1

Programming GUIs using Qt

1.1 An introductory example

To get a small feel for how one programs a GUI using `qtbase`, the R package that interfaces R with the Qt libraries, we show how to produce a simple dialog to collect a date from a user.

If the underlying libraries and package are installed, the package is loaded as any other R package:

```
require(qtbase)
```

Constructors As with all other toolkits, in Qt, GUI components are created with constructors. For this example, we will set various properties later, rather than at construction time. For our GUI we have four basic widgets: a widget used as a container to hold the others, a label, a single line edit area and a button.

```
w <- Qt$QWidget()
l <- Qt$QLabel()
e <- Qt$QLineEdit()
b <- Qt$QPushButton()
```

The constructors are not found in the global environment, but rather are found in the `Qt` environment provided through `qtbase`. As such, the `$` lookup operator is used. For this example, we use a `QWidget` as a top-level window, leaving for to discuss the `QMainWidget` object and its task-tailored features.

Widgets in Qt have various properties that set the state of the object. For example, the window object, `w`, has the `windowTitle` property that is adjusted as follows:

```
w$windowTitle <- "An example"
```

Qt objects are essentially environments. In the above, the named component `windowTitle` of the environment holds the value of the `windowTitle` property of the object, so the `$` use is simply that for environments.

Figure 1.1: Screenshot of our sample GUI to collect a date from the user.

More typical, is a method call. Qt overloads the `$` operator for method calls (as does `RGtk2`). For example, both the button object and label object have a `text` property. The setter `setText` can be used to assign a value. For example,

```
l$setText("Date:")
b$setText("Ok")
```

Although, the calling mechanism is more complicated than just the lookup of a function stored as the component `setText` (cf.), as the object is passed into the body of the function, the usage is similar.

Layout Managers Qt uses layout managers to organize widgets. This is similar to Java/Swing and `tcltk`, but not `RGtk2`. Layout managers will be discussed more thoroughly in , but in this example we will use a grid layout to organize our widgets. The placement of child widgets into the grid is done through the `addWidget` method and requires a specification, by index and span, of the cells the child will occupy.

```
lyt <- Qt$QGridLayout()
lyt$addWidget(l, row=0, column=0, rowSpan=1, columnSpan=1)
lyt$addWidget(e, 0, 1, 1, 1)
lyt$addWidget(b, 1, 1, 1, 1)
```

One can adjust properties of the layout, but we leave that discussion for later.

We need to attach our layout to the widget `w`, which is done through the `setLayout` method:

```
w$setLayout(lyt)
```

To view our GUI (Figure 1.1), we must call its `show` method.

```
w$show()
```

Callbacks As with outher GUI toolkits, we add interactivity to our GUI by binding callbacks to certain events. To add a command to the clicking or pressing of the button is done by attaching a handler to the "pressed" signal for the button (the "clicked" signal is only for mouse clicks). Widgets have various signals they emit. Additionally, there are window-manager events that may be of interest, but using them requires more work than is used below. The `qconnect` function is used to add a handler for a signal. The function needs, as a minimum, the object, the signal name and the handler. Herr we print the value stored in the "Date" field.

```
handler <- function(checked)  print(e$text)
id <- qconnect(b, "pressed", handler)
```

We will discuss callbacks more completely in .

Refinements At this point, we have a working dialog built with `qtbase`. There is much room for refinement, which due to Qt's many features are relatively easy to implement. For this example, we want to guide the user to fill out the date in the proper format. We could have used Qt's `QDateEdit` widget to allow point-and-click selection, but instead show two ways to help the user fill in the information with the keyboard

The `QLineEdit` widget has a number of ways to adjust its behavior. For example, an input mask provides a pattern for the user to fill out. For a date, we may want the value to be in the form "year-month-date." This would be specified with `"0000-00-00"`, as seen by consulting the help page for `QLineEdit`. To add an input mask we have:

```
e$setInputMask("0000-00-00")
```

Further, for the line edit widget Qt can implement validation of the entered text. There are a few built-in validators, and for this purpose the regular expression validator could be used, but instead we wish to determine if we have a valid date by seeing if we can coerce the string value to a date via R's `as.Date` function with a format of `"%Y-%m-%d"`. The method `setValidator` can be used to set the validator that is in charge of the validation. However, rather than passing a function, one must pass an instance of a validator class. For our specific needs, we need to create a new class.

