy","----","paste")) {
  if(i == "----")
    popup$append(gtkSeparatorMenuItem())
  else
    popup$append(gtkMenuItem(i))
}
```

This menu will be shown by `gtkMenuPopup`. This function is called with the menu, some optional arguments for placement, and values for the button that was clicked and the time of activation. These values can be retrieved from the second argument of the callback (a `GdkEvent`), as shown.

```
b <- gtkButton("Click me with right mouse button")
w <- gtkWindow(); w$setTitle("Popup menu example")
w$add(b)
ID <- gSignalConnect(b,"button-press-event",
                     f = function(w, e, userData) {
                         if(e$getButton() == 3 ||
                            (e$getButton() == 1 && # a mac
                             e$getState() == GdkModifierType['control-mask'])
                            ) {
                           gtkMenuPopup(userData$mb,
                                        button = e$getButton(),
                                        activate.time = e$getTime())
                         }
                         return(FALSE)
                     },
                     data=list(mb=popup)
                     )
```

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

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

The above can easily be automated. The menubar widget in `gWidgetsRGtk2` simply maps a list with named components to the above, by setting menu items for each top-level component, submenus for each component that contains children, and menu items for components that do not have children. However, the next approach is preferred for larger menubars, as it separates out the presentation from the actions.

4.3   Toolbars

Toolbars are like menubars only they only contain actions, there are no submenus. Toolbar objects are constructed by `gtkToolbar`. The placement of the widget at the top of a top-level window is done by the programmer. Toolbar items are added to the toolbar using the `Add` method. Once added, items can be referred to by index using the `[[` method.

Toolbar items have some common properties. The buttons are comprised of an icon and text, and the style of their layout is specified by the toolbar method `SetStyle`, with values coming from the `GtkToolbarStyle` enumeration. Toolbar items can have a tooltip set for them through the methods `SetTooltipText` or `SetTooltipMarkup`, the latter if PANGO markup is desired. Toolbar items can be disabled, through the method `SetSensitive`.

The items can be one of a few different types. A stock toolbar item is constructed by `gtkToolbarButtonNewFromStock`, with the stock id as the argument. The constructor `gtkToolbarButton` creates a button that can have its label and icon value set through methods `SetLabel` and `SetIconWidget`. Additionally, there are methods for setting a tooltip or specifying a stock id after construction. A toggle button, which toggles between looking depressed or not when clicked is created by `gtkToggleToolButton` or `gtkToggleToolButtonNewFromStock`. Additionally there are constructors to place menus (`gtkMenuToolButton`) and radio groups (`gtkRadioToolButton`).

The `clicked` signal is emitted when a toolbar button is pressed. For the toggle button, the `toggle` signal is emitteed. Other

Example 4.4: Basic toolbar usage
We illustrate with a toolbar, whose buttons are produced in various ways.

79

```
tb <- gtkToolbar ()
```

A button with a stock icon is produced by a call to the appropriate constructor.

```
b1 <-  gtkToolButtonNewFromStock ("gtk-open")
tb$add(b1)
```

To use a custom icons, requires a few steps.

```
f <- system.file("images/dataframe.gif", package="gWidgets")
image <- gtkImageNewFromFile(f)
b2 <- gtkToolButton ()
b2$setIconWidget(image)
b2$setLabel("Edit")
tb$add(b2)
```

Adding a toggle button also is just a matter of calling the appropriate constructor. In this, example we illustrate how to initiate the callback only when the button is depressed.

```
b3 <- gtkToggleToolButtonNewFromStock ("gtk-fullscreen")
tb$add(b3)
QT <- gSignalConnect(b3, "toggled", f=function(button, data) {
  if(button$getActive ())
    cat("toggle button is depressed\n")
  })
```

We give the other buttons a simple callback when clicked:

```
QT <- sapply(1:2, function(i) gSignalConnect(tb[[i]], "clicked", function(button, 
  cat("You clicked", button$getLabel(), "\n")
}))
```

```
w <- gtkWindow (show=FALSE)
w['title'] <- "Toolbar example"
g <- gtkVBox ()
w$add(g)
g$packStart(tb, expand=FALSE)
g$packStart(gtkLabel("filler"), expand=TRUE, fill=TRUE)
w$showAll ()
```

4.4   Statusbars

In GTK+, a statusbar is constructed through by the gtkStatusbar function. Statusbars must be placed at the bottom of a top-level window by the programmer. In GTK+, a statusbar keeps various stacks of messages for display. One adds a message to display for given stack through the Push method by specifying first an integer value for context.id and a message. To pop the top message on a stack and display the next, the method Pop method is available.
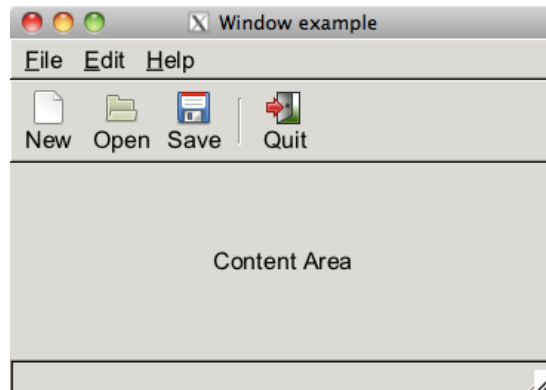
80

Figure 4.1: A GUI made using a UI manager to layout the menubar and toolbar.

4.5 UI Managers

A GUI is designed around actions that are accessible through the menubar and the toolbar. The notion of a *user interface manager* (UI manager) separates out the definitions of the actions from the user interface. The steps required to use GTK+'s UI manager are

1. define a UI manager,
2. set up an accelarator group for keyboard shortcuts,
3. define our actions,
4. create action groups to specify the name, label (with possible mnue-monic), keyboard accelerator, tooltip, icon and callback for the graphi-cal elements that call the action,
5. specify where the menu items and toolbar items will be placed,
6. connect the action group to the UI manager, and finally
7. display the widgets.

We show by an example how this is done.

Example 4.5: UI Manager example
We define the UI manager as follows

