

---

## Preface

### About this book

R has a number of packages that provide a link between the R user and a graphical toolkit, such as `tcltk`, `RGtk2` and `rJava` (for Swing). In addition, an R user can interface with python or other external languages to provide access to graphical toolkits within those languages. This book is about writing graphical user interfaces (GUI) within R that do not rely on knowing an external language.

Currently, there is a range of graphical interfaces for R that are programmed within R. For example, several package authors have provided GUIs for their functions. Examples include `limmaGUI`, `caGUI`, `clustTool`, `Metabonomic`, . . . There are a few tools to automatically generate such GUIs, such as the `fgui` package and the `guiDlgFunction` function from the `svDialogs` package. Other authors have provided graphical interfaces to explore data sets, such as `playwith`, `lattice` and `aplpack`. While others have provided packages with GUIs aimed at allowing students to perform some simulation, e.g., `teachingDemos`. The `rattle` package provides an interface for several data mining operations. The `Rcmdr` package provides a menu- and dialog-driven interface to a wide range of R's functionality. There are several user-contributed plugins that extend the `Rcmdr`. Additionally, as R finds wider usage outside of academia, it is not uncommon for people who work in a team setting to desire an interface to their R code that allows non-R users access.

Such examples are all within the scope of this book. We set out to show that for many purposes adding a graphical interface to one's work is not terribly sophisticated nor time-consuming (atleast each piece isn't). This book does not attempt to cover GUIs for R that require knowledge of another programming language, although several such projects provide some of the most sophisticated interfaces. Examples are the JGR GUI or `iPlots` GUI written in Java through `rJava`, the `rkward` GUI written within KDE, the `biocep` GUI written using Java and the `Rserve` package, or even the Windows GUI that

comes with R's Windows package.

The bulk of this text covers three different approaches to writing GUIs. The `gWidgets` package is covered first. This provides a common programming interface to some of the different graphical toolkits provided with R. This interface is much simpler (and less powerful) than the native toolkits, so is useful for a programmer who does not wish to invest too much time into perfecting their GUI. There are a few other packages that provide an R interface to a toolkit such as `rpanel` or `svDialogs`, but we focus on this one.

Next, we discuss the `RGtk2` package which provides a link between R and the cross-platform GTK+ libraries. These libraries are feature rich and used by several widely used projects. For the R user, they provide the most complete and modern-looking interfaces.

Finally, we discuss the `tcltk` package. This package provides the R user access to the Tk libraries. Although not as modern as GTK+, these libraries come pre-installed with the Windows binary and so there are no installation issues for the average end-user. The bindings to Tk were the first ones to appear for R and the several of the GUI projects above, notably `Rcmdr`, use this toolkit.

In addition to chapters on each of these three packages, the book has an introductory chapter on GUIs, , and a chapter on web GUIs.

The text is written with the belief that much can be learned by studying examples, and so several examples are given. Some of these are meant as sketches of what can be done, while a few illustrate how to code actual useful interfaces. This text can't expect to cover all of the features of a graphical toolkit. For the `tcltk` and `RGtk2` packages, both underlying toolkits have well documented APIs.

This text comes with an accompanying package `ProgGUIinR`. This package includes the complete code for all the examples. In order to save space, some examples in the text have code that is not shown. The package provides the functions `browseGWidgetsFiles`, `browseTclTkFiles` and `browseRGtk2Files` for browsing the examples from the respective chapters. Additionally, this package will contain vignettes describing aspects that did not make it into the text.

This text was written with the Sweave package. To suppress superfluous output an assignment to a variable named `QT` is made at times.

---

## Contents



# The basic ideas of Graphical User Interfaces

## 1.1 Introduction

There are many reasons why one would want to create a graphical user interface for a package or piece of R functionality. For example,

- GUIs allow others easier access to your functions,
- GUIs are useful if a function has many arguments,
- GUIs can be dynamic, they can direct the user how to fill in arguments, can give feedback on the choice of an argument, they can prevent or allow user input as appropriate,
- GUIs make dealing with large data sets easier both visually and as an alternative to sometimes difficult command line usage.
- GUIs can tightly integrate with graphics, for example the rggobi interface among others.

Even though R has so much new interest, especially commercially supported interest it does not have a common graphical user interface. Many reasons exist. Historically those most capable of doing so in the R community had the least desire for one, the cross-platform nature of R makes it difficult to provide, the wide variety of user types lends a single interface unlikely to satisfy all. Even if it did have a common interface, as much of R's functionality depends on user contributions there will always be a demand for package programmers to provide convenient interfaces to their work.

## 1.2 A simple GUI in R

We begin with an example showing how one can use R's standard graphics device, as a canvas for drawing a GUI, for playing a game of tic-tac-toe against the computer (Figure ??). Although this example has nothing to do with statistics, it illustrates, in a familiar way, some of the issues that this text will address with using GUIs.

Many GUIs can be thought of as different views of some data model. This thought leads itself to the prevalent model-view-controller architecture

## 1. THE BASIC IDEAS OF GRAPHICAL USER INTERFACES

---

of many large GUI applications. For many our the examples in the text, we avoid that level of design, but do try to keep separate the different aspects of GUI design. In this example, the data simply consists of information holding the state of the game. Here we define a global variable, `board`, to store the current state of the game.

```
board <- matrix(rep(0,9), nrow=3)      # a global
```

The GUI provides the representation of the data for the user. This example just uses a canvas for this, but most GUIs have a combination of components to represent the data and allow for user interaction. The layout of the GUI directs the user as to how to interact with it and is an important factor as to whether the GUI will be well recieved. Here we define a function to layout the game board using the graphics device as a canvas.

```
layoutBoard <- function() {  
  plot.new()  
  plot.window(xlim=c(1,4), ylim=c(1,4))  
  abline(v=2:3)  
  abline(h=2:3)  
  mtext("Tic Tac Toe. Click a square:")  
}
```

A GUI is designed to respond to user input typically by the mouse or keyboard. The underlying toolkit allows the programmer to assign functions to be called when some specific event occurs. Typically, the toolkit *signals* that some action has occurred, and then calls *callbacks* or *event handlers* that have been assigned by the programmer. How this is implemented varies from toolkit to toolkit.

R's interactive graphics devices implement the `locator` function which responds to mouse clicks by user. When using this function, one specifies how many mouse clicks to gather and the *control* of the program is suspended until these are gathered (or the process is terminated). The suspension of control makes this a *modal* GUI. This design is common for simple dialogs that require immediate user attention, but not common otherwise. To make non-modal dialogs possible, the writers of the R packages that interface with the GUI toolkits have to interface with R's event loop mechanism.

Here we define a function to call to collect the player's input.

```
doPlay <- function() {  
  iloc <- locator(n=1, type="n")  
  clickHandler(iloc)  
}
```

In the above fuction, `clickHandler` is an *event handler*. Its job is to process the output of the `locator` function, checking first if the user terminated `locator` using the keyboard. If not it proceeds to draw the move, and then,

if necessary, the computer's move. Afterwards, play is repeated until there is a winner or a "cat's" game.

```
clickHandler <- function(iloc) {
  if(is.null(iloc))
    stop("Game terminated early")
  move <- floor(unlist(iloc))
  drawMove(move,"x")
  board[3*(move[2]-1) + move[1]] <- 1
  if(!isFinished())
    doComputerMove()
  if(!isFinished())
    doPlay()
}
```

The use of `<-` in the handler illustrates a typical issue in GUI designing within R. After a GUI is created, the state is typically modified within the scope of the callback functions. These callbacks need to be able to modify values outside of their scope, yet even if the values are passed in as argument, this is usually not possible while assigning within the scope of the function call, due to R's pass by copy function calls. As such, global variables are often employed along with some strategies to avoid an explosion of variable names.

Validation of user input is an important task for a GUI, especially for Web GUIs. In the above, the `clickHandler` function checks to see if the user terminated the game early and issues a message, more helpful forms of validation are possible in general.

At this point, we have a data model, a view of the model and the logic connects the two, but we still need to implement some of the functions to tie it together.

This function draws either an "x" or an "o". It does so using the `lines` function. The `z` argument contains the coordinates of the square to draw.

```
drawMove <- function(z,type="x") {
  i <- max(1,min(3,z[1])); j <- max(1,min(3,z[2]))
  if(type == "x") {
    lines(i + c(.1,.9),j + c(.1,.9))
    lines(i + c(.1,.9),j + c(.9,.1))
  } else {
    theta <- seq(0,2*pi,length=100)
    lines(i + 1/2 + .4*cos(theta), j + 1/2 + .4*sin(theta))
  }
}
```

One could use `text` to place a text "x" or "o", but this may not look good if the GUI is resized. Most GUI layouts allow for dynamic resizing. Overall this is a great advantage, for example it allows translations to just worry

## 1. THE BASIC IDEAS OF GRAPHICAL USER INTERFACES

---

about the text and not the layout even though some languages are much more verbose and hence require more space.

This function is used to test if a game is finished. The matrix `m` allows us to easily check all the possible ways to get three in a row.

```
isFinished <- function() {  
  if(sum(abs(board)) >= 9)  
    return(TRUE)  
  m <- matrix(1:9,nrow=3)  
  ways <- list(m[,1], m[,2], m[,3],  
              m[1,], m[2,], m[3,],  
              diag(m),diag(apply(m,2,rev)))  
  sums <- sapply(ways, function(i) abs(sum(board[i])))  
  if(any(sums == 3))  
    return(TRUE)  
  return(FALSE)  
}
```

This function picks a move for the computer. The move is converted into coordinates using `%%` to get the remainder and `%%` to get the quotient when dividing an integer by an integer. This function just chooses at random from the left over positions, we leave implementing a better strategy to the interested reader.

```
doComputerMove <- function() {  
  ## random !  
  newMove <- sample(which(board == 0),1)  
  board[newMove] <- -1 # otherwise a copy of board  
  z <- c((newMove-1) %% 3, (newMove-1) %% 3) + 1  
  drawMove(z,"o")  
}
```

This function provides the main entry point for our GUI. To play a game it first lays out the board and then calls `doPlay`. When this function terminates, a message is written on the screen.

```
playGame <- function() {  
  layoutBoard()  
  doPlay()  
  mtext("All done\n",1)  
}
```

Finally, the above example illustrates a common issue when designing GUIs – they seemingly can always be improved. In this case, there are many obvious improvements: localizing the text messages so different languages can be used, implementing a better logic for the computer's moves, drawing a line connecting three in a row when there is a win, indicating who won when there is a win, etc. For many GUIs there is a necessary trade-off between offering sufficient usability versus increased complexity.



### 1.3 GUI Design Principles

A typical interface design consists of a top-level window referred to as the *document window* that shows the current state of a “document”, whatever that is taken to be. In R it could be a data frame, a command line, a function editor, a graphic etc.. The actions that can be done on the “document” are called through the menubar or toolbar and have their parameters controlled in sub-windows. These sub-windows are termed *application windows* by Apple (?), but we prefer the term *dialogs*, or *dialog boxes*. These terms may also refer to smaller sub-windows that are used for alerts or confirmation. As these are often used in a modal manner, we refer to them as *modal dialog boxes*.

Each window or dialog typically consists of numerous widgets or controls layed out in some manner to facilitate the user interaction. However, there are numerous means to achieve the same goals. For example, Figure ?? contains three dialogs that perform the same task – collect arguments from the user to customize the printing of a document. Although all were designed to do the same thing, there are many differences.

In some cases, typical usage suggests one control over another. The choice of printer for each is specified through a combo box. However, in other cases there can be a variety. For example, the control to indicate the number of copies for the mac and firefox dialogs is a simple text entry window, whereas for the KDE dialog it is a spinbutton. The latter minimizes user error, say through entering a non-positive integer. The KDE and Mac dialogs have icons to compactly represent actions, whereas the firefox one has none. The landscape icon for the Mac is very clear and provides this feature without having to use a sub dialog.

How the interfaces are layed out also varies. All are read top to bottom, although the Mac interface also has a very nice preview feature on the left side. The firefox and KDE dialogs use frames to separate out the printer arguments from the arguments that specify how the print job is to proceed. The Mac uses a vertical arrangement to guide the user through this. For the Mac, horizontal separators are used instead of frames to break up the areas, although a frame is used towards the bottom.

Apple uses a center balance for its controls. They are not left justified as are the KDE and firefox dialogs. Apple has strict user-interface guidelines and this center balance is a design decision.

The Mac GUI provides a very nice preview of the current document indicating to the user clearly what is to be printed and how much. Adjusting GUIs to the possible state is an important user interface property. In each GUI areas that are not currently sensitive to user input are grayed out. For example, the “collate” feature of the GUI only makes sense when more than 1 copy is selected, so the designers have it grayed out until then. It is common practice when designing GUIs to only enable controls that initiate actions when those actions can actually be completed given the state of the applica-

## 1. THE BASIC IDEAS OF GRAPHICAL USER INTERFACES

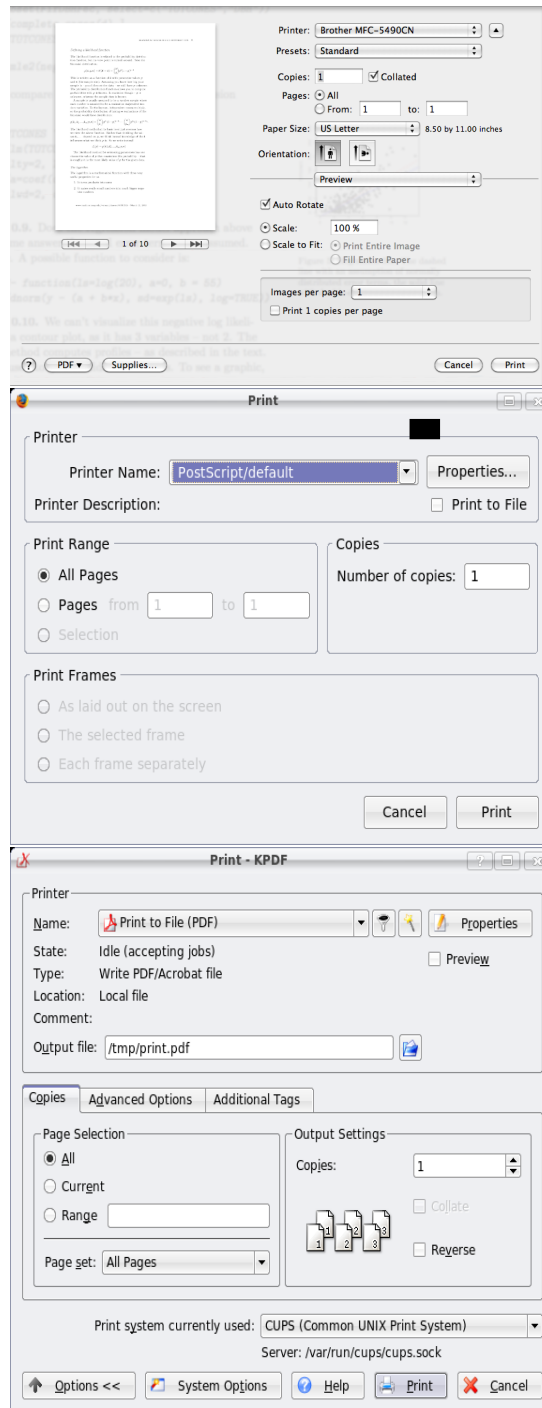


Figure 1.1: Three print dialogs. One from Mac OS X 10.6, one from firefox 2.0 and one from KDE 3.5.

tion.

The Mac GUI has the number of pages in focus, whereas for the firefox one the printer is focused. This allows the user to interact with the GUI without the mouse. Typically the tab key is used to step through the controls. GUIs often have keyboard accelerators, such as the KDE dialog where the underlined letters indicate the accelerator. Such additions help power users navigate through the interface quicker. As well most dialogs have a default button, which will initiate the action when clicked on or if the return key is pressed. In the KDE dialog, you can see by the shading that this is the “print” button.

For such a common dialog, it is unlikely the user will need help. As such the firefox dialog does not provide a link. However, the KDE and Mac dialogs do. A typical dialog should if the functions it provides are not commonly carried out.

The Apple human interface guidelines suggest putting buttons that can cause the destruction of data separate from other control buttons. As this isn’t directly applicable here, we see that Apple does separate buttons that are common to many dialogs (cancel, print) from the ones specific to the dialog. The KDE buttons have nice icons, but their similar, but irregular, sizing is a bit unusual.

One of the greatest differences between these dialogs is the amount of features they choose to expose. The firefox dialog is essentially minimal by comparison. As this is for printing from a browser, and not other documents it makes sense. The Mac GUI uses “disclosure buttons” to allow access to printer properties and the PDF settings, whereas KDE uses a notebook container to show possible options without revealing all their detail.

## Choice of widget

A GUI is comprised of one or more widgets. Typically these widgets allow a user to make a selection from one or more possible values. The choice of widget varies greatly depending on the length and type of values, on the desired number to select, and on the pixel-size allowable in the GUI layout. Table ?? lists several different types of widgets used for selection from a set of values.

Figure ?? shows several such controls in a single GUI. A checkbox is used to include an intercept, a radio group is used to select either full factorial or a custom model, a combobox is used to select the “sum of squares” type, and a list box is used to allow for multiple selection from the available variables in the data set.

For many R object types there are natural choices of widget. For example, values from a sequence map naturally to a slider or spin button; a data frame maps naturally to a table widget; or a list with similar structure can map naturally to a tree widget. However, certain R types have less common

## 1. THE BASIC IDEAS OF GRAPHICAL USER INTERFACES

Table 1.1: Table of possible selection widgets by data type and size

| Type of data  | Single                             | Multiple                  |
|---------------|------------------------------------|---------------------------|
| Boolean       | Checkbox                           |                           |
| Small list    | radiogroup<br>combobox<br>list box | checkboxgroup<br>list box |
| Moderate list | combobox<br>list box               | list box                  |
| Large list    | list box                           | list box                  |
| Sequential    | slider spinbutton                  |                           |
| Tabular       | table                              | table                     |
| Heirarchical  | tree                               | tree                      |

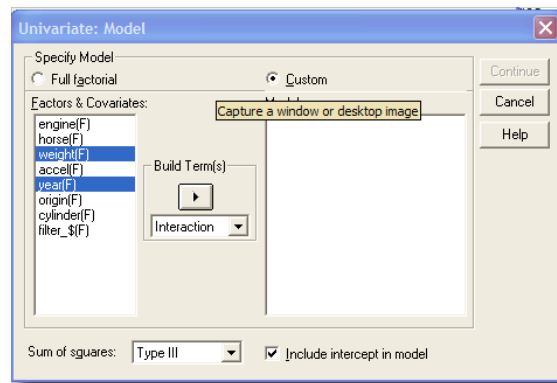


Figure 1.2: A dialog box from SPSS version 11 for specifying terms for a linear model. The graphic shows a dialog that allows the user to specify individual terms in the model using several types of widgets for selection of values, such as a radio group, a checkbox, combo boxes, and list boxes.

metaphors. For instance, a formula object can be a fairly complex thing. Figure ?? shows an SPSS dialog to build terms in a model. R power users may be much faster specifying the formula through a text entry box, but beginning R users coming to grips with the concept of a command line and the concept of a formula may benefit from the assistance a well designed GUI can provide. Designing an interface to balance both of these types of users may be desired, or one may decide to use an interface such as the cumbersome SPSS one. Knowing the potential user base is important.

## 1.4 Controls

This section provides an overview of many common GUI controls.

### Selection

A major task for a GUI for statistics is the selection of a value. For R there are conceivably several different possible types of selections. For example selecting a data frame from a list of data frames, selecting a variable in a data frame, selecting certain cases in a data frame, selecting a logical value for a function argument, selecting a numeric value for a confidence level, selecting a string to specify an alternative hypothesis. Clearly there can be no one-size-fits all widget to handle the selection of a value. We describe some standard selection widgets next.

### Checkboxes

A *checkbox* allows the user to select a logical value for a variable. Checkboxes have labels to indicate which variable is being selected. A *checkbox group* is a linked collection of checkboxes that allows the selection of one or more values at a time.

### Radio Button Groups

A *radio button group* allows a user to select exactly one value from a vector of possible values. The analogy dates back to old car radios where there were 6 buttons to press to select a channel. When a new button was pushed in, the old button popped up. Radio button groups are useful, provided there are not too many values to choose from, as all the values are shown. These values can be arranged in a row, a column or rows and columns to better use screen space.

### Sliders and spinbuttons

A *slider* is a widget that allows the selection of a value from a sequence of possible values (typically) through the manipulation of a knob that can visu-

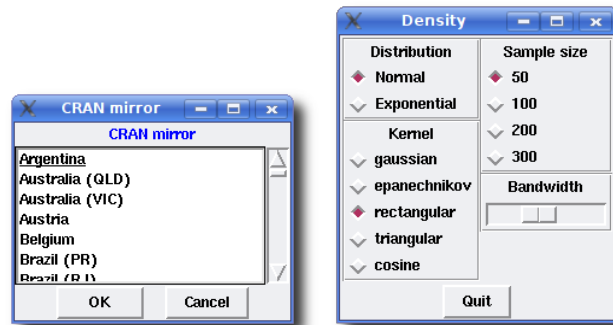


Figure 1.3: Two applications of the `tcltk` package. The left graphic is produced by `chooseCRANmirror` and uses a list box to allow selection from a long list of possibilities. The right graphic is the `tkdensity` demo from the package. It has various widgets that allow for the selection of parameter values for a density plot.

ally range over the possible values. Some toolkits (e.g. Java) only allow for the sequence to have integer values. The slider is a good choice for offering the user a selection of parameter values. The `tkdensity` demo of the `tcltk` package (Figure ??) uses a slider to dynamically adjust the bandwidth of a density estimate.

A *spin button* also allows the user to specify a value from a possible sequence of values. Typically, this widget is drawn with a text box displaying the current value and two arrows to increment or decrement the selection. The text box can usually be edited. Some toolkits allow the sequence to be a more general than a numeric sequence. A spin button has the advantage of using less screen space, but is much harder for the user to use if the sequence to choose from is long. A spin button is used in the KDE print dialog of Figure ?? to adjust the number of copies.

### Popup menus and Combo boxes

A *popup menu* is a widget that allows the selection of one or more fixed values while only showing just the currently selected one. Usually an arrow is drawn beside the selected value to indicate to the user a choice is possible. Clicking on this arrow will cause a pre-specified list of values to either drop down or up depending on the location of the widget on the screen. If there are too many choices, then typically further arrows are used to indicate more available choices. From a screen-space perspective, they can efficiently allow the selection of a value from many values, although a choice from too many values can be annoying to the user, as anyone who has had to select their country out of over 100 in a web form can attest. Some toolkits, such as

GTK+ allow the specification of an icon next to the list of values. A *combo box* is a popup menu combined with a text entry area to allow the user to specify their own value.

### List boxes

A *list box* is a widget that displays in a column the list of possible choices. Often a scrollbar is used when the list is too long to show in the allocated space. Some toolkits have list boxes that allow the values to spread out over several columns. Selection typically occurs with a right mouse click or through the keyboard, whereas a double-click will typically be programmed to initiate some action. Figure ?? shows a list box created by R that is called from the function `chooseCRANmirror`.

List boxes are a good alternative to drop lists when the number of selections gets bigger than 30. Additionally, list boxes can be programmed to allow for multiple selection. This is typically done by holding down either the shift or ctrl keys.

Some toolkits allow widgets to be placed next to the entries, such as checkboxes or icons. The right-most graphic in Figure ?? shows how SPSS places an icon indicating the type of variable next to the variable name in a list box, to aid in selection of the proper type of variable.

### Tabular display

A *table widget* shows tabular data, such as a data frame, where typically each column is rendered in the same manner but different columns need not be. The table widget need not be as flexible as a spread sheet, despite their similar appearances. For example, in GTK+ the selection is done by row, as in a list box.

Table widgets may be used to display and select tabular data, or may be intended for the user to also edit the data. Figure ?? shows the data editor from the JGR GUI that uses Java's table widget.

In RGtk2 and rJava, the table widget has a separation of the data (the model) and its display. This has many advantages, such as, allowing for sorting and filtering of the values displayed without changing the underlying data, allowing for multiple views of the same data; or allowing for efficient ways to store, perhaps, sparsely filled tables. Figure ?? shows a table object and a graph object in a Fathom session. These are two different views for the same data model. The two are dynamically linked, as changes to the underlying model are propagated to its views so that they are updated.

The view part of a table, for efficiency, is geared around the display of its columns rather than on a cell-by-cell basis (although this may be overridden, such as when a cell is being edited). The programmer then needs to specify

## 1. THE BASIC IDEAS OF GRAPHICAL USER INTERFACES

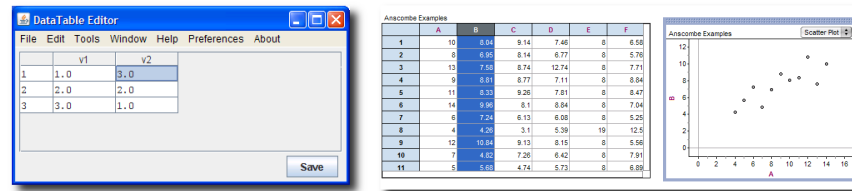


Figure 1.4: Two windows showing the use of table widgets. The left graphic shows the data editor from JGR using the table widget in Java. The right graphic shows a data table and a graph in Fathom 2.1 with two views of the same data. One view uses a table widget, the other a graph. Changes to one or the other views cause an update to the underlying model. This model then will notify its various displays to update. This arrangement allows for dynamic linking of the table and the graph.

a mapping of data in the model to column numbers in the display and the means of rendering the data in the column.

### Tree widgets

List boxes essentially display vectors of similar data, essentially tables display data frames. If the data has a hierarchical structure, then a *tree widget* can be used to show that. Examples of hierarchical data in R, are directory structures, the components of a list, or the inheritance of methods. The object browser in JGR uses a tree widget to show the components of the objects in a users session (the left graphic of Figure ??). The root node of the tree is the “data” folder, each data object in the global workspace is treated as an offspring of this root node. For the data frame *iraq*, its variables are considered as offspring of the data frame. In this case these variables have no further offspring, as indicated by the “page” icon.

### Initiating an action

When users use some widgets, they expect some immediate action to occur. Examples of these widgets are the familiar buttons, menubars and toolbars.

### Buttons

A *button* is typically used to give the user a place to click the mouse in order to initiate some immediate action. The “Properties” button, when clicked, causes a dialog for setting a printers properties to open. Similarly, the button with the wizard icons does something similar. As buttons typically lead to an action, they often are labeled with a verb. (?) In Figure ?? we see how SPSS uses buttons in its dialogs: buttons which are not valid in the current state,



as the user has not input enough information are disabled; buttons which are designed to open subsequent dialogs have trailing dots; and the standard actions of resetting the data, canceling the dialog or requesting help are given their own buttons on the right edge of the dialog box.

The look of the button can usually be manipulated. A button is given a relief through its border, shading, and perhaps a color gradient along its face. Some toolkits allow these to be optionally drawn, thereby making a button look more like a label, as described below. The button text may have some markup or an indication of an accelerator keyboard binding, such as the Contrasts... button in the dialog shown in the right graphic of Figure ??.

### icons

An *icon* is used to augment or replace a text label on a button, a toolbar, in a list box, etc.. When icons are used on toolbars and buttons, they are associated with actions, so the icons should have some visual association. Well drawn icons make a big visual difference in a GUI. The KDE print dialog, contrasted to the firefox print dialog in Figure ?? illustrates this point.

Except for the default installation of `tk`, images and icons may be specified in a variety of different formats. Icons can come in several different sizes from 16 by 16 pixels to 128 by 128. For toolbars and menubars, the toolkit takes care of selecting the appropriate icon.

### Menubars

Xerox Parc's revolutionary idea of a WIMP GUI added windows, icons, menubars, and pointing devices to the desktop computing environment. The central role of menu bars has not diminished. For a GUI, the *menubar* gives access to the full range of functionality available. Each possible action should have a corresponding menu item. Menubars are traditionally associated with a top-level window. This is enforced by the toolkit in `wxWidgets` and Java but not `Tcl/Tk` and `GTK+`. In Mac OS X, the menubar appears on the top line of the display, but otherwise they typically appear at the top of the main window. In most modern applications, standard document-based design is used to organize a GUI and its actions, with a main window showing the document and its menu bar calling actions on the document, some of which may need to open subsequent application windows or dialogs for gathering additional user input. In this model, only the main window has a menu bar not the application windows or dialogs. In a statistics application, the "document" may be viewed as the active data frame, a report, or a graphic so there may be many different menu bars needed.

The styles used for menubars are fairly standardized, as this allows new users to quickly orient themselves with a GUI. The visible menu names are often in the order File, Edit, View, Go, then application specific menus, and

finally a Help menu. Each visible menu item when clicked opens a menu of possible actions. The text for these actions traditionally use a . . . to indicate that a subsequent dialog will open so that more information can be gathered to complete the action, as opposed to some immediate action being taken. The text may also indicate a key-board accelerator, such as Find Next F3 indicating that both “N” as a keyboard accelerator and F3 as a shortcut will initiate this same action.

Not all actions will be applicable at any given time. It is recommended that rather than deleting these menu items, they be disabled, or greyed out, instead. ??

Menus can get very long. To help the user the menu items are usually grouped together, first by being under the appropriate menu title, then with either horizontal separators to define subsequent groupings, or hierarchical submenus. The latter are indicated with an arrow. Several different levels are possible, but navigating through to many can be tedious.

The use of menus has evolved to also allow the user to set properties or attributes of current state of the GUI. There may be checkboxes drawn next to the menu item or some icon indicating the current state.

Another use of menus is to bind contextual menus (popup menus) to certain mouse clicks on GUI elements. Typically right mouse clicks will pop up a menu that lists often-used commands that are appropriate for that widget and the current state of the GUI. In Mac OS X, one-button users these menus are bound to a control-click.

### Toolbars

Toolbars are used to give immediate access to the frequently used actions defined in the menubar. Toolbars typically have icons representing the action and perhaps accompanying text. They traditionally appear on the top of a window, but sometimes are used along the edges.

### Displaying text for editing

As much as possible, GUIs are designed around using the pointing device to select value for user input. Perhaps this is because it is difficult to both type and move the mouse at the same time. For statistical GUIs, especially for R with its powerful command line, it is essential that text entry widgets be used for any complicated GUI. Except with wxWidgets there is a distinction made between widgets for handling just single lines versus multiple lines of text.

#### Single line text

A text entry widget for editing a single line of text is used in the Firefox print dialogs in Figure ?? to specify the number of copies. An advantage to

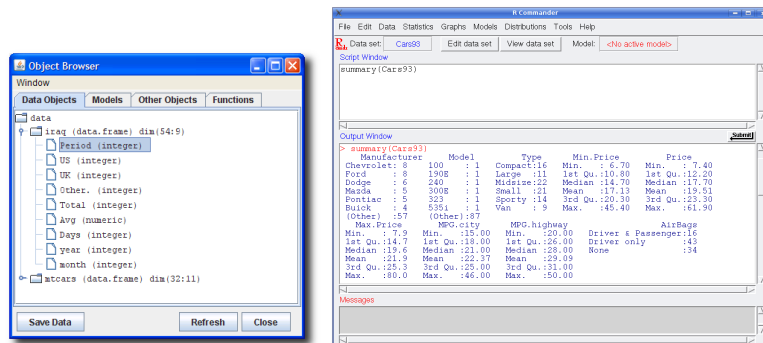


Figure 1.5: Some windows from R GUIs. The left graphic shows the object browser in the JGR GUI using a tree widget to display the possibly heirarchical nature of R objects. The right graphic shows the main Rcmdr (1.3-11) window illustrating the use of multi-line text entry areas for a command area, an output area and a message area.

using a text entry area for this specification, as opposed to a spinbutton, is the ease of editing for a user who uses the tab key to move between fields rather than the mouse. A disadvantage of using this type of widget is the need to validate the user's input, as it only makes sense to have a positive integer for this value.

### Text edit boxes

Multi-line text entry areas are used in many GUIs. The right graphic of Figure ?? shows a text entry areas used by Rcmdr to enter in R commands; to show the output of R commands and to provide a message area (in lieu of a status bar). The “Output Window” shows that text may have formatting attributes applied to it, in this case the color of the commands differs from that of the output. This is done by specifying text-changing attributes to different portions of the text. The difficulty is that when the text widget is changed, these positions must also reflect the change. Alternately, some of the toolkits allow HTML to be used for markup.

### Display of information

Some widgets are typically used to just display information and are not expected to have any response to mouse clicks. These are called static controls in wxWidgets.

### Labels

A label is a widget for placing text into a GUI that is typically not intended for editing, or even selecting with a mouse. This widget is used to label other controls, so the user understands what will happen when that control is changed.

A Label's text can be marked up in some toolkits using HTML in Java/Swing and pango markup in GTK+. As well, regular font attributes can be applied to the label in the other toolkits. Figure ?? shows labels marked in red and blue in `tcltk`.

### Statusbars

A typical top-level window will have a menubar and toolbar for access to the possible actions, an area to display the document being worked on, and at the bottom of the window a statusbar for giving the user immediated feedback on the actions that have been initiated. Figure ?? show a text area being used as a status bar for messages in the `Rcmdr` GUI. Likely this was chosen as there is no ready-made status bar widget in the base `tcltk` package. The other toolkits do provide one.

### Progress bars

A progress bar is used to notify how much time is left during a long computation. They are often used during software installation.

### Tooltips

A tooltip is a small window that pops up when a user hovers their mouse over a tooltip-enabled widget. These are used to give quick feedback on that particular widget. They can be useful for guiding new users of a GUI and annoying for familiar users. In R their implementation for the toolkits can be hampered due to eventloop issues.

### Modal dialog boxes

A *modal dialog box* is a dialog box that keeps the focus until the user takes an action to dismiss the box. They are used to notify the user of some action, perhaps asking for confirmation in case the action is destructive, such as overwriting of a file name. Modal dialog boxes can be disruptive to a users flow, so should be used sparingly. As the flow essentially stops until the window is dismissed, functions that call modal dialogs can return a value when an event occurs, rather than have a handler respond to the dismiss event. The `file.choose` function, mentioned below, is a good example. When used dur-

ing an interactive R session, the user is unable to interact with the command line until a file has been specified or the dialog dismissed.

### **File choosers**

A file chooser allows for the selection of existing files, existing directories, or the specification of new files. They are familiar to any user of a GUI. A typical R installation has the functions `file.choose` and `tkchooseDirectory` (in the `tcltk` package) to select files and directories, the latter using `tcltk` and the former using the windows file selector if applicable. The use of

Other common choosers are color choosers and font choosers.

### **Message dialogs**

A *message dialog* generally has a pretty standard form, a small rectangular box that appears in the middle of the screen with an icon on the left and a message on the right. At the bottom is a button to dismiss the dialog, often labeled “OK.” A *confirmation dialog* would add a “cancel” button which the programmer can use to invalidate the proposed action.

## **1.5 Containers**

Widgets are arranged in a window to produce a GUI. How they are laid out is done using containers. The simplest containers are like boxes that get packed in left to right or top to bottom. These boxes may be decorated with a frame or label, or may have some means of being hidden or displayed by the user. The nesting of box containers can provide a great deal of flexibility, but usually not enough. Additionally, layout can be done using grids to specify where to place the widgets.

### **Top level windows**

The top-level window of a GUI or its sub-windows typically is drawn with window decorations including a title and buttons to iconify, maximize, or close. In addition, a top-level window may have room for a menu bar, a tool bar and a status bar. In between these, if present, is the area referred to as the *client area* or *content pane* where other containers or widgets are placed.

The title is a property of the window and may be specified at the time of construction or afterwards.

On a desktop, only one window may have the focus at a time. It may or may not be desired that a new window receive the focus so some means to specify the focus at construction or later is provided by the toolkit.

The initial placement of the window also may be specified at the time of construction. The top-level window of a GUI may generally be placed

whereever it is convenient for the user, but sub-windows are often drawn on top of their parent window, as are modal dialog boxes.

The initial size of the window may also be specified at construction. The size of a top-level window may be specified as a default size, a preferred size, or a minimum size, depending on the toolkit. All of these allow for resizing of the window with the mouse. When this is done, the laying out of widgets must be updated.

The window manager usually decorates a top-level window with a close button. It may be necessary or desirable to specify some action to take place when this button is clicked. For instance, a user might be prompted if they wish to save changes to their work when the close button is pressed.

it may take some time to initially layout a top-level window. Rather than have the window drawn and then have a blank window while this time passes, it is preferable to not make the window visible until the window is ready to draw.

The layout of widgets in a top-level container follows a heirarchical structure. Widgets are packed inside containers which may again be packed inside containers. Eventually the widget or container is packed inside a top-level window. In `tk` a parent container must be specified when a widget is constructed. Reversing the direction, one can view the top-level window as a root node of a tree with each container packed directly into it as children. (In `GTK2` only one component can be packed into a top-level window.) Each child container, in turn, has children which are the components packed into it. This heirarchy is important when a window is resized. The new dimensions can be passed along to the children, and they can resize themselves accordingly. The allocation of this space varies from toolkit to toolkit.

We now describe some of the main containers.

### Box containers

A box container places its children in it from left to right or from top to bottom. If each child is viewed as a box, then this container holds them by packing them next to each other. The upper left figure in Figure?? illustrates this.

When the boxes have different sizes, then some means to align them must be decided on. Several possibilities exists. The alignment could be around some center, aligned at a baseline, the top line, or each child can specify where to anchor itself within the allotted space (the upper right graphic in Figure ??).

If the space allocated for a box is larger than the space requested by a child component then a decision as to how that component gets placed needs to be made. If the component is not enlarged, then there are nine reasonable places – the center and the 8 compass directions N, NW, W, ... Otherwise,

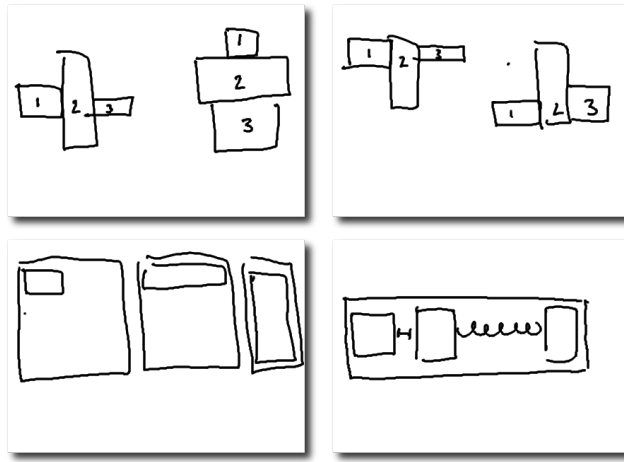


Figure 1.6: XXX REPLACE WITH REAL GRAPHICS XXX. Different possibilities for packing child components within a box. The upper left shows horizontal and vertical layout. The upper right shows some possible alignments. The lower left shows that a child could possibly be anchored in one of 9 positions. As well, it could “expand” to fill the space either horizontally (as shown) or vertically, or both. The lower right shows both a fixed amount of space between the children and an expanding spring between the child components.

it may be desirable for the component to expand horizontally, vertically or both (the lower left graphic in Figure ??). Additionally, it is desirable to be able to place a fixed amount of space between child components or a spring between components. A spring forces all subsequent to children to the far right or bottom of the container (the lower right graphic in Figure ??).

When a top-level window is resized, these space allocations must be made. To help, the different toolkits allows the components to have a size specified. Some combination of a minimum size, a preferred size, a default size, a specific size, or a maximum size are allowed. Specifying fixed sizes for components is generally frowned upon, as they don’t scale well when a user resizes a window and they don’t work well when different languages are used on the controls when an application is localized.

### Frames

A box container may have a border drawn around it to visually separate its contents from others. This border may also have a title. In GTK+ these are called frames, but this word is reserved in Java and wxWidgets for windows.

### Expanding boxes

In order to save screen space, a means to hide a boxes contents can be used. This hiding/showing is initiated by a mouse click on a button or label.

### Paned boxes

If automatic space allocation between two child components is not desired, but rather a means for the user to allocate the space is, then a *paned container* may be used. These offer either a horizontal or vertical sash that can be clicked and dragged to apportion space between two components.

### Grid layout

By nesting box containers, a great deal of flexibility in layout can be achieved. However, there is still a need for the alignment of child components in a tabular manner. The most flexible alignments allow for different sizes for each column and each row, and additionally, the ability for the child components to span multiple columns or rows. Within each cell (or cells) the placement of a child component mirrors that of the lower left graphic of Figure ?? . Some specification where to anchor the component when there are nine possible positions plus expanding options must be made.

### Notebooks

A notebook is a common container to hold one or more pages (or children). The different pages are shown by the user through the clicking of a corresponding tab. The metaphor being a tabbed notebook. Modern web browsers take advantage of this container to allow several web pages to be open at once.

### Example

The KDE print dialog of Figure ?? shows most of the containers previously described.

The top-level window has the generic title “Print – KPDF.” This window appears to have four child components: a frame labeled *Printer*, a notebook with open tab *Copies*, a grid layout for specifying the print system, and a box for holding five buttons at the bottom.

The lower left *Options* button has << to indicate that clicking this will close an expanding box, in this case a box that contains the lower three components above. So in fact, there are two visible child components of the top-level window.

The framed box holds a grid layout with five columns and 6 rows. The sizes allocated to each column are visible in the first row. It is quickly seen



that each column has a different size. The last row has a text entry area that spans the second and third columns. The first column has only labels. These are anchored to the left side of the allowed space. The Apple human interface guide (?, p. 124) suggests using colons for text that provides context for controls, and the KDE designers do to.