Object-oriented support The underlying Qt libraries are written in C++. The object oriented nature is preserved by `qtbase`. Not only are the classes and methods implemented in R, the ability to implement new subclasses and methods is also possible. For this task, we need to implement a subclass of the `QValidator` class, and for this subclass implement a `validate` method. More detail on working with classes and methods in `qtbase` is provided in .

The `qsetClass` function is used to set a new class. To derive a subclass, we need just this:

```
qsetClass("dateValidator", Qt$QValidator, function(parent = NULL) {
  super(parent)
})
```

The `validate` method is implemented as a virtual class in Qt, in R we implement a method of our sub class. The `qsetMethod` is employed. The signature of the `validate` method is a string containing the input and an index indicating where the cursor is in the text box. The return value of this method indicates a state of "Acceptable", "Invalid", or if neither can be determined "Intermediate." These values are actually integers, and in this case are stored within an enumeration in the `Qt$QValidator` class.

```
qsetMethod("validate", dateValidator, function(input, pos) {
  if(!grepl("^[0-9]{4}-[0-9]{1,2}-[0-9]{1,2}$", input))
    return(Qt$QValidator$Intermediate)
  else  if(is.na(as.Date(input, format="%Y-%m-%d")))
    return(Qt$QValidator$Invalid)
  else
    return(Qt$QValidator$Acceptable)
})
```

To use this new class, we call its constructor, which has the same name as the class, and then set it as a validator for the line edit widget:

```
validator <- dateValidator()
e$setValidator(validator)
```

1.2 Overview

Qt is an open-sourced, cross-platform application and UI framework. Its history begins with Haavard Nord and Eirik Chambe-Eng in 1991, the Trolltech company until 2008, and now Nokia, a major cell-phone producer. While originally not an open-source project, it now has licensing under the LGPL that allows its use in open-source software.

Qt is developed in C++ with extensions. There are several languages with bindings to Qt with R being one through the `qtbase` and `qtpaint` packages. While these packages are quite new as compared to `tcltk` and `RGtk2`, they are included here, as Qt provides arguably the richest GUI environment from within R will likely be the GUI toolkit of choice going forward.

Qt, a commercially supported package, has excellent documentation of its API and has several examples of its use using C++.

1.3 The `qtbase` package

The `qtbase` provides the primary interface between R and the underlying Qt libraries, provided the latter are installed. The Qt framework is available as a binary install from `http://qt.nokia.com/`.

The package exports very few items. The main one is an environment, Qt, that contains the bulk of the functionality. [1] The components of this environment are class objects that represent an actual C++ class, an R derivative or a C++ namespace. For example, the `QWidget` class being represented through the component `Qt$QWidget`.

These components inherit from class `RQtClass`

[1] The `Qt` object provides a binding between a library and an R object. There are others, and these all inherit from the `RQtLibrary` class which in turn inherits from `environment`. As such, these library objects can be used where environments are, such as with `ls` or `$`.

```
class(Qt$QWidget)
```

```
[1] "RQtSmokeClass" "RQtClass"      "function"
```

As well, they inherit from the `functions` class, as they serve as constructors for instances of the class. For example

```
w <- Qt$QWidget()
```

The `w` object has a class structure that reflects the class inheritance structure of Qt:

```
class(w)
```

```
[1] "QWidget"             "QObject"             "QPaintDevice"
[4] "UserDefinedDatabase" "environment"         "RQtObject"
```

The base class, `RQtObject`, is also an environment and the properties and methods for this instance of the Qt class that are available from within R comprise its components. For `w` the first few listed using `ls`:

```
ls(w)
```

```
[1] "setWindowRole"  "QWidget"        "topLevelWidget" "logicalDpiY"
[5] "mapToParent"    "pos"
```