```
uimanager = gtkUIManager()
```

Our actions either open a dialog to gather more information or issue a command. A GtkAction element is passed to the action. We define a stub here, that simply updates a gtkStatusbar instance, defined below.

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

81

To show how we can sequentially add interfaces, we break up our action group definitions into one for "File" and "Edit" and another one for "Help." The key is the list defining the entries. Each component specifies (in this order) the name; the icon; the label, with _ specifying the mnemonic; the keyboard accelerator, with <control>, <alt>, <shift> as possible prefixes, a tooltip, which may not work with the R event loop, and finally the callback. Empty values can be defined as NULL or, except for the callback, an empty string.

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

We now add the actions to the action group, then add this action group to the first spot in the UI manager.

```
QT <- firstActionGroup$addActions(firstActionEntries)
uimanager$insertActionGroup(firstActionGroup,0) # 0 — first spot
```

The "Help" actions we do a bit differently. We define a "Use tooltips" mode to be a toggle, as an illustration of that feature. One can also incorporate radio groups, although this is not shown.

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

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

```
toggleAction <- gtkToggleAction("UseTooltips",label="_Use tooltips",
                                tooltip="Use tooltips ")
toggleAction$setActive(TRUE)            # initially set
ID <- gSignalConnect(toggleAction,signal = "toggled",
                     f=function(ta,userData) cat(userData,ta$getName(),"\n"),
```

82

```
                      data="toggled")
helpActionGroup$addAction(toggleAction)
```

We insert the help action group in position 2.

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

The `SetActive` method can set the state, use `GetActive` to retrieve the state.

Our UI Manager's layout is specified in a file. The file uses XML to specify where objects go. The structure of the file can be grasped quickly from the example. Each entry is wrapped in `ui` tags. The type of UI is either a `menubar`, `toolbar`, or `popup`. The `name` properties are used to reference the widgets later on. Menuitems are added with a `menuitem` entry and toolbar items the `toolitem` entry. These have an `action` value and an optional name (defaulting to the `action` value). The `separator` tags allow for some formatting. The nesting of the menuitems is achieved using the `menu` tags. A `placeholder` tag can be used to add entries at a later time.

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

This file is loaded into the UI manager as follows

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

The `id` value can be used to merge and delete UI components, but this is not illustrated here. The menus can also be loaded from strings.

Now we can setup a basic window template with menubar, toolbar, and status bar. We first get the three main widgets. We use the names from the UI layout to get the widgets through the `GetWidget` method of the UI manager. The menubar and toolbar are returned as follows, for our choice of names in the XML file.

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

The statusbar is constructed with

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

Statusbars have a simple API. The push ethod, as used in the definition of the callback `f`, is used to add new text to the statusbar. The pop method reverts to the previous message.

Now we define a top-level window and attach a keyboard accelerator group to the window so that when the window has the focus, the specified keyboard shortcuts can be used.

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

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

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

The redo feature should only be sensitive to mouse events after a user has undone an action and has not done another. To set the sensitivity of a menu item is done through the `SetSensitive` ethod called on the widget. We again retrieve the menuitem or toolbar item widgets through their names.

```
uimanager$getWidget("/menubar/Edit/EditRedo")$setSensitive(FALSE)
```

To reenable, use `TRUE` for the argument to `SetSensitive`

We can also use the `SetText` ethod on the menuitems. For instance, instead of a generic "Undo" label, one might want to change the text to list

84

the most previous action. The method is not for the menu item though, but rather a `gtkLabel` which is the first child. We use the list notation to access that.