The displayed page of the notebook shows a two child components, both framed boxes. A pleasant amount of space between the frames and their child components has been chosen. The Page Selection frame has components including radio buttons, a text area, a horizontal separator, and a combo box.

The print system information is displayed in a grid layout that has been right aligned within its parent container – the expanded group, but its children are center-balanced with the label “Print system currently used” is right aligned and “Server...” is left aligned within their cells.

The button box shows five buttons as child components. At first glance the sizing appears to show that each button is drawn to fully show its label with some fixed space placed between the buttons. If the dialog is expanded, it is seen that there is a spring between the 3rd and 4th buttons, so that the first 3 are aligned with the left side of the window and the last two the right side.

The Apple guidelines(?, Ch. 15) suggest using “center equalization” for arranging widgets within a window. This means that the visual weight is balanced between the right and left side of the content area. This is not the case with the KDE print dialog.

## 1.6 End of chapter notes

More documentation on GUIs is available in book format or online.

For GTK+ there is the gtk tutorial (pygtk); GTK API; DTL’s notes; example code in the RGtk2 package; php-gtk cookbook

For wxWidgets the book; DTL omegahat pages; wxWidgets API;

For Tcl/Tk ActiveStates API; wettenhall examples (sciviews); Dalgaard’s papers; R mailing list; book

For Java Sun’s website tutorials; API; rJava package page;

Event loops



---

## R Programming Practices for GUIs



## gWidgets: Overview

The `gWidgets` package provides a toolkit-independent interface for the R user to program graphical user interfaces from within R. Although the package provides much less functionality than using the native toolkits through their R bindings, `gWidgets` can be used to create moderately large GUIs quickly and easily using a programming interface that is simpler and familiar for the R user to learn.

The `gWidgets` package started as a port to `RGtk2` of the `iWidgets` interface for `rJava` (?). The `gWidgets` package enhances that original interface in terms of functionality and is now toolkit independent.

### 3.1 Installation, toolkits

The `gWidgets` package is installed and loaded as other R packages that reside on CRAN. This can be done through the function `install.packages` or on some OSes through a dialog called from the menu bar. The `gWidgets` package provides the programming interface only (API). To actually create a GUI, one needs to have the underlying toolkit libraries, an underlying R package that provides an interface to the libraries, and a `gWidgetsXXX` package to link `gWidgets` to the R package. The installation of the libraries varies depending on toolkit and OS. Some details are presented by the function `installing_gWidgets_toolkits`.

As of this writing, there are basically complete `gWidgets` packages for the toolkit packages `RGtk2` and `tcltk`; `gWidgetsWWW` ports the API to allow interactive web pages (cf. Chapter ??). Also there is `gWidgetsrJava`, but that package is not being supported. Here, only the implementations for the `RGtk2` and `tcltk` packages are covered.

Not all features of the API are available in each package. The help pages in `gWidgets` describe the API, with the help pages in the toolkit packages indicating differences or omissions from the API. For the most part, the omissions are gracefully handled by simply providing less functionality. We make note of these differences for `gWidgetsRGtk2` and `gWidgetstcltk` here, realiz-

### 3. gWIDGETS: OVERVIEW

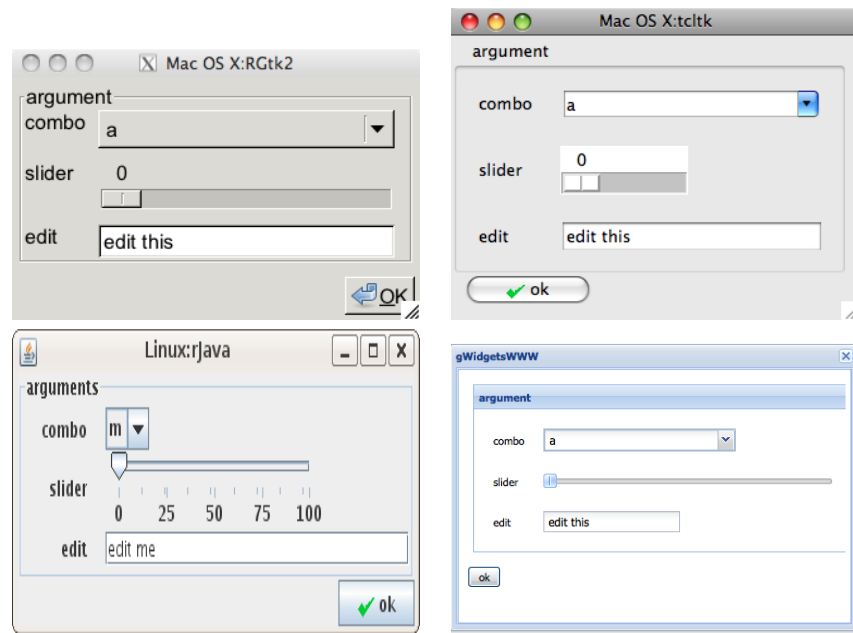


Figure 3.1: The gWidgets package works with different operating systems and different GUI toolkits. This shows, the same code for RGtk2, tcltk, rJava and gWidgetsWWW. Note, each toolkit has it's own sizing ideas for the controls.

ing that over time they may may be resolved. Consult the package documentation if in doubt.

Figure ?? shows how the same GUI code can be rendered differently depending on the OS and the toolkit.

## 3.2 Startup

The gWidgets package is loaded as other R packages:

```
require(gWidgets)
```

A toolkit package is loaded when the first command is issued. If a user does not have a toolkit installed, then a message indicating the need to install an appropriate package is given.

If a user has exactly one toolkit package installed, then that will be used. But it is possible for more than one to be installed, in which case the user is prompted to choose one through an interactive menu. This choice can be avoided by setting the option `guiToolkit` to the underlying R package name, as in

```
options("guiToolkit"="RGtk2")
```

The value is the name of one of the R packages that gWidgets can use.<sup>1</sup> Although in theory the different toolkits can be used together, in practice the different eventloops created by each often lead to issues that can lockup the R process.

### 3.3 Constructors

GUI objects are produced by constructors. In gWidgets most constructors have the following form:

```
gname(arguments, handler = NULL, action = NULL,
       container = NULL, ..., toolkit=guiToolkit())
```

where the arguments vary depending on the object being made.

#### Return value

Not only do constructors create visible GUI objects they also return a useful R object. Except for modal dialog constructors, this is an S4 object of a certain class containing two components `toolkit` and `widget`. The `toolkit` can be specified at time of construction allowing toolkits, in theory, to be mixed. Otherwise, the `guiToolkit` function returns the currently selected toolkit, or queries for one if none is selected point about dispatch Constructors dispatch on the `toolkit` value to call the appropriate constructor in the toolkit implementation. The return value from the toolkit's constructor is kept in the `widget` component. Generic methods have a double dispatch when called. The first dispatch is based on the `toolkit` value which calls a second generic implemented in the toolkits with a different name (`svalue` dispatches to `.svalue`). The toolkit generic, then dispatches based on the class of `widget` component and perhaps other arguments given to the generic. The actual class of the S4 object returned by the first constructor is (mostly) not considered, but when we refer to methods for an object, we gloss over this double dispatch and think of it as a single dispatch. This design allows the toolkit packages the freedom to implement their own class structure.

As with most R objects, one calls generic functions to interact programmatically with the object. The gWidgets package provides some familiar S3 methods, for the appropriate objects, for the familiar generics `l`, `[<-`, `dim`, `length`, `names`, `names<-`, `dimnames`, `dimnames<-`, `update`. In addition, it provides new generics listed in Table ??.

These new generics provide a means to query and set the primary value of the widget (`svalue`, `svalue <-` ), and various methods to effect the display of the widget (`visible <-` , `font <-` , `enabled <-` , `focus <-` ).

<sup>1</sup>As of writing, this is either `RGtk2`, `tcltk`, or `rJava`. The `gWidgetsWWW` package does not use `gWidgets` for dispatch, rather it is loaded directly.

Table 3.1: Generic functions provided or used in gWidgets API.

| Method                              | Description   |
|-------------------------------------|---|
| <code>svalue, svalue &lt;-</code>   | Get or set value for widget                                 |
| <code>[, [ &lt;-</code>             | Refers to values in data store                              |
| <code>length</code>                 | length of data store  |
| <code>dim</code>                    | dim of data store   |
| <code>names</code>                  | names of data store   |
| <code>dimnames</code>               | dimnames of data store                                      |
| <code>update</code>                 | Update widget values  |
| <code>size &lt;-</code>             | Set size of widget in pixels                                |
| <code>show</code>                   | Show widget if not visible                                  |
| <code>dispose</code>                | Destroy widget or its parent                                |
| <code>isExtant</code>               | Does R object refer to GUI object that still exists         |
| <code>enabled, enabled &lt;-</code> | Adjust sensitivity to user input                            |
| <code>visible, visible &lt;-</code> | Adjust widget visibility.                                   |
| <code>focus &lt;-</code>            | Sets focus to widget  |
| <code>defaultWidget &lt;-</code>    | Set widget to have initial focus in a dialog                |
| <code>insert</code>                 | Insert text into a multi-line text widget                   |
| <code>font &lt;-</code>             | Set a widget's font   |
| <code>tag, tag &lt;-</code>         | Sets an attribute for a widget that persists through copies |
| <code>id, id &lt;-</code>           | A unique ID for a widget                                    |
| <code>getToolkitWidget</code>       | Returns underlying toolkit widget for low-level use         |

The methods `tag` and `tag <-` are implemented to bypass the pass-by-copy issues that can make GUI programming awkward at times.

The gWidgets API provides just a handful of generic functions for manipulating an object compared to the number of methods typically provided by a GUI toolkit for a similar object. Although this simplicity makes gWidgets easier to work with, one may wish to get access to the underlying toolkit object to work at that level. The `getToolkitWidget` will provide that object. We don't illustrate this, as we try to stay toolkit agnostic in our examples.

A few constructors create modal dialogs. These do not return objects, as when they are visible the R session is unresponsive. Consequently they have no methods defined for them. Instead, these constructors return values produced by the dialogs, which in turn can be used as arguments to functions.

### The container argument

The constructors produce two types of objects: containers (Table ??) and components (the basic controls in Table ?? and the compound widgets in Table ??).



A GUI consists of a hierarchical nesting of containers which in turn contain other containers or components. In a GUI, except for top-level windows and modal dialogs, every component and container is the child of some parent container. In `gWidgets` this parent is specified with the `container` argument when an object is constructed. This argument name can always be abbreviated `cont`. In the construction of a widget in `gWidgets`, the `add` method for the parent container is called with the new object as an argument and the values passed through the `...` argument as arguments. We remark that not all the toolkits (e.g., `RGtk2`) require one to combine the construction of an object with the specification of the parent container. We don't illustrate this, as the resulting code is not cross-toolkit.

### **The handler and action arguments**

For all the toolkits, when the user initiates some event with the mouse or keyboard, the underlying toolkit will emit some signal. The toolkits allow callbacks to be called when these signals are emitted allowing the GUI to be made interactive. In `gWidgets`, the callbacks are functions with signature `(h, ...)` where `h` is a list that contains user data. Some toolkits pass information through the `...` argument. As this is not portable across toolkits, we do not use this here. In general, the `obj` component of `h` contains the widget that the callback is assigned to, and the component `action` contains user-specified data passed through the `action` argument of the constructor or the `addHandlerXXX` method (the `XXX` may be one of several values). For some classes, extra information is passed along, for instance for the drop target generic, the component `dropdata` contains a string holding the drag-and-drop information.

A callback for an event can be specified through the `handler` argument of a constructor, or added at a later time through the "`addHandlerXXX`" methods. The generic `addHandlerChanged` can be used to add a callback for the most reasonably defined event. In many cases, more than one event is reasonable. For example, for single line text widgets the `addHandlerChanged` responds when the user finishes editing, whereas `addHandlerKeystroke` is called each time the keyboard is used. Table ?? shows a list of these other methods. If these few methods are insufficient, and toolkit-portability is not of interest, then the `addHandler` generic can be used to specify a toolkit-specific signal and a callback. When a `addHandlerXXX` method is used, the return value is an ID. This ID can be used with the method `removeHandler` to remove the callback, or with the methods `blockHandler` and `unblockHandler` to temporarily block a handler from being called.

## 3.4 Drag and Drop

Drag and drop support is implemented through three methods: one to set a widget a drag source, one to set a widget as a drop target, and one to call a handler when a drop event passes over a widget. The `addDropSource` method needs a widget and a handler specified to call when a drag and drop event is initiated. This handler should return the value that will be passed to the drop target. The default value is the value returned by `svalue` method on the object. The `addDropTarget` method is used to allow a widget to receive a dropped value and to specify a handler to call when a value is dropped. The `dropdata` component of the first-argument list, `h`, holds the drop data in the call to the handler. The `addDropMotion` is used to call a handler for the event that a drag event passes over a widget.

Unfortunately, the drag and drop implementations in the toolkits are not all the same. Some toolkits simply use the native drag and drop support, which can not be changed.

Table 3.2: Generic functions to add callbacks in gWidgets API.

| Method                             | Description  |
|------------------------------------|--|
| <code>addHandlerChanged</code>     | Refers to the signal that is bound to when the handler argument is used by the constructor. Interpretation varies from widget to widget. |
| <code>addHandlerClicked</code>     | Sets handler for when widget is clicked with (left) mouse button. May return position of click through components x and y of the h-list. |
| <code>addHandlerDoubleClick</code> | Sets handler for when widget is double clicked   |
| <code>addHandlerRightclick</code>  | Sets handler for when widget is right clicked  |
| <code>addHandlerKeystroke</code>   | Sets handler for when key is pressed. The key component is set to this value if possible.  |
| <code>addHandlerFocus</code>       | Sets handler for when widget gets focus  |
| <code>addHandlerBlur</code>        | Sets handler for when widget loses focus   |
| <code>addHandlerExpose</code>      | Sets handler for when widget is first drawn  |
| <code>addHandlerDestroy</code>     | Sets handler for when widget is destroyed  |
| <code>addHandlerUnrealize</code>   | Sets handler for when widget is undrawn on screen  |
| <code>addHandlerMouseMotion</code> | Sets handler for when widget has mouse go over it  |
| <code>addHandler</code>            | For non cross-toolkit use, allows one to specify an underlying signal from the graphical toolkit   |
| <code>removeHandler</code>         | Remove a handler from a widget   |
| <code>blockHandler</code>          | Temporarily block a handler from being called  |
| <code>unblockHandler</code>        | Restore handler that has been blocked  |
| <code>addHandlerIdle</code>        | Call a handler during idle time  |
| <code>addPopupMenu</code>          | Bind popup menu to widget  |
| <code>add3rdMousePopupMenu</code>  | Bind popup menu to right mouse click   |
| <code>addDropSource</code>         | Specify a widget as a drop source  |
| <code>addDropMotion</code>         | Sets handler to be called when drag event mouses over the widget   |
| <code>addDropTarget</code>         | Sets handler to be called on a drop event. Adds the component dropdata.  |



## gWidgets: Containers

The `gWidgets` package provides a few useful containers: top-level windows, box containers, grid-like containers and notebook containers.

### 4.1 Top-level windows

The `gwindow` constructor creates top-level windows. The title of the window can be set during construction via the `title` argument or later through the `svalue <-` method. As well, the initial size can be set through the `width` and `height` arguments. This initial size is the default size, but may be adjusted later through the `size` method or through the window manager. The `visible` argument controls whether the window is initially drawn. If not drawn initially, the `visible <-` method, taking a logical value, can be used draw the window later in a program. The default is to initially draw the window, but often it is good practice to suppress the initial drawing, especially for displaying GUIs with several controls as the incremental drawing of subsequent child components can make the GUI seem sluggish.

Windows can be closed programatically with the `dispose` method. Windows may also be closed through the window manager, by clicking a close icon in the title bar. The `handler` argument is called just before the window is destroyed, but will not prevent that from happening. The `addHandlerUnrealize` method can be used to call a handler between the initial click of the close icon and the subsequent destroy event of the window. This handler must return a logical value: if `TRUE` the window will not be destroyed, if `FALSE` the window will be, as illustrated in the example.

The initial placement of a window will be decided by the window manager, unless the `parent` argument is specified. If this is done with a vector of  $x$  and  $y$  pixel values, the upper left corner will be placed there. If it is specified as a `gwindow` instance, the new window will be positioned over the specified window and be disposed of when the parent widget is. This is useful, say, when a main window opens a dialog window to gather values.

Table 4.1: Constructors for container objects

| Constructor               | Description  |
|---------------------------|--|
| <code>gwindow</code>      | Creates a top-level window   |
| <code>gggroup</code>      | Creates a box-like container   |
| <code>gframe</code>       | Creates a container with a text label  |
| <code>gexpandgroup</code> | Creates a container with a label and trigger to expand/collapse                        |
| <code>gpandedgroup</code> | Creates a container for two child widgets with a handle to assign allocation of space. |
| <code>glayout</code>      | A grid container   |
| <code>gnotebook</code>    | A tabbed notebook container for holding a collection of child widgets                  |

In the document object model, the use of menubars, toolbars and statusbars is often reserved for the main window, while dialogs are not decorated so. In `gWidgets` it is suggested that these be added only to a top-level window and required by some toolkits.

**Example 4.1: An example of `gwindow`**

To illustrate, the following will open a new window. The initial drawing is postponed until after a button is placed in the window.

```
w1 <- gwindow("parent window", visible=FALSE)
b <- gbutton("a button", cont=w1)
visible(w1) <- TRUE
```

This shows how one might use the parent argument to specify where a sub-window will be placed.

```
w2 <- gwindow("child window", width=100, height=100,
              parent=w1)           # center on w1
b <- gbutton("button on child", cont = w2)
dispose(w1)                       # closes w2 also
```

This shows how the `addHandlerUnrealize` method can be used to intercept the closing of the window through the “close” icon of the window manager. The `gconfirm` dialog returns `TRUE` or `FALSE` depending on the button clicked, as will be explained in ??.

```
w <- gwindow("Close through the window manager")
id <- addHandlerUnrealize(w, handler=function(h,...) {
  !gconfirm("Really close", parent=h$obj)
})
```

Table 4.2: Container methods

| Method              | Description  |
|---------------------|--|
| <code>add</code>    | Adds a child object to a parent container. Called when a parent container is specified to the <code>container</code> argument, in which case, the <code>...</code> arguments are passed to this method |
| <code>delete</code> | Remove a child object from a parent container  |

## 4.2 Box containers

The container produced by `gwindow` is intended to contain just a single child widget, not several. This section demonstrates the box containers produced by `ggroup` that can be used to hold multiple child components. Through nesting, fairly complicated layouts can be produced.

### The `ggroup` container

The `ggroup` box container provides an argument `horizontal` to specify whether the child widgets are packed in horizontally left to right (the default) or vertically from top to bottom. Unlike with the underlying graphical toolkits, there is no means to specify other styles of packing such as from the ends, or in the middle by some index.

**add** When packing in child widgets, the `add` method is used. In our examples, this is called internally by the constructors when the `container` argument is specified. The appropriate `...` values for a constructor are passed to the `add` method. For `ggroup` the important ones are `expand` and `anchor`. When more space is allocated to a child, then is needed by that child, the `expand=TRUE` argument will cause the child to grow to fill the available space in both directions. (No means is available in `gWidgets` to restrict to just one direction.) If `expand=TRUE` is not specified, then the `anchor` argument will instruct how to anchor the child into the space allocated. The direction is specified by  $x$ - $y$  coordinates with both values from either  $-1$ ,  $0$  or  $1$ , where  $1$  indicates top and right, whereas  $-1$  is left and bottom. The example will demonstrate their use.

**delete** The `delete` method can be used to remove a child component from a box container. In some toolkits, this child may be added back at a later time, but this isn't part of the API.

**Spacing and sizing** For spacing between the child components, the argument `spacing` may be used to specify, in pixels, the amount of space between

#### 4. gWIDGETS: CONTAINERS

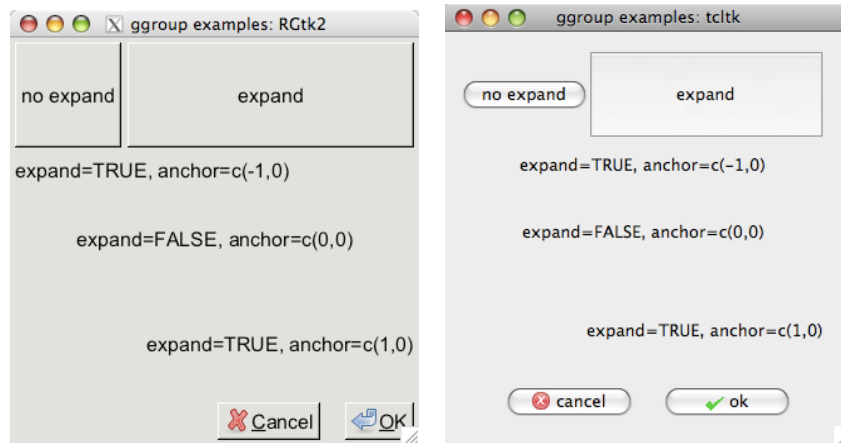


Figure 4.1: Use of `expand`, `anchor`, `addSpace` and `addSpring` with the `ggroup` constructor in `gWidgetsRGtk2` and `gWidgetstcltk`

the child widgets. This can later be set through the `svalue` method. The method `addSpace` can add space between two widgets packed next to each other, whereas the method `addSpring` will place an invisible spring between two widgets, forcing them apart. Both are useful for laying out buttons.

The overall size of `ggroup` container is controlled through it being a child of its parent container. However, a size can be assigned through the `size <-` method. This will be a preferred size, but need not be the actual size, as the container may need to be drawn larger to accomodate its children. The argument `use.scrollwindow` when specified as `TRUE` will add scrollbars to the box container so that a fixed size can be maintained. Although, it is generally considered a poor idea to use scrollbars when there is a chance the key controls for a dialog will be hidden.

##### Example 4.2: Example of `ggroup` usage

This example shows the nesting of vertical and horizontal box containers and the effect of the `expand` and `anchor` arguments. Figure ?? shows how it is implemented in two different toolkits.

```
w <- gwindow("ggroup examples")
g <- ggroup(cont=w, horizontal=FALSE, expand=TRUE)
g1 <- ggroup(cont=g, expand=TRUE)
b <- gbutton("no expand", cont=g1)
b <- gbutton("expand", cont=g1, expand=TRUE)
g2 <- ggroup(cont=g)
l <- glabel("expand=TRUE, anchor=c(-1,0)", anchor=c(-1,0),
           expand=TRUE, cont=g2)
```



```

g3 <- ggroup(cont=g, expand=TRUE)
l  <- glabel("expand=FALSE, anchor=c(0,0)", anchor=c(0,0),
            expand=TRUE, cont=g3)
g4 <- ggroup(cont=g, expand=TRUE)
l  <- glabel("expand=TRUE, anchor=c(1,0)", anchor=c(1,0),
            expand=TRUE, cont=g4)

```

This demonstrates how one might use the `addSpace` and `addSpring` methods in a button bar.

```

g5 <- ggroup(cont=g, expand=FALSE)
addSpring(g5)
b  <- gbutton("cancel", cont=g5, handler=function(h,..) dispose(w))
addSpace(g5, 10)
b  <- gbutton("ok", cont=g5)

```

The next example shows an alternative to the `expand` group widget.

#### Example 4.3: The `delete` method of `ggroup`

This example shows nested `ggroup` containers and the use of the `delete` method to remove a child widget from a container. In this application, a box is set aside at the top of the window to hold a message that can be set via `openAlert` and closed with `closeAlert`. This example works better under `RGtk2`, as the space allocated to the alert is reclaimed when it is closed.

This code sets up the area for the alert box to appear from.

```

w <- gwindow("Alert box example")
g <- ggroup(horizontal=FALSE, cont = w)
alertBox <- ggroup(cont = g)
mainBox <- ggroup(cont = g, expand=TRUE)
l <- glabel("main box label", cont = mainBox, expand=TRUE)
ig <- NULL                                     # global

```

These two functions will open and close the alert box respectively. In this example we use the global value, `ig`, to store the inner group.

```

openAlert <- function(message="message goes here") {
  ig <- ggroup(cont=alertBox)
  glabel(message, cont = ig)
}
closeAlert <- function() delete(alertBox, ig)

```

The state of the box can be toggled programmatically via

```

QT <- openAlert("new message")           # open
QT <- closeAlert()                       # close

```

#### The `gframe` and `gexpandgroup` containers

Framed containers are used to set off elements and are provided by `gframe`. Expandable containers are used to preserve screen space unless requested

and are provided by `gexpandgroup`. Both of these containers can be used in place of the `ggroup` container.

In addition to the `ggroup` arguments, the `gframe` constructor has the arguments `text` to specify the text marking the frame and `pos` to specify the positioning of the text, using 0 for left and 1 for right. If the toolkit supports markup, such as RGtk2, the `markup` argument takes a logical indicating if markup is being used in the specification of text. The `names` method can be used to get and set the label after construction of the widget.

The `gexpandgroup` constructor, like `gframe`, has the `text` argument, but no `pos` argument for positioning the text label. The widget has two states, which may be toggled either by clicking the trigger or through the `visible <-` method. A value of `TRUE` means the child is visible. The `addHandlerChanged` method is used to specify a callback for when the widget is expanded.

##### Example 4.4: The `gframe` and `gexpandgroup` containers

This example shows how the `gframe` container can be used.

```
w <- gwindow("gframe example")
f <- gframe(text="title", pos=1, cont=w)
l <- glabel("Some text goes here", cont=f)
names(f) <- "new title"
```

This is a similar example for `gexpandgroup`.

```
w <- gwindow("gexpandgroup example")
g <- gexpandgroup(text="title", cont=w)
l <- glabel("Some text goes here", expand=TRUE, cont=g)
visible(g) <- FALSE
visible(g) <- TRUE # toggle visibility
```

### 4.3 Paned containers: the `gpanedgroup` container

The `gpanedgroup` constructor produces a container which has two children. The children are aligned side-by-side by default, or top to bottom if the `horizontal` argument is given as `FALSE`. These two children are separated by a visual gutter which can be adjusted using the mouse to allocate the space between the two children. This can also be done programatically using the `svalue <-` method where a value from 0 to 1 specifies the proportion of space allocated to the leftmost (topmost) child.

To add children, the container should be used as the parent container for two constructors. These can be other container constructors which is the typical usage for more complicated layouts. (For toolkits which support the separation of widget construction and layout, the `gpanedgroup` constructor can have two children specified to the arguments `widget1` and `widget2`.)

#### Example 4.5: Paned groups

This example shows how one could use this container.

```
w <- gwindow("gpanedgroup example", visible=FALSE)
pg <- gpanedgroup(cont=w)
g <- ggroup(cont=pg)           # left child
l <- glabel("left child", cont=g)
b <- gbutton("right child", cont=pg)
visible(w) <- TRUE
```

To adjust the sash position, one can do:

```
svalue(pg) <- .5
```

## 4.4 Tabbed notebooks: the `gnotebook` container

The `gnotebook` constructor produces a tabbed notebook container. The constructor has the argument `tab.pos` to specify the location of the tabs. A value of 1 through 4 with 1 being bottom, 2 left side, 3 top and 4 right side being used, with the default being 3. (Not available for all toolkits.) The `closebuttons` argument takes a logical indicating whether the tabs should have close buttons on them. In this case, the argument `dontCloseThese` can be used to specify which tabs, by index, should not be closable. (Some toolkits do not implement these features though.)

The `add` method for the notebook container uses the `label` argument to specify the tab label. As this is called implicitly when a widget is constructed, this argument is specified to the constructor.

**Methods** The `svalue` method returns the index of the currently raised tab, whereas `svalue <-` can be used to switch the page to the specified tab. The currently shown tab can be removed using the `dispose` method. To remove a different tab, use this method in combination with `svalue <-` . When removing many tabs, you may want to start from the end as otherwise the tab positions change, which can be confusing when using a loop. The `names` method can be used to retrieve the tab names, and `names <-` to set the names. The `length` method returns the number of pages held by the notebook.

#### Example 4.6: Tabbed notebook example

A simple example follows. The `label` argument is passed along from the constructor to the `add` method for the notebook instance.

```
w <- gwindow("gnotebook example")
nb <- gnotebook(cont=w, tab.pos=3)
l <- glabel("first page", cont=nb, label="one")
b <- gbutton("second page", cont=nb, label="two")
```

To set the page to the first one:

```
svalue(nb) <- 1
```

To remove the first page (the current one)

```
dispose(nb)
```

### 4.5 Grid layout: the `glayout` container

The layout of dialogs and forms is usually seen with some form of alignment between the widgets. The `glayout` constructor provides a grid container to do so, using matrix notation to specify location of the children. The argument `homogeneous` can be used to specify that each cell take up the same size, the default is `FALSE`. Spacing between each cell may be specified through the `spacing` argument.

Children may be added to the grid at a specific row and column, and a child may span more than one row or column. To specify this, R's matrix notation, `[ <-` , is used with the indices indicating the row and column. When a child is to span more than one row or column, the corresponding index should be a vector of indices indicating so. There is no `[` method defined to return the child components. To add a child, the `glayout` container should be specified as the container and be on the left hand side of the `[ <-`  call. For convenience, if the right hand side is a string, a label will be generated. To align a widget within a cell, the `anchor` argument of the `[ <-`  `glayout` method is used. The example illustrates how this can be used to achieve a center balance.

#### Example 4.7: Layout with `glayout`

This example shows how a simple form can be given an attractive layout using a grid container. It uses the `gedit` constructor to provide a single-line text entry widget. As the matrix notation does not have a means to return the child widget (a `[` method say), we store the values of the `gedit` widgets into variables.

```
w <- gwindow("glayout example")
tbl <- glayout(cont=w)
right <- c(1,0); left <- c(-1,0)
tbl[1,1, anchor=right] <- "name"
tbl[1,2, anchor=left ] <- (name <- gedit("", cont=tbl))
tbl[2,1, anchor=right] <- "rank"
tbl[2,2, anchor=left ] <- (rank <- gedit("", cont=tbl))
tbl[3,1, anchor=right] <- "serial number"
tbl[3,2, anchor=left ] <- (snumber <- gedit("", cont=tbl))
```

## gWidgets: Control Widgets

### 5.1 Basic controls

This section discusses the basic controls provided by gWidgets.

#### Buttons, Menubars, Toolbars

The button widget allows a user to initiate an action through clicking on it. Buttons have labels – usually verbs indicating action – and often icons. The `gbutton` constructor has an argument `text` to specify the text. For text that matches the stock icons, an icon will also be rendered. A list of stock icons is returned by `getStockIcons`. In common with the other controls, the argument handler is used to specify a callback and the action argument will be passed along to this callback (unless it is a `gaction` object, whose case is described below).

The click handler can be specified at construction, or afterward through the `addHandlerClicked` which is also aliased to the generic `addHandlerChanged`.

A new button may or may not have the focus when a GUI is constructed. If it does have the focus, then the return key will initiate the button click signal. To make a GUI start with its focus on a button, the `defaultWidget` method is available.

The `svalue` method will return button's label. The method `svalue <-` is used to set the label text. Most GUIs will make a button insensitive to user input if the button's action is not currently permissible. Toolkits draw such buttons in a greyed out state. The `enabled <-` method can set or disable whether a widget can accept input.

#### Example 5.1: Hello world button

This example shows how a button is assigned a handler to respond to click events.<sup>1</sup> When working with handlers, one can use an object name that will

---

<sup>1</sup>Each toolkit has its idiosyncracies. If this example is run using `RGtk2` the button will stretch to fill the space. At times this is not desired. Placing the button within a `ggroup` container can

## 5. gWIDGETS: CONTROL WIDGETS

Table 5.1: Table of constructors for control widgets in gWidgets. Most, but not all, are implemented for each toolkit.

| Constructor    | Description  |
|----------------|--|
| glabel         | A text label   |
| gbutton        | A button to initiate an action                         |
| gradio         | A radio button group                                   |
| gcheckbox      | A checkbox   |
| gcheckboxgroup | A group of checkboxes                                  |
| gcombobox      | A drop-down list of values                             |
| gtable         | A table (vector or data frame) of values for selection |
| gslider        | A slider to select a value                             |
| gspinbutton    | A spinbutton to select from a set of values            |
| gedit          | Single line of editable text                           |
| gtext          | Multi-line text edit area                              |
| ghtml          | Display text marked up with HTML                       |
| gdf            | Data frame viewer and editor                           |
| gtree          | A display for heirarchical data                        |
| gimage         | A display for icons and images                         |
| ggraphics      | A widget containing a graphics device                  |
| gfilebrowser   | A widget to select a file or directory                 |
| gcalendar      | A widget to select a date                              |
| gaction        | A reusable definition of an action                     |
| gmenubar       | Puts a menubar on a top-level window                   |
| gtoolbar       | Adds a toolbar to a top-level window                   |
| gstatusbar     | Adds a status bar to a top-level window                |
| gtooltip       | Add tooltip to widget                                  |
| gseparator     | A widget to display a horizontal or vertical line      |

be found through R's scoping rules, or the components passed through the `h` argument, as below.

```
w <- gwindow("Button example")
b <- gbutton("Click me", cont=w)
id <- addHandlerChanged(b, action=w, handler=function(h,...) {
  btnText <- svalue(h$obj) # or svalue(b)
  svalue(h$obj) <- paste("don't", btnText, "again") # set text
  enabled(h$obj) <- FALSE
  svalue(h$action) <- "Button example is finished" # set title
})
```

prevent this. Whereas, under `tcltk` the parent window will shrink to fit the button. The `size` method can prevent this if it is not desired.

## Actions

In GUI programming an action is a reusable code object that can be shared among buttons, toolbars, and menubars. Common to these three controls are that the user expects some “action” to occur when a value is selected. For example, some save dialog is summoned, or some page is printed. Actions contain enough information to be displayed in several manners. An action would contain some text, an icon, perhaps some keyboard accelerator, and some handler to call when the action is selected. When a particular action is not possible due to the state of the GUI, it should be disabled, so as not to be sensitive to user interaction.

Actions in `gWidgets` are created through the `gaction` constructor. The arguments are `label`, `tooltip`, `icon`, `key.accel` and the standard handler and `action`. The label appears as the text on a button, the menu item or toolbar text, whereas the icon will decorate the same if possible. For some toolkits, the tooltip pops up when the mouse hovers, as may be done through the `tooltip<-` method for `gWidgets` objects.

**methods** The main methods for actions are `svalue <-` to set the label text and `enabled <-` to adjust whether the widget is sensitive to user input. All instances of the action are set through one call. In some toolkits, such as `RGtk2`, actions are bundled together into action groups. This allows one to easily set the sensitivity of related actions at once. In R, one can store like actions in a list, and get similar functionality by using `sapply`.

**buttons** An action can be assigned to a button by setting it as the `action` argument of the `gbutton` constructor, in which case all other arguments for the constructor are ignored.

Otherwise, actions are used as list components which define the toolbar or menubar, as described in the following.

## Toolbars

The document object model, when employed, often relies on the custom that the main window showing the document possibly has a menu bar, a toolbar and a status bar. Subwindows, such as those for dialogs, would not have these decorations to emphasize their role in the GUI.

In `gWidgets`, toolbars (and menubars) are specified ahead of time as a named list. This is similar to how `RGtk2` can use an XML specification to define a user interface, but unlike how menubars and toolbars can be created one item at a time in the toolkits.

For a toolbar, the list has a simple structure. The list has named components each of which either describes a toolbar item or a separator. The toolbar items are specified by `gaction` instances and separators by `gseparator`

instances with no container specified. (Alternatively, these items can be specified through lists as described in the manual page.)

The `gtoolbar` constructor takes as its first argument the list. As toolbars belong to the window, the corresponding `gWidgets` objects use a `gwindow` object as the parent container. (Some of the toolkits relax this.) The argument style can be one of "both", "icons", "text", or "both-horiz" to specify how the toolbar is rendered. Toolbars in `gWidgetstcltk` are not native widgets, so the implementation uses aligned buttons.

### Menubars, popup menus

Menubars and popup menus are specified in a similar manner as toolbars with menu items being defined through `gaction` instances, and visual separators by `gseparator` instances. Menus differ from toolbars, as sub-menus give a nested structure. This structure is specified using a nested list as the component to describe the sub menu. The lists all have named components, in this case the corresponding name is used to label the sub menu item. For menu bars, it is typical that all the top-level components be lists, but for popup menus, this wouldn't necessarily be the case.

The main constructor `gmenu` has its first argument to specify the list, and the container argument to specify the top-level window.

In Mac OS X, toolbars may be drawn along the top of the screen, as is the custom of that OS.

**Methods** The main methods for toolbar and menubar instances are the `svalue` method which will return the list. Whereas, the `svalue <-` method can be used to redefine the menubar or toolbar. Use the `add` method to append to an existing menubar or toolbar, again using a list to specify the new items.

### Example 5.2: Menubar and toolbar example

The following commands create some standard looking actions. The handler `f` is just a stub to be replaced in a real application.

```
f <- function(...) print("stub")          # a stub
aOpen <- gaction("open", icon="open", handler = f)
aQuit <- gaction("quit", icon="quit", handler = f)
aUndo <- gaction("undo", icon="undo", handler = f)
```

A menubar and toolbar are specified through a named list, as is illustrated next. The menubar list, has a nested list specifying a submenu.

```
tl <- list(open = aOpen, quit = aQuit)
ml <- list(File = list(
  open = aOpen,
```



```

        sep = gseparator(),
        quit = aQuit),
    Edit = list(
        undo = aUndo
    ))

```

Menubars and toolbars are added to top-level windows, so their parent containers should be gwindow objects.

```

w <- gwindow("Example of menubars, toolbars")
mb <- gmenu(ml, cont=w)
tb <- gtoolbar(tl, cont=w)
l <- glabel("Test of DOM widgets", cont=w)

```

By disabling a gaction instance, we change the sensitivity of all its realizations. Here this will only affect the menu bar.

```
enabled(aUndo) <- FALSE
```

An “undo” menubar item, often changes its label when a new command is performed, or the previous command is undone. The `svalue <-` method can set the label text. This shows how a new command can be added and how the menu item can be made sensitive to mouse events.

```

svalue(aUndo) <- "undo: command"
enabled(aUndo) <- TRUE

```

Good GUI building principles suggest that one should not replace values in the a menu, rather one should simply disable those that are not being used. This allows the user to more easily become familiar with the possible menu items. However, it may be useful to add to a menu or toolbar. The `add` method can do so. For example, to add a help menu item to our example one could do:

```

hl <- list(help = list(
    help = gaction("manual", handler=f)
))
add(mb, hl)

```

**Popup menus** Popup menus can be created for a right click event through the `add3rdMousePopupmenu` constructor. (Or `control-button-1` for Mac OS X.) This constructor has arguments `obj` to specify a widget, like a button, to initiate the popup, `menulist` to specify the menu and optionally an action argument.

### Example 5.3: Popup menus

```
w <- gwindow("Popup example")
```

```
b <- gbutton("click me or right click me", cont=w,  
            handler=function(h, ...) {  
              cat("You clicked me\n")  
            })  
f <- function(h,...) cat("you right clicked on", h$action, "\n")  
mbList <- list(one = gaction("one", action="one", handler=f),  
              two = gaction("two", action="two", handler=f)  
              )  
add3rdMousePopupmenu(b, mbList)
```

### 5.2 Text widgets

A number of widgets are geared toward the display or entry of text. The `gWidgets` API defines `glabel` for displaying a single-or multiple-line string of static text, `gstatusbar` to place message labels at the foot of a window, `gedit` for a single line of editable text, and `gtext` for multi-line, editable text. For some toolkits, a `ghtml` widget is also defined, but neither `RGtk2` or `tc1tk` have this implemented.

#### Labels

The `glabel` constructor produces a basic label widget. The label's text is specified through the `text` argument. This is a character vector of length 1 or is coerced into one by collapsing the vector with newlines. The `svalue` method will return the text as a single string, and the `svalue <-` method can be used to set the text programatically. The `font <-` method can also be used to set the text markup (Table ??). For some toolkits, the argument `markup` for the constructor takes a logical value indicating if the text is in the native markup language (PANGO for `RGtk2`).

The widget constructor also has the argument `editable`, which when specified as `TRUE` will add a handler to the event that the label is clicked that allows the text to be edited. Although this is popular in some GUIs, say the tab in a spreadsheet, it has not proven to be intuitive to most users, as typically labels are not expected to change. In the `wxWidgets` toolkit labels are constructed by a function named `staticText` to emphasize this permanence.

#### Statusbars

Statusbars are simply labels placed at the bottom of a window to leave informative, but non-disruptive, messages for the user. The `gstatusbar` constructor provides this widget. The argument `text` can be given to set the initial text away from its default of no message. Subsequent changes are made through the `svalue <-` method. As with toolbars and menubars, a top-level window should be specified for the `container` argument.

### Single-line, editable text

The `gedit` constructor produces a widget to display a single line of editable text. The initial text can be set through the `text` argument. If it is desirable to set the width of the widget, the `width` argument allows the specification in terms of number of characters allowed to display without horizontal scrolling. The width of the widget may also be specified in pixel size through the `size` method.

**Methods** The text is returned by the `svalue` method and may be set through the `svalue <-` method. The `svalue` method will return a character vector by default. However, it may be desirable to use this widget to collect numeric values or perhaps some other type of variable. One could write code to coerce the character to the desired type, but it is sometimes convenient to have the return value be a certain non-character type. In this case, the `coerce.with` argument can be used to specify a function of a single argument to call before the value is returned by `svalue`.