Properties and methods are accessed from the environment in the usual manners available. The most convenient extractor is the `$` operator, but `[[` and `get` will also work. (With the `$` operator at the command line, completion works.) The properties may be accessed like a component of an environment. For example, a `QWidget` has a `windowTitle` property which is used when the widget draws itself with a window. The following shows how it can be accessed and set.

```
w$windowTitle
```

```
NULL
```

```
w$windowTitle <- "a new title"
w$windowTitle
```

```
[1] "a new title"
```

However, most properties in Qt are accessed through getter and setter methods. In this case, we have the setter `setWindowTitle` available

```
w$setWindowTitle("an even newer title")
```

Setter methods are typically named with the word "set" followed by the property name written in lower camel case, the convention Qt uses for its properties and method. (Class names are in upper camel case.)

The environment structure of the object masks the fact that the methods may be defined in a parent class of the object. For example, a button widget is provided by the `QPushButton` constructor, as in

```
b <- Qt$QPushButton()
```

This too has a `windowTitle` property, but this is inherited from the fact that the `QPushButton` subclasses the `QWidget` class, as may be seen from:

```
head(class(b), n=3)
```

```
[1] "QPushButton"     "QAbstractButton" "QWidget"
```

The reason this distinction is important to know, is that the documentation for the method will be found with the class where the method is defined, not in the subclass. As there is no easy way even to tell the signature of these methods, being able to consult the documentation is crucial.

Constructors

As mentioned, the class name is the same as the constructor, but constructors may have different signatures. For example, a simple push button can be produced in several different ways:

```
b <- Qt$QPushButton()
```

Qt allows one to specify a parent object at construction time, although generally this happens when the widget is added to a layout. The child gets added to the list of children of the parent thereby creating an object heirarchy. This allows such things as the communication between components during resizing of layouts or the automatic deletion of ancestors when a parent widget is destroyed. This happens by assigning `NULL` as a parent.

```
w <- Qt$QWidget()
b <- Qt$QPushButton(parent=w)
```

In addition, there are convenience constructors. To set the text property for a button, one can pass the value to the `text` argument:

```
b <- Qt$QPushButton(text="Button text")
```

We used a named argument, but the matching is done by position and type of object.

Buttons may also have icons, for example

```
i <- Qt$QIcon(system.file("images/ok.gif", package="gWidgets"))
b <- Qt$QPushButton(icon=i, text="Ok")
```

It Qt, the class name prefaced with a tilde is the destructor for a widget, but in `qtbase` it suffices to assign `NULL` as the parent through `setParent`

```
b$setParent(NULL)
```

Common methods for QWidgets and QObjects

The widgets we discuss in the sequel inherit many properties and methods from the base QObject and QWidget classes. The QObject class is the base class and forms the basis for the object heirarchy and the event processing system. The QWidget class is the base class for objects with a user interface. Defined in this class are several methods inherited by the widgets we discuss.

The function qmethods will show the methods defined for a class. It returns a data frame with variables indicating the name, return value, signature, and whether the method is protected and static. For example, for a simple button we have many methods.

```
out <- qmethods(Qt$QPushButton)
dim(out)                                    # many methods
```

```
[1] 435   5
```

Showing or hiding a widget Widgets must have their show method called in order to have them drawn to the screen. This call happens through the print method for an object inheriting from QWidget, but more typically is called by Qt recursively showing the children when a top-level window is drawn. The method raise will raise the window to the top of the stack of windows, in case it is covered. The method hide will hide the widget.

A widget can also be hidden by calling its setVisible method with a value of FALSE and reshown using a value of TRUE. Similary, the method setEnabled can be used to toggle whether a widget is sensitive to user input, including mouse events.

Only one widget can have the keyboard focus. This is changed by the user through tab-navigation or mouse clicks (unless customized, see focusPolicy), but can be set programatically through the setFocus method, and tested through the hasFocus method.

Qt has a number of means to notify the user about a widget when the mouse hovers over it. The setTooltip method is used to specify a tooltip as a string. The message can be made to appear in the status bar of a top-level window through the method setStatusTip.

The size of a widget A widget may be drawn with its own window, or typically embedded in a more complicated GUI. The size of the widget can be adjusted through various methods.