```
a <- uimanager$getWidget("/menubar/Edit/EditUndo")
a[[1]]$setText("Undo add text")
```

4.6   Dialogs

GTK+ comes with a variety of dialogs to create simple, usually single purpose, popup windows for the user to interact with.

The `gtkDialog` constructor

The constructor `gtkDialog` creates a basic dialog box, which is a display containing a top section with optionally an icon, a message, and a secondary message. The bottom section, the action area, shows buttons, such as yes, no and/or cancel. The convenience functions `gtkDialogNewWithButtons` and `gtkMessageDialog` simplify the construction.

In GTK+ dialogs can be modal or not. Thre are a few ways to make a dialog modal. The method window `setModal` will do so, as will passing in a `modal` flag to some of the constructors. These make other GUI elements inactive, but not the R session. Whereas, calling the `run` method, will stop the flow until the dialog is dismissed, The return value can then be inspected for the action, such as what button was pressed. These values are from `GtkResponseType`, which lists what can happen.

Basic message dialogs   The `gtkMessageDialog` has an argument `parent`, to specify a parent window the dialog should appear relative to. The `flags` argument allows one to specify values (from `GtkDialogFlags`) of `destroy-with-parent` or `modal`. The `type` is used to specify the message type, using a value in `GtkMessageType`. The `buttons` is used to specify which buttons will be drawn. The message is the following argument. The dialog has a `secondary-text` property that can be set to give a secondary message.

```
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"
response <- dlg$run()
if(response == GtkResponseType["cancel"] || # for other buttons
   response == GtkResponseType["close"] ||
   response == GtkResponseType["delete-event"]) {
```

85

```
   ## pass
} else if(response == GtkResponseType["ok"]) {
  print("Ok")
}
dlg$Destroy()
```

Making your own dialogs The `gtkDialog` constructor returns a dialog object which can be customized for more involved dialogs. In the example below, we illustrate how to make a dialog to accept user input. We use the `gtkDialogNewWithButtons`, which allows us to specify a stock buttosn and a response value. We use standard responses, but could have used custom ones by specifying a positive integer. The dialog is a window object containing a box container, which is returned by the `getVbox` method. This box has a separator and button box packed in at the end, we pack in another box at the beginning below to hold a label and our entry widget.

When one of the buttons is clicked, the `response` signal is emitted by the dialog. We connect to this close the dialog.

```
dlg <- gtkDialogNewWithButtons(title="Enter a value",
                               parent=NULL, flags=0,
                               "gtk-ok", GtkResponseType["ok"],
                               "gtk-cancel", GtkResponseType["cancel"],
                               show=FALSE)
g <- dlg$getVbox()                            # content area
vg <- gtkVBox()
vg['spacing'] <- 10
g$packStart(vg)
vg$packStart(gtkLabel("Enter a value"))
entry <- gtkEntry()
vg$packStart(entry)
ID <- gSignalConnect(dlg, "response", f=function(dlg, resp, user.data) {
  if(resp == GtkResponseType["ok"])
    print(entry$getText())
  dlg$Destroy()
})
dlg$showAll()
dlg$setModal(TRUE)
```

File chooser

GTK+ has a `GtkFileChooser` backend to implement selecting a file from the file system. The same widget allows one to open or save a file and select or create a folder (directory). The action is specified through one of the `GtkFileChooserAction` flags. This backend presented in various ways through `gtkFileChooserDialog`, which pops up a modal dialog; `gtkFileChooserButton`,

which pops up the dialog when the button is clicked; and `gtkFileChooserWidget`, which creates a widget that can be placed in a GUI to select a file.

The dialog constructor allows one to spcify a title, a parent and an action. In addition, the dialog buttons must be specified, as with the last example using `gtkDialogNewWithButtons`.

Example 4.6: An open file dialog
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"])
```

One can use the `run` method to make this modal or connect to the `response` signal. The file selected is found from the file chooser method `getFilename`. One can enable multiple selections, by passing `setSelectMultiple` a `TRUE` value. In this case, the `getFilenames` returns a list of filenames,

```
ID <- gSignalConnect(dlg, "response", f=function(dlg, resp, data) {
  if(resp == GtkResponseType["ok"]) {
    filename <- dlg$getFilename()
    print(filename)
  }
  dlg$destroy()
})
```

For the open dialog, one may wish to specify one or more filters, to narrow the available files for selection. A filter object is returned by the `gtkFileFilter` function. This object is added to the file chooser, through its `addFilter` method. The filter has a name property set through the `setName` method. The user can select a filter through a combobox, and this provides the label. To use the filter, one can add a pattern (`addPattern`), a MIME type (`addMimeType`), or a custom filter.

```
fileFilter <- gtkFileFilter()
fileFilter$setName("R files")
dlg$addFilter(fileFilter)
QT <- sapply(c("*.R", "*.Rdata"),
             function(i) fileFilter$addPattern(i))
QT <- sapply(c("text/plain"),
             function(i) fileFilter$addMimeType(i))
```

The save file dialog is similar. The `setFilename` can be used to specify a default file and `setFolder` can specify an initial directory. To be careful as to not overwrite an existing file, the method `setDoOverwriteConfirmation` can be passed a `TRUE` value.

Date picker

A calendar widget is produced by `gtkCalendar`. This widget allows selection of a day, month or year. To specify these values, the properties `day`, `month` (0-11), and `year` store these values as integers. One can assign to these directly, or use the methods `selectDay` and `selectMonth` (no select year method). The method `getData` returns a list with components for the year, month and day. If there is no selection, the `day` component is 0.

The widget emits various signals when a selection is changed. The `day-selected` and `day-selected-double-click` ones are likely the most useful of these.

Draft version, do not circulate – April 17, 2010