Some toolkits allow type-ahead values to be set. These values anticipate what a user wishes to type and offers a means to complete a word. The `[ <-` method allows these values to be specified through a character vector, as in `obj[] <- values`.

**Handlers** The default handler for the `gedit` widget is called when the text area is “activated” through it losing focus or the return key being pressed. The `addHandlerKeystroke` method can assign a handler to be called when a key is released. For the toolkits that support it, the specific key is given in the `key` component of the list `h` (the first component).

#### Example 5.4: Validation

In web programming it is common to have textarea entries be validated prior to their values being submitted. By validating ahead of time, the programmer can avoid the lag created by communicating with the server when the input is not acceptable. However, despite this lag not being the case for the GUIs considered now, it may still be a useful practice to validate the values of a text area when the underlying handlers are expecting a specific type of value.

The `coerce.with` argument can be used to specify a function to coerce values after an action is initiated, but in this example we show how to validate the text widget when it loses focus. The use of a modal dialog is a bit extreme here, a more user-friendly correction is suggested.

```
w <- gwindow("Validation example")
validRegexpr <- "[[:digit:]]{3}-[[:digit:]]{4}"
tbl <- glayout(cont=w)
tbl[1,1] <- "Phone number (XXX-XXXX)"
tbl[1,2] <- (e <- gedit("", cont = tbl))
```

## 5. gWIDGETS: CONTROL WIDGETS

Table 5.2: Possible specifications for setting font properties. Font values of an object are changed with named vectors, as in `font(obj) <- c(weight="bold", size=12, color="red")`

| Attr   | Possible value                 |
|--------|--------------------------------|
| weight | light, normal, bold            |
| style  | normal, oblique, italic        |
| family | normal, sans, serif, monospace |
| size   | a point size, such as 12       |
| color  | a named color                  |

```
tbl[2,2] <- (b <- gbutton("submit", cont = tbl,
                          handler=function(h,...) print("hi")))
## Blur is focus out event
addHandlerBlur(e, handler = function(h,...) {
  curVal <- svalue(h$obj)
  if(length(grep(validRegexpr, curVal)) == 0) {
    focus(h$obj) <- TRUE
    gmessage("not valid")
  }
})
```

### Multi-line, editable text

The `gtext` constructor produces a multi-line text editing widget with scroll-bars to accomodate large amounts of text. The `text` argument is for specifying the initial text. This text can have a font attribute specified through the `font.attr` argument. This argument takes the same values as the `font <-` method. The initial width and height can be set through similarly named arguments, which is useful under `tcltk`.

The `svalue` method retrieves the text stored in the buffer. If the argument `drop=TRUE` is specified, then only the currently selected text will be returned. Text in multiple lines is returned as a single string with “\n” separating the lines.

The contents of the text buffer can be replaced with the `svalue <-` method. To add text to a buffer, the `insert` method is used. The signature is `insert(obj, text)` where `text` is a character vector. New text is added to the end of the buffer. The font for newly added text can be set with the `font.attr` argument. The font for the selected text can be set with the `font <-` method. To clear the text buffer, the `dispose` method is used.

As with, `gedit`, the `addHandlerKeystroke` method is used to set a handler to be called for each keystroke. This is the default handler.

### Example 5.5: A calculator

The following example shows how one might use the widgets just discussed to make a GUI that resembles a calculator. Which may offer familiarity to new R users, although certainly is no replacement for a command line.

The `glayout` container is used to neatly arrange the widgets. This example illustrates how a child widget can span a block of multiple cells by using the appropriate indexing. Furthermore, the `spacing` argument is used to tighten up the appearance. The example also illustrates a useful strategy of storing the widgets using a list for subsequent manipulations.

The following sets up the layout of the display and buttons.

```
buttons <- rbind(c(7:9, "(", ")"),
                c(4:6, "*", "/"),
                c(1:3, "+", "-"))
bList <- list()
w <- gwindow("glayout for a calculator")
g <- ggroup(cont=w, expand=TRUE, horizontal=FALSE)
tbl <- glayout(cont=g, spacing=2)
tbl[1, 1:5, anchor=c(-1,0)] <- # span 5 columns
  (eqnArea <- gedit("", cont=tbl))
tbl[2, 1:5, anchor=c(1,0)] <-
  (outputArea <- glabel("", cont=tbl))
for(i in 3:5) {
  for(j in 1:5) {
    val <- buttons[i-2, j]
    tbl[i,j] <- (bList[[val]] <- gbutton(val, cont=tbl))
  }
}
tbl[6,2] <- (bList[["0"]] <- gbutton("0", cont=tbl))
tbl[6,3] <- (bList[["."]] <- gbutton(".", cont=tbl))
tbl[6,4:5] <- (eqButton <- gbutton("=", cont=tbl))
outputArea <- gtext("", cont = g)
```

This code defines the handler for each button except the equals button and then assigns the handler to each button. This is done efficiently, using the generic `addHandlerChanged`. The handler simply pastes the text for each button into the equation area.

```
addButton <- function(h, ...) {
  curExpr <- svalue(eqnArea)
  newChar <- svalue(h$obj) # the button's value
  svalue(eqnArea) <- paste(curExpr, newChar, sep="")
  svalue(outputLabel) <- "" # clear label
}
out <- sapply(bList, function(i)
  addHandlerChanged(i, handler=addButton))
```

When the equals sign is clicked, the expression is evaluated and if there are no errors, the output is displayed in the label.

```
addHandlerClicked(eqButton, handler = function(h,...) {
  curExpr <- svalue(eqnArea)
  out <- try(capture.output(eval(parse(text=curExpr))))
  if(inherits(out,"try-error")) {
    gmessage("There is an error")
    return()
  }
  svalue(outputArea) <- out
  svalue(eqnArea) <- ""           # restart
})
```

### 5.3 Selection controls

A common task for a GUI control is to select a value or values from a set of numbers or a table of numbers. Many toolkits implement these widgets using a model-view-controller paradigm whereby the control is just one of possibly many views of the data store (the model). This approach isn't taken with `gWidgets`. Rather, each widget has its own data store containing the data for selection, and familiar R methods are used to manipulate this underlying data store. The controls in `gWidgets` that display such data have the methods `[], [] <-`, `length`, `dim`, `names` and `names <-`, as appropriate.

This section discusses several different controls that do basically the same thing, but which exist primarily because they use screen space differently.

#### Checkbox widget

The simplest selection control is the checkbox widget that allows the user to set a state as `TRUE` or `FALSE`. The constructor has an argument `text` to set a label and `checked` to indicate if the widget should initially be checked. The default is `TRUE`.

The `svalue` method returns a logical indicating if the widget is in the checked state. Use `svalue <-` to set the state. The label can be adjusted, if the underlying toolkit allows it, with the `[] <-` method and is returned by the `[]` method.

The default handler would be called on a click event, when the state toggles. If it is desired that the handler be called only in the `TRUE` state, say, one needs to check within the handler for this.

#### Radio button widget

A radio button group allows the user to choose one of a few items. A radio button group object is returned by `gradio`. The items to choose from are specified as a vector of values to the `items` argument. These items may be displayed horizontally or vertically (the default) as specified by the `horizontal`

argument which expects a logical. The `selected` argument specifies the initially selected item, with a default of the first.

The currently selected item is returned by `svalue` as the label text or by the index if the argument `index` is `TRUE`. The item may be set with the `svalue <-` method. Again, the item may be specified by the label or by an index, the latter when the argument `index=TRUE` is specified. The data store is the set of labels so are referenced through the `[]` method, and may be set (if the underlying toolkit allows it) with the `[] <-` method. If `gWidgetstcltk` one can not change the number of radio buttons. For convenience, the `length` method returns the number of labels.

The default handler would be called on a click event.

### A group of checkboxes

The checkbox group widget, produced by the `gcheckboxgroup` constructor, allows the selection of one or more of a set of items. The `items` argument is used to specify the values. The state of whether an item is selected can be set with a logical vector of the same size as the number of items to the `checked` argument, recycling is used. The item layout can be controlled by the `horizontal` argument. The default is a vertical layout (`horizontal=FALSE`).

The state is retrieved as a character vector through the `svalue` method. The `index=TRUE` argument instructs `svalue` to return the indices instead. As a checkboxgroup is like both a checkbox and a radio button group, one can set the selected values two different ways. As with a checkbox, the selected values can be set by specifying a logical vector through the `svalue <-` method. As with radio button groups, the selected values can also be set with a character vector indicating which labels should be selected, or if `index=TRUE` is given, using a numeric index vector.

The labels are returned through the `[]` method and if the underlying toolkit allows it, set through the `[] <-` method. As with `gradio`, the `length` method returns the number of items.

### A combobox

A combobox is used as an alternative to a radio button group when there are too many choices to comfortably fit on the screen. Comboboxes are constructed by `gcombobox`.<sup>2</sup> The possible choices are specified to the argument `items`. This may be a vector of values or a data frame whose first column defines the choices. For toolkits which support icons in the combobox, if the data is specified as a data frame, the second column can be used to signify which stock icon is to be used. By design, a third column can be used to specify a tooltip, but this is not implemented in `RGtk2` and `tcltk`.

---

<sup>2</sup>This was at one time called `gdropplist` in `gWidgets`, as comboboxes appear like drop-down lists.

## 5. gWIDGETS: CONTROL WIDGETS

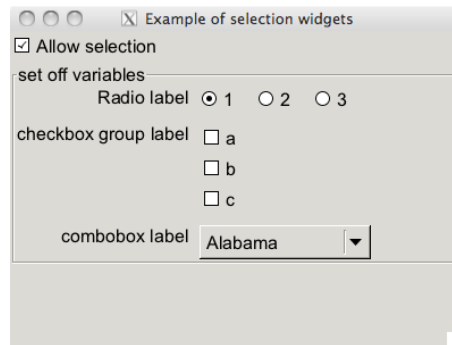


Figure 5.1: A template for a GUI using some of the widgets for selection.

The argument `editable` accepts a logical value indicating if the user can supply their own value by typing into a text entry area. The default is `FALSE`. When editing is possible, the constructor also has the `coerce.with` argument like `gedit`.

The currently selected value is returned through the `svalue` method. If `index` is `TRUE`, the index of the selected item is given if possible. The state can be set through the `svalue <-` method. This is specified by a character unless `index` is `TRUE`, in which case as a numeric index with respect to the underlying items. The `[` method returns the items of the data store, and `[ <-` is used to assign new values to the data store. The `length` method returns the number of items.

The default handler is called when the state of the widget is changed. This is also aliased to `addHandlerClicked`. When `editable` is `TRUE`, then the `addHandlerKeystroke` can be used to set a handler to response to keystroke events.

### Example 5.6: Selection widgets

This example provides template for a possible GUI that would allow a specification of arguments for a function (Figure ??). A checkbox is used to toggle whether the other controls are enabled or not.

```
w <- gwindow("Example of selection widgets", visible=FALSE)
g <- ggroup(horizontal=FALSE, cont=w)
cb <- gcheckbox("Allow selection", cont=g, checked=FALSE,
             handler = function(h, ...) {
               enabled(f) <- svalue(cb)
             })
f <- gframe("set off variables", cont=g)
tbl <- glayout(cont=f)
right <- c(1, 1); left <- c(-1, 1)
```



```
tbl[1,1, anchor=right] <- "Radio label"
tbl[1,2, anchor=left] <- (rb <- gradio(1:3, horizontal=TRUE,
  cont = tbl))
tbl[2,1, anchor=right] <- "checkbox group label"
tbl[2,2, anchor=left] <- (chb <- gcheckboxgroup(letters[1:3],
  horizontal=FALSE, cont = tbl))
tbl[3,1, anchor=right] <- "combobox label"
tbl[3,2, anchor=left] <- (combo <- gcombobox(state.name,
  cont = tbl))
enabled(f) <- FALSE
visible(w) <- TRUE
```

## Display of tabular data

The `gtable` constructor produces a widget that displays data in a tabular form from which the user can select one (or more) rows. The widgets performance under `RGtk2` is much faster and able to handle larger data stores than under `tcltk`, as there is no native table widget in `Tcl/Tk`. Both perform well on moderate-sized data sets (10 or so columns and fewer than 500 rows),

The data is specified through the `items` argument. This may be a data frame, matrix or vector. Vectors are coerced to data frames. The data is presented in a tabular form, with column headers derived from the `names` attribute of the data frame. The `icon.FUN` argument is used to place a stock icon in a left-most column. This argument takes a function of a single argument – the data frame being shown – and should return a character vector of stock icon names, one for each row.

**Filtering** The arguments `filter.column` and `filter.FUN` allow one to specify whether the user can filter, or limit, the display of the values in the data store. If a column number is specified to `filter.column` then a combobox is added to the widget with values taken from the unique values in the specified column. Changing the value of the combobox, restricts the display of the data to just those rows which match that column's values. More advanced filtering can be specified using the `filter.FUN` argument. If this is a function, then it takes arguments (`data_frame`, `filter.by`) where the data frame is the data, and the `filter.by` value is the state of a combobox whose values are specified through the argument `filter.labels`. This function should return a logical vector with length matching the number of rows in the data frame. Only rows corresponding to `TRUE` values will be displayed. If `filter.FUN` is the character string "manual" then the `visible <-` method can be used to control the filtering, again by specifying a logical vector of the proper length. See Example ?? for an application.

**Selection** Users can select a row, not a cell from this widget. The value returned by a selection is controlled by the arguments `chosencol`, which specifies which column value will be returned, as the user can only specify the row; and `multiple` which controls whether the user may select more than one row. The `svalue` method will return the currently selected value. If the argument `index` is specified as `TRUE`, then the selected row index (or indices) will be returned. These refer to the data store, not the visible data when filtering is being used. The argument `drop` specifies if just the chosen column's value is returned (the default) or if specified as `FALSE` the entire row.

**Methods** The underlying data store is referenced by the `[]` method. Indices may be used to access just a portion. Values may be set using the `[] <-` method, but be warned it is not as flexible as assigning to a data frame. The underlying toolkits may not like to change the type of data displayed in a column, so when updating a column do not assume some underlying coercion, as is done with R's data frames. To replace the data store, the `[] <-` can be used via `obj[] <- new_data_frame`. The methods `names` and `names <-` refer to the column headers, and `dim` and `length` the underlying dimensions of the data store.

**Handlers** Selection is done through a single click. The `addHandlerClick` method can be used to assign a handler to those events. The default handler is `addHandlerDoubleClick`, which assigns a handler for a double click event. Also of interest are the `addHandlerRightclick` and `add3rdMousePopupMenu` methods for assigning handlers to right-click events.

The `gtable` widget shows clearly the trade offs between using `gWidgets` and a native toolkit under R. As will be seen in later chapters, setting up a table to display a data frame using the `RGtk2` or `tcltk` packages can involve a fair amount of coding as compared to `gtable`, which makes it very easy. However, there is far less power possible from `gWidgets`. For example, there is no method to adjust the column sizes programatically (although they can be adjusted with the mouse), there is no means to adjust the formatting of the displayed text, or to embed other widgets into the tabular display.

### Example 5.7: Simple filtering

We use the `Cars93` data set from the `MASS` package to show how to set up a display of the data which provides simple filtering based on the type of car, whose value is stored in column 3.

```
require(MASS)
w <- gwindow("gtable example")
tbl1 <- gtable(Cars93, chosencol=1, filter.column=3, cont=w)
```

Adding a handler for the double click event is illustrated below. This handler prints both the manufacturer and the model of the currently selected row when called.

```
addHandlerChanged(tbl, handler=function(h,...) {
  val <- svalue(h$obj, drop=FALSE)
  cat(paste("You selected the", val[,1], val[,2], "\n", sep=" "))
})
```

### Example 5.8: More complex filtering

Even with moderate-sized data sets, the number of rows can be quite large, in which case it is inconvenient to use a GUI for selection unless some means of searching or filtering the data is used. This example uses the possible CRAN sites, to show how a `gedit` instance can be used as a search box to filter the display of data. The `addHandlerKeystroke` method is used so that the search results are updated as the user types.

The `available.packages` function returns a data frame of all available packages. If a CRAN site is not set, the user will be queried to set one.

```
d <- available.packages()      # pick a cran site
```

This basic GUI is barebones, for example it has no text labels to guide the user.

```
w <- gwindow("test of filter")
g <- ggroup(cont=w, horizontal=FALSE)
ed <- gedit("", cont=g)
tbl <- gtable(d, cont=g, filter.FUN="manual", expand=TRUE)
```

The `filter.FUN` provides a means to have a combobox control the display of the table. For this example, we desire more flexibility, so we specify the value of "manual".

Different search criteria may be desired, so it makes sense to separate out this code from the GUI code using a function. The one below uses `grep` to match, so that regular expressions can be used. Another reasonable choice would be to use the first letter of the package. (That filtering could also be specified easily through the `filter.FUN` argument.)

```
ourMatch <- function(curVal, vals) {
  ind <- grep(curVal, vals)          # indices
  vis <- rep(FALSE, length(vals))
  if(length(ind) > 0)
    vis[ind] <- TRUE
  return(vis)                       # logical
}
```

Finally, the `addHandlerKeystroke` method calls its handler everytime a key is released while the focus is in the edit widget. In this case, the handler

finds the matching indices using the `ourMatch` function, converts these into logical format, and then updates the display using the `visible <-` method for `gtable`.

```
id <- addHandlerKeystroke(ed, handler=function(h, ...) {  
  vals <- tbl[, 1, drop=TRUE]  
  curVal <- svalue(h$obj)  
  vis <- ourMatch(curVal, as.character(vals))  
  visible(tbl) <- vis  
})
```

### An editor for tabular data

The `gdf` constructor returns a widget for editing data frames. This is similar to the GUI provided by the `data.entry` function, but uses the underlying toolkit in use by `gWidgets`. Each cell can be edited. Users can click (or double click) in a cell to select it, or use the arrow and tab keys to navigate. For `gWidgetstcltk`, there is no native widget for editing tabular data, so the `tktable`, add-on widget is used (`tktable.sourceforge.net`). A warning will be issued if this is not installed. Again, the widget under `RGtk2` is much faster than that under `tcltk`, but both can load a moderately sized data frame in a reasonable time.

The constructor has argument `items` to specify the data frame to edit and `name` to specify the data frame name, if desired. The column types are important, in particular factors and character types are treated differently, although they may render in a similar manner.

**Methods** Under both `RGtk2` and `tcltk` there are bindings for right-mouse clicks that allow the user to modify the data frame displayed. In addition, there are several methods defined that follow those of a data frame. The `[` and `[ <-` methods can be used to extract and set values from the data frame by index. As with `gtable`, these are not as flexible as for a data frame though. In particular, it may not be possible to change the type of a column, or add new rows or columns through these methods. Using no indices, as in `obj[,]` will return the current data frame, which can be assigned to some value for saving. The current data frame can be completely replaced, when no indices are specified in the replacement call. Additionally, the data frame methods `dimnames`, `dimnames <-`, `names`, `names <-`, and `length` are defined.

The `gdfnotebook` constructor produces a notebook that can hold several data frames at once.

## 5.4 Selection from a sequence of numbers

The previous widgets allowed selection from a user-specified set of values. When these values are a sequence of numbers, the slider control and spin button control are also commonly used. Both of these widgets have arguments to specify the sequence that match those of the `seq` function in R: `from`, `to`, and `by`.

### A slider control

The `gslider` constructor creates a slider that allows the user to select a value from the specified sequence. In `gWidgetstcltk` the sequence must have integer steps. If this is not the case, the spin button control is used instead. In addition to the arguments to specify the sequence, the argument `value` is used to set the initial value of the widget and `horizontal` controls how the slider is drawn, `TRUE` for horizontal, `FALSE` for vertical.

The `svalue` method returns the currently chosen value. The `[ <-` method can be used to update the sequence of values to choose from.

The default handler is called when the slider is changed. Example ?? shows how this can be used to update a graphic.

### A spin button control

The spin button control constructed by `gspinbutton` is similar to `gslider`, but presents the user a different way to select the value. The argument `digits` specifies how many digits are displayed.

#### Example 5.9: Example of sliders and spin buttons

The use of sliders and spin buttons to dynamically adjust a graphic is common in R GUIs targeted towards teaching statistics. Here is an example, similar to the `tkdensity` example of `tcltk`, where the slider controls the bandwidth of a density estimation and the spin button the sample size of a random sample.

```
w <- gwindow("Slider and Spin Button example")
tbl <- glayout(cont=w)
tbl[1,1] <- "sample size"
tbl[1,2] <- (spinner <- gspinbutton(from=10, to=100, by=5,
                                   value=25, cont=tbl))

tbl[2,1] <- "adjusted bandwidth"
tbl[2,2, expand=TRUE] <- (slider <- gslider(from=0.1, to=1,
                                             by=0.01, value=1, cont=tbl))
plotGraph <- function(h,...) {
  x <- rexp(svalue(spinner))
  plot(density(x, adj=svalue(slider)))
}
```

```
QT <- sapply(list(spinner, slider), function(i)
  addHandlerChanged(i, handler=plotGraph))
```

## 5.5 Display of heirarchical data

The `gtree` constructor can be used to display heirarchical structures, such as a file system. This constructor parameterizes the data to be displayed in terms of the node of the tree that is currently selected. The `offspring` argument is assigned a function of two variables, the path in the tree that the node in question is on and any data passed through the optional `offspring.data` argument. This function should return a data frame with each row referring to an offspring for the node and whose first column is a key that characterizes the node of the offspring, unless the argument `chosencol` is used to specify otherwise.

To indicate if a node has offspring, a function can be passed through the `hasOffspring` argument. This function takes the data frame returned by the `offspring` function and should return a logical vector with each value indicating which rows have offspring. If it is more convenient to compute this within the `offspring` function, then when `hasOffspring` is left unspecified and the second column returned by `offspring` is a logical, then that column will be used.

A single click is used to select a row. Multiple selections are possible if the `multiple` argument is given a `TRUE` value.

For some toolkits the `icon.FUN` can be used to specify a stock icon to be displayed next to the first column. This function, like `hasOffspring` has as an argument the data frame returned by `offspring` and should return a character vector with each entry indicating which stock icon is to be shown.

For some toolkits, the column type must be determined prior to rendering. By default, a call to `offspring` with argument `c()` indicating the root node is made. The returned data frame is used to determine the column types. If that is not correct, the argument `col.types` can be used. It should be a data frame with column types matching those returned by `offspring`.

**methods** The `svalue` method returns the currently selected key, or node label. There is no assignment method. The `[]` method returns the path for the currently node. This is what is passed to the `offspring` function. The method `addHandlerDoubleClick` can be used to specify a function to call on a double click event.

### Example 5.10: Using `gtree` to explore a recursive partition

The `party` package implements a recursive partitioning algorithm for tree-based regression and classification models. The package provides an

excellent plot method for the object, but in this example we demonstrate how the `gtree` widget can be used to display the heirarchical nature of the fitted object. First, we fit a model from an example appearing in the package's vignette.

```
require(party)
data("GlaucomaM", package="ipred")      # load data
gt <- ctree(Class ~ ., data=GlaucomaM)  # fit model
```

The `party` object tracks the heirarchical nature through its nodes. This object is a complex structure using lists to store data about the nodes. We define an `offspring` function next that tracks the node by number, as is done in the `party` object; records whether a node has offspring through the terminal component (by passing the `hasOffspring` function); and computes condition on the variable that creates the node. For this example, the trees are all binary trees with 0 or 2 offspring so this data frame has only 0 or 2 rows.

```
offspring <- function(key, offspring.data) {
  if(missing(key) || length(key) < 1) # which node?
    node <- 1
  else
    node <- as.numeric(key[length(key)]) # key is a vector

  if(nodes(gt, node)[[1]]$terminal) # return if terminal
    return(data.frame(node=node, hasOffspring=FALSE,
                      description="terminal"))

  df <- data.frame(node=integer(2), hasOffspring=logical(2),
                  description=character(2),
                  stringsAsFactors=FALSE)

  ## party internals
  children <- c("left", "right")
  ineq <- c("<=", ">")
  varName <- nodes(gt, node)[[1]]$psplit$variableName
  splitPoint <- nodes(gt, node)[[1]]$psplit$splitpoint

  for(i in 1:2) {
    df[i,1] <- nodes(gt, node)[[1]][[children[i]]][[1]]
    df[i,2] <- !nodes(gt, df[i,1])[1]$terminal
    df[i,3] <- paste(varName, splitPoint, sep=ineq[i])
  }
  df # returns a data frame
}
```

We make a simple GUI to show the widget (Figure ??)

```
w <- gwindow("Example of gtree")
g <- ggroup(cont=w, horizontal=FALSE)
```

## 5. gWIDGETS: CONTROL WIDGETS

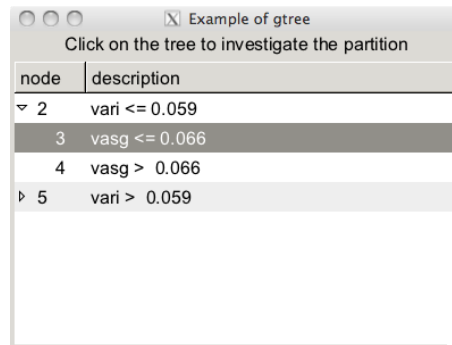


Figure 5.2: GUI to explore return value of a model fit by the party package.

```
l <- glabel("Click on the tree to investigate the partition",
            cont=g)
tr <- gtree(offspring, cont=g, expand=TRUE)
```

A single click is used to expand the tree, here we create a binding to a double click event to create a basic graphic. The vignette shows how to make more complicated – and meaningful – graphics for this model fit.

```
addHandlerDoubleClick(tr, handler=function(h,...) {
  node <- as.numeric(svalue(h$obj))
  if(nodes(gt, node)[[1]]$terminal) { # if terminal plot
    weights <- as.logical(nodes(gt,node)[[1]]$weights)
    plot(response(gt)[weights, ])
  })
})
```

### 5.6 Selecting from the file system

The `gfile` dialog allows one to select a file or directory from the file system. This is a modal dialog, which returns the name of the selected file or directory. The `gfilebrowse` constructor creates a widget that has a button that allows the user to initiate this selection.

The selection type is specified by the `type` argument with values of `open`, to select an existing file; `save` to select a file to write to; and `selectdir` to select a directory. For `RGtk2`, the `filter` argument can be used to narrow the listed files. The dialog returns the path of the file, or `NA` if the dialog was canceled. One can also specify a handler to the constructor to call on the file or directory name. The component `file` of the first argument to the handler contains the file name.

```
if(!is.na(tmp <- gfile()))
  source(tmp)
## or
```



```
gfile(handler=function(h,...) {  
  if(!is.na(h$file))  
    source(h$file)  
})
```

### Selecting a date

The `gcalendar` constructor returns a widget that can be used to select a date if the underlying toolkit supports such a widget or a text edit box to allow the user to enter a date. The argument `text` argument can be used to specify the initial text. The `format` is used to specify the format of the date.

The methods for the widget inherit from `gedit`. In particular, the `svalue` method returns the text in the text box as a character vector formatted by the value specified by the `format` argument. To return a value of a different class, pass a function, such as `as.Date` to the `coerce.with` argument.

## 5.7 Display of graphics

### Displaying icons and images stored in files

Graphics files can be displayed by the `gimage` widget. (Not all file types may be displayed by each toolkit, in particular `gWidgetstcltk` can only display gif, ppm, and xbm files.) The file to display is specified through the `filename` argument of the constructor. This value is combined with that of the `dirname` argument to specify the file path.

The `gWidgets` package provides a few stock icons, that simplify the above. Stock icons, can be specified by using their name for the `filename` argument and the character string "stock" for the `dirname` argument. A list of the defined stock icons is returned by the function `getStockIcons`. The names attribute defines the valid stock icon names. For `gWidgetsRGtk2`, the size of a stock icon can be adjusted through the `size` argument, with a value from "menu", "small\_toolbar", "large\_toolbar", "button", or "dialog".

The `svalue <-` method can be used to change the graphics file. In this case, a full path name is specified, or the stock icon name.

More stock icon names may be added through the function `addStockIcons`. This function requires a vector of stock icon names and a vector of corresponding file paths, and is illustrated in the example.

The default handler is called on a click event.

#### Example 5.11: Adding and using stock icons

This example shows how to add to the available stock icons and use `gimage` to display them. It creates a table to select a color from, as an alternative to a more complicated color chooser dialog. Under `gWidgetstcltk` the image

## 5. gWIDGETS: CONTROL WIDGETS

---

files would need to be converted to gif format, as png format is not a natively supported image type.

We begin by defining 16 arbitrary colors.

```
someColors <- c("black", "red", "blue", "brown",  
               "green", "yellow", "purple",  
               paste("grey", seq(10,90,by=10), sep=""))
```

This is the function that is used to create an icon file. We use some low-level grid functions to draw the image to a png file.

```
require(grid)  
iconDir <- tempdir(); iconSize <- 16;  
makeColorIcon <- function(i) {  
  filename <- paste(iconDir, "/color-", i, ".png",  
                    sep="", collapse="")  
  png(file=filename, width=iconSize, height=iconSize)  
  grid.newpage()  
  grid.draw(rectGrob(gp=gpar(fill=i)))  
  dev.off()  
  return(filename)  
}
```

To add icons, we need to define the stock names and the file paths for addStockIcons.

```
icons <- sapply(someColors, makeColorIcon)  
iconNames <- paste("color-", someColors, sep="")  
QT <- addStockIcons(iconNames, icons)
```

We use a table layout to show the 16 colors. As an illustration of assigning a handler for a click event, we assign one that returns the corresponding stock icon name.

```
w <- gwindow("Icon example")  
f <- function(h,...) print(h$action)  
tbl <- glayout(cont = w, spacing=0)  
for(i in 1:4) {  
  for(j in 1:4) {  
    ind <- (i - 1) * 4 + j  
    tbl[i,j] <- gimage(icons[(i-1)*4 + j], handler=f,  
                       action=iconNames[ind], cont=tbl)  
  }  
}
```

### A graphics device

Some toolkits support an embeddable graphics device (RGtk2 through cairoDevice). In which case, the ggraphics constructor produces a widget that can be

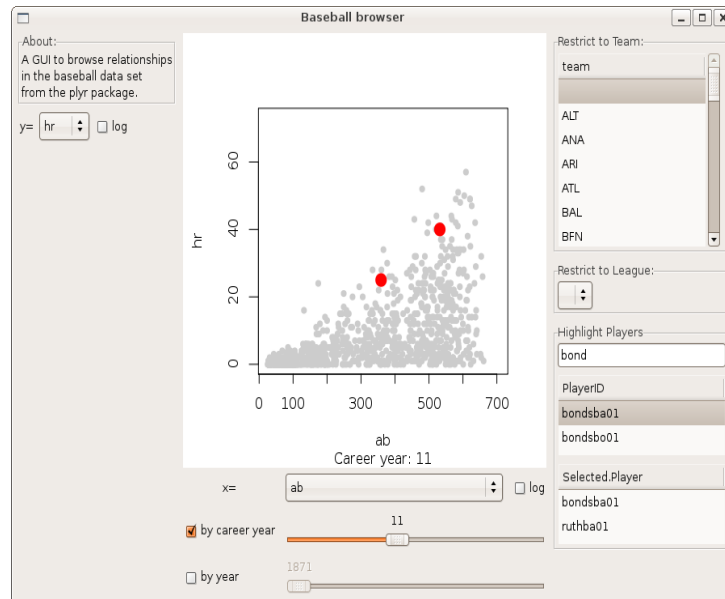


Figure 5.3: A RGtk2 GUI for exploring the `baseball` data set of the `plyr` package. One can subset by year or career year through the slider widgets.

added to a container. The arguments `width`, `height`, `dpi`, `ps` are similar to other graphics devices.

The `ggraphicsnotebook` creates a notebook that allows the user to easily navigate multiple graphics devices.

### Example 5.12: A GUI to explore a data set

The hash **package** provided by:

/ \_ \_ \_ \ | ' \_ \_ \_ \ / \_ \_ \_ \ ' \_ \_ \_ \ / \_ \_ \_ \ | \_ \_ \_ \ | \_ \_ \_ \  
 | ( \_ ) | | \_ ) | \_ \_ / | | | ( \_ | | ( \_ | | | ( \_ |  
 \ \_ \_ \_ / | \_ \_ \_ / \ \_ \_ \_ | \_ | \_ | \_ \_ \_ , | \ \_ \_ \_ , | \ \_ \_ \_ , |  
 | \_ | <http://www.opendatagroup.com>

This example creates a GUI to explore the baseball data set of the `plyr` package. The baseball data set contains information by year for players who had 15-year or longer careers. Several interesting things can be seen by looking at specific players, such as Babe Ruth (coded `ruthba01`) or Barry Bonds (`bondsba01`). Before beginning, we follow an example from the `plyr` package to create a new variable to hold the career year of a player.

```
data(baseball, package="plyr")
```

## 5. gWIDGETS: CONTROL WIDGETS

---

```
calc <- function(df)
  transform(df,
            cyear = year - min(year),
            cpercent = (year - min(year))/(max(year) - min(year)))
b <- ddply(baseball, .(id), calc)
b <- subset(b, ab >= 25)
nVars <- names(b)[-c(1:5,23:24)]      # numeric variables
```

This example uses the hash package to store our data and an environment to store our widgets.

```
require(hash)
dat <- hash()
e <- new.env()
```

The following function transfers values from the GUI to our data store, dat, returning TRUE if all goes well. The widgets are all stored in an environment, e below, using names which are again used as keys to the hash. We also define a function plotIt to produce a graphic based on the current state of the data store, but don't reproduce it here.

```
transferData <- function() {
  out <- try(sapply(e, svalue, drop=TRUE), silent=TRUE)
  if(inherits(out, "try-error"))
    return(FALSE)
  dat[[names(out)]] <- out          # hash keys
  dat$id <- e$id[]                  # not svalue
  return(TRUE)                     # works!
}
```

We now create a GUI so that the user can select which graphic to make. Our GUI will have a main plot window to show a scatter plot, and controls to adjust the variables that are plotted, and to filter that values plotted.

Our layout will use box containers to split the top-level window into three panes. The middle one holds the graphic, so we set it to expand when the window is resized.

```
w <- gwindow("Baseball browser", visible=FALSE)
g <- ggroup(cont=w, horizontal=TRUE)
lp <- ggroup(cont=g, horizontal=FALSE)
cp <- ggroup(cont=g, horizontal=FALSE, expand=TRUE)
rp <- ggroup(cont=g, horizontal=FALSE, spacing=10)
```

The left panel holds a short description and a combobox to select the y-variable plotted.

```
f <- gframe("About:", cont=lp)
l <- glabel(paste("A GUI to browse relationships",
                  "in the baseball data set",
                  "from the plyr package."),
```

```

      sep="\n"),
      cont=f)
g1 <- ggroup(cont=lp)
l <- glabel("y=", cont=g1)
e$y <- gcombobox(nVars, selected=4, cont=g1)
e$ylog <- gcheckbox("log", checked=FALSE, cont=g1)

```

The center panel holds the `ggraphics` object, along with controls to select the  $x$  variable. As well, we add controls to filter out the display by either the year a player played and/or their career year. A `gtable` instance is used for layout.

```

gg <- ggraphics(cont=cp)
tbl <- glayout(cont=cp)
tbl[1,1] <- "x="
tbl[1,2, expand=TRUE] <- (e$x <- gcombobox(nVars, selected=2,
      cont=tbl))
tbl[1,3] <- (e$xlog <- gcheckbox("log", checked=FALSE,
      cont=tbl))
##
tbl[2,1] <- (e$doCareerYear <- gcheckbox("by career year",
      checked=TRUE, cont=tbl))
tbl[2,2:3, expand=TRUE] <- (e$cyyear <-
      gslider(min(b$cyyear), max(b$cyyear), by=1, cont=tbl))
enabled(e$cyyear) <- TRUE
##
tbl[3,1] <- (e$doYear <- gcheckbox("by year",
      checked=FALSE, cont=tbl))
tbl[3,2:3, expand=TRUE] <- (e$year <-
      gslider(min(b$year), max(b$year), by=1, cont=tbl))
enabled(e$year) <- FALSE

```

The right panel includes a few means to filter the display of values. We use a simple `gtable` widget to allow the user to restrict the display to one or more teams. A combobox allows the user to restrict to one of the historic leagues. To allow certain players to stand out, a compound widget is made using a `gedit` object to filter values, a `gtable` object to show all possible IDs, and a `gtable` object to show the selected IDs to highlight. Frame are used to visually combine these controls.

```

rpWidth <- 200
f <- gframe("Restrict to Team:", cont = rp)
teams <- data.frame(team=c("", sort(unique(b$team))),
      stringsAsFactors=FALSE)
e$team <- gtable(teams, cont=f, multiple=TRUE, width=rpWidth)
size(e$team) <- c(200,200)
svalue(e$team, index=TRUE) <- 1
##
f <- gframe("Restrict to League:", cont=rp)

```

## 5. gWIDGETS: CONTROL WIDGETS

---

```
leagues <- names(table(b$lg))[-1]      # drop ""
e$lg <- gcombobox(c("", leagues), cont=f)
##
f <- gframe("Highlight Players", horizontal=FALSE, cont=rp)
searchPlayer <- gedit("", cont=f)
listPlayers <- gtable(data.frame("PlayerID"=unique(b$id),
                                stringsAsFactors=FALSE),
                      filter.FUN="manual", cont=f)
e$id <- gtable(data.frame("Selected Player"=character(0),
                          stringsAsFactors=FALSE), cont=f)
```

We define several handlers to make the GUI responsive to user output. Rather than write an `updateUI` function to update the GUI at periodic intervals, we use an event-driven model. These first two handlers, simply toggle whether the user can control the display by year or career year.

```
addHandlerChanged(e$doYear, handler = function(h,...) {
  val <- ifelse(svalue(e$doYear), TRUE, FALSE)
  enabled(e$year) <- val
})
addHandlerChanged(e$doCareerYear, handler = function(h,...) {
  val <- ifelse(svalue(e$doCareerYear), TRUE, FALSE)
  enabled(e$year) <- val
})
```

This next handler updates the graphic when any of several widgets is changed.

```
QT <- sapply(list(e$x, e$xlog, e$y, e$ylog, e$year, e$year,
                 e$doYear, e$doCareerYear, e$lg), function(i)
  addHandlerChanged(i, handler=function(h, ...)
    transferData() && plotIt()))
```

For `gtable` objects, it is more natural here to bind to a single mouse click, rather than the default double click.

```
QT <- sapply(list(e$team, e$id), function(i)
  addHandlerClicked(i, handler=function(h, ...)
    transferData() && plotIt()))
```

These handlers are used to select the IDs to highlight.

```
addHandlerKeystroke(searchPlayer, handler=function(h, ...) {
  cur <- svalue(h$obj)
  ind <- grep(cur, unique(b$id))
  tmp <- rep(FALSE, length(unique(b$id)))
  if(length(ind) > 0) {
    tmp[ind] <- TRUE
    visible(listPlayers) <- tmp
  } else if(length(grep("^\\s$", cur))) {
    visible(listPlayers) <- !tmp
  }
```

```

    } else {
      visible(listPlayers) <- tmp
    }
  })
  addHandlerChanged(listPlayers, handler=function(h, ...) {
    val <- svalue(h$obj)
    e$id[] <- sort(c(val, e$id[]))
  })
  addHandlerChanged(e$id, handler=function(h, ...) {
    val <- svalue(h$obj)
    cur <- e$id[]
    e$id[] <- setdiff(cur, val)
  })

```

Finally, we implement functionality similar to the `locator` function for the graphic. This handler labels the point nearest to a mouse click in the plot area.

```

distance <- function(x,y) {
  ds <- apply(y, 1, function(i) sum((x-i)^2))
  ds[is.na(ds)] <- max(ds, na.rm=TRUE)
  ds
}
addHandlerClicked(gg, function(h,...) {
  x <- c(h$x, h$y)
  ds <- distance(x, curdf[,2:3])
  ind <- which(ds == min(ds))
  ids <- curdf[ind, 1]
  points(y[ind,1], y[ind,2], cex=2, pch=16, col="blue")
  text(y[ind,1], y[ind,2], label=ids, adj=c(-.25,0))
})

```

To end, we show the GUI and initialize the plot.

```

visible(w) <- TRUE
QT <- transferData() && plotIt()

```

## 5.8 Dialogs

The `gWidgets` package provides a few constructors to quickly make some basic dialogs for showing messages or gathering information. Mostly these are modal dialogs that take control of the eventloop, not allowing any other part of the GUI to be active for programmatic interaction. As such, the constructors do not return an object to manipulate through its methods, but rather the value of the dialog specified by the user. Hence, they are used differently than other constructors. For example, the `gfile` dialog, previously described, is a modal dialog that pops up a means to select a file returning the selected file path or `NA`.

Table 5.3: Table of constructors for basic dialogs in gWidgets

| Constructor               | Description                                      |
|---------------------------|--|
| <code>gfile</code>        | File and directory selection dialog              |
| <code>gmessage</code>     | Dialog to show a message                         |
| <code>galert</code>       | Unobtrusive (non-modal) dialog to show a message |
| <code>gconfirm</code>     | Confirmation dialog                              |
| <code>ginput</code>       | Dialog allowing user input                       |
| <code>gbasicdialog</code> | Flexible modal dialog                            |

The dialogs pop up a window with a common appearance. The constructors have arguments `message` for a message; `title` for the window title; and `icon` to specify an icon, whose value is one of "info", "warning", "error", or "question". Buttons will appear at the bottom of the dialog, and are determined by choice of the constructor. The parent argument will place the dialog near the `gWidgets` instance specified. Otherwise, placement will be controlled by the window manager.