First, we can get the size of the widget through the methods frameGeometry and frameSize. The frameGeometry method returns a QRect instance, Qt's rectangle class. Rectangles are parameterized by an $x - y$ position and two dimensions (x, y, width and height). In this case, the position refers to the upper left coordinate and dimensions are in pixels. The convenience function qrect is provided to construct QRect instances. The frameSize method

returns a `QSize` object with properties `width` and `height`. The `qsize` function is a convenience constructor for objects of this class.

The widget's fixed size can be adjusted by modifying the rectangle and then resetting the geometry with `setGeometry`, or directly through the same method when integer values are given for the arguments.

«ChangeGeometry, results=hid>e>= w <- Qt$QWidget()rect < -wframeGeometry rectwidth()rectsetWidth(2 * rectwidth())wsetGeometry(rect)

Although the above sets the size, it does not fix it. If that is desired, the methods `setFixedSize` or `setFixedWidth` are available.

When a widget is resized, one can constrain how it changes by specifying a minimum size or maximum size. These values work in combination with the size policy of the widget. The properties `minimumSize`, `minimumWidth`, `minimumHeight`, `maximumSize` `maximumWidth` and `maximumHeight`, and their corresponding setters, are the germane ones. How these get used is determined by the `sizePolicy`roperty. For example, buttons will only grow in the x direction – not the y direction due to their default size policy.

Properties and enumerations

As mentioned, widget properties are set via setters. For example a button may be drawn "flat" to remove the typical beveling indicating it is a button. The method `setFlat`ccepts a logical indicating if the button is to be "flat."

Often there can be more than two states for a property, in which case values other than logical ones must be used. For example, the label widget (among others) has a property for how its text is aligned. For alignment there are options left, right, center, top, bottom, etc. In `qtbase` these enumerations are stored in the `Qt$Qt` class object, or for some widgets the `RQtClass` object. For example, alignment values are from the `AlignmentFlag` enumeration and right alignment is specified by:

```
Qt$Qt$AlignRight
```

```
AlignRight
         2
attr(,"class")
[1] "QtEnum"
```

Whereas, the size policy enumeration are in the `QSizePolicy` class and these are in the `Qt$QSizePolicy` object:

```
Qt$QSizsePolicy$Exanding
```

```
NULL
```

Although these enumerations can be specifed as integers, in `qtbase` they are of class `QtEnum` and have the overloaded operator `|` to combine values. For example, aligning text in label in the upper right can be done through

```
l <- Qt$QLabel("Our text")
l$setAlignment(Qt$Qt$AlignRight | Qt$Qt$AlignTop)
```

Fonts

Fonts in Qt are handled through the `QFont` class. In addition to the basic constructor, one constructor allows the programmer to specify a `family`, such as `helvetica`; `pointsize`, an integer; `weight`, an enumerated value such as `Qt`*`QFont`*`Light` (or `Normal`, `DemiBold`, `Bold`, or `Black`); and whether the italic version should be used, as a logical.

For example, a typical font specification may be given as follows:

```
f <- Qt$QFont(family="helvetica", ps=12,
              weight=Qt$QFont$Bold,
              italics=TRUE)
```

For widgets, the `setFont` method can be used to adjust the font. For example, to change the font for a label we have

```
l <- Qt$QLabel("Text for the label")
l$setFont(f)
```

The `QFont` class has several methods to query the font and to adjust properties. For example, there are the methods `setFamily`, `setUnderline`, `setStrikeout` and `setBold` among others.

Styles

Qt uses styles to provide a means to customize the look and feel of an application for the underlying operating system. Each style implements a palette of colors to indicate the states "active" (has focus), "inactive" (does not have focus), and "disabled" (not sensitive to user input). Many widgets do not have a visible distinction between active or inactive.

The role an object plays determines the type of coloring it should receive. A palette has an enumeration of `ColorRoles`. Sample ones are `Qt$QPalette$Highlight`, to indicate a selected item, or `Qt$QPalette$WindowText` to indicate a foreground color.

These roles are used for setting the foreground or background role to give a widget a different look, as illustrated in Example 1.2.

Style Sheets

Cascading style sheets (CSS) are used by web deisgners to decouple the layout and look and feel of a web page from the content of the page. Qt implements the mechanism in the `QWidget` class to customize a widget through the CSS syntax. The implemented syntax is described in the overview on

stylesheets provided with Qt documentation and is not rewritten here, as it is quite readable.

To implement a change through a style sheet involves the `setStyleSheet` method. For example, to change the background and text color for a button we could have

```
b <- Qt$QPushButton("Style sheet example")
b$show()
b$setStyleSheet("QPushButton {color: red; background: white}")
```

One can also set a background image:

```
ssheet <- sprintf("* {background-image: url(%s)}", "logo.png")
b$setStyleSheet(ssheet)
```

1.4 Signals

Qt in C++ uses an architecture of signals and slots to have components communicate with each other. A component emits a signal when some event happens, such as a user clicking on a button. Qt allows one to register a slot in another component (or the same) that will be passed information when the signal is emitted. The two components are decoupled as the emitter does not need to know about the receiver except through the coupling. In R, this isn't quite the case. A function takes the role of a slot (similar then to how `RGtk2` works via callbacks) and is called when the the signal is emitted. The function `qconnect` does the work. For example

```
b <- Qt$QPushButton("click me")
qconnect(b, "clicked", function(checked) print("ouch"))
b$show()
```

The signal names are defined within a class or inherited through subclasses. Sometimes the callback has arguments. [2] The `clicked` signal used above inherits from `QAbstractButton`, which also is a base class for check boxes. As such, this signal passes in information if the button is checked.

The optional argument `user.data` can be used to specify data to parameterize the callback. This data is passed in through the last argument.

The `qconnect` function has no return value. In `RGkt2` the return value is used to remove a callback. For Qt, the `disconnect` method can be used to remove connections with some degree of granularity, but this method is not implemented in `qtbase`. Rather, one can block all signals from being emitted with the `blockSignals` method, which takes a logical value to toggle whether the signals should be blocked.

In addition to signals, Qt widgets can also have event handlers for various events. For example, the button may have event handlers for things such as

[2] One advantage of the signal-slot architecture is the type-checking of arguments.

mouseMoveEvent. In C++ one uses virtual function (functions defined for instances), but in R these are implemented as methods of sub-classes in R. That is, you define a subclass, and then implement the desired methods, such as mouseMoveEvent. This will be illustrated in Example 1.2.

1.5 Defining Classes and Methods

The qtbase package allows the R user the ability to define classes and sub-classes to extend the features of Qt, as deemed necessary. Classes are related to the constructor that produces an instance of the class. In R classes are implemented as functions along with static methods in an environment. This is done with a bit of R voodoo and carried out when the base constructor is defined.

The qsetClass creates a new subclass and defines the constructor. The signature includes the arguments x to specify the name of the new class; parent to specify the class the new class will be a subclass of (for example, Qt$QWidget – the class, not the constructor, so no parentheses); constructor to specify the function used for construction; and a where argument to override where the class is defined. When qsetClass is called, a variable is assigned into the global environment (with scoping similar to methods::setClass.

In the introductory example we saw this minimal use of qsetClass.

```
qsetClass("dateValidator", Qt$QValidator, function(parent = NULL) {
  super(parent)
})
```

Here we see dateValidator is the new class name, a sub-class of the QValidator class. The constructor has a single call to super. The super function does not exist outside the scope of a constructor or a method. Within a constructor, super invokes the constructor of the parent (super) class, with the given arguments. Whereas, within a method implementation, super will call a method (named in the first parameter) in the parent class.

Within the body of the constructor and a method, the variable this is a reference to the instance of the class and the inherited method names are also attached, so need not be referenced through the $ notation.

To define a method for a class the qsetMethod function is used. This overrides the virtual method for the class. The signature includes name, for the new name; class, for the class being extended; and FUN to define the method. The dispatch happens on the class and method name, not on the signature of the method. In addition, there is an argument access to specify if the method is "public" (the default), "protected", or "private".