The dialogs, except for `galert`, have the standard `handler` and `action` arguments, for calling a handler, but typically it is easier to use the return value when programming.

**A message dialog** The simplest dialog is produced by `gmessage`, which is used to display a message. The user has a cancel button to dismiss the dialog.

**An alert dialog** The `galert` dialog is similar to `gmessage` only it is meant to be less obtrusive, so it is non-modal. It does not take the focus and vanishes after a time delay.

**A confirmation dialog** The constructor `gconfirm` produces a dialog that allows the user to confirm the message. This dialog returns `TRUE` or `FALSE` depending on the user's selection.

**An input dialog** The `ginput` constructor produces a dialog which allows the user to input a single line of text. If the user confirms the dialog, the value of the string is returned, otherwise if the user cancels the dialog through the button a value of `NA` is returned.

**A basic dialog** The `gbasicdialog` constructor allows one to place an arbitrary widget within a modal window with OK and Cancel. The handler, if specified, will be called if the user clicks the OK button. This allows users to create their own modal dialogs.



As with the others, the argument `title` is used to specify the window title, but there is no icon or message arguments, as there is no standard appearance. Rather, the widget argument specifies a widget to pack into the dialog. This can be a simple control, or a container containing other widgets.

As with `gconfirm`, this widget returns `TRUE` or `FALSE` depending on the user's selection. To do something more complicated than `gconfirm`, a handler should be specified at construction. If the user selects OK, the handler, if specified, is called before the value `TRUE` is returned.

This dialog is called a bit awkwardly, to allow it to work when controls need a parent container specified at construction time (e.g., `tcltk`). The construction is in three stages: an initial call to `gbasicdialog` to return a container which is used as the parent container for a child component; a construction of the dialog; then a call to the `visible` method on the dialog with `set=TRUE` value (not though `visible(obj) <- TRUE`).

#### Example 5.13: Modal dialogs

The basic input dialog requires just the first argument.

```
ginput("Message goes here", title="example dialog")
```

Here we use the question icon for a confirmation dialog, as the message is a question.

```
ret <- gconfirm("Really delete file?", icon="question")
```

This illustrates how to use the return value.

```
ret <- ginput("Enter your name", icon="info")
if(!is.na(ret))
  cat("Hello",ret,"\n")
```

The `gbasicdialog` constructor can be used to make modal dialogs. This example will force the user to select a color before proceeding with anything else.

```
## create a parent container
dlg <- gbasicdialog("Pick a color", handler =
  function(h,...) print(svalue(widget)))
## create the dialog using dlg as the parent container
widget <- gtable(colors(), cont = dlg)
## show the modal dialog (not visible(dlg) <- TRUE)
visible(dlg, set=TRUE)
```

## 5.9 gWidgets: Compound widgets

The `gWidgets` package provides some R specific widgets for producing GUIs. Table ?? lists them.

Table 5.4: Table of constructors for compound widgets in gWidgets

| Constructor               | Description   |
|---------------------------|---|
| constructorgvarbrowser    | GUI for browsing variables in the workspace                                   |
| constructorghelp          | GUI for a help page   |
| constructorghelpbrowser   | A help browser  |
| gcommandline              | Command line widget   |
| constructorgformlayout    | Uses list to specify layout of a GUI  |
| constructorggenericwidget | Creates a GUI for a function based on its formal arguments or a defining list |

### Workspace browser

A workspace browser is constructed by `gvarbrowser`, providing a means to browse and select the objects in the current global environment. The quality of the implementation varies depending on the toolkit. The default handler object calls `do.call` on the object for the function specified through the `action` argument. The default is to print a summary of the object. This handler is called on a double click. A single click is used for selection. The name of the currently selected value is returned by the `svalue` method.

### Help browser

The `ghelp` constructor produces a widget for showing help pages using a notebook container. This widget does not use the html help pages or the chm help pages, so it may not work for all operating systems. (For Windows, the help browser of the GUI is much better anyways.) To add a help page, the `add` method is used, where the `value` argument describes the desired page. This can be a character string containing the topic, a character string of the form `package::topic` to specify the package, or a list with named components `package` and `topic`. The `dispose` method of notebooks can be used to remove the current tab.

The `ghelpbrowser` constructor produces a stand-alone GUI for displaying help pages, running examples from the help pages or opening vignettes provided by the package. This GUI provides its own top-level window and does not return a value for which methods are defined.

### Command line widget

A simple command line widget is created by the `gcommandline` constructor. This is not meant as a replacement for R's typical commandlines, but is there for lightweight usage. A text box allows users to type in R commands. The programmer may issue commands to be evaluated and displayed through the `svalue <-` method. The value assigned is a character string holding

the commands. If there is a `names` attribute, the results will be assigned to a variable in the global workspace with that name. The `svalue` and `[]` methods return the command history.

## Simplifying creation of dialogs

The `gWidgets` package has two means to simplify the creation of GUIs. The `gformlayout` constructor takes a list defining a layout and produces a GUI, the `ggenericwidget` constructor can take a function name and produce a GUI based on the formal arguments of the function. This too uses a list, that can be modified by the user before the GUI is constructed.

## Laying out a form

The `gformlayout` constructor takes a list defining a layout and creates the specified widgets. The design borrows from the `ext.js` javascript libraries for web programming, where a similar function can be used to specify the layout of web forms. Several toolkits have a means to specify a layout using XML (eg. `glade`), this implementation uses a list, assuming this is more familiar to the R user. By defining the layout ahead of time, pieces of the layout can be recycled for other layouts.

To define the layout, each component is specified using a list with named components. The component type specifies what component to be created, as a string. This can be the name of a container constructor, a widget constructor or the special value `"fieldset"`. Field sets are used to group a set of common controls. If the component name is specified, then the component that is created will be stored in the list returned by the `[]` method.

The `label` component can be specified to add a descriptive label to the layout. When specified, the component `label.pos` can be specified with value `"top"` to have the label on top of the widget, or `"side"` to place the label on the side (the default positioning). The `label.font` component can be used to specify the label's font properties using a label's `font <-` method.

If the type is a container or fieldset, then the `children` component is a list whose components specify the children as above. Except for fieldsets, these children can contain other containers or components. Fieldsets only allow components as children.

Whether a widget is enabled or not can be controlled by specifying values for `depends.on`, `depends.FUN`, and `depends.signal`. If the component `depends.on` specifies the name of a previous component, then the function `depends.FUN` will be consulted when the signal specified by `depends.signal` is emitted. This uses the `addHandlerXXX` names with a default value of `addHandlerChanged`. The `depends.FUN` function has a single argument consisting of the value returned by `svalue` when called on the widget specified

## 5. gWIDGETS: CONTROL WIDGETS

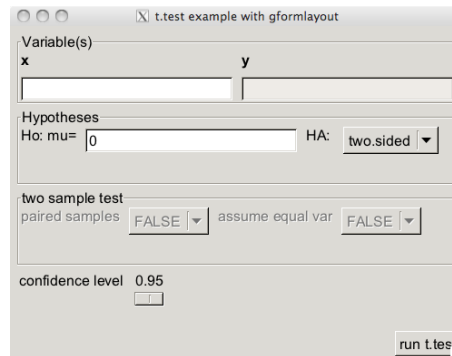


Figure 5.4: A dialog to collect arguments for a  $t$ -test made with `gformlayout`.

through `depends.on`. This function should return a logical indicating if the widget is enabled or not.

**Methods** The constructor returns an object with a few methods. The `[` method will return a list with components being the widgets that were named in the defining list. The `svalue` method simply applies the `svalue` method for each component of the list returned by the `[` method. The `names` method returns the names of the widgets in the list.

### Example 5.14: The `gformlayout` constructor

This example uses `gformlayout` to make a GUI for a  $t$ -test (Figure ??). The first task is to define the list that will set up the GUI. We do this in pieces. This first piece will define the part of the GUI where the null and alternative hypotheses are specified. The null is specified as a numeric value with a default of 0. We use the `gedit` widget which by default will return a character value, so the `coerce.with=as.numeric` argument is specified. For the alternative, this requires a selection for just 3 possibilities, so a combo box is employed.

```
hypotheses <-  
  list(type = "fieldset",  
        label = "Hypotheses",  
        columns = 2,  
        children = list(  
          list(type="gedit",  
                name="mu", label="Ho: mu=",  
                text="0", coerce.with=as.numeric),  
  
          list(type="gcombobox",  
                name="alternative", label="HA: ",  
                items=c("two.sided", "less", "greater"))
```

```
)))
```

Basic usage of the `t.test` function allows for an `x`, or `x` and `y` variable to be specified. Here we disable the `y` variable until the `x` one has been entered. The `addHandlerChanged` method is called when the enter key is pressed after the `x` value is specified.

```
variables <-  
  list(type="fieldset",  
        columns = 2,  
        label = "Variable(s)",  
        label.pos = "top",  
        label.font = c(weight="bold"),  
        children = list(  
          list(type = "gedit",  
                name = "x", label = "x",  
                text = ""),  
          list(type = "gedit",  
                name = "y", label = "y",  
                text = "",  
                depends.on = "x",  
                depends.FUN = function(value) nchar(value) > 0,  
                depends.signal = "addHandlerChanged"  
          )))
```

If a `y` value is specified, then the two-sample options make sense. This enables them dependent on that happening.

```
two.sample <-  
  list(type = "fieldset",  
        label = "two sample test",  
        columns = 2,  
        depends.on = "y",  
        depends.FUN = function(value) nchar(value) > 0,  
        depends.signal = "addHandlerChanged",  
        children = list(  
          list(type = "gcombobox",  
                name = "paired", label = "paired samples",  
                items = c(FALSE, TRUE)  
          ),  
          list(type = "gcombobox",  
                name = "var.equal", label = "assume equal var",  
                items = c(FALSE, TRUE)  
          )))
```

The confidence interval specification is specified using a slider for variety.

```
conf.level <-  
  list(type = "fieldset",  
        columns = 1,
```

## 5. gWIDGETS: CONTROL WIDGETS

---

```
children = list(
  list(type = "gslider",
        name = "conf.level", label = "confidence level",
        from=0.5, to=1.0, by=.01, value=0.95
      )))
```

Finally, the constituent pieces are placed inside a box container.

```
tTest <- list(type = "gggroup",
             horizontal = FALSE,
             children = list(
               variables,
               hypotheses,
               two.sample,
               conf.level
             )))
```

The layout of the GUI is primarily done by the `gformlayout` call. The following just places the values in a top-level window and adds a button to initiate the call to `t.test`.

```
w <- gwindow("t.test example with gformlayout")
g <- gggroup(horizontal=FALSE, cont=w)
fl <- gformlayout(tTest, cont=g, expand=TRUE)
```

```
DEBUG$weight
weight
"bold"

[1] " bold"
DEBUG$weight
weight
"bold"

[1] " bold"
```

```
bg <- gggroup(cont=g)
addSpring(bg)
b <- gbutton("Run t.test", cont=bg)
```

The handler is very simple, as the names chosen match the argument names of `t.test`, so the list returned by the `svalue` method can be used with `do.call`. The only needed adjustment is for the one-sample case.

```
addHandlerChanged(b, function(h, ...) {
  out <- svalue(fl)
  out$x <- svalue(out$x) # turns text string into numbers
  if(out$y == "") {
    out$y <- out$paired <- NULL
  } else {
```

```
    out$y <- svalue(out$y)
  }
  print(do.call("t.test", out))
})
```

### Automatically creating a GUI

The `ggenericwidget` constructor can create a basic GUI for a function using the function's formal arguments as a guide for the proper widget to use to collect values for an argument of the function. The `fgui` package provides a similar function using just the `tcltk` package, only it improves `ggenericwidget` by parsing the function's help page.

The implementation actually has two stages, the first creates a list specifying the layout of the GUI and the second a call to layout the GUI. This list is different from that used by `gformlayout`. It does not provide as much flexibility and is described in the help page for `ggenericwidget`. This list can be edited if desired and then used directly.

The formal arguments of an S3 method may be different from those of its generic. For instance, those for the `t.test` generic are much different (and less useful for this purpose) than the `t.test.default` method for numeric values for `x`. Knowing this, a useful GUI can be quickly created for the `t.test` with the commands:

```
w <- gwindow("t.test through gggenericwidget")
f <- stats::t.test.default;
widget <- gggenericwidget("f", cont=w)
```

```
DEBUG$style
[1] "bold"

[1] ""
DEBUG$style
[1] "bold"

$size
[1] 10

[1] " 10"
DEBUG$style
[1] "bold"

[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 GdkgPixbuf libraries for image manipulation; libglade for designing GUI layouts from an XML description; ATK for the accessibility 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+.

### 6.1 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`.

## 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 hierarchy by calling `class` on a `gtkWindow` instance: <sup>1</sup>

```
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

---

<sup>1</sup>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 function 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 more 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

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]])}
```

**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 callback. 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()
```

## 6.2 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

`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.

### 7.1 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 `SetPosition` 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 propagated 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")
```

```
w$add(1)
```

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 shortcut for list extraction from the return value of the `GetChildren` method.

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

## 7.2 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 analogous 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
```

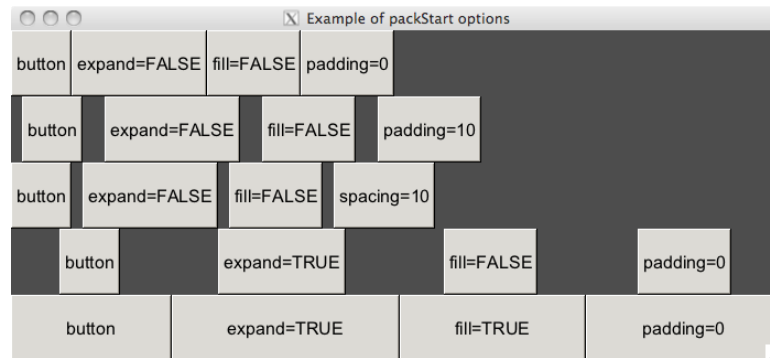


Figure 7.1: Examples of packing widgets into a box container. The top row shows no padding, whereas the 2nd and 3rd illustrate the difference between padding (an amount around each child) and spacing (an amount between each child). The last two rows show the effect of `fill` when `expand=TRUE`. Illustration follows one in original 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 ?? provides an example and Figure ?? 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

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 ?? 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.

### 7.3 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 7.1: 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") )
```

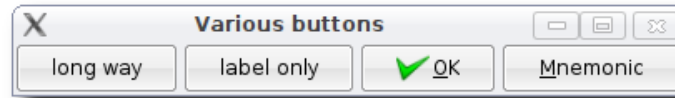


Figure 7.2: Various buttons

```
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 ??.

#### Example 7.2: Callback example for `gtkButton`

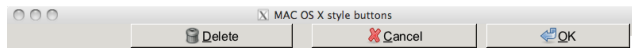


Figure 7.3: Example using stock buttons with extra spacing added between the delete and cancel buttons.

```
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)        # propagate
                    })
ID <- gSignalConnect(b,"clicked",           # click or keyboard
                    f = function(w,...) {
                        print("clicked")
                    })
```

### Example 7.3: 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 separation between it and the `cancel` button. This is given by adding 12 pixels of padding when this button is packed in, giving 24 in total. The blank label is there to fill out space if the parent container expands.

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

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

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

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

### 7.4 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, `<u>` 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 7.4: Label formatting

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



```

g <- gtkVBox(spacing=2); g$setBorderWidth(5); w$add(g)
string = "the quick brown fox jumped over the lazy dog"
## wrap by setting number of characters
basicLabel <- gtkLabel(string)
basicLabel$setLineWrap(TRUE);
basicLabel$setWidthChars(35) # 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 ?? 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 7.5: 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)
```

### 7.5 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 7.6: 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. Unlike 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()
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)
  }
}
```

## 7.6 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 7.7: `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"])
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 7.8: 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)
  }
}
```

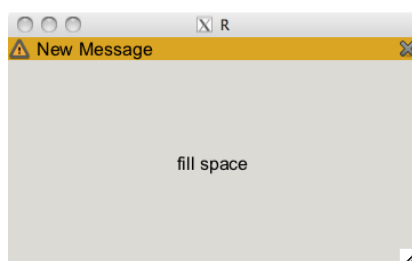


Figure 7.4: The alert panel showing a message.

```

    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[grepl("^new", nms)])

```

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

### Example 7.9: 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,
                          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)
```

```
## 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`.

## 7.7 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.

**Example 7.10: 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 7.11: 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)
```



## 7.8 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 `gtkCheckButtonNewWithMnemonic`.

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.

## 7.9 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 7.12: Radio group construction

Creating a new radio button group follows this pattern:

## 7. RGtk2: BASIC COMPONENTS

---

```
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 converts a list in R to the appropriate list for GTK+. However, you may wish 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 7.13: 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)
```

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

## 7.10 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 be 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 7.14: 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
```

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

## Sliders

The slider widget and spinbutton widget allow selection from a regularly 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 vertically. The decision is made through the choice of constructor: `gtkHScale` or `gtkVScale`. For these widgets, the adjustment can be specified – if desired, 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 `setValue` 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 widget. 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 7.15: A slider controlling histogram bin selection

A simple mechanism to make a graph interactive, is to have the graph redraw whenever 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 7.16: 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 propagate 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 adjustment 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)
```

### 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.

### 7.11 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.

## 7.12 Divided containers

The `gtkHPaned` and `gtkVPaned` create containers with a “gutter” to allocate the space between its two children. An example is presented in Example ???. 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.

## 7.13 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`, indicating 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 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 7.17: 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$insertPageWithCloseButton(gtkButton("hello"), label.text="page 1")
QT <- nb$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 adjust the visible portion of the component. 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 7.18: 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 locate 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$setTitle("Scrolled window example")
w$setSizeRequest(-1, 300)
w$add(sw)
w$show()
```

### 7.14 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 may 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 `ypadding` 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 7.19: 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)
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 7.20: 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")

```

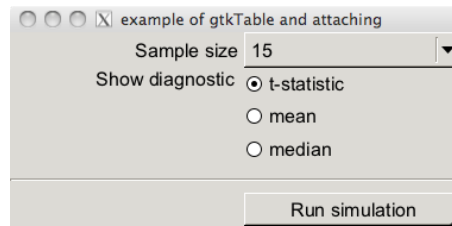


Figure 7.5: A basic dialog using a gtkTable container for layout.

```
w1 <- gtkComboBoxNewText()
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` property set, so that it stays at the top of the cell, not the middle.

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

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

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

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

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

## 7.15 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.

### 8.1 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 overridden for parts of the text buffer through the use of text tags. The left and right margins are adjusted through the `left-margin` and `right-margin` properties.

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 `setModified`, 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 8.1: 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()
```



```
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 modified 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 specified 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 `backwardChars`. As well, there are methods to move to the end or be-

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 ?? 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` iterators. 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 8.2: 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$forwardWordEnd()
  val <- obj$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 of any added text.

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.

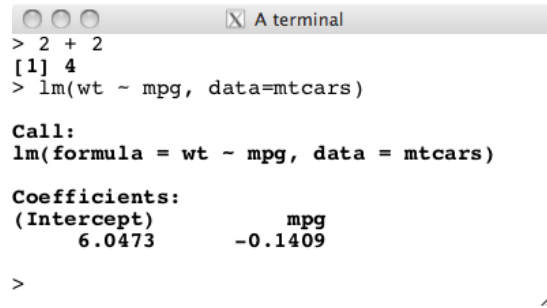
**Tags** Marks and iterators can be used to specify different properties for different parts of the text buffer. GTK+ uses tags to specify how pieces of text will differ from those of the textview overall. To create a tag, the `createTag` method is used. This has optional argument `tag.name` `gtkTextBufferCreateTag` 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 8.3: 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 `copyClipboard` to copy the current selection to the clipboard. The method `cutClipboard` has an extra argument, `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.



```
> 2 + 2
[1] 4
> lm(wt ~ mpg, data=mtcars)

Call:
lm(formula = wt ~ mpg, data = mtcars)

Coefficients:
(Intercept)          mpg
      6.0473        -0.1409

>
```

Figure 8.1: A basic R terminal implemented using a `gtkTextView` widget.

#### Example 8.4: 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)
```

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 visible.

```
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)  
}
```

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) {
```

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

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

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

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

## 8.2 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 specified 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 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

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 8.5: 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 as images) and tree stores are used for hierarchical 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



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 employed 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 position 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 `position=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 position just before or after the sibling.

**Example 8.6: 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 8.7: 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)
  }
}
```

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 iterator 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 *n*th child is desired, the method `iterNthChild` can be used. Again, it returns an iterator referring to the *n*th 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.

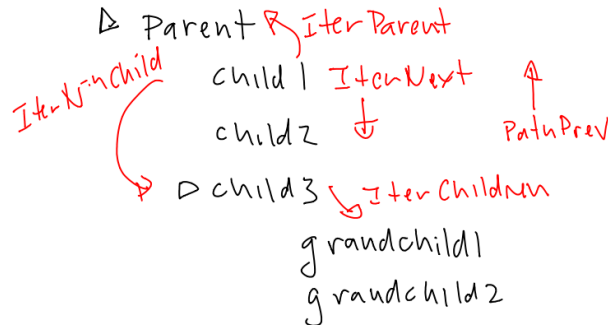


Figure 8.2: [REPLACEME!] Graphical illustration of the functions used by iterators to traverse a tree store.

### 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 laid 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).

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 be changed.

**Pixbuf cell renderers** Graphics can be added to the cell with the renderer `gtkCellRendererPixbuf`. The graphic can be specified by its `stock-id` attribute as a character string, or `icon-name` for a themed icon. It can also be specified 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 ??, 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.

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 8.8: 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.

```
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 8.9: 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 mention 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 iterator pointing to a row in the model and optionally user data that is passed through the `func.data` argument of the `SetMatchFunc` method. This method should return `TRUE` or `FALSE` depending on whether that row should be displayed in the set of completions.

#### Example 8.10: 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).

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 ??, 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 propagate 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)
```

```
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 renderer requires one to get the value from the model and modify it before the cell renderer 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 implement 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 argument, or set as data for the widget if needed within the callback.

```

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 8.11: 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.

## 8. RGtk2: WIDGETS USING MODELS

---

```
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
```

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.

## 8. RGtk2: WIDGETS USING MODELS

---

```
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
```



**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
```

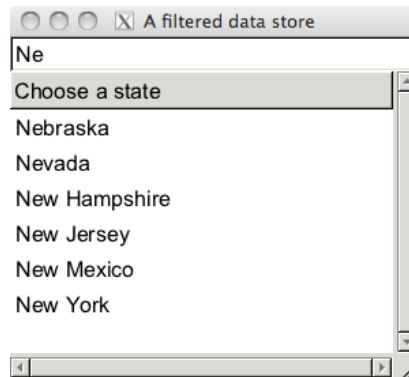


Figure 8.3: Example of a data store filtered by values typed into a text-entry widget.

**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

```
apply(tree.view$getColumnns(), 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 8.12: 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 ?? shows the two widgets placed within a simple GUI.

### Example 8.13: 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 ??, 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)
```

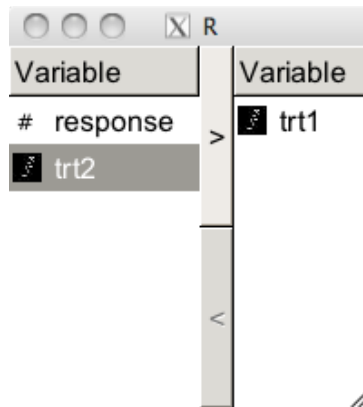


Figure 8.4: An example showing two tree views with buttons to move entries from one to the other. This is a common method for variable selection.

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 render` 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 ?? 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]
  }
}

```

```
    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 8.14: 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 ???. 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")
```

```
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 8.15: 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, modifying 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)
```

```
    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$setValue(iter, column=j-1, children[i,j]))
    ## Add a branch if there are children
    ## no better way, as this adds an extra blank line
    ## we remove it later.
    if(children[i, "offspring"])
      store$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])
}
```



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) {
```

```
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
  })
```

## RGtk2: Menus and Dialogs

### 9.1 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 propagate to all the widgets the action has a proxy connection with.

#### Example 9.1: 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
```

## 9.2 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 sub-menus, 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

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 9.2: 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")
})
```

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 9.3: 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.

### 9.3 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 emitted. Other

#### Example 9.4: Basic toolbar usage

We illustrate with a toolbar, whose buttons are produced in various ways.

```
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, data) {
  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()
```

### 9.4 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.



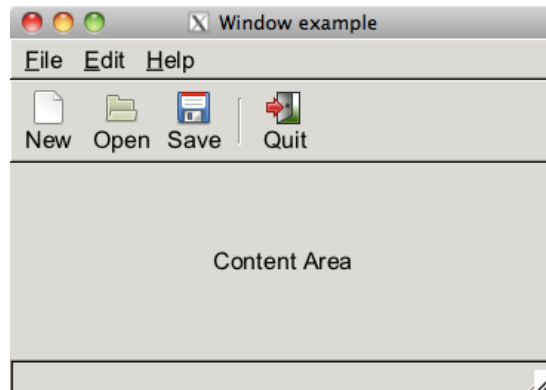


Figure 9.1: A GUI made using a UI manager to layout the menubar and toolbar.

## 9.5 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 accelerator group for keyboard shortcuts,
3. define our actions,
4. create action groups to specify the name, label (with possible mnemonic), keyboard accelerator, tooltip, icon and callback for the graphical 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 9.5: 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()
```

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"),
```

```

        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.

```

<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 `pushmethod`, 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` method 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` method on the menuitems. For instance, instead of a generic "Undo" label, one might want to change the text to list

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")
```

## 9.6 Dialogs

GTK+ comes with a variety of dialogs to create simple, usually single purpose, pop-up 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. There 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"]) {
```

```
## 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 button 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 specify a title, a parent and an action. In addition, the dialog buttons must be specified, as with the last example using `gtkDialogNewWithButtons`.

### Example 9.6: 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.



## Tcl Tk: Overview

Tcl (“tool command language”) is a scripting language and interpreter of that language. Originally developed in the late 80s by John Ousterhout as a “glue” to combine two or more complicated applications together, it evolved overtime to find use not just as middleware, but also as a standalone development tool.

Tk is an extension of Tcl that provides GUI components through Tcl. This was first developed in 1990, again by John Ousterhout. Tk quickly found widespread usage, as it made programming GUIs for X11 easier and faster. Over the years, other graphical toolkits have evolved and surpassed this one, but Tk still has numerous users.

Tk has a large number of bindings available for it, e.g. Perl, Python, Ruby, and through the `tcltk` package, R. The `tcltk` package was developed by Peter Dalgaard, and included in R from version 1.1.0. Since then, the package has been used in a number of GUI projects for R, most notably, the `Rcmdr` GUI.

Tk had a major change between version 8.4 and 8.5, with the latter introducing themed widgets. Many widgets were rewritten, and their API dramatically simplified. In `tcltk` there can be two different functions to construct a similar widget. For example, `tklabel` or `ttklabel`. The latter, with the `ttk` prefix, would be for the newer themed widget. We assume the Tk version is 8.5 or higher, as this was a major step forward. As of version 2.7.0, R for windows has been bundled with this Tk version, so there are no installation issues for that platform. As of writing, some linux distributions and Mac OS X still come with 8.4 which would need to be upgraded for the following.

### 10.1 Interacting with Tcl

The basic syntax of Tcl is a bit unlike R. For example a simple string assignment would be made at `tcsh`, the Tcl shell with (using `%` as a prompt)

```
% set x {hello world}
```

## 10. TCL Tk: OVERVIEW

---

```
hello world
```

Unlike R where braces are used to form blocks, this example shows how Tcl uses braces instead of quotes to group the words as a single string. The use of braces, instead of quotes, in this example is optional, but in general isn't, as expressions within braces are not evaluated.

The example above assigns to the variable `x` the value of `hello world`. Once assignment has been made, one can call commands on the value stored in `x` using the `$` prefix:

```
% puts $x
hello world
```

The `puts` command, in this usage, simply writes back its argument to the terminal. Had we used braces the argument would not have been substituted:

```
% puts {$x}
$x
```

More typical within the `tcltk` package is the idea of a subcommand. For example, the `string` command provides the subcommand `length` to return the number of characters in the string.

```
% string length $x
11
```

The `tcltk` package provides the low-level function `.Tcl` for direct access to the Tcl interpreter:

```
library(tcltk)
.Tcl("set x {some text}")           # assignment
```

```
<Tcl> some text
```

```
.Tcl("puts $x")                     # print
.Tcl("string length $x")             # call a command
```

```
<Tcl> 9
```

the `.Tcl` function simply sends a command as a text string to the Tcl interpreter and returns the result as an object of class `tclObj` (cf. `?Tcl`). These objects print with the leading `<Tcl>` (which we suppress here when there is no output). To coerce these values into characters, the `tclvalue` function is used or the `as.character` function. They differ in how they treat spaces and new lines. Conversion to numeric values is also possible through `as.numeric`, but conversion to logical requires two steps.

The `.Tcl` function can be used to read in Tcl scripts as with `.Tcl("source filename")`. This can be used to run arbitrary Tcl scripts within an R session.

The Tk extensions to Tcl have a complicated command structure, and thankfully, `tcltk` provides some more conveniently named functions. To illustrate, the Tcl command to set the text value for a label object (`.label`) would look like

```
% .label configure -text "new text"
```

The `tcltk` provides a corresponding function `tkconfigure`. Although the Tcl statement appears to have the object oriented form of “object method arguments,” behind the scenes Tcl creates a command with the same name as the widget with `configure` as a subcommand. This is followed by options passed in using the form `-key value`. The Tk API for `ttklabel`’s `configure` subcommand is

*pathName* **configure** *?option? ?value option value ...?*

The *pathName* is the ID of the label widget. In the Tk documentation paired question marks indicate optional values. In this case, one can specify nothing, returning a list of all options; just an option, to query the configured value; the option with a value, to modify the option; and possibly do more than one at a time. For commands such as `configure`, if possible, there will correspond a function in R of the same name with a `tk` prefix, in this case `tkconfigure`. (The package `tcltk` was written before namespaces, so the “tk” prefix serves that role.) To make consulting the Tk manual pages easier in the text we would describe the `configure` subcommand as *ttklabel configure [options]*. (The R manual pages simply redirect you to the original Tk documentation, so understanding this is important for reading the API.) However, if such a function is present, we will use the R function equivalent when we illustrate code. Some subcommands have further subcommands. An example is to set the selection. In the R function, the second command is appended with a dot, as in `tkselection.set`. (There are just a few exceptions to this.)

**The `tcl` function** Within `tcltk`, the `tkconfigure` function is defined by

```
function(widget, ...) tcl(widget, "configure", ...)
```

The `tcl` function is the workhorse function used to piece together Tcl commands. Behind the scenes it turns an R object, *widget*, into the *pathName* above (using its ID component), converts R `key=value` pairs into `-key value` options for Tcl, and adjusts any callback functions. The `tcl` function uses position to create its command, the order of the subcommands needs to match that of the Tk API.

Often, the R object is first, but this is not always the case. As named arguments are only for the `-key value` expansion, we follow the Tcl language and call the arguments “options” in the following. The `tcl` function returns an object of class `tclObj`.

```
tcl(widget, subcommand, key=value, callback)
  /           |           |           \
widget$ID  subcommand  -key value  makeCallback
```

Figure 10.1: How the `tcl` function maps its arguments

To use the `tkconfigure` function to change a labels `text` attribute is done as follows (assuming `l` is a label object):

```
tkconfigure(l, text="new text")
```

## 10.2 Constructors

In this chapter, we will stick to a few basic widgets: labels and buttons; and top-level containers to illustrate the basic usage of `tcltk`, leaving for later chapters more detail on containers and widgets.

Unlike GTK+, say, the construction of widgets in `tcltk` is tightly linked to the widget heirarchy. Tk widgets are constructed as children of a parent container with the parent specified to the constructor. When the Tk shell, `wish`, is used or the Tk package is loaded through the Tcl command package `require Tk`, a top level window named `."` is created. In the variable name `.label`, from above, the dot refers to the top level window. In `tcltk` a top-level window is created separately through the `tktoplevel` constructor, as with

```
w <- toplevel()
```

Top-level windows will be explained in more detail in Section ???. For now, we just use one to construct a label widget. Like all other constructors, the label constructor (`tklabel`) requires a specification of the parent container (`w`) and any other options that are desired. A typical usage would look like:

```
l <- tklabel(w, text="label text")
```

**Options** The first argument of a constructor is the parent container, subsequent arguments are used to specify the options for the constructor given as `key=value` pairs. The Tk API lists these options along with their description.

For a simple label, the following options are possible: `anchor`, `background`, `font`, `foreground`, `justify`, `padding`, `relief`, `text`, and `wraplength`. This is in addition to the standard options `class`, `compound`, `cursor`, `image`, `state`, `style`, `takefocus`, `textvariable`, `underline`, and `width`. (Although clearly lengthy, this list is significantly reduced from the options for `tklabel` where options for the many style properties are also included.)

Many of the options are clear from their name. The padding argument allows the specification of space in pixels between the text of the label and the widget boundary. This may be set as four values `c(left, top, right, bottom)`, or fewer, with bottom defaulting to top, right to left and top to left. The `relief` argument specifies how a 3-d effect around the label should look if specified. Possible values are "flat", "groove", "raised", "ridge", "solid", or "sunken".

**The functions `tkcget`, `tkconfigure`** Option values may be set through the constructor, or adjusted afterwards by `tkconfigure`. A listing (in Tcl code) of possible options that can be adjusted may be seen by calling `tkconfigure` with just the widget as an argument.

```
head(as.character(tkconfigure(l)))

[1] "-background frameColor FrameColor {} {}"
[2] "-foreground textColor TextColor {} {}"
[3] "-font font Font {} {}"
[4] "-borderwidth borderWidth BorderWidth {} {}"
[5] "-relief relief Relief {} {}"
[6] "-anchor anchor Anchor {} {}"
```

The `tkcget` function returns the value of an option (again as a `tclObj` object). The option can be specified two different ways. Either using the Tk style of a leading dash or using the convention that NULL values mean to return the value, and not set it.

```
tkcget(l, "-text")           # retrieve text property

<Tcl> label text

tkcget(l, text=NULL)         # alternate syntax

<Tcl> label text
```

**Coercion to character** The `tclObj` objects can be coerced to character class two ways. The conversion through `as.character` breaks the return value along whitespace:

```
as.character(tkcget(l, text=NULL))

[1] "label" "text"
```

The `tclvalue` function can also be used to extract the value from a `tclObj`, in this case not breaking along white space.

```
tclvalue(tkcget(l, text=NULL))

[1] "label text"
```

**Buttons** Buttons are constructed using the `ttkbutton` constructor.

```
b <- ttkbutton(w, text="click me")
```

Buttons and labels share many of the same options. However, buttons have a `command` option to specify a callback for when it is clicked. Callbacks will be explained in Section ???. Furthermore, buttons have the option `default` to specify which button of a dialog will get the Return signal when the enter key is pressed. A callback can then be set to respond to this signal. This value for `default` may be "active", indicating the button will get the signal; "normal"; or "disabled", to draw the button without space to indicate it

**tkwidget** Constructors call the `tkwidget` function which returns an object of class `tkwin`. (In Tk the term "window" is used to refer to the drawn widget and not just a top-level window)

```
str(b)
```

```
List of 2
 $ ID : chr ".2.2"
 $ env:<environment: 0xcb6fe4>
- attr(*, "class")= chr "tkwin"
```

The returned widget objects are lists with two components an ID and an environment. The ID component keeps a unique ID of the constructed widget. This is a character string, such as ".1.2.1" coming from the the widget heirarchy of the object. This value is generated behind the scenes by the `tcltk` package using numeric values to keep track of the heirarchy. The `env` component contains an environment that keeps track of subwindows, the parent window and any callback functions. This helps ensure that any copies of the widget refer to the same object (?). As the construction of a new widget requires the ID and environment of its parent, the first argument to `tkwidget`, `parent`, must be an R Tk object, not simply its character ID, as is possible for the `tcl` function. The latter is useful in callback, as only the ID may be known to the callback function.

### Geometry managers

As with GTK+, when a new widget is constructed it is not automatically mapped. Tk uses geometry managers to specify how the widget will be drawn within the parent container. We will discuss two such geometry managers in Section ??, but for now, we note that the simplest way to view a widget in its parent window is through `tkpack`:

```
tkpack(l)
tkpack(b)
```

This command packs the widgets into the top-level window (the parent in this case) in a box-like manner. Unlike GTK+ more than one child can be packed into a top-level window, although we don't demonstrate this further, as later we will use an intermediate `ttkframe` box container so that themes are properly displayed.

## Tcl variables

For several Tk widgets, there is an option `textvariable` for a Tcl variable. These variables are dynamically bound to the widget, so that changes to the variable are propagated to the GUI. (The Tcl variable is a model and the widget a view of the model.) The basic functions involved are `tclVar` to create a Tcl variable, `tclvalue` to get the assigned value and `tclvalue <-` to modify the value.

```
textvar <- tclVar("another label")
l2 <- ttklabel(w, textvariable=textvar)
tkpack(l2)
tclvalue(textvar)
```

```
[1] "another label"
```

```
tclvalue(textvar) <- "new text"
```

The advantages of Tcl variables are like those of the MVC paradigm – a single data source can have its changes propagated to several widgets automatically. If the same text is to appear in different places, their usage is recommended. One disadvantage, is that in a callback, the variable is not passed to the callback and must be found through R's scoping rules.

## Colors and fonts

The label color can be set through its `foreground` property. Colors can be specified by name – for common colors – or by hex RGB values which are common in web programming.

```
tkconfigure(l, foreground="red")
tkconfigure(l, foreground="#00aa00")
```

To find the hex RGB value, one can use the `rgb` function to create RGB values from intensities in  $[0,1]$ . The R function `col2rgb` can translate a named color into RGB values. The `as.hexmode` function will display an integer in hexadecimal notation.

**Fonts** Fonts are more involved than colors. Tk version 8.5 made it more difficult to change style properties of individual widgets. This following the practice of centralizing style options for consistency, ease of maintaining

Table 10.1: Standard font names defined by a theme.

|                    |  |
|--------------------|--|
| TkDefaultFont      | The default for all GUI items not otherwise specified. |
| TkTextFont         | Font for text widgets                                  |
| TkFixedFont        | Fixed-width font.                                      |
| TkMenuFont         | Menu bar fonts   |
| TkHeadingFont      | Font for column headings                               |
| TkCaptionFont      | Caption font (dialogs)                                 |
| TkSmallCaptionFont | Smaller caption font                                   |
| TkIconFont         | Icon and text font                                     |

and ease of theming. To set a font for a label, rather than specify the font properties, one configures the font using a pre-defined font name, such as

```
tkconfigure(l, font="TkFixedFont")
```

The "TkFixedFont" value is one of the standard font names, in this case to use a fixed-width font. A complete list of the standard names is provided in Table ???. Each theme sets these defaults accordingly. The `tkfont.create` function can be used to create a new font, as with the following commands:

```
tkfont.create("ourFont", family="Helvetica", size=12,
             weight="bold")
```

```
<Tcl> ourFont
```

```
tkconfigure(l, font="ourFont")
```

Available font families are system dependent. Only "Courier", "Times" and "Helvetica" are guaranteed to be there. A list of available font families is returned by the function `tkfont.families`. Figure ?? shows a display of some available font families on a Mac OS X machine. See Example ?? for details.

The arguments for `tkfont.create` are optional. The `size` argument specifies the pixel size. The `weight` argument can be used to specify "bold" or "normal". Additionally, a `slant` argument can be used to specify either "roman" (normal) or "italic". Finally, `underline` and `overstrike` can be set with a TRUE or FALSE value.

**Font metrics** The average character size is important in setting the width and height of some components. This can be found through the `tkfont.measure` and `tkfont.metrics` functions as follows:

```
tmp <- tkfont.measure("TkTextFont", paste(c(0:9, LETTERS), collapse=""))
fontWidth <- ceiling(as.numeric(tclvalue(tmp))/36)
```



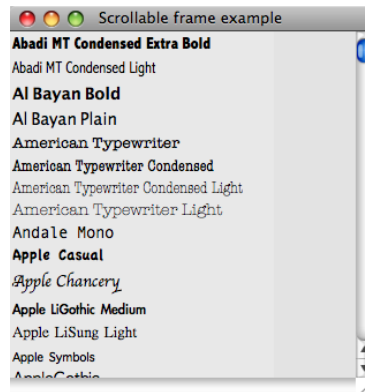


Figure 10.2: A scrollable frame widget (cf. Example ??) showing the available fonts on a system.

```
tmp <- tkfont.metrics("TkTextFont", "linespace"=NULL)
fontHeight <- as.numeric(tclvalue(tmp))
c(width=fontWidth, height=fontHeight)
```

```
width height
      9    16
```

## Images

Many `tcltk` widgets, including both labels and buttons, can show images. In these cases, either with or without an accompanying text label. Constructing images to display is similar to constructing new fonts, in that a new image object is created and can be reused by various widgets. The `tkimage.create` function is used to create image objects. The following command shows how an image object can be made from the file `tclp.gif` in the current directory:

```
tkimage.create("photo", "::img::tclLogo", file = "tclp.gif")
```

```
<Tcl> ::img::tclLogo
```

The first argument, "photo" specifies that a full color image is being used. This option could also be "bitmap" but that is more a legacy option. The second argument specifies the name of the object. We follow the advice of the Tk manual and preface the name with `::img::` so that we don't inadvertently overwrite any existing Tcl commands. The third argument `file` specifies the graphic file. The basic Tk image command only can show GIF and PPM/PNM images. Unfortunately, not many R devices output in these formats. (The GDD device driver can.) One may need system utilities to convert to the allowable formats.

To use the image, one can specify the name for the image option.

```
l <- ttklabel(w, image="::img::tclLogo", text="logo text",  
             compound = "top")
```

By default the text will not show. The compound argument takes a value of either "text", "image" (default), "center", "top", "left", "bottom", or "right" specifying where the label is in relation to the text.

**Image manipulation** Once an image is created, there are several options to manipulate the image. These are found in the Tk man page for photo, not image. For instance, to change the palette so that instead of fullcolor only 16 shades of gray are used to display the icon, one could issue the command

```
tkconfigure("::img::tclLogo", palette=16)
```

Another useful manipulation to draw attention to an image is to change the gamma value when something happens, such as a mouse-over event.

## Themes

The themed widgets have a style that determines how they are drawn. The separation of style properties from the widget, as opposed to having these set for each construction of a widget, makes it much easier to change the look of a GUI and easier to maintain the code. A collection of styles makes up a theme. The available themes depend on the system. The default theme should enable a GUI to have the native look and feel of the operating system. (This was definitely not the case for the older Tk widgets.) There is no built in command to return the theme, so we use `.Tcl` to call the appropriate Tcl command. The `names` sub command will return the available themes and the `use` sub command can be used to set the theme.

```
.Tcl("ttk::style theme names")
```

```
<Tcl> aqua clam alt default classic
```

```
.Tcl("ttk::style theme use classic")
```

The writing of themes will not be covered, but in Example ?? we show how to create a new style for a button.

The example we have shown so far, would not look quite right, as the toplevel window is not a themed widget. To work around that, a `ttkframe` widget is usually used to hold the child components of the top-level window. The following shows how to place a frame inside the window, with some arguments to be explained later that allow it to act reasonably if the window is resized.

```
w <- tktoplevel()
f <- ttkframe(w, padding=c(3,3,12,12)) # a little breathing room
tkpack(f, expand=TRUE, fill="both")    # handle window expansion
l <- ttklabel(f, image="::img::tclLogo", text="label", compound="top")
tkpack(l)
```

### Window properties and state: tkwinfo

Widgets have options which can be set through `tkconfigure` and additionally, when mapped, the “window” they are rendered to has properties, such as a class or size. These properties are queried through the `tkwinfo` function. There are several such properties, and may take different forms. If the API is of the form

```
winfo subcommand_name window
```

the specification to `tkwinfo` is in the same order (the widget is not the first argument). For instance, the class of a label is returned by the `class` subcommand as

```
tkwinfo("class", l)
```

```
<Tcl> TLabel
```

The window, in this example `l`, can be specified as an R object, or by its character ID. This is useful, as the return value of some functions is the ID. For instance, the `children` subcommand returns IDs. Below the `as.character` function will coerce these into a vector of IDs.

```
(children <- tkwinfo("children",f))
```

```
<Tcl> .3.1.1
```

```
sapply(as.character(children), function(i) tkwinfo("class", i))
```

```
$ '.3.1.1 '
```

```
<Tcl> TLabel
```

There are several possible subcommands, here we list a few. The `tkwinfo geometry` sub command returns the location and size of the widgets’ window in the form `width x height + x + y`; the sub commands `tkwinfo height`, `tkwinfo width`, `tkwinfo x`, or `tkwinfo y` can be used to return just those parts. The `tkwinfo exists` command returns 1 (TRUE) if the window exists and 0 otherwise; the `tkwinfo ismapped` sub command returns 1 or 0 if the window is currently mapped (drawn); the `tkwinfo viewable` sub command is similar, only it checks that all parent windows are also mapped. For traversing the widget heirarchy, one has available the `tkwinfo parent` sub command which

returns the immediate parent of the component, *tkwinfo* `toplevel` which returns the ID of the top-level window, and *tkwinfo* `children` which returns the IDs of all the immediate child components, if the object is a container, such as a top-level window.

### 10.3 Events and Callbacks

The button widget has the `command` option for assigning a callback for when the user clicks the mouse button on the button. In addition to this, one can specify callbacks for many other events that the user may initiate.

#### Callbacks

The `tcltk` package implements callbacks in a manner different from Tk, as the callback functions are R functions, not Tk procedures. This is much more convenient, but introduces some slight differences. In `tcltk` these callbacks can be expressions (unevaluated calls) or functions. We use only the latter, for more clarity. The basic callback function need not have any arguments. For instance, here we show how to print a message when the user clicks a button:

```
w <- tktoplevel()
callback <- function() print("hi")
b <- ttkbutton(w, text="Click me", command = callback)
tkpack(b)
```

The callback's return value is generally not important, although we shall see with validation in Section ??, it can matter. This is not the case in Tk. As well, in Tk callbacks are evaluated in the global environment, but this is not so in `tcltk`.

```
<Tcl> .w
```

```
<Tcl> .w.b
```

#### Events

When a user interacts with a GUI, they initiate events. The `tcltk` package allows the programmer to bind callbacks to these events, through the `tkbind` function. This function is called as `tkbind(tag, events, command)`. The `command` is a callback, as described above.

The `tag` argument allows for quite a bit of flexibility. It can be:

- the name of a widget** in which the command will be bound to that widget;
- q top-level window** in which case the command will be bound to the event for the window and all its internal widgets;

**a class of widget** such as "TButton", in which case all such widgets will get the binding; or

**the value "all"** in which case all widgets in the application will get the binding.

The possible events (or sequences of events) vary from widget to widget. Events can be specified in a few ways. A single keypress event, can be assigned by specifying the ASCII character generated. For instance, to bind to C for the "Click me" button above using the same callback could be done with

```
tkbind(b, "C", callback)
tkfocus(b)
```

The `tkfocus` function is used to set the focus to the button so that it will receive the keypress.

**Events with modifiers** More complicated events can be described with the pattern

`<modifier-modifier-type-detail>`.

Examples are `<KeyPress>`, `<ButtonPress>`, `<Double-Button-1>`, or `<Control-c>`. The full list of modifiers and types are described in the man page for `bind`. Some familiar modifiers are `Control`, `Alt`, `Button1` (or its shortening `B1`), `Double` and `Triple`. The event types are the standard X event types along with some abbreviations. These are also listed in the `bind` man page. Some commonly used ones are `ButtonPress`, `ButtonRelease`, `KeyPress`, `KeyRelease`, `FocusIn`, and `FocusOut`. The detail may be a button number or key, such as `a` or `C`.

**Window manager events** Some events are based on window manager events. The `<Configure>` event happens when a component is resized. The `<Map>` and `<Unmap>` events happen when a component is drawn or undrawn.

**Virtual events** Finally, the event may be a "virtual event." These are represented with `<<EventName>>`. There are predefined virtual events listed in the event man page. These include `<<MenuSelect>>` when working with menus, `<<Modified>>` for text widgets, `<<Selection>>` for text widgets, and `<<Cut>>`, `<<Copy>>` and `<<Paste>>` for working with the clipboard. New virtual events can be produced with the `tkevent.add` function. This takes atleast two arguments, an event name and a sequence that will initiate that event. The event man page has these examples coming from the Emacs world:

```
tkevent.add("<<Paste>>", "<Control-y>")
tkevent.add("<<Save>>", "<Control-x><Control-s>")
```

In addition to virtual events occurring when the sequence is performed, the `tkevent.generate` can be used to force an event for a widget. This function requires a widget (or its ID) and the event name. Other options can be used to specify substitution values, described below. To illustrate, this command will generate the `<<Save>>` event for the button `b`:

```
tkevent.generate(b, "<<Save>>")
```

In `tcltk` only one callback can be associated with a widget and event through the call `tkbind(widget,event,callback)`. (Although, callbacks for the widget associated with classes or toplevel windows can differ.) Calling `tkbind` another time will replace the callback. To remove a callback, simply assign a new callback which does nothing.<sup>1</sup>

### % Substitutions

Tk provides a mechanism called *percent substitution* to pass information about the event to callbacks bound to the event. The basic idea is that in the Tcl callback expressions of the type `%X` for different characters `X` will be replaced by values coming from the event. In `tcltk`, if the callback function has an argument `X`, then that variable will correspond to the value specified by `%X`. The complete list of substitutions is in the `bind` man page. Useful ones are `x` and `X` to specify the relative or absolute *x*-position of a mouse click (the difference can be found through the `rootx` property of a widget), `y` and `Y` for the *y* position, `k` and `K` for the keycode (ASCII) and key symbol of a `<KeyPress>` event, and `W` to refer to the ID of the widget that signaled the event the callback is bound to. Example ?? will illustrate some of these.

**The after command** The Tcl command `after` will execute a command after a certain delay (specified in milliseconds as an integer) while not interrupting the control flow while it waits for its delay. The function is called in a manner like:

```
afterID <- tcl("after", 1000, function() print("1 second passed"))
```

The ID returned by `after` may be used to cancel the command before it executes.

### Example 10.1: Drag and Drop

This example shows how to implement drag and drop between two widgets. Steps are needed to make a widget a drop source, and other steps are needed to make a widget a drop target. The basic idea is that when a value is being dragged, virtual events are generated for the widget the cursor is over. If that widget has callbacks bound to these events, then the drag and drop can be

---

<sup>1</sup>This event handling can prevent being able to port some Tk code into `tcltk`. In those cases, one may consider sourcing in Tcl code directly.

processed. The idea for the code below originated with <http://wiki.tcl.tk/416>.

To begin, we create a simple GUI to hold three widgets. We use buttons for drag and drop, but only because we haven't yet discussed more natural widgets such as the text widgets.

```
w <- tkoplevel()
bDrag <- ttkbutton(w, text="Drag me")
bDrop <- ttkbutton(w, text="Drop here")
tkpack(bDrag)
tkpack(ttklabel(w, text="Drag over me"))
tkpack(bDrop)
```

Before beginning, we define three global variables that can be shared among drop sources to keep track of the drag and drop state. A more elegant example might store these in an environment.

```
.dragging <- FALSE           # currently dragging?
.lastWidgetID <- ""         # last widget dragged over
.dragValue <- ""            # value to transfer
```

To set up a drag source, we bind to three events: a mouse button press, mouse motion, and a button release. For the button press, we set the values of the three global variables.

```
tkbind(bDrag, "<ButtonPress-1>", function(W) {
  .dragging <-> TRUE
  .lastWidgetID <-> as.character(W)
  .dragValue <-> as.character(tkcget(W, text=NULL))
})
```

For mouse motion, we do several things. First we set the cursor to indicate a drag operation. The choice of cursor is a bit outdated. The commented code shows how one can put in a custom cursor from an xbm file, but this doesn't work for all platforms (e.g., OS X). After setting the cursor, we find the ID of the widget the cursor is over. This uses `tkwinfo` to find the widget containing the  $x, y$ -coordinates of the cursor position. We then generate the `<<DragOver>>` virtual event for this widget, and if this widget is different from the previous last widget, we generate the `<<DragLeave>>` virtual event.

```
tkbind(bDrag, "<B1-Motion>", function(W, X, Y) {
  if(!.dragging) return()
  ## see cursor help page in API for more
  ## For custom cursors cf. http://wiki.tcl.tk/8674.
  tkconfigure(W, cursor="coffee_mug") # set cursor

  w = tkwinfo("containing", X, Y)      # widget mouse is over
  if(as.logical(tkwinfo("exists", w))) # does widget exist?
```

## 10. TCL Tk: OVERVIEW

---

```
    tkevent.generate(w, "<<DragOver>>")

    ## generate drag leave if we left last widget
    if(as.logical(tkwininfo("exists", w)) &&
        length(as.character(w)) > 0 &&
        length(as.character(.lastWidgetID)) > 0
    ) {
        if(as.character(w)[1] != .lastWidgetID)
            tkevent.generate(.lastWidgetID, "<<DragLeave>>")
    }
    .lastWidgetID <- as.character(w)
})
```

Finally, if the button is released, we generate the virtual events `<<DragLeave>>` and most importantly `<<DragDrop>>` for the widget we are over.

```
tkbind(bDrag, "<ButtonRelease-1>", function(W, X, Y) {
    if(!.dragging) return()
    w = tkwininfo("containing", X, Y)

    if(as.logical(tkwininfo("exists", w))) {
        tkevent.generate(w, "<<DragLeave>>")
        tkevent.generate(w, "<<DragDrop>>")
        tkconfigure(w, cursor="")
    }
    .dragging <- FALSE
    tkconfigure(W, cursor="")
})
```

To set up a drop target, we bind callbacks for the virtual events generated above to the widget. For the `<<DragOver>>` event we make the widget active so that it appears ready to receive a drag value.

```
tkbind(bDrop, "<<DragOver>>", function(W) {
    if(.dragging)
        tkconfigure(W, default="active")
})
```

If the drag event leaves the widget without dropping, we change the state back to normal.

```
tkbind(bDrop, "<<DragLeave>>", function(W) {
    if(.dragging) {
        tkconfigure(W, cursor="")
        tkconfigure(W, default="normal")
    }
})
```

Finally, if the `<<DragDrop>>` virtual event occurs, we set the widget value to that stored in the global variable `.dragValue`.



```
tkbind(bDrop,"<<DragDrop>>", function(W) {  
  tkconfigure(W, text=.dragValue)  
  .dragValue <- ""  
})
```

## 10.4 Rcmdr

[Adding to the Rcmdr. Write me.]

## 10.5 Other sources of documentation

In writing this chapter we used the a few primary resources. The Tk API (?) lays out the details of each widget in a clear concise manner. There is an excellent tutorial demonstrating the themed widgets by Mark Roseman (?). The Tcl language is written about in several books. We used the authoritative text by Welch, Jones and Hobbs (?). The tcltk package was described in three very informative articles by the package Peter Dalgaard (?), (?), and (?). Finally, a number of examples of the package compiled by James Wettenhall may be seen at (?).



## Tcl Tk: Containers and Layout

### 11.1 Top-level windows

Top level windows are created through the `tktoplevel` constructor. The arguments `width` and `height` may be specified to give a requested size. Negative values means the window will not request any size. Top-level windows can have a menubar specified through the `menu` argument. Menus will be covered in Section ??.

The `tkdestroy` function can be called to destroy the window and its child components.

The Tk command `wm` is used to interact with top-level windows. This command has several subcommands, leading to `tcltk` functions with names such as `tkwm.title`. This function is used to set the window title. As with all such functions, either the top-level window object, or its ID must be the first argument. In this case, the new title is the second.

When a top-level window is constructed there is no option for it not to be shown. However, one can use the `tclServiceMode` function to suspend/resume drawing of any widget through Tk. This function takes a logical value to change the state. After a window is drawn. To iconify an already drawn window can be down through the `tkwm.withdraw` function and reversed with the `tkwm.deiconify` function. Together these can be useful to use in the construction of complicated GUIs, as the drawing of the widgets can seem slow. (The same can be done through the `tkwm.state` function with an option of `"withdraw"` or `"normal"`.)

**Window sizing** The preferred size of a top-level window can be configured through the `width` and `height` options. The absolute size and position of a top-level window in pixels can be queried or specified through the `tkwm.geometry` function. The geometry is specified as a string in the form `=wxh+x+y` (or `-`) where any of `=`, `wxh` or `+x+y` can be omitted. The value for `x` (if using `+`) indicates how many pixels to the right from the left edge should the window be placed (if using `-` then the left side of the screen is used as a

reference). For y the top (or bottom) of the screen is the reference.

The `ttksizegrip` widget can be used to add a visual area (usually the lower right corner) for the user to grab on to with their mouse for resizing the window. On some OSes (e.g., Mac OS X) these are added by the window manager automatically.

The `tkwm.resizable` function can be used to prohibit the resizing of a top-level window. The syntax allows either the width or height to be constrained. The following command would prevent resizing of both the width and height of the toplevel window `w`.

```
tkwm.resizable(w, FALSE, FALSE)    # width first
```

When a window is resized, you can constrain the minimum and maximum sizes with `tkwm.minsize` and `tkwm.maxsize`. The aspect ratio (width/height) can be set through `tkwm.aspect`.

For some uses it may be desirable to not have the window manager decorate the window with a title bar etc. Tooltips, for example, can be constructed using this approach. The command `tktoplevel wm overrideRedirect logical` takes a logical value indicating if the window should be decorated. Though, not all window managers respect this.

**bindings** Bindings for top-level windows are shared by all of their child widgets. If a common binding is desired for all the children, then it need only be specified once for the top-level window.

The `tkwm.protocol` function is used to assign commands to window manager events, most commonly, the delete event when the user clicks the close button on the windows decorations. A top-level window can be removed through the `tkdestroy` function, or through the user clicking on the correct window decorations. When the window decoration is clicked, the window manager issues a "WM\_DELETE\_WINDOW" event. To bind to this, a command of this form `tkwm.protocol(win,"WM_DELETE_WINDOW", callback)` is used.

To illustrate, if `w` is a top-level window, and `e` a text entry widget (cf. Section ??) then the following snippet of code would check to see if the text widget has been modified before closing the window. This uses a modal message box described in Section ??.

```
tkwm.protocol(w,"WM_DELETE_WINDOW", function() {
    modified <- tcl(e, "edit", "modified")
    if(as.logical(modified)) {
        response <-
            tkmessageBox(icon="question",
                        message="Really close?",
                        detail="Changes need to be saved",
                        type="yesno",
                        parent=w)
```

```

        if([as.character(response) == "no"])
            return()
    }
    tkdestroy(w)                                # otherwise close
})

```

Sometimes, say with dialogs, a top-level window should be related to a different top-level window. The function `tkwm.transient` allows one to specify the master window as its second argument. The new window will mirror the state of the master window, including if the master is withdrawn.

A window can be made to always be the topmost window through the attributes subcommand of the `wm` command. However, there is no direct `tcltk` function, so if `w` was to be on top, one would use the `tcl` function as follows:

```
tcl("wm", "attributes", w, topmost=TRUE)
```

## 11.2 Frames

The `ttkframe` constructor produces a themable box container that can be used to organize visible components within a GUI. It is often the first thing packed within a top-level window.

The options include `width` and `height` to set the requested size, `borderwidth` to specify a border around the frame of a given width, and `relief` to set the border style. The value of `relief` is chosen from the default "flat", "groove", "raised", "ridge", "solid", or "sunken". The `padding` option can be used to put space within the border between the border and subsequent children.

### Label Frames

The `ttklabelframe` constructor produces a frame with an optional label that can be used to set off and organize components of a GUI. The label is set through the option `text`. Its position is determined by the option `labelanchor` taking values labeled by compass headings (combinations of n, e, w, s). The default is theme dependent, although typically "nw" (upper left).

**Separators** To use a single line to separate out areas in a GUI, the `ttkseparator` widget can be used. The lone widget-specific option is `orient` which takes values of "horizontal" (the default) or "vertical". This widget must be told to stretch when added to a container, as described in the next section.

### 11.3 Geometry Managers

Tcl uses geometry managers to place child components within their parent windows. There are three such managers, but we describe only two, leaving the lower-level `place` command for the official documentation. The use of geometry managers, allows Tk to quickly reallocate space to a GUI's components when it is resized. The `tkpack` function will place children into their parent in the box-like manner. We have seen in several examples throughout the text, that through the use of nested boxes, one can construct quite flexible layouts, and Example ?? will illustrate that once again. When simultaneous horizontal and vertical alignment of child components is desired, the `tkgrid` function can be used to manage the components.

A GUI may use a mix of `pack` and `grid` to manage the child components, but all siblings in the widget heirarchy must be managed the same way. Mixing the two will typically result in a lockup of the R session.

#### Pack

We have illustrated how `tkpack` can be used to manage how child components are viewed within their parent. The basic usage `tkpack(child)` will pack in the child components from top to bottom. The `side` option can take a value of "left", "right", "top" (default), or "bottom" to adjust where the children are placed. These can be mixed and matched, but sticking to just one direction is easier to understand. Using nested frames is a more transparent approach.

**after, before** The `after` and `before` options can be used to place the child before or after another component. These are used as with `tkpack(child1, after=child2)`. The object `child2` can be the R object or an ID. The latter might be useful, say when all the children are listed using the command `tkwininfo("children",parent)` which returns the IDs of the immediate child components.

```
## Code to make padding/pady/ipady figure
## padding
w <- tktoplevel();f <- ttkframe(w, padding=20 + 10);tkpack(f, expand=TRUE, fill="both")
tkpack(ttkbutton(f, text="padding"))
tkpack(ttkbutton(f, text="padding"))
## pady outside border
w <- tktoplevel();f <- ttkframe(w, padding=10);tkpack(f, expand=TRUE, fill="both")
tkpack(ttkbutton(f, text="pady"), pady=10)
tkpack(ttkbutton(f, text="pady"), pady=10)
## ipady within border
w <- tktoplevel();f <- ttkframe(w, padding=10);tkpack(f, expand=TRUE, fill="both")
```



Figure 11.1: Various ways to put padding between widgets using a box container and `tkpack`. The `padding` option for the box container puts padding around the cavity for all the widgets. The `pady` option for `tkpack` puts padding around the top and bottom on the border of each widget. The `ipady` option for `tkpack` puts padding within the top and bottom of the border for each child (modifying the theme under Mac OS X).

```
tkpack(ttkbutton(f, text="ipady"), ipady=10)
tkpack(ttkbutton(f, text="ipady"), ipady=10)
```

In addition to the `padding` option for a frame container, the `ipadx`, `ipady`, `padx`, and `pady` options can be used to add space around the child components. Figure ?? has an example. The `x` and `y` indicate left-right space or top-bottom space. The `i` stands for internal padding that is left on the sides or top and bottom of the child within the border, for `padx` the external padding added around the border of the child component. The value can be a single number or pair of numbers for asymmetric padding.

This sample code shows how one can easily add padding around all the children of the frame `f` using the `tkpack` "configure" subcommand.

**Cavity model** The packing algorithm, as described in the Tk documentation, is based on arranging where to place a slave into the rectangular unallocated space called a cavity. We use the nicer terms child component and box to describe these. When a child is placed inside the box, say on the top, the space allocated to the child is the rectangular space with width given by the width of the box, and height the sum of the requested height of the child plus twice the `ipady` amount (or the sum if specified with two numbers). The packer then chooses the dimension of the child component, again from the requested size plus the `ipad` values for `x` and `y`. These two spaces may, of course, have different dimensions.

**anchor** By default, the child will be placed centered along the edge of the box within the allocated space and blank space, if any, on both sides. If there

is not enough space for the child in the allocated space, the component can be drawn oddly. Enlarging the top-level window can adjust this. When there is more space in the box than requested by the child component, there are other options. The `anchor` option can be used to anchor the child to a place in the box by specifying one of the valid compass points (eg. "n" or "se" leaving blank space around the child. Padding between the child and the box can be set through the `padx` and `pady` options.

**expand, fill** When there is more space in the original box than needed by the children the extra space will be left blank unless some children have the option `expand` set to `TRUE`. In this case, the extra space is allocated evenly to each child with this set. The `fill` option is often used when `expand` is set. The `fill` option is used to base the size of the child on the available cavity in the box – not on the requested size of the child. The `fill` option can be "x", "y" or "both". The former expanding the child's size in just one direction, the latter in both.

**forget** Child components can be forgotten by the window manager, unmapping them but not destroying them, with the `tkpack forget` subcommand, or the convenience function `tkpack.forget`. After a child component is removed this way, it can be re-placed in the GUI using a geometry manager. In `gWidgetstcltk` this is used to create a `gexpandgroup` container, as such a container is not provided by Tk.

**Introspection** The subcommand `tkpack slaves` will return a list of the child components packed into a frame. Coercing these return values to character via `as.character` will produce the IDs of the child components. The subcommand `tkpack info` will provide the packing info for a child.

### Example 11.1: Packing dialog buttons

This example shows how one can pack in action buttons, such as when a dialog is created.

The first example just uses `tkpack` without any arguments except the side to indicate the buttons are packed in left to right, not top to bottom.

```
f1 <- ttklabelframe(f, text="plain vanilla")
tkpack(f1, expand=TRUE, fill="x")
l <- list(ttkbutton(f1, text="cancel"), ttkbutton(f1, text="ok"))
QT <- sapply(l, function(i) tkpack(i, side="left"))
```

Typically the buttons are right justified. One way to do this is to pack in using side with a value of "right". This shows how to use a blank expanding label to take up the space on the left.



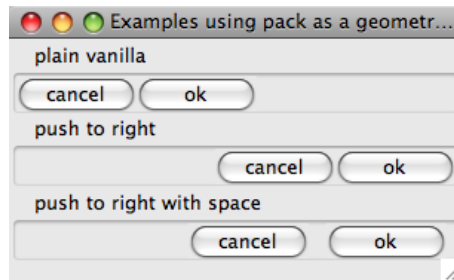


Figure 11.2: Demonstration of using tkpack options showing effects of using the side and padx options to create dialog buttons.

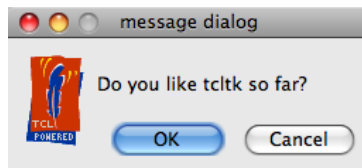


Figure 11.3: Example of a simple dialog

```
f2 <- ttklabelframe(f, text="push to right")
tkpack(f2, expand=TRUE, fill="x")
l <- list(ttkbutton(f2, text="cancel"), ttkbutton(f2, text="ok"))
tkpack(ttklabel(f2, text=" "), expand=TRUE, fill="x", side="left")
QT <- sapply(l, function(i) tkpack(i, side="left"))
```

Finally, we add in some padding to conform to Apple's design specification that such buttons should have a 12 pixel separation.

```
f3 <- ttklabelframe(f, text="push to right with space")
tkpack(f3, expand=TRUE, fill="x")
tkpack(ttklabel(f3, text=" "), expand=TRUE, fill="x", side="left")
l <- list(ttkbutton(f3, text="cancel"), ttkbutton(f3, text="ok"))
QT <- sapply(l, function(i) tkpack(i, side="left", padx=6))
```

### Example 11.2: A non-modal dialog

This example shows how to use window, frames, labels, buttons, icons, packing and bindings to create a non-modal dialog.

Although not written as a function, we set aside the values that would be passed in were it.

```
title <- "message dialog"
message <- "Do you like tcltk so far?"
parent <- NULL
QT <- tkimage.create("photo", "::img::tclLogo",
```

## 11. TCL Tk: CONTAINERS AND LAYOUT

---

```
file = system.file("images","tclp.gif",
package="ProgGUIinR"))
```

The main top-level window is then given a title, then withdrawn while the GUI is created.

```
w <- tktoplevel(); tkwm.title(w, title)
tkwm.state(w, "withdrawn")
```

If the parent is non-null and is viewable, then the dialog is made transient to a parent. The parent need not be a top-level window, so `tkwinfo` is used to find the parent's top-level window. For Mac OS X, we use the `notify` attribute to bounce the dock icon until the mouse enters the window area.

```
if(!is.null(parent)) {
  parentWin <- tkwinfo("toplevel", parent)
  if(as.logical(tkwinfo("viewable", parentWin))) {
    tkwm.transient(w, parent)
    tcl("wm","attributes",parentWin, notify=TRUE) # bounce
    tkbind(parentWin,"<Enter>", function()
      tcl("wm","attributes",parentWin, notify=FALSE)) #stop
  }
}
```

Finally, a frame is packed into the top-level window. Since we want the frame to expand to fill all the space if the window is resized, we use `fill="both"`.

```
## frame for theme purposes, give some space
f <- ttkframe(w, padding=c(5,5,10,10))
tkpack(f, expand=TRUE, fill="both")
```

We will use a standard layout for our dialog with an icon on the left, a message and buttons on the right. We pack the icon into the left side of the frame,

```
l <- ttklabel(f, image="::img::tclLogo", padding=5) # recycle
tkpack(l,side="left")
```

A nested frame will be used to layout the message area and button area. Since the `tkpack` default is to pack in top to bottom, no side specification is made.

```
f1 <- ttkframe(f)
tkpack(f1, expand=TRUE, fill="both")
#
m <- ttklabel(f1, text=message)
tkpack(m, expand=TRUE, fill="both")
```

The buttons have their own frame, as they are layed out horizontally.

```
f2 <- ttkframe(f1)
tkpack(f2)
```

The callback function for the OK button prints a message then destroys the window.

```
okCB <- function() {  
  print("That's great")  
  tkdestroy(w)  
}  
okButton <- ttkbutton(f2, text="OK", default="active")
```

We bind the callback to both a left mouse click on the button, and if the user presses return when the button has the focus. The `default="active"` argument, makes this button the one that gets the Return event when the return key is pressed.

```
tkbind(okButton, "<Button-1>", okCB)  
tkbind(okButton, "<Return>", okCB)  
cancelButton <- ttkbutton(f2, text="Cancel",  
                          command=function() tkdestroy(w))  
tkpack(okButton, side="left", padx=12) # give some space  
tkpack(cancelButton)
```

Now we bring the dialog back from its withdrawn state, fix the size and set the focus on the OK button.

```
tkwm.state(w, "normal")  
tkwm.resizable(w, FALSE, FALSE)  
tkfocus(okButton)
```

Finally, the following bindings make the buttons look active when the keyboard focus is on them, generating a FocusIn event, then a FocusOut event. We make a binding for the top-level window, then within the callback check to see if the widget emitting the signal is of a themed button class.

```
isTButton <- function(W)  
  as.character(tkwininfo("class",W)) == "TButton"  
tkbind(w,"<FocusIn>", function(W) {  
  if(isTButton(W)) tkconfigure(W,default="active")  
})  
tkbind(w,"<FocusOut>", function(W) {  
  if(isTButton(W)) tkconfigure(W,default="normal")  
})
```

## Grid

The `tkgrid` geometry manager is used to place child widgets in rows and columns. In its simplest usage, a command like

```
tkgrid(child1, child2, ..., childn)
```

will place the  $n$  children in a new row, in columns 1 through  $n$ . However, the specific row and column can be specified through the row and column

options. Counting of rows and columns starts with 0. Spanning of multiple rows and columns can be specified with integers 2 or greater by the `rowspan` and `colspan` options. These options, and others can be adjusted through the `tkgrid.configure` function.

**The `tkgrid.rowconfigure`, `tkgrid.columnconfigure` commands** When the managed container is resized, the grid manager consults weights that are assigned to each row and column to see how to allocate the extra space. These weights are configured with the `tkgrid.rowconfigure` and `tkgrid.columnconfigure` functions through the option `weight`. The weight is a value between 0 and 1. If there are just two rows, and the first row has weight 1/2 and the second weight 1, then the extra space is allocated twice as much for the second row. The specific row or column must also be specified. Rows and columns are referenced starting with 0 not the usual R-like 1. So to specify a weight of 1 to the first row would be done with a command like:

```
tkgrid.rowconfigure(parent, 0, weight=1)
```

**The sticky option** When more space is available than requested by the child component, the `sticky` option can be used to place the widget into the grid. The value is a combination of the compass points "n", "e", "w", and "s". A specification "ns" will make the child component "stick" to the top and bottom of the cavity that is provided, similar to the `fill="y"` option for `tkgrid`. Similarly a value of "news" will make the child component expand in all the direction like `fill="both"`. A value of "w" will anchor the widget on the left side of the cavity

**Padding** As with `tkpack`, `tkgrid` has options `ipadx`, `ipady`, `padx`, and `pady` to give internal and external space around a child.

**Size** The function `tkgrid.size` will return the number of columns and rows of the specified parent container that is managed by a grid. This can be useful when trying to position child components through the options `row` and `column`.

**Forget** To remove a child from the parent, the `tkgrid.forget` function can be used with the child object as its argument.

### Example 11.3: Using `tkgrid` and `tkpack` to draw some world flags

This example shows how the `tkpack` and `tkgrid` geometry managers can be used to draw some of the world flags. For these, we consulted <https://www.cia.gov/library/publications/the-world-factbook/docs/flagsoftheworld.html>.



Figure 11.4: Example of world flags to illustrate `tkpack` and `tkgrid` usage. The Mali flag uses `expand=TRUE` to allocate space evenly, `fill="both"` to have the child fill the space and `side="left"` to place the children, whereas Lithuania uses `side="top"`. The Benin flag takes advantage of `tkgrid` to layout the colors in a grid. The left color has `rowspan=2` set. The Togo flag could be done using just `grid`, but a mix is demonstrated.

We will make the dimensions of the flags true to the flag proportions. These we found at <http://flagspot.net/flags/xf-size.html>. Here we define the proportions for the flags of interest.

```
dims <- cbind(Benin=2:3, Cameroon=2:3, Guinea=2:3, Mali=2:3,
              Bolivia=2:3, Lithuania=1:2, Congo=2:3, Guyana=1:2,
              Togo= 2:3)
```

This is a convenience function to create `tkframes` with different background colors. We use `tkframe` here – not `ttkframe` – as it has a background property.

```
makeColors <- function(parent)
  list(green = tkframe(parent, background="green"),
       red   = tkframe(parent, background="red"),
       yellow = tkframe(parent, background="yellow"))
```

This convenience function packs a frame into a top-level window.

```
makeTopLevel <- function(country) {
  w <- tktoplevel()
  tkwm.title(w, country)
  f <- ttkframe(w, padding=c(3,3,3,12))
```

## 11. Tcl Tk: CONTAINERS AND LAYOUT

---

```
tkpack(f, expand=TRUE, fill="both")
return(list(w=w, f= f, country=country))
}
```

This function resizes the toplevel window and forces it to have a fixed proportion using the global dims defined previously.

```
resizeWin <- function(win) {
  height <- 200;
  setWidth <- function(h)
    floor(h * dims[2, win$country] / dims[1, win$country])
  width <- setWidth(height)
  tkwm.geometry(win$w, paste(width, height, sep="x"))
  ## constrain the resize to flag proportions
  ## alternate to tkwm.aspect(w, a,b,c,d)
  tkbind(win$w, "<Configure>", function(W) {
    t1 <- tkwininfo("toplevel",W)
    geo <- tclvalue(tkwm.geometry(t1))
    geo <- as.numeric(unlist(strsplit(geo,"^[:digit:~]"))[1:2])
    tkwm.geometry(t1, paste(setWidth(geo[2]), geo[2], sep="x"))
  })
}
```

Our first flags are Cameroon (GRY), Guinea (RYG), and Mali (GYR). These are flags with 3 equal vertical strips of color. We use tkpack with side="left" to pack in the colors from left to right. The expand=TRUE option causes extra space to be allocated equally to the three children, preserving the equal sizes in this case.

```
win <- makeTopLevel("Cameroon")
w <- win$w; f <- win$f
l <- makeColors(f)
tkpack(l$green, l$red, l$yellow, expand=TRUE,
  fill="both", side="left")
resizeWin(win)
```

To create Guinea's flag we simply move the green strip to the end.

```
## Guinea just moves colors around
tkpack("forget", l$green)
tkpack(l$green, expand=TRUE, fill="both", side="left")
tkwm.title(win$w, "Guinea")
```

For Mali, we flip the position of green and red. We pack them in relative to the yellow strip using the before and after options to tkpack.

```
tkpack("forget", l$green)
tkpack("forget", l$red)
tkpack(l$green, before=l$yellow, expand=TRUE, fill="both", side="left")
tkpack(l$red, after=l$yellow, expand=TRUE, fill="both", side="left")
tkwm.title(win$w, "Mali")
```

Lithuania is similar, only the stripes run horizontally. We pack from top to bottom to achieve this.

```
win <- makeToplevel("Lithuania")
l <- makeColors(win$f)
tkpack(l$yellow, l$green, l$red, expand=TRUE, fill="both", side="top")
resizeWin(win)
```

Benin's flag is better suited for the grid geometry manager. We use a combination of rowspan and colspan to get the proper arrangement. In this case, the proportions of the colors are achieved through equal weights when we configure the row and columns.

```
## benin is better suited for grid
win <- makeToplevel("Benin")
l <- makeColors(win$f)
tkgrid(l$green, row=0, column=0, rowspan=2, sticky="news")
tkgrid(l$yellow, row=0, column=1, colspan=2, sticky="news")
tkgrid(l$red, row=1, column=1, colspan=2, sticky="news")
## use grid in equal sizes to get spacing right
tkgrid.rowconfigure(win$f, 0:1, weight=1)
tkgrid.columnconfigure(win$f, 0:2, weight=1)
resizeWin(win)
```

Togo is trickier. We could use grid, as above, with the proper combinations of row and colspan. Instead we do this less directly to illustrate the mixing of the tkgrid and tkpack geometry managers.

```
win <- makeToplevel("Togo")
f <- win$f
l <- makeColors(f)
upperR <- ttkframe(f); bottom <- ttkframe(f)
tkgrid(l$red, row=0, column=0, sticky="news")
tkgrid(upperR, row=0, column=1, sticky="news")
tkgrid(bottom, row=1, column=0, colspan=2, sticky="news")
l1 <- makeColors(upperR)
tkpack(l1$yellow, expand=TRUE, fill="both", side="top")
tkpack(l1$green, expand=TRUE, fill="both", side="top")
l2 <- makeColors(bottom)
tkpack(l2$yellow, expand=TRUE, fill="both", side="top")
tkpack(l2$green, expand=TRUE, fill="both", side="top")
tkgrid.rowconfigure(f, 0:1, weight=1)
tkgrid.columnconfigure(f, 0, weight=8)
tkgrid.columnconfigure(f, 1, weight=10) # not quite uniform
resizeWin(win)
```

#### Example 11.4: Using tkgrid to create a toolbar

## 11. Tcl Tk: CONTAINERS AND LAYOUT

---

Tk does not have a toolbar widget. Here we use `tkgrid` to show how we can add one to a top-level window in a manner that is not affected by resizing. We begin by packing a frame into a top-level window.

```
w <- tktoplevel(); tkwm.title(w, "Toolbar example")
f <- ttkframe(w, padding=c(3,3,12,12))
tkpack(f, expand=TRUE, fill="both")
```

Our example has two main containers: one to hold the toolbar buttons and one to hold the main content.

```
tbFrame <- ttkframe(f, padding=0)
contentFrame <- ttkframe(f, padding=4)
```

The `tkgrid` geometry manager is used to place the toolbar at the top, and the content frame below. The choice of sticky and the weights ensure that the toolbar does not resize if the window does.

```
tkgrid(tbFrame, row=0, column=0, sticky="we")
tkgrid(contentFrame, row=1, column=0, sticky = "news")
tkgrid.rowconfigure(f, 0, weight=0)
tkgrid.rowconfigure(f, 1, weight=1)
tkgrid.columnconfigure(f, 0, weight=1)
## some example to pack into the content area
tkpack(ttklabel(contentFrame, text="Some content"))
```

Now to add some buttons to the toolbar. We first show how to create a new style for a button, slightly modifying the themed button to set the font and padding, and eliminate the border if the OS allows. This `makeIcon` function finds stock icons from the `gWidgets` package.

```
tcl("ttk::style","configure","Toolbar.TButton",
    font="helvetica 12", padding=0, borderwidth=0)
makeIcon <- function(parent, stockName, command=NULL) {
    iconFile <- system.file("images",
                           paste(stockName,"gif",sep="."),
                           package="gWidgets")
    if(nchar(iconFile) == 0) {
        b <- ttkbutton(parent, text=stockName, width=6)
    } else {
        iconName <- paste("::img::",stockName, sep="")
        tkimage.create("photo", iconName, file = iconFile)
        b <- ttkbutton(parent, image=iconName,
                       style="Toolbar.TButton", text=stockName,
                       compound="top", width=6)
        if(!is.null(command))
            tkconfigure(b, command=command)
    }
    return(b)
}
```



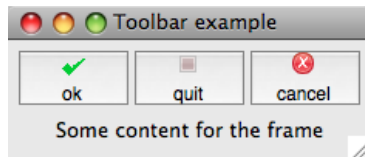


Figure 11.5: Illustration of using tkpack to make a toolbar.

To illustrate, we pack in some icons. Here we use tkpack. One does not use tkpack and tkgrid to manage children of the same parent, but these are children of tbFrame, not f.

```
tkpack(makeIcon(tbFrame, "ok"), side="left")
tkpack(makeIcon(tbFrame, "quit"), side="left")
tkpack(makeIcon(tbFrame, "cancel"), side="left")
```

These two bindings show how to slightly highlight the icon when the mouse is over that button, so that the user has some extra feedback.

```
changeGamma <- function(W, gamma=1.0) {
  if(as.character(tkwininfo("class",W)) == "TButton") {
    img <- tkcget(W,"image"=NULL)
    tkconfigure(img, gamma=gamma)
  }
}
tkbind(w,"<Enter>", function(W) changeGamma(W, gamma=0.5))
tkbind(w,"<Leave>", function(W) changeGamma(W, gamma=1.0))
```

## 11.4 Other containers

Tk provides just a few other basic containers, here we describe paned windows and notebooks.

### Paned Windows

A paned window is constructed by the function `ttkpanedwindow`. The primary option, outside of setting the requested width or height with `width` and `height`, is `orient`, which takes a value of `"vertical"` (the default) or `"horizontal"`. This specifies how the children are stacked, and is opposite how the sash is drawn.

The returned object can be used as a parent container, although one does not use the geometry managers to manage them. Instead, the `add` command is used. For example:

```
w <- ttoplevel(); tkwm.title(w, "Paned window example")
```

## 11. TCL Tk: CONTAINERS AND LAYOUT

---

```
pw <- ttkpanedwindow(w, orient="horizontal")
tkpack(pw, expand=TRUE, fill="both")
left <- ttklabel(pw, text="left"); right <- ttklabel(pw, text="right")
#
tkadd(pw, left, weight=1)
tkadd(pw, right, weight=2)
```

When resizing which child gets the space is determined by the associated weight, specified as an integer. The default uses even weights. Unlike GTK+ more than two children are allowed.

**Forget** The subcommand *ttkpanedwindow* *forget* can be used to unmanage a child component. For the paned window, we have no convenience function, so we call as follows:

```
tcl(pw, "forget", right)
tkadd(pw, right, weight=2) ## add back
```

**Sash position** The sash between two children can be adjusted through the subcommand *ttkpanedwindow* *sashpos*. The index of the sash needs specifying, as there can be more than one. Counting starts at 0. The value for *sashpos* is in terms of pixel width (or height) of the paned window. The width can be returned as follows:

```
tcl(pw, "sashpos", 0, 150)
```

```
<Tcl> 59
```

```
as.integer(tkwininfo("width", pw)) # or "height"
```

```
[1] 71
```

### Notebooks

The *ttknotebook* constructor returns a notebook object that can be subsequently manipulated. In Tk the object itself, becomes a command with the subcommands being important. There are no convenience functions for these, so we will use the *tcl* function directly.

Notebook pages can be added through the *ttknotebook* *add* subcommand or inserted after a page through the *ttknotebook* *insert* subcommand. The latter requires a tab ID to be specified, as described below. The tab label is configured similarly to *ttklabel* through the options *text* and the optional *image*, which if given has its placement determined by *compound*. The placement of the child component within the notebook page is manipulated similarly as *tkgrid* through the *sticky* option with values specified through compass points. Extra padding around the child can be added through the

padding option. Typically, the child components would be containers to hold more complicated layouts.

**Tab identifiers** Many of the commands for a notebook require a specification of a desired tab. This can be given by index, starting at 0; by the values "current" or "end"; by the child object added to the tab, either as an R object or an ID; or in terms of  $x$ - $y$  coordinates in the form "@ $x$ , $y$ " (likely found through a binding).

To illustrate, if `nb` is a `ttknotebook` object, then these commands would add pages.

```
iconFile <- system.file("images",paste("help","gif",sep="."),
                        package="gWidgets")
iconName <- "::tcl::helpIcon"
QT <- tkimage.create("photo", iconName, file = iconFile)
l1 <- ttklabel(nb, text="label 1")
l2 <- ttklabel(nb, text="label 2")
tkadd(nb, l2, sticky="nswe", text="label 2",
      image=iconName, compound="right")
## put l1 first (tabID is 0)
tkinsert(nb, 0, l1, sticky="nswe", text="label 1")
```

There are several useful subcommands to extract information from the notebook object. For instance, `index` to return the page index (0-based), `tabs` to return the page IDs, `select` to select the displayed page, and `forget` to remove a page from the notebook. Except for `tabs`, these require a specification of a tab ID.

```
tcl(nb, "index", "current")          # current page for tabID
```

```
<Tcl> 1
```

```
length(as.character(tcl(nb,"tabs"))) # number of pages
```

```
[1] 2
```

```
tcl(nb, "select", 0)                 # select viewable page by index
tcl(nb, "forget", l1)                 # forget removes page from notebook
tcl(nb, "add", l1)                    # can be managed again.
```

The notebook state can be manipulated through the keyboard, provided traversal is enabled. This can be done through

```
QT <- tcl("ttk::notebook::enableTraversal", nb)
```

If enabled, the shortcuts such as control-tab to move to the next tab are implemented. If new pages are added or inserted with the option `underline`, which takes an integer value (0-based) specifying which character in the label is underlined, then a keyboard accelerator is added for that letter.

**bindings** Beyond the usual events, the notebook widget also generates a <<NotebookTabChanged>> virtual event after a new tab is selected.

The notebook container in Tk has a few limitations. For instance, there is no graceful management of too many tabs, as there is with GTK+, as well there is no easy way to implement close buttons as an icon.

## Tcl Tk: Widgets

Tk has widgets for the common GUI controls. As mentioned in Chapter ??, the constructors for these widgets call the function `tkwidget` which calls the appropriate Tk command and adds in extra information including an ID and an environment. As with labels and buttons, one primarily uses `tkconfigure` and `tkcget` to set and get properties of the widget.

The label, button and image constructors were discussed in Section ??. For buttons and labels, we saw that a Tcl variable can be used to store the data for a widget.

### 12.1 Selection Widgets

This section covers the many different ways to present data for the user to select a value. The widgets can use Tcl variables to refer to the value that is displayed or that the user selects. Recall, these were constructed through `tclVar` and manipulated through `tclvalue`. For example, a logical value can be stored as

```
value <- tclVar(TRUE)
tclvalue(value) <- FALSE
tclvalue(value)
```

```
[1] "0"
```

As `tclvalue` coerces the logical into the character string "0" or "1", some coercion may be desired.

#### Checkbutton

The `ttkcheckbutton` constructor returns a check button object. The checkbutton's value (TRUE or FALSE) is linked to a Tcl variable which can be specified using a logical value. The checkbutton label can also be specified through a Tcl variable using the `textvariable` option. Alternately, as with the `ttklabel` constructor, the label can be specified through the `text` option.

## 12. TCl Tk: WIDGETS

---

This allows one to specify an image as well and arrange its display, as is done with `ttklabel`, using the `compound` option.

The `command` argument is used at construction time to specify a callback when the button is clicked. The callback is called when the state toggles, so often a callback considers the state of the widget before proceeding. Binding through `<Button-1>` can be problematic, as the callback is called before the variable is updated, use `<ButtonRelease-1>`.

For example, if `f` is a frame, we can create a new check button with the following:

```
value <- tclVar(TRUE)
callback <- function() print(tclvalue(value))      # uses global
labelVar <- tclVar("check button label")
cb <- ttkcheckbutton(f, variable=value,
                    textvariable=labelVar, command=callback)
tkpack(cb)
```

To avoid using a global variable is not trivial here. There is no easy way to pass user data through to the callback, and there is no easy way to get the R object from the values passed through the `%` substitution values. The variable holding the value can be found through

```
tkcget(cb, "variable"=NULL)
```

```
<Tcl> ::RTcl3
```

But manipulating that is difficult. A more general strategy within R would be to use a function closure to encapsulate the variables or an environment to store the global values.

### Radio Buttons

Radiobuttons are checkbuttons linked through a shared Tcl variable. Each button is constructed through the `ttkradiobutton` constructor. Each button has a value and a label, which need not be the same. The variable refers to the value. As with labels, the radio button labels may be specified through a text variable or the `text` option, in which case, as with a `ttklabel`, an image may also be incorporated through the `image` and `compound` options. In Tk the placement of the buttons is managed by the programmer.

This small example shows how radio buttons could be used for selection of an alternative hypothesis, assuming `f` is a parent container.

```
values <- c("less", "greater", "two.sided")
var <- tclVar(values[3])                # initial value
callback <- function() print(tclvalue(var))
for(i in values) {
  rb <- ttkradiobutton(f, text=i, variable=var, value=i,
                      command=callback)
```

```
tkpack(rb, side="top", anchor="w")
}
```

## Comboboxes

The `ttkcombobox` constructor returns a combobox object to select from a list of values, or with the appropriate option, allowing the user to specify a value. Like radiobuttons and checkbuttons, the value of the combobox can be specified using a Tcl variable to the option `textvariable`, making the getting and setting of the displayed value straightforward. The possible values to select from are specified as a character vector through the `values` option. (This may require one to coerce the results to the desired class.) Unlike GTK+ there is no option to include images in the displayed text. One can adjust the alignment through the `justify` options. By default, a user can add in additional values through the entry widget part of the combobox. The `state` option controls this, with the default `"normal"` and the value `"readonly"` as an alternative.

To illustrate, again suppose `f` is a parent container. Then we begin by defining some values to choose from and a Tcl variable.

```
values <- rownames(mtcars)
var <- tclVar(values[1])
```

The constructor call is as follows:

```
cb <- ttkcombobox(f,
                  values=values,
                  textvariable=var,
                  state="normal", # or "readonly"
                  justify="left")
tkpack(cb)
```

The possible values the user can select from can be configured after construction through the `values` option.

```
tkconfigure(cb, values=tolower(values))
```

**Setting the value** Setting values can be done through the Tcl variable, or by value or index (0-based) using the `ttkcombobox set` sub command through `tkset` or the `ttkcombobox current` sub command.

```
tclvalue(var) <- values[2]           # using tcl variable
tkset(cb, values[4])                 # by value
tcl(cb, "current", 4)                 # by index
```

The values can also be retrieved in this manner, but coercing into a usable R object is not supported in general.

## 12. TCL Tk: WIDGETS

---

**Getting the value** One can retrieve the selected object in various ways: from the Tcl variable. Additionally, the *ttkcombobox* *get* subcommand can be used through *tkget*.

```
| tclvalue(var)                                # TCL variable
```

```
| [1] "hornet sportabout"
```

```
| tkget(cb)                                    # get subcommand
```

```
<Tcl> hornet sportabout
```

```
| tcl(cb, "current")                          # 0-based index
```

```
<Tcl> 4
```

**Events** The virtual event `<<ComboboxSelected>>` occurs with selection. When the combobox may be edited, a user may expect some action when the return key is pressed. This triggers a `<Return>` event, so to bind to this event, one can do something like the following:

```
| tkbind(cb, "<Return>", function(W) {  
|     val <- tkget(W)  
|     cat(as.character(val), "\n")  
| })
```

For editable comboboxes, the widget also supports some of the *ttkentry* commands discussed in Section ??.

### Scale widgets

The *ttkscale* constructor to produce a themable scale (slider) control is missing. You can define your own:

```
| ttkscale <- function(parent, ...) tkwidget(parent, "ttk::scale", ...)
```

The orientation is set through the option *orient* taking values of "horizontal" (the default) or "vertical". For sizing the slider, the *length* option is available. To set the range, the basic options are *from* and *to* to set the range. There is no *by* option as of Tk 8.5. The constructor *ttkscale*, for a non-themable slider, has the option *resolution* to set that. The *variable* option is used for specifying a Tcl variable to record the value of the slider. Otherwise the *value* option is available. The *tkget* and *tkset* function (using the *ttkscale* *get* and *ttkscale* *set* sub commands) can be used to get and set the value shown. They are used in the same manner as the same subcommands for a combobox. Again, the *command* option can be used to specify a callback for when the slider is manipulated by the user.



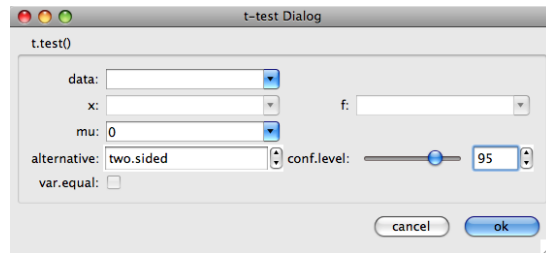


Figure 12.1: A dialog to collect values for a t.test.

## Spinboxes

In Tk version 8.5 there is no themable spinbox widget. In Tk the `spinbox` command produces a non-themable spinbox. Again, there is no direct `tkspinbox` constructor, but one can be defined:

```
tkspinbox <- function(parent, ...) tkwidget(parent, "tk::spinbox", ...)
```

The non-themable widgets have many more options than the themable ones, as style properties can be set on a per-widget basis. We won't discuss those here. The spinbox can be used to select from a sequence of numeric values or a vector of character values.

The basic options to set the range for a numeric spinbox are `from`, `to`, and `increment`. The `textvariable` option can be used to link the spinbox to a Tcl variable. As usual, this allows the user to easily get and set the value displayed. Otherwise, the `tkget` and `tkset` functions may be used for these tasks. The option `state` can be used to specify whether the user can enter values, the default of "normal"; not edit the value, but simply select one of the given values ("readonly"), or not select a value ("disabled").

In Tk, spinboxes can also be used to select from a list of text values. These are specified through the `values` option. For the latter, the `wrap` option, as in `wrap=TRUE`, can be used to wrap around the list of values when the end is reached. As with a combobox, when the Tk spinbox displays character data and is in the "normal" state, the widget can be controlled like the entry widget of Section ??.

### Example 12.1: A GUI for `t.test`

This example illustrates how the basic widgets can be combined to make a dialog for gathering information to run a *t*-test. A realization is shown in Figure ??.

We will use a data store to hold the values to be passed to `t.test`. For the data store, we use an environment to hold Tcl variables.

```
### Data model
e <- new.env()
```

## 12. TCL Tk: WIDGETS

---

```
e$x <- tclVar(""); e$f <- tclVar(""); e$data <- tclVar("")
e$mu <- tclVar(0); e$alternative <- tclVar("two.sided")
e$conf.level <- tclVar(95); e$var.equal <- tclVar(FALSE)
```

Our layout is basic. Here we place a label frame into the window to give the dialog a nicer look. We will use the `tkgrid` geometry manager below.

```
### GUI
w <- tktoplevel()
tkwm.title(w, "t-test Dialog")
f <- ttkframe(w, padding=c(3,3,3,12))
tkpack(f, expand=TRUE, fill="both")
lf <- ttklabelframe(f, text="t.test()", padding=10)
tkpack(lf, expand=TRUE, fill="both", padx=5, pady=5)
```

This next function simplifies the task of adding a label.

```
putLabel <- function(parent, text, row, column) {
  label <- ttklabel(parent, text=text)
  tkgrid(label, row=row, column=column, sticky="e")
}
```

Our first widget will be one to select a data frame. For this, a combobox is used, although if large number of data frames are a possibility, a different widget may be better suited. The `getDfs` function is not shown, but simply returns the names of all data frames in the global environment. Also not shown are two similar calls to create comboboxes `xCombo` and `fCombo` which allow the user to specify parts of a formula.

```
putLabel(lf, "data:", 0, 0)
dataCombo <- ttkcombobox(lf, values=getDfs(), textvariable=e$data)
tkgrid(dataCombo, row=0, column=1, sticky="ew", padx=2)
tkfocus(dataCombo) # give focus
```

```
## not shown
putLabel(lf, "f:", 1, 2)
fCombo <- ttkcombobox(lf, values=getTwoLevelFactor(), textvariable=e$f)
tkgrid(fCombo, row=1, column=3, sticky="ew", padx=2)
```

The combobox may not be the most natural widget to gather a numeric value for the mean when the data is continuous, but at this point we haven't discussed the `tkentry` widget.

```
putLabel(lf, "mu:", 2, 0)
muCombo <- ttkcombobox(lf, values=c(""), textvariable=e$mu)
tkgrid(muCombo, row=2, column=1, sticky="ew", padx=2)
```

The alternative is a natural choice for a combo box, but in `tcltk` we can use a spin box with `wrap=TRUE`.

```
putLabel(lf, "alternative:", 3, 0)
```

```
altCombo <- tkspinbox(lf, values=c("two.sided","less","greater"),
                      textvariable=e$alternative, wrap=TRUE)
tkgrid(altCombo, row=3, column=1, sticky="ew", padx=2)
```

Here we use two widgets to specify the confidence level. The slider is quicker to use, but less precise than the spinbox.

```
putLabel(lf, "conf.level:", 3, 2)
altFrame <- ttkframe(lf)
tkgrid(altFrame, row=3, column=3, columnspan=2,
       sticky="ew", padx=2)
altScale <- ttkScale(altFrame, from=75, to=100,
                    variable=e$conf.level)
tkpack(altScale, expand=TRUE, fill="y", side="left")
altSpin <- tkspinbox(altFrame, from=75, to=100, increment=1,
                    textvariable=e$conf.level, width=5)
tkpack(altSpin, side="left")
```

A checkbox is used to set the binary variable for `var.equal`

```
putLabel(lf, "var.equal:", 4, 0)
veCheck <- ttkcheckboxbutton(lf, variable=e$var.equal)
tkgrid(veCheck, row=4, column=1, stick="w", padx=2)
```

When assigning grid weights, we don't want the labels (columns 0 and 2) to expand the same way we want the other columns to do, so we assign different weights.

```
tkgrid.columnconfigure(lf, 0, weight=1)
tkgrid.columnconfigure(lf, 1, weight=10)
tkgrid.columnconfigure(lf, 2, weight=1)
tkgrid.columnconfigure(lf, 1, weight=10)
```

The dialog has two control buttons we wish to include.

```
bf <- ttkframe(f)
tkpack(bf, fill="x", padx=5, pady=5)
cancel <- tkbutton(bf, text="cancel", command=function() {
  tcl("after", "cancel", updateID$ID)
  tkdestroy(w)
})
ok <- tkbutton(bf, text="ok")
tkpack(ttklabel(bf, text=" "), expand=TRUE, fill="y",
       side="left")
tkpack(cancel, padx=6, side="left")
tkpack(ok, padx=6, side="left")
tkconfigure(ok, default="active")
```

The ok button is made to look active. As such we should bind to the button click and "Return" signals. First we define the callback. The `runTTest` function is not shown, but is written to make good use of the structure of the data store.

## 12. TCL Tk: WIDGETS

---

```
okCallback <- function() {  
  l <- lapply(e, tclvalue)  
  runTTest(l)  
}  
tkbind(ok, "<Button-1>", okCallback)  
tkbind(w, "<Return>", okCallback)      # for active binding
```

Our `updateUI` function does two basic things: it searches for new data frames and it adjusts the controls depending on the current state.

```
updateUI <- function() {  
  tkconfigure(dataCombo, values=getDfs())  
  dfName <- tclvalue(e$data)  
  
  if(dfName == "") {  
    tkconfigure(xCombo, state="disabled")  
  } else {  
    df <- get(dfName, envir=.GlobalEnv)  
    tkconfigure(xCombo, state="normal", values=getNumericVars(df))  
    if(! tclvalue(e$x) %in% getNumericVars(df))  
      tclvalue(e$x) <- ""  
  
    tkconfigure(fCombo, values=getTwoLevelFactor(df))  
    if(! tclvalue(e$f) %in% getTwoLevelFactor(df))  
      tclvalue(e$f) <- ""  
  }  
  
  tkconfigure(fCombo, state=  
    ifelse(tclvalue(e$x) == "", "disabled", "normal"))  
  
  if(tclvalue(e$f) == "") {  
    tkconfigure(muCombo, state="normal")  
    tkconfigure(veCheck, state="disabled")  
  } else {  
    tclvalue(e$mu) <- 0  
    tkconfigure(muCombo, state="disabled")  
    tkconfigure(veCheck, state="normal")  
  }  
}
```

We use the `after` command to repeat a function call every so often. We also define a flag to stop the polling if desired. When polling, we make sure to test for existence of the parent window.

```
updateID <- new.env()  
updateID$flag <- TRUE  
updateID$ID <- NA  
repeatFun <- function() {  
  if(updateID$flag && as.logical(tkwininfo("exists",w))) {
```

```

    updateUI()
    updateID$ID <- tcl("after",1000, repeatFun)
  }
}
repeatFun()

```

## 12.2 Text widgets

Tk provides both single- and multi-line text entry widgets. The section describes both and introduces scrollbars which are often desired for multi-line text entry.

### Entry Widgets

The `ttkentry` constructor provides a single line text entry widget. The widget can be associated with a Tcl variable at construction to facilitate getting and setting the displayed values through its argument `textvariable`. The width of the widget can be adjusted at construction time through the `width` argument. This takes a value for the number of characters to be displayed, assuming average-width characters. The text alignment can be set through the `justify` argument taking values of "left" (the default), "right" and "center". For gathering passwords, the argument `show` can be used, such as with `show="*"`, to show asterisks in place of all the characters.

The following constructs a basic example

```

eVar <- tclVar("initial value")
e <- ttkentry(w, textvariable=eVar)
tkpack(e)

```

We can get and set values using the Tcl variable.

```
tclvalue(eVar)
```

```
[1] "initial value"
```

```
tclvalue(eVar) <- "set value"
```

The `get` command can also be used.

```
tkget(e)
```

```
<Tcl> set value
```

**Indices** The entry widget uses an index to record the different positions within the entry box. This index can be a number (0-based), an *x*-coordinate of the value (`@x`), the values "end" and "insert" to refer to the end of the current text and the insert as set through the keyboard or mouse. The mouse

## 12. TCL Tk: WIDGETS

---

can also be used to make a selection. In this case the indices "sel.first" and "sel.last" describe the selection.

With indices, we can insert text with the *ttkentry* insert command

```
tkinsert(e, "end", "new text")
```

Or, we can delete a range of text, in this case the first 4 characters, using *ttkentry* delete. The first value is the left most index to delete (0-based), the second value the index to the right of the last value deleted.

```
tkdelete(e, 0, 4) # e.g., 123456 -> 56
```

The *ttkentry* icursor command can be used to set the cursor position to the specified index.

```
tkicursor(e, 0) # move to beginning
```

Finally, we note that the selection can be adjusted using the *ttkentry* selection range subcommand. This takes two indices. Like delete, the first index specifies the first character of the selection, the second indicates the character to the right of the selection boundary. The following example would select all the text.

```
tkselection.range(e, 0, "end")
```

The *ttkentry* selection clear subcommand clears the selection and *ttkentry* selection present signals if a selection is currently made.

**Events** Several useful events include <KeyPress> and <KeyRelease> for a key presses and <FocusIn> and <FocusOut> for focus events.

The examples show a bit more about the entry widget. The first shows how styles can be used to adjust the look of an entry widget, and the second how to validate the users data entry in an entry widget.

### Example 12.2: Using styles to adjust the look of an entry widget

This example follows one found at <http://wiki.tcl.tk/18188> where a entry widget has its style set so that it resembles a search box for Mac OS X. This involves setting a background to a gif image to curve the borders of the entry widget and adding a magnifying class icon. We show the commands, but do not offer much description. Much more on using styles is available online in Mark Roseman's Tk tutorial at <http://www.tkdocs.com/tutorial/>.

The use of uuencoded gif images allows one to store image data as text. The variable .imageData contains this for the background image of our entry widget. The following commands create two images, the latter to display when the widget has the focus.

```
QT <- tkimage.create("photo", "::image::search1",  
                    data=.imageData, format="gif -index 0")
```

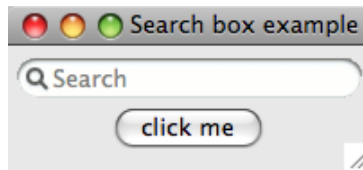


Figure 12.2: Illustration of using Tk styles to configure an entry widget to look like a Mac OS X search entry box.

```
QT <- tkimage.create("photo", "::image::search2",
                    data=.imageData, format="gif -index 1")
```

The following command creates a new style, `Search.field`, which has an image specified for when it is drawn, another for when it has the focus, a border set so that the magnifying glass icon isn't written over with text, and a value of `sticky` so that it stretches from left to right – but not top to bottom – when resized.

```
.Tcl(paste("ttk::style element create Search.field ",
           "image {::image::search1 focus ::image::search2} ",
           "-border {22 7 14} -sticky ew",
           sep=""))
```

```
<Tcl> Search.field
```

The following lays out the entry style. The search field above is given. It has children of padding and textarea which are simply told to stretch in all directions when resized.

```
.Tcl(paste("ttk::style layout Search.entry {",
           "Search.field -sticky nswe -border 1 -children {",
           "Entry.padding -sticky nswe -children {",
           "Entry.textarea -sticky nswe",
           "}}}",
           sep="\n"))
```

Finally, this command configures the background for the `Search.entry` style.

```
tcl("ttk::style", "configure", "Search.entry",
    background="#b2b2b2")
```

At this point, we can set up a basic GUI to illustrate the use. First a toplevel window and themed frame are defined.

```
w <- tktoplevel()
tkwm.title(w, "Search box example")
f <- ttkframe(w, padding=c(3,3,3,12)); tkpack(f, expand=TRUE, fill="both")
```

## 12. TCL Tk: WIDGETS

---

The entry widget has the style set through its style argument, as follows.

```
eVar <- tclVar("")
e <- ttkentry(f, textvariable=eVar, style="Search.entry")
tkpack(e)
b <- ttkbutton(f, text="click me"); tkpack(b)
```

At this point, we define two bindings to insert some initial text into the widget to give direction to the user. The first handler is called on the expose event, when the widget is first drawn, and simply sets the initial text in gray.

```
initialText <- "Search"
tkbind(e,"<Expose>", function(W) {
  cur <- as.character(tcl(W,"get"))
  if(length(cur) == 0 || cur == "") {      # first time
    tclvalue(eVar) <- initialText
    tkconfigure(W, foreground="gray50")
  } else {
    tkconfigure(W, foreground="black")
  }
})
```

The second binding is to catch the first edit of the entry widget to replace the initial text with the value input by the user.

```
tkbind(e,"<FocusIn>", function(W) {
  cur <- as.character(tcl(W, "get"))
  if(cur == initialText) {                # searching for initial text?
    tclvalue(eVar) <- ""
    tkconfigure(W, foreground="black")
  }
})
```

### Example 12.3: Using validation for dates

There is no native calendar widget in tcltk. This example shows how one can use the validation framework for entry widgets to check that user-entered dates conform to an expected format.

Validation happens in a few steps. A validation command is assigned to some event. This call can come in two forms. Prevalidation is when a change is validated prior to being committed, for example when each key is pressed. Revalidation is when the values is checked after it is sent to be committed, say when the entry widget loses focus or the enter key is pressed.

When a validation command is called it should check whether the current state of the entry widget is valid or not. If valid, it returns a value of TRUE and FALSE otherwise. These need to be Tcl Boolean values, so in the following,



the command `tcl("eval","TRUE")` (or `"FALSE"`) is used. If the validation command returns `FALSE`, then a subsequent call to an invalidation command is made.

For each callback, a number of substitution values are possible, in addition to the standard ones such as `W` to refer to the widget. These are: `d` for the type of validation being done: 1 for insert prevalidation, 0 for delete prevalidation, or -1 for revalidation; `i` for the index of the string to be inserted or deleted or -1; `P` for the new value if the edit is accepted (in prevalidation) or the current value in revalidation; `s` for the value prior to editing; `S` for the string being inserted or deleted; `v` for the current value of validate and `V` for the condition that triggered the callback.

In the following callback definition we use `W` so that we can change the entry text color to black and format the data in a standard manner and `P` to get the entry widget's value just prior to validations.

To begin, we define some patterns for acceptable date formats.

```
datePatterns <- c()
for(i in list(c("%m","%d","%Y"),
              c("%m","%d","%y"))) {
  for(j in c("/","-", " "))
    datePatterns[length(datePatterns)+1] <- paste(i,sep=" ", collapse=j)
}
```

```
isValidDate <- function(W, P) { # P is the current value
  for(i in datePatterns) {
    date <- try( as.Date(P, format=i), silent=TRUE)
    if(!inherits(date, "try-error") && !is.na(date)) {
      tkconfigure(W,foreground="black") # consult style?
      tkdelete(W,"0","end")
      tkinsert(W,0, format(date, format="%m/%d/%y"))
      return(tcl("expr","TRUE"))
    }
  }
  return(tcl("expr","FALSE"))
}
```

This callback for when a date is invalid simply sets the text color to red.

```
indicateInvalidDate <- function(W) {
  tkconfigure(W,foreground="red")
  tcl("expr","TRUE")
}
```

Now we define a simple GUI to show the entry widget.

```
w <- tktoplevel(); tkwm.title(w,"Validation of date example")
f <- ttkframe(w, padding=c(3,3,3,12)); tkpack(f, expand=TRUE, fill="both")
tkpack(ttklabel(f, text="Enter a date mm/dd/yy"), side="left")
```

The `validate` argument is used to specify when the validation command should be called. This can be a value of "none" for validation when called through the validation command; "key" for each key press; "focusin" for when the widget receives the focus; "focusout" for when it loses focus; "focus" for both of the previous; and "all" for any of the previous. We use "focusout" below, so also give a button widget so that the focus can be set elsewhere.

```
e <- ttkentry(f, validate="focusout",
              validatecommand=isValidDate,
              invalidcommand=indicateInvalidDate)
tkpack(e, side="left")
b <- ttkbutton(f, text="click")           # something to focus on
tkpack(b, side="bottom")
```

### Scrollbars

Tk has several scrollable widgets – those that use scrollbars. Widgets which accept a scrollbar (without too many extra steps) have the options `xscrollcommand` and `yscrollcommand`. To use scrollbars in `tcltk` requires two steps: the scrollbars must be constructed and bound to some widget, and that widget must be told it has a scrollbar. This way changes to the widget can update the scrollbar and vice versa. Suppose, `parent` is a container and `widget` has these options, then the following will set up both horizontal and vertical scrollbars.

The scrollbars are defined as follows using the `orient` option and a command of the following form.

```
xscr <- ttkscrollbar(parent, orient="horizontal",
                    command=function(...) tkxview(widget, ...))
yscr <- ttkscrollbar(parent, orient="vertical",
                    command=function(...) tkxview(widget, ...))
```

The view commands sets what part of the widget is being shown.

To link the widget back to the scrollbar, the `set` command is used in a callback to the scroll command. For this example we configure the options after the widget is constructed, but this can be done at the time of construction as well. Again, the command takes a standard form:

```
tkconfigure(widget,
            xscrollcommand=function(...) tkset(xscr,...),
            yscrollcommand=function(...) tkset(yscr,...))
```

Although scrollbars can appear anywhere, the conventional place is on the right and lower side of the parent. The following adds scrollbars using the grid manager. The combination of weights and stickiness below will have the scrollbars expand as expected if the window is resized.

```
tkgrid(widget, row=0, column=0, sticky="news")
```

```
tkgrid(yscr,row=0,column=1, sticky="ns")
tkgrid(xscr, row=1, column=0, sticky="ew")
tkgrid.columnconfigure(parent, 0, weight=1)
tkgrid.rowconfigure(parent, 0, weight=1)
```

Although a bit tedious, this gives the programmer some flexibility in arranging scrollbars. To avoid doing all this in the sequel, we turn the above into function `addScrollbars` (not shown).

## Multi-line Text Widgets

The `tktext` widget creates a multi-line text editing widget. If constructed with no options but a parent container, the widget can have text entered into it by the user.

The text widget is not a themed widget, hence has numerous arguments to adjust its appearance. We mention a few here and leave the rest to be discovered in the manual page (along with much else). The argument `width` and `height` are there to set the initial size, with values specifying number of characters and number of lines (not pixels). The actual size is font dependent, with the default for 80 by 24 characters. The `wrap` argument, with a value from "none", "char", or "word", indicates if wrapping is to occur and if so, does it happen at any character or only a word boundary. The argument `undo` takes a logical value indicating if the undo mechanism should be used. If so, the subcommand `tktext edit` can be used to undo a change (or the control-z keyboard combination).

**Indices** As with the entry widget, several commands take indices to specify position within the text buffer. Only for the multi-line widget both a line and character are needed in some instances. These indices may be specified in many ways. One can use row and character numbers separated by a period in the pattern `line.char`. The line is 1-based, the column 0-based (e.g., 1.0 says start on the 1st row and first character). In general, one can specify any line number and character on that line, with the keyword `end` used to refer to the last character on the line. If the *x-y* position of the spot is known (through percent substitutions say) the form `x,y` may be used. Text buffers may carry transient marks, in which case the use of this mark indicates the next character after the mark. The mark `insert` tracks the insertion point in the text buffer were the user to begin typing, whereas the mark `current` follows the character closest to the mouse position. As well, pieces of text may be tagged. The format `tag.first` and `tag.last` index the range of the tag `tag`. Marks and tags are described in the following.

Indices can also be adjusted relative to the above. This adjustment can be by a number of characters (`chars`), index positions (`indices`) or lines. For example, `insert + 1 lines` refers to 1 line under the insert point. The values `linestart`, `lineend`, `wordstart` and `wordend` are also available. For

instance, `insert` `linestart` is the beginning of the line from the insert point, while `end -1 wordstart` and `end - 1 chars wordend` refer to the beginning and ending of the last word in the buffer. (The `end` index refers to the character just after the new line so we go back 2 steps.)

**Inserting text** Inserting text can be done through the *tktext* `insert` subcommand by specifying first the index then the text to add. One can use `\n` to add new lines.

```
tkinsert(t, "end", "more text\n new line")
```

Images and other windows can be added to a text buffer, but we do not discuss that here.

**Panning the buffer: tksee** After text is inserted, the visible part of buffer may not be what is desired. The *tktext* `see` sub command is used to position the buffer on the specified index, its lone argument.

**Getting text** The *tktext* `get` subcommand is used to retrieve the text in the buffer. Coercion to character should be done with `tclvalue` and not `as.character` to preserve the distinction between spaces and line breaks.

```
value <- tkget(t, "1.0", "end")
as.character(value)                # wrong way
```

```
[1] "more" "text" "new" "line"
```

```
tclvalue(value)
```

```
[1] "more text\n new line\n"
```

**tags** Tags are a means to assign a name to characters within the text buffer. Tags may be used to adjust the foreground, background and font properties of the tagged characters from those specified globally at the time of construction of the widget, or configured thereafter. Tags can be set when the text is inserted, as with

```
tkinsert(t, "end", "last words", "lastWords") # lastWords is tag
```

Tags can be set after the text is added through the *tktext* `tag add` subcommand using indices to specify location. The following marks the first word:

```
tktag.add(t, "firstWord", "1.0 wordstart", "1.0 wordend")
```

The *tktext* `tag configure` can be used to configure properties of the tagged characters, for example:

```
tktag.configure(t, "firstWord", foreground="red", font="helvetica 12 bold")
```

There are several other configuration options for a tag. A cryptic list can be produced by calling the subcommand *tktext* tag configure without a value for configuration.

**selection** The current selection, if any, is indicated by the *sel* tag, with *sel.first* and *sel.last* providing indices to refer to the selection. (Provided the option *exportSelection* was not modified.) These tags can be used with *tkget* to retrieve the currently selected text. An error will be thrown if there is no current selection. To check if there is a current selection, the following may be used:

```
hasSelection <- function(W) {
  ranges <- tclvalue(tcl(W, "tag", "ranges", "sel"))
  length(ranges) > 1 || ranges != ""
}
```

The cut, copy and paste commands are implemented through the functions *tk\_textCut*, *tk\_textCopy* and *tk\_textPaste*. Their lone argument is the text widget. These work with the current selection and insert point. For example to cut the current selection, one has

```
tcl("tk_textCut", t)
```

**marks** Tags mark characters within a buffer, marks denote positions within a buffer that can be modified. For example, the marks *insert* and *current* refer to the position of the cursor and the current position of the mouse. Such information can be used to provide context-sensitive popup menus, as in this code example:

```
popupContext <- function(W, x, y) {
  ## or use paste("@", x, ", ", y, sep="")
  cur <- tkget(W, "current wordstart", "current wordend")
  cur <- tclvalue(cur)
  popupContextMenuFor(cur, x, y)      # some function
}
```

To assign a new mark, one uses the *tktext* mark set subcommand specifying a name and a position through an index. Marks refer to spaces within characters. The gravity of the mark can be left or right. When right (the default), new text inserted is to the left of the mark. For instance, to keep track of an initial insert point and the current one, the initial point (marked *leftlimit* below) can be marked with

```
tkmark.set(t, "leftlimit", "insert")
tkmark.gravity(t, "leftlimit", "left")      # keep on left
tkinsert(t, "insert", "new text")
tkget(t, "leftlimit", "insert")
```

```
<Tcl> new text
```

The use of the subcommand *tktext* mark gravity is done so that the mark attaches to the left most character at the insert point. The rightmost one changes as more text is inserted, so would make a poor choice.

**The edit command** The subcommand *tktext* edit can be used to undo text. As well, it can be used to test if the buffer has been modified, as follows:

```
tcl(t, "edit", "undo")           # no output
tcl(t, "edit", "modified")       # 1 = TRUE
```

```
<Tcl> 1
```

**Events** The text widget has a few important events. The widget defines virtual events `<<Modified>>` and `<<Selection>>` indicating when the buffer is modified or the selection is changed. Like the single-line text widget, the events `<KeyPress>` and `<KeyRelease>` indicate key activity. The %*-*substitution *k* gives the keycode and *K* the key symbol as a string (*N* is the decimal number).

#### Example 12.4: Displaying commands in a text buffer

This example shows how a text buffer can be used to display the output of R commands, using an approach modified from Sweave. The following function does the work of evaluating a command chunk then inserting the values into the text buffer, using different markup to indicate commands from output.

```
evalCmdChunk <- function(t, cmds) {
  ## create tags
  tktag.configure(t, "commandTag", foreground="blue",
                 font="courier 12 italic")
  tktag.configure(t, "outputTag", font="courier 12")
  tktag.configure(t, "errorTag", foreground="red",
                 font="courier 12 bold")

  cmdChunks <- try(parse(text=cmds), silent=TRUE)
  if(inherits(cmdChunks,"try-error")) {
    tkinsert(t, "end", "Error", "errorTag") # add tag for markup
  }

  for(cmd in cmdChunks) {
    dcmd <- deparse(cmd, width.cutoff = 0.75 * getOption("width"))
    command <-
      paste(getOption("prompt"),
            paste(dcmd, collapse=paste("\n", getOption("continue")),
                  sep="")),
```

```

        sep="", collapse="")
    tkinsert(t, "end", command, "commandTag")
    tkinsert(t, "end", "\n")
    ## output, should check for errors in eval!
    output <- capture.output(eval(cmd, envir=.GlobalEnv))
    output <- paste(output, collapse="\n")
    tkinsert(t, "end", output, "outputTag")
    tkinsert(t, "end", "\n")
  }
}

```

We envision this as a piece of a larger GUI which generates the commands `t` evaluate. For this example though, we make a simple GUI.

```

w <- tktoplevel(); tkwm.title(w, "Text buffer example")
f <- ttkframe(w, padding=c(3,3,3,12))
tkpack(f, expand=TRUE, fill="both")
t <- tktext(f, width=80, height = 24) # default size
addScrollbars(f, t)

```

This is how it can be used.

```
evalCmdChunk(t, "2 + 2; lm(mpg ~ wt, data=mtcars)")
```

## 12.3 Treeview widget

The themed treeview widget can be used to display rectangular data, like a data frame, or heirarchical data. The usage is similar for each beyond the need to indicate the heirarchical structure of a tree.

### Rectangular data

Rectangular data has a row and column structure. In R, data frames are internally kept in terms of their columns which all must have the same type. The treeview widget is different, it stores all data as character data and one interacts with the data row by row.

The `ttktreeview` constructor creates the tree widget. There is no separate model for this widget, but there is a means to filter what is displayed. The argument `columns` is used to specify internal names for the columns and indicate the number of columns. A value of `1:n` will work here unless explicit names are desired. The argument `displaycolumns` is used to control which of the columns are actually display. The default is `"all"`, but a vector of indices or names can be given. The size of the widget is specified two different ways. The `height` argument is used to adjust the number of visible rows. The width of the widget is determined by the combined widths of each column, whose adjustments are mentioned later. The user may select one or more rows with the mouse, as controlled by the argument `selectmode`. Multiple

rows may be selected with the default value of "extended", a restriction to a single row is specified with "browse", and no selection is possible if this is given as none. The treeview widget has an initial column for showing the tree-like aspect with the data. This column is referenced by #0. The show argument controls whether this column is shown. A value of "tree" leaves just this column shown, "headings" will show the other columns, but not the first, and the combined value of tree headings will display both (the default). Additionally, the treeview is a scrollable widget, so has the arguments xscrollcommand and yscrollcommand for specifying scrollbars.

If f is a frame, then the following call will create a widget with just one column showing 25 rows, like the older, non-themed, listbox widget of Tk.

```
tr <- ttktreeview(f,
                  columns=1,           # column identifier is "1"
                  show="headings",     # not "#0"
                  height=25)
addScrollbars(f, tr)                  # scrollbar function
```

**Column properties** Once the widget is constructed, its columns can be configured on a per-column basis. Columns can be referred to by the name specified through the columns argument or by number starting at 1 with "#0" referring to the tree column. The column headings can be set through the *ttktreeview* heading subcommand. The heading, similar to the button widget, can be text, an image or both. The text placement of the heading may be positioned through the anchor option. For example, this command will center the text heading of the one column:

```
tcl(tr, "heading", 1, text="Host", anchor="center")
```

The *ttktreeview* column subcommand can be used to adjust a column's properties including the size of the column. The option width is used to specify the pixel width of the column (the default is large); As the widget may be resized, one can specify the minimum column width through the option minwidth. When more space is allocated to the tree widget, than is requested by the columns, column with a TRUE value specified to the option stretch are resized to fill the available space. Within each column, the placement of each entry within a cell is controlled by the anchor option, using the compass points.

For example, this command will adjust properties of the lone column of tr:

```
tcl(tr, "column", 1, width=400, stretch=TRUE, anchor="w")
```

**Adding values** Values can be added to the widget through the *ttktreeview* insert parent item [text] [values] subcommand. This requires the specification



of a parent (always "" for rectangular data) and an index for specifying the location of the new child amongst the previous children. The special value "end" indicates placement after all other children, as would a number larger than the number of children. A value of 0 or a negative value would put it at the beginning.

There are a number of options for each row. If column #0 is present, the text option is used to specify the text for the tree row and the option image can be given to specify an image to place to the left of the text value. For filling in the columns the values option is used. If there is a single column, like the current example, care needs to be taken that values separated by spaces are quoted (or in braces), otherwise, they will be split on spaces and treated like a vector of values truncated on the first one. Finally, we mention that tag option for insert that can be used to specify a tag for the inserted row. This allows the use of the subcommand *ttktreeview* tag configure to configure the foreground color, background color, font or image of an item.

In the example this is how we can add a list of possible CRAN mirrors to the treeview display.

```
x <- getCRANmirrors()
Host <- paste(' ', x$Host, ' ', sep=" ") # add quotes!
shade <- c("none", "gray")           # tag names
for(i in 1:length(Host))
  ID <- tkinset(tr, "", "end", tag=shade[i %% 2], values=Host[i])
tktag.configure(tr, "gray", background="gray95") # shade rows
```

**Item IDs** Each row has a unique item ID generated by the widget when a row is added. The base ID is "" (why this is specified for the value of parent for rectangular data). For rectangular displays, the list of all IDs may be found through the *ttktreeview* children sub command, which we will describe in the next section. Here we see it used to find the children of the root. As well, we show how the *ttktreeview* index command returns the row index.

```
children <- tcl(tr, "children", "")
(children <- head(as.character(children))) # as.character
```

```
[1] "I001" "I002" "I003" "I004" "I005" "I006"
```

```
apply(children, function(i) tclvalue(tkindex(tr, i)))
```

```
I001 I002 I003 I004 I005 I006
"0"  "1"  "2"  "3"  "4"  "5"
```

**Retreiving values** The *ttktreeview* item subcommand can be used to get the values and other properties stored for each row. One specifies the item and the corresponding option:

## 12. TCL Tk: WIDGETS

---

```
x <- tcl(tr, "item", children[1], "-values") # no tkitem
as.character(x)
```

```
[1] "Patan.com.ar, Buenos Aires"
```

The value returned from the `item` command can be difficult to parse, as Tcl introduces braces for grouping. The coercion through `as.character` works much better at extracting the individual columns. A possible alternative to using the `item` command, is to instead keep the original data frame and use the index of the item to extract the value from the original.

**Moving and deleting items** The `movettktreeview` subcommand can be used to replace a child. As with the `insert` command, a parent and an index for where the new child is to go among the existing children is given. The item to be moved is referred to by its ID. The `ttktreeview delete` and `ttktreeviews detach` can be used to remove an item from the display, as specified by its ID. The latter command allows for the item to be reinserted at a later time.

**Events and callbacks** In addition to the keyboard events `<KeyPress>` and `<KeyRelease>` and the mouse events `<ButtonPress>`, `<ButtonRelease>` and `<Motion>`, the virtual event `<<TreeviewSelect>>` is generated when the selection changes. The current selection marks 0, 1 or more than 1 items if "extended" is given for the `selectmode` argument. The `ttktreeview selection` command will return the current selection. If converted to a string using `as.character` this will be a 0-length character vector, or a character vector of the selected item IDs. Further subcommands `set`, `add`, `remove`, and `toggle` can be used to adjust the selection programatically.

Within a key or mouse event callback, the selected column and row can be identified by position, as illustrated in this example callback.

```
callbackExample <- function(W, x, y) {
  col <- as.character(tkidentify(W, "column", x, y))
  row <- as.character(tkidentify(W, "row", x, y))
  ## do something ...
}
```

### Example 12.5: Filtering a table

We illustrate the above with a slightly improved GUI for selecting a CRAN mirror. This adds in a text box to filter the possibly large display of items to avoid scrolling through a long list.

```
df <- getCRANmirrors()[, c(1,2,5,4)]
```

We use a text entry widget to allow the user to filter the values in the display as the user types.

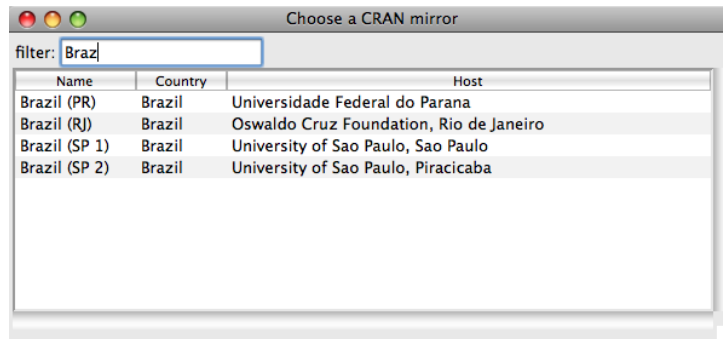


Figure 12.3: Using `ttktreeview` to show various CRAN sites. This illustration adds a search-like box to filter what repositories are displayed for selection.

```
w <- tkoplevel()
tkwm.title(w, "Choose a CRAN mirror")
f <- ttkframe(w, padding=c(3,3,3,12))
tkpack(f, expand=TRUE, fill="both")
f0 <- ttkframe(f); tkpack(f0, fill="x")
l <- ttklabel(f0, text="filter:"); tkpack(l, side="left")
filterVar <- tclVar("")
filterEntry <- ttkentry(f0, textvariable=filterVar)
tkpack(filterEntry, side="left")
```

The treeview will only show the first three columns of the data frame, although we store the fourth which contains the URL.

```
f1 <- ttkframe(f); tkpack(f1, expand=TRUE, fill="both")
tr <- ttktreeview(f1, columns=1:ncol(df),
                  displaycolumns = 1:(ncol(df) - 1),
                  show = "headings",      # not "tree"
                  selectmode = "browse") # single selection
addScrollbars(f1, tr)
```

We configure the column widths and titles as follows:

```
widths <- c(100, 75, 400)          # hard coded
nms <- names(df)
for(i in 1:3) {
  tcl(tr, "heading", i, text=nms[i])
  tcl(tr, "column", i, width=widths[i], stretch=TRUE, anchor="w")
}
```

This following helper function is used to fill in the widget with values from a data frame.

```
fillTable <- function(tr, df) {
```

## 12. TCL Tk: WIDGETS

---

```
children <- as.character(tcl(tr, "children", ""))
for(i in children) tcl(tr, "delete", i)
shade <- c("none", "gray")
for(i in seq_len(nrow(df)))
  tcl(tr, "insert", "", "end", tag=shade[i %% 2],text="",
      values=unlist(df[i,]))
tktag.configure(tr, "gray", background="gray95")
}
```

The initial call populates the table from the entire data frame.

```
fillTable(tr, df)
```

The filter works by grepping the user input against the host value. We bind to `<KeyRelease>` (and not `<KeyPress>`) so we capture the last keystroke.

```
curInd <- 1:nrow(df)
tkbind(filterEntry, "<KeyRelease>", function(W, K) {
  val <- tclvalue(tkget(W))
  possVals <- apply(df,1, function(...) paste(..., collapse=" "))
  ind<- grep(val, possVals)
  if(length(ind) == 0) ind <- 1:nrow(df)
  fillTable(tr, df[ind,])
})
```

This binding is for capturing a users selection through a double-click event. In the callback, we set the CRAN option then withdraw the window.

```
tkbind(tr, "<Double-Button-1>", function(W, x, y) {
  sel <- as.character(tcl(W, "identify", "row", x, y))
  vals <- tcl(W, "item", sel, "-values")
  URL <- as.character(vals)[4] # not tclvalue
  repos <- getOption("repos")
  repos["CRAN"] <- gsub("/$", "", URL[1L])
  options(repos = repos)
  tkwm.withdraw(tkwininfo("toplevel", W))
})
```

**Editing cells of a table** There is no native widget for editing the cells of tabular data, as is provided by the `edit` method for data frames. The `tktable` widget (<http://tktable.sourceforge.net/>) provides such an add-on to the base Tk. We don't illustrate its usage here, as we keep to the core set of functions provided by Tk. However, we note that the `gdf` function of `gWidgetstcltk` provides an example of how it can be used.

### Heirarchical data

Specifying tree-like or heirarchical data is nearly identical to specifying rectangular data for the `ttktreeview` widget. The widget provides column #0 to

display this extra structure. If an item, except the root, has children, a trigger icon to expand the tree is shown. This is in addition to any text and/or an icon that is specified. Children are displayed in an indented manner to indicate the level of ancestry they have relative to the root. To insert hierarchical data in to the widget the same *ttktreeview* insert subcommand is used, only instead of using the root item, "", as the parent item, one uses the item ID corresponding to the desired parent. If the option `open=TRUE` is specified to the insert subcommand, the children of the item will appear, if `FALSE`, the user can click the trigger icon to see the children. The programmer can use the *ttktreeview* item to configure this state. When the parent item is opened or closed, the virtual events `<<TreeviewOpen>>` and `<<TreeviewClose>>` will be signaled.

**Traversal** Once a tree is constructed, the programmer can traverse through the items using the subcommands *ttktreeview* parent *item* to get the ID for the parent of the item; *ttktreeview* prev *item* and *ttktreeview* next *item* to get the immediate siblings of the item; and *ttktreeview* children *item* to return the children of the item. Again, the latter one will produce a character vector of IDs for the children when coerced to character with `as.character`.

#### Example 12.6: Using the treeview widget to show an XML file

This example shows how to display the hierarchical structure of an XML document. The XML package is used to parse the document.

We use the XML library to parse a document from the internet. This example uses just a few functions from this library: The `(htmlTreeParse)` (similar to `xmlInternalTreeParse`) to parse the file, `xmlRoot` to find the base node, `xmlName` to get the name of a node, `xmlValue` to get an associated value, and `xmlChildren` to return any child nodes of a node.

```
library(XML)
fileName <- "http://www.omegahat.org/RFXML/shortIntro.html" # XML's doc
QT <- function(...) {} # quiet next call
doc <- htmlTreeParse(fileName, useInternalNodes=TRUE, error=QT)
root <- xmlRoot(doc)
```

Our GUI is primitive, with just a treeview instance added.

```
w <- tkoplevel()
tkwm.title(w, "Treeview example with XML")
f <- ttkframe(w, padding=c(3,3,3,12))
tkpack(f, expand=TRUE, fill="both")
tr <- ttktreeview(f, displaycolumns="#all", columns=1)
addScrollbars(f, tr)
```

We configure our columns headers and set a minimum width below. Recall, the tree column is designated "#0".

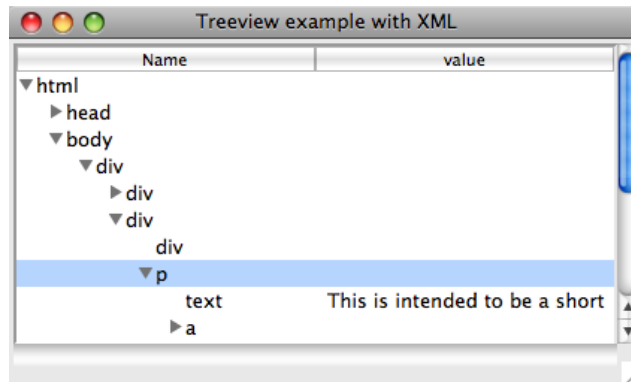


Figure 12.4: Illustration of using `ttktreeview` widget to show heirarchical data returned from parsing an HTML document with the XML package.

```
tcl(tr, "heading", "#0", text="Name")
tcl(tr, "column", "#0", minwidth=20)
tcl(tr, "heading", 1, text="value")
tcl(tr, "column", 1, minwidth=20)
```

To map the tree-like structure of the XML document into the widget, we define the following function to recursively add to the treeview instance. We only add to the value column (through the `values` option) when the node does not have children, `dren`. We use `do.call`, as a convenience, to avoid constructing two different calls to the `insert` subcommand. The `quoteIt` function used is not shown, but similar to `shQuote` only escaping with double quotes, as single quotes are treated differently by Tcl. (Otherwise the `ttktreeview` widget will split values on spaces.)

```
insertChild <- function(tr, node, parent="") {
  l <- list(tr, "insert", parent, "end", text=xmlName(node))
  children <- xmlChildren(node)
  if(length(children) == 0) {
    # add in values
    values <- paste(xmlValue(node), sep=" ", collapse=" ")
    values <- gsub("\n", " ", values) # treeview doesn't like
    values <- quoteIt(values)        # \n and spaces
    l$values <- values
  }
  treePath <- do.call("tcl", l)

  if(length(children))
    for(i in children) insertChild(tr, i, treePath)
}
insertChild(tr, root)
```

At this point, the GUI will allow one to explore the structure of the XML file. We continue this example to show two things of general interest, but are really artificial for this example.

**Drag and drop** First, we show how one might introduce drag and drop to rearrange the rows. We begin by defining two global variables that store the row that is being dragged and a flag to indicate if a drag event is ongoing.

```
.selectedID <- ""                                # globals
.dragging <- FALSE
```

We provide callbacks for three events: a mouse click, mouse motion and mouse release. This first callback sets the selected row on a mouse click.

```
tkbind(tr, "<Button-1>", function(W,x,y) {
  .selectedID <- as.character(tcl(W, "identify","row", x, y))
})
```

The motion callback configures the cursor to indicate a drag event and sets the dragging flag. One might also put in code to highlight any drop areas.

```
tkbind(tr, "<B1-Motion>", function(W, x, y, X, Y) {
  tkconfigure(W, cursor="diamond_cross")
  .dragging <- TRUE
})
```

When the mouse button is released we check that the widget we are over is indeed the tree widget. If so, we then move the rows. One can't move a parent to be a child of its own children, so we wrap the *ttktreeview* move sub command within try. The move command places the new value as the first child of the item it is being dropped on. If a different action is desired, the "0" below would need to be modified.

```
tkbind(tr, "<ButtonRelease-1>", function(W, x, y, X, Y) {
  if(.dragging && .selectedID != "") {
    w = tkwinfo("containing", X, Y)
    if(as.character(w) == as.character(W)) {
      dropID <- as.character(tcl(W, "identify","row", x, y))
      try(tkmove(W, .selectedID, dropID, "0"), silent=TRUE)
    }
  }
  .dragging <- FALSE; .selectedID <- "" # reset
})
```

**Walking the tree** Our last item of general interest is a function that shows one way to walk the structure of the treeview widget to generate a list representing the structure of the data. A potential use of this might be to allow

a user to rearrange an XML document through drag and drop. The subcommand `ttktreeview children` proves useful here, as it is used to identify the heirarchical structure. When there are children a recursive call is made.

```
treeToList <- function(tr) {  
  l <- list()  
  walkTree <- function(child, l) {  
    l$name <- tclvalue(tcl(tr,"item", child, "-text"))  
    l$value <- tclvalue(tcl(tr,"item", child, "-values"))  
    children <- as.character(tcl(tr, "children", child))  
    if(length(children)) {  
      l$children <- list()  
      for(i in children)  
        l$children[[i]] <- walkTree(i, list()) # recurse  
    }  
    return(l)  
  }  
  l <- walkTree("",l)  
  return(l)  
}
```

## 12.4 Menus

Menu bars and popup menus in Tk are constructed with `tkmenu`. The parent argument depends on what the menu is to do. A toplevel menu bar, such as appears at the top of a window has a toplevel window as its parent; a sub-menu of a menu bar uses the parent menu; and a popup menu uses a widget. The menu widget in Tk has an option to be “torn off.” This features was at one time common in GUIs, but now is rarely seen so it is recommended that this option be disabled. The `tearoff` option can be used at construction time to override the default behaviour. Otherwise, the following command will do so globally:

```
tcl("option","add","*tearOff",0)      # disable tearoff menus
```

A toplevel menu bar is to attached to top-level window using `tkconfigure` to set the menu property of the window. For the aqua Tk libraries for Mac OS X, this menu will appear on the top menu bar when the window has the focus. For other operating systems, it appears at the top of the window. For Mac OS X, a default menu bar with no relationship to your application will will shown if a menu is not provided for a toplevel window. Testing for native Mac OS X may be done via the following function:

```
usingMac <- function()  
  as.character(tcl("tk", "windowingsystem")) == "aqua"
```

The `tkpopup` function facilitates the creation of a popup menu. This function has arguments for the menu bar, and the postion where the menu should



be popped up. For example, the following code will bind a popup menu, `pmb`, to the right click event for a button `b`. As Mac OS X may not have a third mouse button, and when it does it refers to it differently, the callback is bound conditionally to different events.

```
doPopup <- function(X, Y) tkpopup(pmb, X, Y) # define call back
if (usingMac()) {
  tkbind(b, "<Button-2>", doPopup)           # right click
  tkbind(b, "<Control-1>", doPopup)          # Control + click
} else {
  tkbind(b, "<Button-3>", doPopup)
}
```

**Adding submenus and action items** `Menus` shows a heirarchical view of action items. Items are added to a menu through the `tkmenu` `add` subcommand. The nested structure of menus is achieved by specifying a `tkmenu` object as an item. The `tkmenu` `add cascade` subcommand is used for this. The option `label` is used to label the menu and the menu option to specify the sub-menu.

Grouping of similar items can be done through nesting, or on occasion through visual separation. The latter is implemented with the `tkmenu` `add separator` subcommand.

There are a few different types of action items that can be added.

An action item is one associated with a command. The simplest case is a label in the menu that activates a command when selected through the mouse. The `tkmenu` `add command` (through `tkadd(widget, "command", ...)`) allows one to specify a label, a command and optionally an image with a value for `compound` to adjust its layout. (Images are not shown in Mac OS X.) Action commands may possibly called for different widgets, so the use of percent substitution is discouraged here. One can also specify that a keyboard accelerator be displayed through the option `accelerator`, but a separate callback must listen for this combination.

Action items may also be checkboxes. To create one, the subcommand `tkmenu` `add checkbutton` is used. The available arguments include `label` to specify the text, `variable` to specify a tcl variable to store the state, `onvalue` and `offvalue` to specify the state to the tcl variable, and `command` to specify a call back when the checked state is toggled. The initial state is set by the value in the Tcl variable.

Additionally, action items may be radiobutton groups. These are specified with the subcommand `tkmenu` `add radiobutton`. The `label` option is used to identify the entry, `variable` to set a text variable and to group the buttons that are added, and `command` to specify a command when that entry is selected.

Action items can also be placed after an item, rather than at the end using the *tkmenu* `insert` command `index` subcommand. The index may be specified numerically with 0 being the first item for a menu. More conveniently the index can be determined by specifying a pattern to match the menu's labels.

**Set state** The `state` option is used to retrieve and set the current state of the a menu item. This value is typically `normal` or `disabled`, the latter to indicate the item is not available. The state can be set when the item is added or configured after that fact through the *tkmenu* `entryconfigure` command. This function needs the menu bar specified and the item specified as an index or pattern to match the labels.

### Example 12.7: Simple menu example

This example shows how one might make a very simple code editor using a text-entry widget. We use the *svMisc* package, as it defines a few GUI helpers which we use.

```
library(svMisc)                                # for some helpers
showCmd <- function(cmd) writeLine(captureAll(Parse(cmd)))
```

We create a simple GUI with a top-level window containing the text entry widget.

```
w <- tktoplevel()
tkwm.title(w, "Simple code editor")
f <- ttkframe(w, padding=c(3,3,3,12));
tkpack(f, expand=TRUE, fill="both")
tb <- tktext(f, undo=TRUE)
addScrollbars(f, tb)
```

We create a `toplevel` menu bar, `mb`, and attach it to our `toplevel` window. Then we create a file and edit submenu.

```
mb <- tkmenu(w); tkconfigure(w, menu=mb)
fileMenu <- tkmenu(mb)
tkadd(mb, "cascade", label="File", menu=fileMenu)
editMenu <- tkmenu(mb)
tkadd(mb, "cascade", label="Edit", menu=editMenu)
```

To these sub menu bars, we add action items. First a command to evaluate the contents of the buffer.

```
tkadd(fileMenu, "command", label="Evaluate buffer",
      command = function() {
        curVal <- tclvalue(tkget(tb, "1.0", "end"))
        showCmd(curVal)
      })
```

Then a command to evaluate just the current selection

```
tkadd(fileMenu, "command", label="Evaluate selection",
      state="disabled",
      command = function() {
        curSel <- tclvalue(tkget(tb, "sel.first", "sel.last"))
        showCmd(curSel)
      })
```

Finally, we end the file menu with a quit action.

```
tkadd(fileMenu, "separator")
tkadd(fileMenu, "command", label="Quit", command=function() tkdestroy(w))
```

The edit menu has an undo and redo item. For illustration purposes we add an icon to the undo item.

```
img <- system.file("images/up.gif", package="gWidgets")
QT <- tkimage.create("photo", "::img::undo",
                    file = img)
tkadd(editMenu, "command", label="Undo",
      image="::img::undo", compound="left",
      command = function() tcl(tb, "edit", "undo"))
tkadd(editMenu, "command", label="Redo",
      command = function() tcl(tb, "edit", "redo"))
```

We now define a function to update the user interface to reflect any changes.

```
updateUI <- function() {
  states <- c("disabled", "normal")
  ## selection
  hasSelection <- function(W) {
    ranges <- tclvalue(tcl(W, "tag", "ranges", "sel"))
    length(ranges) > 1 || ranges != ""
  }
  ## by index
  tkentryconfigure(fileMenu, 1, state=states[hasSelection(tb) + 1])
  ## undo — if buffer modified, assume undo stack possible
  ## redo — no good check for redo
  canUndo <- function(W) as.logical(tcl(W, "edit", "modified"))
  tkentryconfigure(editMenu, "Undo", state=states[canUndo(tb) + 1])
  tkentryconfigure(editMenu, "Redo", state=states[canUndo(tb) + 1])
}
```

We update the UI during idle times. The `after` command can be used to run a command after a period of time. The following repeats a function every second until the toplevel window no longer exists.

```
afterID <- ""
repeatCall <- function(ms=1000, f, w) {
  afterID <- tcl("after", ms, function() {
    if(as.logical(tkwininfo("exists", w))) {
```

## 12. TCL Tk: WIDGETS

---

```
f()
  afterID <- repeatCall(ms, f, w)
} else {
  tcl("after", "cancel", afterID)
}
})
}
repeatCall(100, updateUI, w)
```

We now add an accelerator entry to the menubar and a binding to the top-level window for the keyboard shortcut.

```
if(usingMac()) {
  tkentryconfigure(editMenu, "Undo", accelerator="Cmd-z")
  tkbind(w, "<Option-z>", function() tcl(tb, "edit", "undo"))
} else {
  tkentryconfigure(editMenu, "Undo", accelerator="Control-u")
  tkbind(w, "<Control-u>", function() tcl(tb, "edit", "undo"))
}
```

To illustrate popup menus, we define one within our text widget that will grab all functions that complete the current word, using the CompletePlus function from the svMisc package to find the completions. The use of current wordstart and current wordend to find the word at the insertion point isn't quite right for R, as it stops at periods.

```
doPopup <- function(W, X, Y) {
  cur <- tclvalue(tkget(W, "current wordstart", "current wordend"))
  tcl(W, "tag", "add", "popup", "current wordstart", "current wordend")
  posVals <- head(CompletePlus(cur)[,1, drop=TRUE], n=20)
  if(length(posVals) > 1) {
    popup <- tkmenu(tb) # create menu for popup
    sapply(posVals, function(i) {
      tkadd(popup, "command", label=i, command = function() {
        tcl(W, "replace", "popup.first", "popup.last", i)
      })
    })
    tkpopup(popup, X, Y)
  }
}
```

For a popup, we set the appropriate binding for the underlying windowing system. For the second mouse button binding in OS X, we clear the clipboard. Otherwise the text will be pasted in, as this mouse action already has a default binding for the text widget.

```
if (!usingMac()) {
  tkbind(tb, "<Button-3>", doPopup)
} else {
  tkbind(tb, "<Button-2>", function(W,X,Y) {
    ## UNIX legacy re mouse-2 click for selection copy
  })
}
```

```

tcl("clipboard","clear",displayof=W)
doPopup(W,X,Y)
})      # right click
tkbind(tb, "<Control-1>", doPopup)      # Control + click
}

```

## 12.5 Canvas Widget

The canvas widget provides an area to display lines, shapes, images and widgets. Methods exist to create, move and delete these objects, allowing the canvas widget to be the basis for creating interactive GUIs. The constructor `tkcanvas` for the widget, being a non-themable widget, has many arguments. We mention the standard ones `width`, `height`, and `background`. Additionally, the canvas is a scrollable widget, so has the corresponding arguments `xscrollcommand` and `yscrollcommand`.

**The create command** The subcommand `tkcanvas create type [options]` is used to add new items to the canvas. The options vary with the type of the item. The basic shape types that one can add are "line", "arc", "polygon", "rectangle", and "oval". Their options specify the size using  $x$  and  $y$  coordinates. Other options allow one to specify colors, etc. The complete list is covered in the canvas manual page, which we refer the reader to, as the description is lengthy. In the examples, we show how to use the "line" type to display a graph and how to use the "oval" type to add a point to a canvas. Additionally, one can add text items through the "text" type. The first options are the  $x$  and  $y$  coordinates and the `text` option specifies the text. Other standard text options are possible (e.g., `font`, `justify`, `anchor`).

The type can also be an image object or a widget (a window object). Images are added by specifying an  $x$  and  $y$  position, possibly an anchor position, and a value for the "image" option and optionally, for state dependent display, specifying "activeimage" and "disabledimage" values. The "state" option is used to specify the current state. Window objects are added similarly in terms of their positioning, along with options for "width" and "height". The window itself is added through the "window" option. An example shows how to add a frame widget.

Once created, a screenshot of the canvas can be created through the `tkcanvas postscript` subcommand, as in `tcl(canvas, "postscript", file="filename")`. To store the widget so that it can be recreated is not supported directly. Tcl code to do so can be found at <http://wiki.tcl.tk/9168>.

**Items and tags** The `tkcanvas.create` function returns an item ID. This can be used to refer to the item at a later stage. Optionally, tags can be used to group items into common groups. The "tags" option can be used with

`tkcreate` when the item is created, or the *tkcanvas* `addtag` subcommand can be used. The call `tkaddtag(canvas, "tagName", "withtag", item)` would add a tag to the item returned by `tkcreate`. As well, if one is adding a tag through a mouse click, the call `tkaddtag(W, "tagName", "closest", x, y)` could be used with `W`, `x` and `y` coming from percent substitutions. Tags can be deleted through the *tkcanvas* `dtag tag` subcommand.

There are several subcommands that can be called on items as specified by a tag or item ID. For example, the *tkcanvas* `itemcget` and *tkcanvas* `itemconfigure` subcommands allow one to get and set options for a given item. The *tkcanvas* `delete tag_or_ID` subcommand can be used to delete an item. Items can be moved through the *tkcanvas* `move tag_or_ID x y` subcommand, where `x` and `y` specify the horizontal and vertical shift in pixels. The subcommand *tkcanvas* `coords tag_or_ID [coordinates]` allows one to respecify the coordinates for which the item was defined, thereby allowing the possibility of moving or resizing the object. Additionally, the *tkcanvas* `scale` can be used to rescale items. If items overlap each other, except for windows, an item can be raised to the top through the *tkcanvas* `raise item_or_ID` subcommand.

**Bindings** Bindings can be specified overall for the canvas, as usual, through `tkbind`. However, bindings can also be set on specific items through the subcommand *tkcanvas* `bind tag_or_ID event function` which is aliased to `tkitembind`. This allows bindings to be placed on items sharing a tag name, without having the binding on all items. Only mouse, keyboard or virtual events can have such bindings.

#### Example 12.8: Using a canvas to make a scrollable frame

This example shows how to use a canvas widget to create a box container that scrolls when more items are added than will fit in the display area. The basic idea is that a frame is added to the canvas equipped with scrollbars using the *tkcanvas* `create window` subcommand. The binding to the `<Configure>` event updates the scrollregion of the canvas widget to include the entire canvas. This grows, as items are added to the frame. This is modified from an example found at <http://mail.python.org/pipermail/python-list/1999-June/005180.html>.

This constructor returns a box container that scrolls as more items are added. The parent passed in must use the grid manager for its children.

```
scrollableFrame <- function(parent, width= 300, height=300) {  
  canvasWidget <-  
    tkcanvas(parent,  
             borderwidth=0, highlightthickness=0,  
             background="#e3e3e3", # match themed widgets  
             width=width, height=height)  
  addScrollbars(parent, canvasWidget)
```

```

gp <- ttkframe(canvasWidget, padding=c(0,0,0,0))
gpID <- tkcreate(canvasWidget, "window", 0, 0, anchor="nw",
                 window=gp)

tkbind(gp,"<Configure>",function() { # updates scrollregion
  bbox <- tcl(canvasWidget, "bbox", "all")
  tcl(canvasWidget,"config", scrollregion=bbox)
})

return(gp)
}

```

To use it, we create a simple GUI as follows:

```

w <- tktoplevel()
tkwm.title(w,"Scrollable frame example")
g <- ttkframe(w); tkpack(g, expand=TRUE, fill="both")
gp <- scrollableFrame(g, 300, 300)

```

To display a collection of available fonts requires a widget or container that could possibly show hundreds of similar values. The scrollable frame serves this purpose well (cf. Figure ??). The following shows how a label can be added to the frame whose font is the same as the label text. The available fonts are found from `tkfont.families` and the useful coercion to character by `as.character`.

```

fontFamilies <- as.character(tkfont.families())
## skip odd named ones
fontFamilies <- fontFamilies[grepl("^[[:alpha:]]", fontFamilies)]
for(i in 1:length(fontFamilies)) {
  fontName <- paste("tmp",i,sep="")
  try(tkfont.create(fontName, family=fontFamilies[i], size=14),
      silent=TRUE)
  l <- ttklabel(gp, text=fontFamilies[i], font=fontName)
  tkpack(l, side="top", anchor="w")
}

```

### Example 12.9: Using canvas objects to show sparklines

Edward Tufte, in his book *Beautiful Evidence*?, advocates for the use of *sparklines* – small, intense, simple datawords – to show substantial amounts of data in a small visual space. This example shows how to use a `ttkcanvas` object to display a sparkline graph using a line object. The example also uses `tkgrid` to layout the information in a table. We could have spent more time on the formatting of the numeric values and factoring out the data download, but leave improvements as an exercise.

This function simply shortens our call to `ttklabel`. We use the global `f` (a `ttkframe`) as the parent.

## 12. TCL Tk: WIDGETS

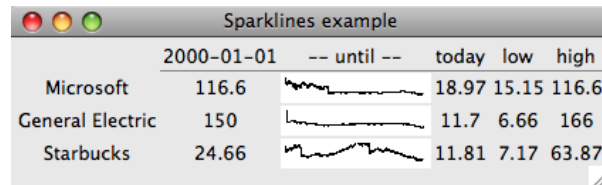


Figure 12.5: Example of embedding sparklines in a display organized using tkgrid. A tkcanvas widget is used to display the graph.

```
mL <- function(label) {  
  if(is.numeric(label))  
    label <- format(label, digits=4)  
  ttklabel(f, text=label) # save some typing  
}
```

We begin by making the table header along with a toprule.

```
tkgrid(mL(""), mL("2000-01-01"), mL("-- until --"),  
       mL("today"), mL("low"), mL("high"))  
tkgrid(ttkseparator(f), row=1, column=1, columnspan=5, sticky="we")
```

This function adds a sparkline to the table. We use financial data in this example, as we can conveniently employ the `get.hist.quote` function from the `tseries` package to get interesting data.

```
addSparkLine <- function(label, symbol="MSFT") {  
  width <- 100; height=15 # fix width, height  
  y <- get.hist.quote(instrument=symbol, start="2000-01-01",  
                     quote="C", provider="yahoo",  
                     retclass="zoo")$Close  
  min <- min(y); max <- max(y)  
  start <- y[1]; end <- tail(y,n=1)  
  rng <- range(y)  
  
  sparkLineCanvas <- tkcanvas(f, width=width, height=height)  
  x <- 0:(length(y)-1) * width/length(y)  
  if(diff(rng) != 0) {  
    y1 <- (y - rng[1])/diff(rng) * height  
    y1 <- height - y1 # adjust to canvas coordinates  
  } else {  
    y1 <- height/2 + 0 * y  
  }  
  ## make line with: pathName create line x1 y1... xn yn  
  l <- list(sparkLineCanvas, "create", "line")  
  sapply(1:length(x), function(i) {  
    l[[2*i + 2]] <- x[i]  
    l[[2*i + 3]] <- y1[i]  
  })  
}
```



```

    })
    do.call("tcl",l)

    tkgrid(mL(label),mL(start), sparkLineCanvas,
           mL(end), mL(min), mL(max), pady=1)
  }

```

We can then add some rows to the table as follows:

```

addSparkLine("Microsoft","MSFT")
addSparkLine("General Electric", "GE")
addSparkLine("Starbucks","SBUX")

```

### Example 12.10: Capturing mouse movements

This example is a stripped-down version of the `tkcanvas.R` demo that accompanies the `tcltk` package. That example shows a scatterplot with regression line. The user can move the points around and see the effect this has on the scatterplot. Here we focus on the moving of an object on a canvas widget. We assume we have such a widget in the variable `canvas`.

This following adds a single point to the canvas using an `oval` object. We add the "point" tag to this item, for later use. Clearly, this code could be modified to add more points.

```

x <- 200; y <- 150; r <- 6
item <- tkcreate(canvas, "oval", x - r, y - r, x + r, y + r,
                 width=1, outline="black",
                 fill="SkyBlue2")
tkaddtag(canvas, "point", "withtag", item)

```

In order to indicate to the user that a point is active, in some sense, the following changes the fill color of the point when the mouse is over the point. We add this binding using `tkitembind` so that it will apply to all point items and only the point items.

```

tkitembind(canvas, "point", "<Any-Enter>", function()
  tkitemconfigure(canvas, "current", fill="red"))
tkitembind(canvas, "point", "<Any-Leave>", function()
  tkitemconfigure(canvas, "current", fill="SkyBlue2"))

```

There are two key bindings needed for movement of an object. First, we tag the point item that gets selected when a mouse clicks on a point and update the last position of the currently selected point.

```

lastPos <- numeric(2) # global to track position
tagSelected <- function(W, x, y) {
  tkaddtag(W, "selected", "withtag", "current")
  tkitemraise(W, "current")
  lastPos <- as.numeric(c(x, y))
}
tkitembind(canvas, "point", "<Button-1>", tagSelected)

```

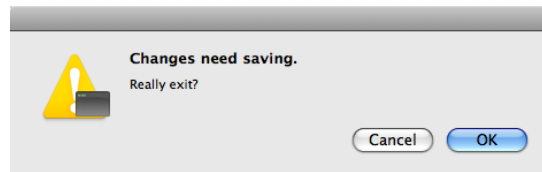


Figure 12.6: A basic modal dialog constructed by `tkmessageBox`.

When the mouse moves, we use `tkmove` to have the currently selected point move too. This is done by tracking the differences between the last position recorded and the current position and moving accordingly.

```
moveSelected <- function(W, x, y) {  
  pos <- as.numeric(c(x,y))  
  tkmove(W, "selected", pos[1] - lastPos[1], pos[2] - lastPos[2])  
  lastPos <- pos  
}  
tkbind(canvas, "<B1-Motion>", moveSelected)
```

A further binding, for the `<ButtonRelease-1>` event, would be added to do something after the point is released. In the original example, the old regression line is deleted, and a new one drawn. Here we simply delete the "selected" tag.

```
tkbind(canvas, "<ButtonRelease-1>", function(W) tkdtag(W, "selected"))
```

## 12.6 Dialogs

### Modal dialogs

The `tkmessageBox` constructor can be used to create a simple modal dialogs allowing a user to confirm an action. This replaces the older `tkdialog` dialogs. The `tkmessageBox` dialogs use the native toolkit if possible. The arguments `title`, `message` and `detail` are used to set the text for the dialog. The `title` may not appear for all operating systems. A `messageBox` dialog has an `icon` argument. The default icon is "info" but could also be one of "error", "question" or "warning". The buttons used are specified by the `type` argument with values of "ok", "okcancel", "retrycancel", "yesno", or "yesnocancel". When a button is clicked the dialog is destroyed and the button label returned as a value. The argument `parent` can be given to specify which window the dialog belongs to. Depending on the operating system this may be used when drawing the dialog.

A sample usage is:

```
tkmessageBox(title="Confirm",  
             message="Really exit?", detail="Changes need saving.",
```

```
icon="question", type="okcancel")
```

If the default modal dialog is not enough – for instance there is no means to gather user input – then a toplevel window can be made modal. The `tkwait.window` will cause a top-level window to be modal and `tkgrab.release` will return the interactivity for the window.

## File and directory selection

Tk provides constructors for selecting a file, for selecting a directory or for specifying a filename when saving. These are implemented by `tkgetOpenFile`, `tkchooseDirectory`, and `tkgetSaveFile` respectively. Each of these can be called with no argument, and returns a `tclObj` that can be converted to a character string with `tclvalue`. The value is empty

There are arguments that can be useful. The dialog will appear related to a toplevel window if the argument `parent` is specified. The `initialdir` and `initialfile` can be used to specify the initial values in the dialog. The `defaulttextension` argument can be used to specify a default extension for the file.

When browsing for files, it can be convenient to filter the available file types that can be selected. The `filetypes` argument is used for this task. However, the file types are specified using Tcl brace-notation, not R code. For example, to filter out various image types, one could have

```
tkgetOpenFile(filetypes = paste(
    "{{jpeg files} {.jpg .jpeg} }",
    "{{png files} {.png}}",
    "{{All files} {*}}", sep=" ")) # the space is impt. here
```

Extending this is hopefully clear from the pattern above.

### Example 12.11: A “File” menu

To illustrate, a simple example for a file menu could be:

```
w <- tktoplevel(); tkwm.title(w, "File menu example")
mb <- tkmenu(w); tkconfigure(w, menu=mb)
fileMenu <- tkmenu(mb)
tkadd(mb, "cascade", label="File", menu=fileMenu)
tkadd(fileMenu, "command", label="Source file...",
      command= function() {
        fname <- tkgetOpenFile(fileTypes="{{R files} {.R}} {{All files} *}")
        source(tclvalue(fname))
      })
tkadd(fileMenu, "command", label="Save workspace as...",
      command=function() {
        fname <- tkgetSaveFile(defaulttextension="Rsave")
        save.image(file=tclvalue(fname))
      })
```

## 12. TCL Tk: WIDGETS

---

```
tkadd(fileMenu, "command", label="Set working directory...",
      command=function() {
        fname <- tkchooseDirectory()
        setwd(tclvalue(fname))
      })
```

### Choosing a color

Tk provides the command `tk_chooseColor` to construct a dialog for selection of a color by RGB value. There are three optional arguments `initialcolor` to specify an initial color such as `"#efefef"`, `parent` to make the dialog a child of a specified window and `title` to specify a title for the dialog. The return value is in hex-coded RGB quantiles. There is no constructor in `tcltk`, but one can use the dialog as follows:

```
w <- tktoplevel(); tkwm.title(w, "Select a color")
f <- ttkframe(w, padding=c(3,3,3,12))
tkpack(f, expand=TRUE, fill="both")
colorWell <- tkcanvas(f, width=40, height=16,
                     background="#ee11aa",
                     highlightbackground="#ababab")
tkpack(colorWell)
tkbind(colorWell, "<Button-1>", function(W) {
  color <- tcl("tk_chooseColor", parent=w, title="Set box color")
  color <- tclvalue(color)
  if(nchar(color))
    tkconfigure(W, background = color)
})
```

## 12.7 End of chapter notes

## Web-based GUIs

The internet allows one to distribute their work in a convenient, standardized way that allows people from around the globe to share. Indeed, the R project would never have reached the point it is had it started 10 years prior, as the web technologies that enabled user participation from disparate points did not exist in widespread form.

This chapter shows some of the means to produce interactive interfaces between the user and R through web technologies, at the time of writing. Web interfaces to some process have many obvious advantages over the desktop interfaces discussed in previous chapters: no installation issues for R and the toolkit libraries, user familiarity with a browser, operating system independence, etc. This makes it much easier to share ones work, but also puts an added burden on the GUI writer, who must have some familiarity with new technologies and the security implications contained therein.

This chapter does not even pretend to be comprehensive. It covers an enormous array of technologies. Rather, its focus is to show how R can be used with these technologies. The interested reader will likely need to seek additional help before implementation.

The web programmer coming to R will find relatively simple tools as compared say to some open-source tools available for the python programmer (Django [djangoproject.com](http://djangoproject.com), pyjamas [pyjs.org](http://pyjs.org), ...) or the ruby programmer (Ruby on Rails [rubyonrails.org](http://rubyonrails.org)) or even the web programmer used to one of the many available frameworks built on PHP (Drupal [drupal.org](http://drupal.org), Joomla! [joomla.org](http://joomla.org), ...). However, we will see that there are useful tools for R that make it possible to develop R-driven websites. Of course, web technologies are changing quite rapidly, and R package writers are hard at work, so one should check to see if newer, more powerful resources, have been added to the mix.

## 13. WEB-BASED GUIs

---

```
browser -> request -> server -> page lookup -> return page to browser -> display  
XXX -- REPLACE ME -- XXX
```

Figure 13.1: Basic flow of how a static HTML file is displayed on a browser.

### 13.1 Authoring Web Pages

The simplest web page is a static page that is returned when a user makes a request. The basic architecture involves a browser (or some other client) requesting a document from a web server. The request must encode what document is desired, so the web server can find it. The request is specified in terms of a *URI*, or uniform resource identifier (a URL is technically a type of URI). The web server in turn maps the URI request to a file on the file system which the web server returns to the browser.

The type of HTML file just described is known as a static file, in contrast to a dynamically generated file, as its contents do not reflect any possible extra information in the request. The authoring of static HTML files may involve three different technologies described next.

#### Markup languages

Typically a static web page is marked up in HTML. This now familiar markup language allows the page author to indicate structure in various parts of the document. Typical structures are paragraphs, headers, images, etc. Additionally, tags can denote presentation, such as color, font etc.

HTML is centered around the concept of a tag which is used to wrap a portion of the text of a file. A tag has a name or keyword, in lower case, and is enclosed in angle brackets. If the tag encloses some text, it has a start and end style. The start tag for a tag *x* would be `<x>` where the end tag would be `</x>` (an extra slash). All text between these tags would carry this tag. Some tags, such as the image tag `img`, are used to define their attributes only (a link in this case) so do not come in pairs, in this case it is common practice to end the tag with `/>`.<sup>1</sup>

A few typical tags are specified in Table ???. Tags may indicate how text is supposed to be formatted (e.g. `b`), others indicate what type of text it is (e.g. `code`), others the document structure (e.g., `h1`, `p`, etc.).

---

<sup>1</sup>There are two common variants of HTML one coming from SGML, another, XHTML, deriving from XML. Both are similar, but XHTML is stricter with its use of tags. Some basic rules (as opposed to conventions) include all tags are either ended with a closing tag, or with `/>`; tags are lower case; attributes must be enclosed in quotes and specified; the root element is different from that given in the examples. The Web Hypertext Application Technology Working Group (<http://whatwg.org>) has proposed specifications for the two that seem likely to become the standard for HTML5.

Table 13.1: Table of common tags in HTML.

| Tag    | Description                             |
|--------|---|
| html   | Denotes an HTML file                    |
| head   | Marks header of file                    |
| body   | Marks off main body of file             |
| script | Used to include other types of files    |
| p      | A paragraph. Also, br for a line break  |
| h1     | First level header. Also h2,...,h6      |
| ul     | Unordered list. Also ol.                |
| li     | Denotes a list item                     |
| a      | An anchor for a hyperreference          |
| img    | Denotes an image                        |
| div    | A text division, indicates a line break |
| span   | A text division, no implied break       |
| b      | Denotes text to be set in bold          |
| code   | Denotes text that is code               |
| em     | Denotes text to be emphasized           |
| table  | Creates a table element                 |

A tag may have one or more *attributes* specified. For example, an anchor tag, `a`, has an attribute to specify the image by URI, `href`. This attribute is indicated by name with an equals sign. Quotes are optional for HTML, but recommended in general. They are mandatory if there is white space involved. An example might be `<a href="http://www.r-project.org" />`.

All tags have an `id` attribute, which is used to give a unique ID to the part enclosed by the tag. This is used to identify the tag within the document object model (DOM) described in brief later. All tags also have a `class` attribute to indicate if the tagged content should be treated as a member of a class. This provides a means to classify and treat similar objects as a group. Some tags also allow one to specify style attributes, but a more modern approach is to use a stylesheet to specify those. The `span` and `div` tags are primarily used to specify attributes for the tagged text.

Some characters, such as angle brackets, have a reserved meaning. As such, to use an angle bracket in an HTML document requires the use of an *HTML entity reference*. There are many such entities – they are also used for character encodings. Entities are denoted by a leading ampersand `&` and trailing semicolon, as with `&lt;`.

### Example 13.1: Simple HTML file

A basic HTML file would include a structure similar to the following which shows the head and body. Within the head, a title is set.

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
```

```
<html>
  <head>
    <title>Hello World Page</title>
  </head>
  <body>
    Hello world
  </body>
</html>
```

A basic xhtml file has a different header, for example the following which specifies a version for the XML and a default name space through the `xmlns` attribute.

```
<?xml version="1.0" ?>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
  lang="en">
<head>
<meta http-equiv="content-type"
  content="text/html; charset=UTF-8"/>
<title>Page title</title>
</head>
```

#### Example 13.2: A basic table

Displaying tables is a common task for web pages. The `table` tag encloses a table. New rows are enclosed in a `tr` tag, and each cell can be a header cell, `th`, or a data cell `td`. The following shows one way alternate rows can be striped by hard coding a color attribute to the rows.

```
<table border="0" cellpadding="0">
  <tr>
    <th>Header 1</th><th>Header 2</th>
  </tr>
  <tr>
    <th>1</th> <th>2</th>
  </tr>
  <tr bgcolor="goldenrod">
    <th>3</th> <th>4</th>
  </tr>
</table>
```

#### Example 13.3: R helpers

Writing a tag and specifying a table can be tedious, especially if the data is in an R object. Some helper functions are useful. The `hwriter` package includes a few, for now we mention `hmakeTag` which will produce a tag around some specified content along with attributes, that can be passed in through R's `name=value` syntax.



```
require(hwriter)
out <- hmakeTag("td",1:2, bgcolor="red")
cat(out, sep="\n")
```

```
<td bgcolor="red">1</td>
<td bgcolor="red">2</td>
```

The function is vectorized, as can be gathered from the output.

## Style sheets

Cascading Style Sheets (CSS) may be used to specify the presentation of the text on a page. Common practice is to use the markup language to specify document structure and a separate document to specify the layout of the first document. The advantage is a clean separation of tasks so that one can make changes to the layout, say, without affecting the text (and vice versa). A typical usage is to be able to provide different layouts depending on the type of device.

Without going into detail, the style sheet syntax provides a means to specify what type of tagged content the style will apply to (the selector) and a means to specify what styles of markup will be applied. For example, the specification below has `h1`, `h2`, `h3`, `h4`, `h5`, `h6` as a selector to indicate that it applies to all header tags. In the *declaration block* are style specifications for the color of the text and the font weight. Additionally, specifications for margins and padding are given, along with a border on the bottom around the element.

```
h1, h2, h3, h4, h5, h6 {
  color: Black;
  font-weight: normal;
  margin: 0;
  padding-top: 0.5em;
  border-bottom: 1px solid #aaaaaa;
}
```

The full specification allows for much more complicated selections, be they based on id of the tag, class of the tag, or relation of tag to an enclosing tag. Style sheets can also refer to positioning of the object within the page. Most modern web pages use style sheets for layout, rather than tables, as it allows for greater accessibility and offers advantages with search engines.

### Example 13.4: A striped table using style sheets

Using the `bgcolor` attribute of a table is deprecated in favor of style sheets for good reason. Here we illustrate a style sheet approach to striping a table. The style sheet may be defined in the HTML file itself with the `style` tag that appears within the document's head.

```
<style type="text/css">
table { border: 1px solid #8897be; border-spacing: 0px}
tr.head { background-color:#ababab;}
tr.even { background-color: #eeeeee;}
tr.odd { background-color: #ffffff;}
</style>
```

A more common alternative, is to use the link tag to include the stylesheet through a url. For example,

```
<link rel="stylesheet" href="the.url.of.the.sheet" />
```

For the table itself, we need only replace the specification of the attribute with a class specification.

```
<table>
  <tr class='head'>
    <th>Header 1</th><th>Header 2</th>
  </tr>
  <tr class='odd'>
    <th>1</th> <th>2</th>
  </tr>
  <tr class='even'>
    <th>3</th> <th>4</th>
  </tr>
</table>
```

## JavaScript

The third primary component of most modern web pages is JavaScript. This is a scripting language that runs within the browser that allows manipulation of the document. The document object model (DOM) specifies the elements of the text that may be referenced from within JavaScript. For example, individual elements can be found by unique ID, or common elements by class, or elements sharing a tag, say p. JavaScript provides methods for manipulating these elements. The simplest uses might be to change the text when the mouse hovers over an element.

JavaScript allows web pages to be dynamic interfaces. The language allows for callbacks to be defined for certain events, as with the other GUI toolkits we've discussed. We don't pursue this, but note that the gWidgetsWWW package uses JavaScript to make dynamic web pages (cf. Figure ??).

### Example 13.5: Simple use of JavaScript to make a button have an action

The button tag produces a visual button. This tag has several event attributes, including onmouseover and onclick. Here we show how to change the documents background color on a mouseover, and how to display a message on a mouse click.

```
<button
  onMouseOver="document.bgColor='red'; return true;"
  onMouseOut="document.bgColor=''; return true;"
  onClick="alert('clicked button'); return true;" >
  Click me ...
</button>
```

There are several open source JavaScript libraries available that offer convenient interfaces to JavaScript and UI widgets. We mention ExtJS ([www.extjs.com](http://www.extjs.com)), jQuery ([jquery.com](http://jquery.com)), YUI ([developer.yahoo.com/yui](http://developer.yahoo.com/yui)) and Dojo ([www.dojotoolkit.org](http://www.dojotoolkit.org)).

## R tools to assist with authoring web pages

There are quite a few packages for R to facilitate authoring web pages from R objects. We mention a couple.

### The hwriter package

The `hwriter` package simplifies the task of creating HTML tables for R objects, such as a matrix or vector. The package has a self-generated example page in HTML which is created by its `showExample` function. The main function `hwrite` maps R objects into table objects (by default) and has many options to modify the attributes involved. By default, it writes its output to a file. The helper function `openPage` takes a file name and returns a text connection. The `closePage` function will close it. In the examples below, so as the output will print, we use the `stdout` function instead for the connection.

The package's examples show many different usages, we illustrate a few below.

To create a simple table, we simply call the constructor on a matrix or `data.frame` object:

```
m <- matrix(1:4, ncol=2)
hwrite(m, page=stdout())
```

```
<table border="1">
<tr>
<td>1</td><td>3</td></tr>
<tr>
<td>2</td><td>4</td></tr>
</table>
```

To get alternate rows to be striped we could have:

```
hwrite(m, page=stdout(),
      row.class=rep(c("odd", "even"), nrow(m))[1:nrow(m)])
```

```
<table border="1">
<tr>
<td class="odd">1</td><td class="odd">3</td></tr>
<tr>
<td class="even">2</td><td class="even">4</td></tr>
</table>
```

(Or using the `bgcolor` attribute setting `row.bgColor`. Although, most of the `hwrite` arguments recycle, in this case we do so manually. as the “row.” attributes need such assistance.)

Finally, a hyperlink can be generated through the `link` argument.

```
hwrite("R project", link="http://www.r-project.org",
      page=stdout())
```

```
<a href="http://www.r-project.org">R project</a>
```

### The R2HTML package

The R2HTML provides the generic function `HTML` for creating HTML output from R objects based on their class. As with `hwrite`, this function writes its output to a connection for ease of generating a file.

As `HTML` is a generic function, its usage is straightforward. For a numeric vector we have:

```
library(R2HTML)
HTML(1:4, file=stdout())
```

```
<p class='integer'>1&nbsp; 2&nbsp; 3&nbsp; 4</p>
```

The class is written using the `class` attribute, so a style sheet can be used:

```
HTML(c(TRUE, FALSE), file=stdout())
```

```
<p class='logical'>TRUE&nbsp; FALSE</p>
```

Functions may be formatted:

```
HTML(mean, file=stdout())
```

```
<br><xmp class=function>function (x, ...)
UseMethod("mean")
<environment: namespace:base></xmp><br>
```

For more complicated objects, such as matrices and data frames, the `HTML` function has other arguments. For example, a border and inner border can be set (we omit the output).

```
HTML(iris[1:3,1:2], Border=10, innerBorder=5, file=stdout())
```

The package also includes a number of functions to facilitate the drafting of HTML files within R, including `HTMLInitFile`, `HTMLCSS`, `HTMLInsertGraph` and `HTMLEndFile`. The functions `HTMLStart` and `HTMLEnd`

|       |     | Evaluate                  |                           |
|-------|-----|---------------------------|---------------------------|
|       |     | Yes                       | No                        |
| Print | Yes | <code>&lt;%= %&gt;</code> | no delimiters             |
|       | No  | <code>&lt;% %&gt;</code>  | <code>&lt;%# %&gt;</code> |

Table 13.2: The brew delimiters and how they are processed

### The brew package

R has the wonderful facility Sweave that passes through a  $\text{\LaTeX}$  file and can replace R code with the code and output generated by evaluating the code. The R2HTML provides a means to do the same with HTML files. Whereas, the `ascii` package provides a means to do so for several `ascii`-based syntaxes for markup, many of which have tools to create HTML pages.

The `brew` package does something similar, yet different. It allows one to place a template within an HTML file that R will eventually populate when called accordingly. In the next section, we illustrate how this can be used dynamically in a web page. For now, we mention how to make a template and how to process it.

A template is a file which is processed by the `brew` function. This function returns the contents of the file after modification to store in a file or to print out.

A template is a file with parts of it marked by delimiters (cf. Table ??). All text not within delimiters is processed as is. Whereas, text within delimiters may be evaluated by R, and if evaluated the contents may be inserted into the output or simply used to adjust the evaluation environment.

### Example 13.6: Differences in brew delimiters

To illustrate the differences in the `brew` delimiters, the left side has `brew` commands and the right side is their output.

|  |   |
|--|---|
| <pre>Run, no print &lt;% x &lt;- 4 %&gt; Eval and print &lt;%= x^2 %&gt; Comment &lt;%# A comment %&gt; Inline &lt;%= x -%&gt; value</pre> | <pre>Run, no print Eval and print 16 Comment Inline 4 value</pre> |
|--|---|

The inline example has a dash before the closing delimiter to suppress a newline.

### Example 13.7: Dynamically formatted text

This example shows how `brew` can be used to insert dynamic text.

This template

```
<%
```

### 13. WEB-BASED GUIs

---

```
require(fortunes)
out <- fortune(155)
%>
<h3 class="fortune">Fortune</h3>
<%= wrap(out$quote) %> <br>
--<em><%= out$author %></em>, <%= out$date %>
```

produces

```
<h3 class="fortune">Fortune</h3>
It might surprise many R-help posters, but R has
manuals as well... <br>
--<em>Uwe Ligges</em>, January 2006
```

#### Example 13.8: Recursively calling brew

Typically there will be more than one page on a web site with each sharing common features: a banner, a footer, navigation links, a side bar, ... Using templates for these pieces and then including the template in a file is one way to centralize these common pieces. The `brew` function can easily be used to do this.

For example, here we define a header and footer and then call them in from a page. Our header is basic template, but includes a variable `title` to be defined in the page.

```
<html>
<title><%= title %></title>
</html>
<body>
```

Our basic footer is

```
<div id="footer">
  [boilerplate text goes here]
</div>
</body>
```

And a typical page has this structure. We set the variable `title` in the scope of this page, but it is seen withing the scope of the call to process the header page.

```
<% title <- "A sample page" %>
<%= brew("brew-header.brew") %>

A basic page

<%= brew("brew-footer.brew") %>
```

#### Example 13.9: Creating a template within a template

This example shows how one can define a template within a template, as an

alternative to a separate file. The basic idea is to use `paste` to bypass the issue of being unable to nest `brew` delimiters. We evaluate the template within a context, so that each time we get the values from different rows.

This template

```
<% tmpl <- paste("<a href=<",
  "%= URL %", ">><",
  "%= Name %", "></a><br />",
  sep="")
%>
<%
df <- getCRANmirrors() ## some data frame
for(i in 2:3) {
  context <- df[i,]
  with(context, brew(text=tmpl))
}
%>
```

produces

```
<a href=http://cran.ms.unimelb.edu.au/>Australia</a><br />
<a href=http://cran.at.r-project.org/>Austria</a><br />
```

## Graphics in web pages

Web pages may be plain text, but most contain images or graphics. The `img` tag allows one to display a graphics file in an HTML page by specifying its `src` attribute. This is an image file, often in `png`, `gif` or `jpeg` format. In this section, we describe how R can be used to generate images by using different device drivers. To list all possible stock devices, see the help page for `Devices`. The function `capabilities` lists which devices are available for a given R installation.

### png

Typically when a plot command is issued, an interactive plot device is opened or reused, however, the user can specify a device to save the output to a file for further use. For example, the `pdf` and `postscript` functions will turn R commands into files for inclusion in written documents. For web pages, the `png` and `jpeg` device drivers are available for many systems. These may be used to insert a graphic into a web page.

The basic usage is like that of the `pdf` driver illustrated below – open the device, issue graphics commands, close the device:

```
pdf(file="test.pdf", width=6, height=6) # in inches
hist(rnorm(100), main="Some graphic")
invisible(dev.off()) # close device
```

To use the png driver on a linux server, the option `type` should be set to `cairo` either through the constructor, or by setting the option `bitmapType`.

The Cairo device driver is an alternative which can also output in png format.

### SVG graphics

The web has other means to display graphics than an inclusion of an image file. For example, Flash is a very popular method.<sup>2</sup> Scalable vector graphics (SVG, <http://www.w3.org/Graphics/SVG/>) is another way to specify a graphical object using XML. Many modern web browsers have support for the display of SVG graphics. To insert the file, we have the `object` tag and its attributes `data` and `type`, as in

```
<object data="image-svg.svg" type="image/svg+xml"></object>
```

Not all browsers support svg, so one might also have a fall back image, as in:

```
<object data="image-svg.svg" type="image/svg+xml">

</object>
```

There are a few drivers to create SVG files in R, for example In the base `grGraphics` package, the driver `svg` is available. This non-interactive driver is used as the `png` one illustrated above.

The `RSVGTipsDevice` package provides an alternate driver, `devSVGTips`. The “Tips” part of the package, is provided by the function `setSVGShapeToolTip`, which allows one to specify a tooltip to popup when the mouse hovers over an element. The tooltip specified is placed over the next shape drawn, such as a point.

For example, here we add a tip and a URL to each point in a scatterplot. We initially call `plot` without plot characters to set up the axes, etc.

```
require(RSVGTipsDevice)
f <- "image-svg.svg"
devSVGTips(f, toolTipMode=2, toolTipOpacity=.8)
plot(mpg ~ wt, mtcars, pch=NA)
nms <- rownames(mtcars)
for(i in 1:nrow(mtcars)) {
  ## need to add tooltip shape by shape
  setSVGShapeToolTip(title=nms[i]) # add tooltip
  setSVGShapeURL("http://www.google.com") # some URL
  with(mtcars, points(wt[i], mpg[i], cex=2, pch=16)) # add
}
invisible(dev.off())
```

---

<sup>2</sup>The `FlashMXML` from [omegahat.org](http://omegahat.org) provides a means to generate flash files from within R.



### The canvas tag

HTML5 is a major extension to HTML that is being implemented in most browsers at the time of the writing of this book. One of the new features of HTML5 is the canvas element, which allows JavaScript code to manipulate objects, similar to the tkcanvas widget of tcltk.

R has the canvas device driver, that can be used to generate JavaScript code to produce the graphic in a canvas element. The basic usage involves creating the JavaScript:

```
require(canvas)
f <- "canvas-commands.js"
canvas(width=480, height=480, file=f)
hist(rnorm(100), main="Some graphic")
invisible(dev.off())
```

Then, within the HTML File, code along the lines of the following is needed. The first script tag is used to define the variable `ctx` to hold the canvas object, as this is assumed by the canvas package.

```
<canvas id="canvas_id" width=480 height=480></canvas>
<script type="text/javascript" language="javascript">
var ctx = document.getElementById("canvas_id").getContext("2d");
</script>
<script type="text/javascript" src="canvas-commands.js"></script>
```

An alternative to canvas is possible through the RGraphicsDevice device from [omegahat.org](http://omegahat.org).

## 13.2 The rapache package

While websites can consist of just static files, many webpages viewed are dynamically generated in response to user input. In order to implement this, the process of returning a page for a user request is more complicated. Rather than simply look up a file, the web server may call an external program that prepares the text to return. This text may be HTML for a web page, or in the case of web services, may be XML or some other form of data markup. For R users, there have been a few projects in the past that allow an R process to be used to generate the response. At this point, the best one is the rapache package. The package web page lists a few projects that use this technology to create web pages, including some highly interactive web pages by Jeroen Ooms. The `gWidgetsWWW` package ports the `gWidgets` API to the web using rapache.

The *Apache* web server is one of several open-source projects supported by the Apache Software Foundation. It is extremely successful – its website (<http://www.apache.org>) boasts it has been the most popular web server on the internet since 1996. Like R, Apache's open source nature allows

developers to customize its standard behaviour, in this case using modules. The RApache package (<http://biostat.mc.vanderbilt.edu/rapache/>) provides such a module that inserts R in the processing phase of a request to the web server.

The rapache package works under linux but not directly under windows. However, one can use a virtual machine to run a linux version of Apache under windows or Mac OS X. A “virtual machine” containing a pre-built linux system is available from the rapache website.

### Configuration

The rapache package requires the Apache web server to be properly configured. There are a number of steps in the process. The rapache homepage has detailed instructions, we mention just the steps here.

First, a module for Apache must be created by running rapache’s configure script. For Debian users, the package can be installed through the usual mechanism. Afterwards, Apache must be configured.

The R module must be loaded into Apache. This is done in the standard way for Apache, through its `LoadModule` directive. This is done before any other R-centric directives are given in Apache’s configuration.

The `REvalOnStartup` directive is used to specify any packages that should be loaded whenever the web server starts. The web server embeds a copy of R in itself and spawns copies of this as it spawn copies of itself to handle requests. The startup can be slow, so this offers a chance to pre-load common packages to speed things up at the cost of a larger memory footprint. `RSourceOnStartup` is similar, only it used to specify a file to be sourced on startup.

**The Directory Directive** There are a few directives to configure rapache to process an incoming request. A standard configuration for Apache, is to have the URL specify a file on the file system after some mangling of the name, exchanging the base part of the URL with a document root. One can have rapache process the file prior to being returned by creating the appropriate directive

The rapache manual specifies a typical usage is to call `brew` on the file. That is, to make a dynamic web page a `brew` template is created and placed into the appropriate directory.

To configure rapache to do this, the following example directive may be added to Apache’s configuration.

```
<Directory /var/www/brew>
  SetHandler r-script
  RHandler brew::brew
</Directory>
```

```
request url -> mangle file name -> lookup, return file  
to  
request url -> mangle file name -> run function file through rapache handler  
-> return output
```

Figure 13.2: Inserting rapache in the request lookup

```
request -> rapache calls function -> returns output to client
```

Figure 13.3: Creating a web page from a script and inputs

If the “DocumentRoot” of Apache is `/var/www`, then a request such as `http://servername/brew/file.brew` will resolve first to Apache finding `file.brew` in the `/var/www/brew/` directory, and then that file will be processed by the `brew` function in the `brew` package. The output will then be returned to the client making the request. The `SetHandler` directive, specifies the handler will be script, thereby passing along the file information and an environment. The `RHandler` directive is used to specify a function to call. This example assumes the `brew` package is not loaded, so the `brew` is found from within the package. An alternative to `brew`, may be `sys.source`.

**The Location Directive** Requests need not map to a file system, but can simply map to a function call. For example, an application might be designed around data stored in a data base and all pages are generated dynamically. To have a URL call a script without reference to a file, the `LOCATION` directive is used. For example,

```
<Location /myapp>  
  SetHandler r-handler  
  RFileHandler /path/to/R/scripts/myapp.R  
</Location>
```

A request to `http://servername/myapp/extra` will call the script `myapp.R`. The `extra` part of the request can be found from one of the rapache variables discussed in Section ?? and the script can adjust its output based on this.

## Creating files

The typical use of rapache is to return an HTML file, but this is not the only possibility.

A `brew` template can easily have the necessary header/body structure of an HTML file to be returned by the server to the client. However, if the

page is generated by a function call, as with the `Location` directive, `rapache` provides some convenience functions.

Response headers can be added through the `setHeader` function. The set of headers is long and technical.<sup>3</sup>

Web servers typically return HTML files, but are possible of much more. For example, the server may be asked to dynamically generate a graphic, and the output would be an image file. As well, web services are used to pass some resource, say some data to a client requesting it. This data may be stored in XML format, or JSON or YAML etc. The `setContentType` function is used to set the MIME type of the response. It must be called before any `print` or `cat` statements in the file. To send back binary data, the function `sendBin` is available.

**Return Codes** The return value of the handler call indicates the failure or success of the request. The return value should be an integer, `rapache` provides named variables instead. For success a return value of `DONE` will indicate success, whereas a value such as `HTTP_BAD_REQUEST` will signal an error.<sup>4</sup>

The function `RApacheOutputErrors` can be used to direct what happens to the error, in particular it can be used to have errors print out to the browser rather than the log file. This is useful when developing a program.

### **rapache variables**

When a script of function is being evaluated within `rapache` certain variables holding information about the request and web server are created. The variables are lists with named arguments, the names matching Apache variables.

**SERVER** The `SERVER` variable holds a large amount of information on the request. For example, the `status` component returns the status code. Some of the most useful, decompose the URL requesting the page.

The response depends on the configuration. If the we use `/var/www/brew` to process requests through `brew`, as above, then a request like `http://localhost/brew/test.brew?some=brew` results in values of `uri` being `/brew/test.brew`, `filename` being `/var/www/brew/test.brew`, `path_info` being an empty string and `args` holding the string `some=brew`.

However, if we use the `Location` directive above, then the request `http://localhost/myapp/detail?some=brew` has `uri` being `/myapp/detail`, `path_info` being `/detail` (the “virtual” part of the request), and `args` again holding the string `some=brew`.

---

<sup>3</sup>The definitions can be found at <http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html>.

<sup>4</sup>A list of the `rapache` variables appear in its manual. A list of status codes can be found at <http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>

**GET** Both of the example urls above result have `SERVER$method` being `GET`. HTTP has a few conventions that are not enforced, but are associated with it providing RESTful web services. One being that one uses a certain set of methods to interact with the service. A `GET` request is meant to return data, a `POST` request is meant to create new data, a `PUT` request is meant to update data and a `DELETE` request to delete data.

The two example requests above, result in `GET` requests and the `GET` variable contains some useful information, namely the arguments passed through the URL after the `?`. (URLs use a `?` to pass arguments in the form `key=value` with multiple arguments separated by an `&`. So in the above, `GET` is a list with component `some` whose value is `brew`.

**POST** A `POST` request usually comes from within a form. As with a `GET` request, arguments can be passed in with the request, although they do not appear in a URL as illustrated above. As with the `GET` variable, the arguments appear as named components in the `POST` variable. `POST` request can contain more information – they are not limited in length the same way – and must be used to upload files, say.

**COOKIES** By design HTTP is a stateless protocol. This means that between requests the web server remembers nothing about the past requests. For large web sites, this has an advantage when multiple servers are used to process requests. However, it has disadvantages as the request must relay the state of a web page. Several mechanisms have been developed to deal with this issue. Sessions, where information is kept server side and an ID kept with the client allows a state to be maintained. Another solution is to store information on the client side. This is implemented through cookies. Although cookies have privacy issues, their use is widespread.

A basic cookie consists of a name and a value (a character vector of length 1). Cookies must satisfy certain validity constraints which are specified through a time to expire, a path to which the cookie pertains and a domain for which the cookie is valid. The rapache function `setCookie` can be used to set a cookie. The first argument is the cookie name, the second the value, and others are available to set properties, such as an expiry time. Cookies are placed in the outgoing header of a document, so this call is done before the head tag.

When a page is loaded, the `COOKIES` variable contains cookie information. Again, as a list. In this case, the names are the valid cookie names and the component's value is the cookie.

## Forms

User input can be passed to the server through the URL request or through a form. Forms are specified with the `form` tag, which has a few important

attributes.<sup>5</sup> The `action` attribute specifies the URI that will process the form information. In our example, this will match a `Location` directive. The `method` attribute is used to specify a `GET` request or a `POST` request. For a post request that includes a file upload, the `enctype` attribute should contain `"multipart/form-data"`. In addition to these, the `onsubmit` attribute is often used to specify some JavaScript to call as the form is submitted. For example, this may be used to specify code to validate the form entries.

**The `input` tag** With the form tags control elements may be placed. The `input` tag is used to specify several types of controls. The `type` attribute is used to specify which control. The default is `text` for a single line text entry, but other values are `password` for a password entry; `checkbox` and `radio` for selection of items; `file` for a file upload control; `image` for an image; `button` to make a button; and `submit` for a submit button.

The usual attributes `class` and `id` apply, as do many others that are type specific. The `name` attribute specifies the name for the element. This is processed as a key in the `POST` variable. The `value` attribute is used to specify an initial value. For sizing, the attributes `size` and `maxlength` are used to specify the control size and length of text string. For images `src` is used to specify the image source as a URL. For the selection widgets, `checked` is used to specify if the button is on.

To illustrate, this HTML would produce a simple text entry area

```
<input type="text" value="initial text" />
```

This would be used to specify a submit button

```
<input type="submit" value="submit" />
```

A radio group is created by having multiple inputs sharing a common name

```
<input type="radio" name="key" value="TRUE" checked="TRUE">
<input type="radio" name="key" value="FALSE">
```

**The `select` tag** The `select` tag is used to create a control to select one or more values from a list of options. This control may be a combobox or a table display. The attribute `multiple` is used to specify if the user can select one or more values. When specified, the `POST` or `GET` variables have multiple components of the same name. The `size` attribute specifies the number of entries to make initially visible.

The possible values for selection are given within option tags. The attribute `selected` is used to specify if the value is initially selected. The attribute `value` can be used to specify a different value than that displayed.

For example,

---

<sup>5</sup>See <http://www.w3.org/TR/html401/interact/forms.html> for a specification

```
<select name="id">
  <option value="1">one</option>
  <option value="2" selected="true">two</option>
</select>
```

**A textarea tag** Single line text entries are created by the input tag by default, but multiple line entries are formed by the textarea tag. The attributes cols and rows specify the size.

## Security

Forms allow users to specify values. These value may then be processed by the underlying R process within rapache. As such a malicious user may try to have code run that could compromise the web server. More benignly, the user may specify responses that include malformed HTML. If these are simply printed back when the web page is created, a rendering error may occur. Regardless of the user base for a web application, one should assume that user input for web sites should never be trusted.

**Unclosed or malicious tags** To avoid malformed HTML one should encode any user input that is echoed back to a web page. The following function will replace certain characters with their HTML entity for safe inclusion with a page.

```
HTMLEncode <- function(str) {
  str <- as.character(str)
  vals <- list(c('&', '&amp;'),
              c('"', '&quot;'),
              c('<', '&lt;'),
              c('>', '&gt;'))
  for(i in vals)
    str <- gsub(i[1], i[2], str)
  str
}
```

**Whitelists, Blacklists** Even in the event of a fixed list of values for a user to choose from, user input should always be checked. It is very easy to fabricate a request without going through the web form, for example the R package Rcurl can do this.

When checking values, one can use a whitelist – a list of acceptable values, or a blacklist – a list of unacceptable values. The use of a whitelist is better if possible, as it is very easy to miss something in a blacklist.

In either case, it is a good idea to never evaluate directly a users input.

**SQL injection** Many web sites are built around queries to a data base. Web-sites powered by rapache can take this approach, as the Rdbi package allows an interface within the R process between a data base and R. The basic use is to create a query within R and then call one of Rdbi's functions to get the results from the query. The technique of SQL injection, takes advantage of carelessly constructed SQL queries that are made by pasting together SQL commands with user-given input.

#### Example 13.10: Using rapache to explore a data store

This example shows how one can use rapache to allow a user to explore a data set. This basic application is simple, but the structure of it is typical and very extendible. There are three pages to display: a page to greet the user, a page to select one of many items, and a page to display detail on an item.

We use a Location directive for this application which allows us to specify which page to display using the path\_info variable.

```
<Location /simpleapp>
  SetHandler r-handler
  RFileHandler /var/www/GUI/simpleapp/app.R
</Location>
```

The script app.R is responsible for processing the request and dispatching to the appropriate page. Our script contains the following to load packages and set the current working directory to match that of the script. This is needed for our calls to brew.

```
require(brew, quietly=TRUE)
require(hwriter, quietly=TRUE)
dir <- "/var/www/GUI/simpleapp"
setwd(dir)
```

We have four main pages, one for any errors, and the three mentioned. The dispatch to the page will call these functions which are responsible for setting the context for the brew templates. Each template has a title variable that we set within the function. This then will be within the scope of the call to brew. The variable df is assumed to contain a data frame of interest. This could be retrieved by some call to a data base, for example.

Our error page is called by

```
processError <- function(e) {
  title <- "Error"
  with(e, brew("error.brew"))
}
```

The error.brew template has

```
<% brew("brew-header.brew") %>
```



```
<h2>
  <%= message %>
</h2>
<% brew("brew-footer.brew") %>
```

where the value for message is passed in through the error call. The header and footer templates are straightforward, and are used to give a consistent look to each page. In this case, as we use xhtml, we have for the header:

```
<?xml version="1.0" ?>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
  lang="en">
<head>
<meta http-equiv="content-type"
  content="text/html; charset=UTF-8"/>
<title> <%= title %> </title>
</head>
<body>
<%
  if(exists("user_name") && nchar(user_name))
    cat(sprintf("<h2>Welcome %s</h2>", HTMLencode(user_name)))
%>
```

The user\_name variable is set in the greeting page, so may not be present. Note the call to HTMLencode to ensure that the value for the name, which comes from the user, does not contain any malformed HTML.

The footer simply closes the body and html tags. In both cases, these templates could be much more complicated.

Our greeting page illustrates how to use a form to gather user input, in this case a name, but in general this might be used for authentication etc.

```
showLogon <- function() {
  title <- "Logon"
  brew("login-form.brew")
}
```

The main part of the brew template is a basic form using the input tag in two different ways.

```
<form method="POST" action="/simpleapp/select">
<label>Enter your name:</label>
<input type="text" name="name" />
<input type="submit" value="submit" />
</form>
```

We use a POST call, as this may be used to modify a data source. As well, the action specification uses select so that the path\_info variable can be used to determine which page to call.

## 13. WEB-BASED GUIs

---

After logging on, the user may be asked to narrow the search for data. In this example, the user is asked to select one of the rows of the data source. We generically refer to the row identifier as ID.

```
selectID <- function() {  
  title <- "Select an ID"  
  context <- list(nms=rownames(df))  
  with(context, brew("select-id.brew"))  
}
```

The context variable is used to pass in different contexts to the brew template. Of course this could also appear directly in the template, but it is better to separate the logic from the presentation. In this case, the template for ID selection includes this

```
<form method="GET" action="/simpleapp/id">  
<select name="id">  
  <%  
    cat(hmakeTag("option", nms))  
  %>  
</select>  
<input type="submit" value="submit" />  
</form>
```

We use GET for the method, as we assume this is merely a request to narrow the display of data, not modify the data store. The useful `hmakeTag` function is employed to vectorize the creation of the HTML option tags.

Finally, our call to show detail on the selected identifier includes matching the user specified ID against a list of possible values (a whitelist). If no match occurs, an error message is printed.

```
showID <- function() {  
  title <- "Show an ID"  
  id <- GET$id  
  if(! id %in% rownames(df)) {  
    processError(list(message="id does not match"))  
  } else {  
    context <- list(d=df[id,], id=id)  
    with(context, brew("show-id.brew"))  
  }  
}
```

For the display, we have this basic template which uses `hwrite` to put the output into a table.

```
<h3> Detail on <%= id %> </h3>  
<%  
  hwrite(unlist(d), page=stdout())  
%>
```

The main script must figure out the `user_name` variable. This may come from the greeting page through a POST request, or may be stored using a cookie to make the name persistent. This leads to the following (`get_d` is used to provide a default, if the variable is NULL):

```
user_name <- ""
if (!is.null(POST)) {
  user_name <- get_d(POST$name, "")
}
if(user_name == "" && !is.null(COOKIES)) {
  user_name <- get_d(COOKIES$name, "")
}
```

Finally, the script is used to dispatch to the proper page. We start by setting the content type and a cookie to store the `user_name` variable.

```
setContentType("text/html")
if(user_name != "")
  setCookie("name",user_name)
```

Following how django processes URLs we set up a list of regular expressions to check against `path_info` and function names to handle the dispatch.

```
urls <- list(select=list(regex = "~/select", call="selectID"),
             id = list(regex = "~/id", call="showID" )
            )
default_call <- "showLogon"
```

With this, we then process the request as follows.

```
path_info <- SERVER$path_info
flag <- FALSE
for(i in urls) {
  if(!flag && grepl(i$regex, path_info)) {
    flag <- TRUE
    tryCatch(do.call(i$call, list()), error=processError)
  }
}
if(!flag)
  tryCatch(do.call(default_call, list()), error=processError)
```

We wrap the call inside `tryCatch` in case the page creation throws an error.

The last line of the script is simply `DONE` to indicate to the client that the request is finished.

### 13.3 Web 2.0

The term web 2.0 is used to describe highly interactive web sites. A key feature of many of these is the use of *Ajax technologies*. The packages `Rpad`

and `gWidgetsWWW` use Ajax technologies for interactive web sites. “Ajax” comes from asynchronous Javascript and XML. The term asynchronous refers to pieces of a web page being updated independently of others, unlike in the previous section where each request creates a new page. The JavaScript term is a substitute for a browser side language to manipulate the web pages DOM, and XML simply a means to encode data, and shouldn’t be taken literally, as other common text-based encodings are used, such as JSON.

Several JavaScript libraries are built around Ajax technologies, such as the `extjs` library and `jQuery`. These provide a means to query a server asynchronously through an *XMLHttpRequest*. This section discusses briefly how to use `rapache` to provide the data for such a request.

### Example 13.11: Creating a web service using `rapache`

This example will illustrate how make a web service with `rapache`. There are two pieces, the JavaScript code in the web page, and the server code. For the JavaScript piece, we use the `jQuery` library, as the use is somewhat straightforward.

We illustrate how to return HTML, JSON and XML. First, the HTML.

In the header of our web page, we must call in the `jQuery` JavaScript library. These files may be on local webserver, or called in with the following HTML code:

```
<script
  src="http://ajax.googleapis.com/ajax/libs/jquery/1.3/jquery.min.js"
  type="text/javascript">
</script>
```

Inside the HTML page, we have a place holder to put the text from the web service. We use `div` tag, with a unique id.

```
<div id="htmlTarget"> [HTML target] </div>
```

(There are also similar areas for JSON and XML.)

We want the request for data to happen when the page loads. The `jQuery` library provides a means to have a function called as the page loads (before any images are downloaded, say). We place the command within this bit of JavaScript.

```
<script type="text/javascript">
  $(document).ready(function(){
    // JavaScript commands go here
  })
</script>
```

As for the JavaScript commands, the following `jQuery` code will produce the Ajax request. This assumes the webserver is running locally. One would replace `localhost` with the appropriate site.

```
$.ajax({
```

```

type: "GET",
url: "http://localhost/ajaxapp/html",
dataType: "html",
success: function(data) {
    $("#htmlTarget").html(data);
},
error: function(e) {
    $("#htmlTarget").html("<em>Service is unavailable</em>");
}
});

```

To explain, the \$ is a jQuery variable, the first occurrence is a call to its ajax method. The arguments are specified to make a GET request to a certain url. The return data will be HTML. The request, if a success, will replace the HTML code within the node with id htmlTarget with that returned by the Ajax request. If an error is returned, an error message is placed there instead.

Within the R script run by rapache, we have a call like this

```

show_html <- function() {
  require(hwriter, quietly=TRUE)
  setContentType("text/html")
  hwrite(d[1:5,], page=stdout())
}

```

Which specifies the content type and some HTML text. No headers are needed here. The d variable refers to some data frame. If there were an error, we would return an error code, say 404L for file not found. In this case the error handler is called.

Using JSON is not much different. This example will use the package rjson to create encode the data into json code, but RJSONIO can be used instead (from omegahat.org) or one can create the JSON within R directly. Here is the server side code (not written with any generality):

```

show_json <- function() {
  require(rjson, quietly=TRUE)
  n <- as.integer(GET$n)
  n <- min(max(n,1), 32) # check
  out <- toJSON(list(mpg=mtcars$mpg[1:n],
                    car=rownames(mtcars)[1:n]))
  setContentType("application/json")
  cat(out)
}

```

We allow a variable n to be passed in through the Ajax call. The function toJSON prefers lists to data frames, so we make a list with our data, in this case we have two named variables mpg and car.

Within the HTML file we have this JavaScript code.

```
$.getJSON(
```

```
"http://localhost/ajaxapp/json",
{n:"5"},
function(data) {
  $("#jsonTarget").html(""); // clear out
  for(i=0; i < data.mpg.length; i=i+1) {
    $("#jsonTarget").append(data.car[i] + " gets " +
      data.mpg[i] + " miles per gallon" + "<br />");
  }
});
```

The `getJSON` method is a convenience for the `ajax` method. The second argument is how we pass in the parameter `n`. Finally, the last function is called on a success, and simply loops over the vector and pieces together some HTML, appending it to the target. (The last bit is much easier in R, but not too hard in JavaScript.)

Finally, we illustrate doing a similar task only using XML. The server side code might look like

```
show_xml <- function() {
  require(XML, quietly=TRUE)
  n <- as.integer(GET$n)
  n <- min(max(n,1), 32) # check
  children <- sapply(1:n, function(i)
    newXMLNode("car",
              newXMLNode("make", d[i,1]),
              newXMLNode("mpg", d[i,2])
            ))
  out <- saveXML(newXMLNode("data", .children=children))
  setContentType("text/xml")
  cat(out)
}
```

We use the XML library to piece together our response. In this case we make several car nodes, each with a make and mpg value.

The JavaScript to parse this response can look like this:

```
$.ajax({
  type: "GET",
  url:"http://localhost/ajaxapp/xml",
  data: {n:"4"},
  dataType: "xml",
  success: function(data) {
    $("#xmlTarget").html("");
    $(data).find("car").each(function() {
      $("#xmlTarget").append($(this).find("make").text() +
        " gets " + $(this).find("mpg").text() +
        " miles per gallon" + "<br />")
    })
  }
})
```

```
| })
```

The `data` argument is used to pass in a parameter. As for the success callback, as before we append `tet` to the target after clearing it out. To find the text, is a bit tricky, as it uses jQuery's selector methods. Basically, the variable `this` stands for each car node, and the `find` method gets the child node for that variable. The `text` method converts the object to text that can be appended to the target.