Within the method, the special super function can be used to call the next method, if the sub class overrides a method. An example is in Example 1.2. The basic call looks like super(meth_name, arg1, arg2, ...).

Example 1.1: A "error label"
A common practice when validation is used for text entry is to have a "buddy label." That is an accompanying label to set an error message. As Qt uses "buddy" for something else, we call this an "error label" below. We show how to implement such a widget in `qtbase` where we subclass the single-line text edit widget constructed by `QLineEdit`. We begin by defining our subclass and constructor

```
qsetClass("ErrorLabel", Qt$QLineEdit,
          function(text, parent=NULL, message="") {
  super(parent)

  this$widget <- Qt$QWidget()           # for attaching
  this$error <- Qt$QLabel()             # set height=0
  this$error$setStyleSheet("* {color: red}")  # set color


  lyt <- Qt$QGridLayout()               # layout
  lyt$setVerticalSpacing(0)
  lyt$addWidget(this, 0, 0, 1, 1)
  lyt$addWidget(error, 1, 0, 1, 1)
  this$widget$setLayout(lyt)

  if(nchar(message) > 0)
    setMessage(message)
  else
    setErrorHeight(FALSE)
  if(!missing(text)) setText(text)
})
```

In addition to the call to `super`, we define a `QWidget` instance to contain the line edit widget and its label. These are placed within a grid layout. The use of `this` refers to the object we are creating. The new method `setErrorHeight` is used to flatten the height of the label when it is not needed and is defined below. The final line sets the initial text in the line edit widget. The R environment where `setText` is defined is extended by the environment of this constructor, so no prefix is needed in the call.

The `widget` component is needed to actually show the widget. We create an accessor method for this

```
qsetMethod("widget", ErrorLabel, function() widget)
```

```
[1] "widget"
```

We extend the API of the line edit widget for this sub class to modify the message. We define three methods, one to get the message, one to set it and a convenience function to clear the message.

```
qsetMethod("message", ErrorLabel, function() error$text)
```

```
qsetMethod("setMessage", ErrorLabel, function(msg="") {
  if(nchar(msg) > 0)
    error$setText(msg)
  setErrorHeight(nchar(msg) > 0)
  })
qsetMethod("clear", ErrorLabel, function()  setMessage())
```

Finally, we define the method to set the height of the label, so that when there is no message it has no height. We use a combination of setting both the minimum and maximum height.

```
qsetMethod("setErrorHeight", ErrorLabel, function(do.height=FALSE)
{
  if(do.height) {
    m <- 18; M <- 100
  } else {
    m <- 0; M <- 0
  }
  error$setMinimumHeight(m)
  error$setMaximumHeight(m)
})
```

To use this widget, we have the extra call to `widget()` to retrieve the widget to add to a GUI. In the following, we just show the widget.

```
e <- ErrorLabel()
w <- e$widget()                 # get widget to show
w$show()                        # to view widget
e$setMessage("A label")         # opens message
e$clear()                       # clear message, shrink space
```

1.6 Drag and drop

Qt has native support for basic drag and drop activities for some of its widgets, such as text editing widgets, but for more complicated situations such support must be programmed in. The toolkit provides a clear interface to allow this.

A drag and drop event consists of several stages: the user selects the object that initiates the drag event, the user then drags the object to a target, and finally a drop event occurs. In addition, several decisions must be made, e.g., will the object "move" or simply be copied. This is determined by XXX. Or, what kind of object will be transferred (an image? text?, ...) etc. The type is specified using a standard MIME specification.

Initiating a drag and drop source When a drag and drop sequence is initiated, the widget receiving the mouse press event needs to set up a `QDrag` instance that will transfer the necessary information to the receiving widget.

In addition, the programmer specifies the type of data to be passed, as an instance of the `QMimeData` class. Finally, the user must call the `exec` method with instructions indicating what happens on the drop event (the supported actions) and possibly what happens if no modifier keys are specified. These are given using the enumerations `CopyAction`, `MoveAction`, or `LinkAction`. This is all specified in a new method for `mousePressEvent`, so must be done in a subclass of the widget you wish to use.

Creating a drop target The application must also set up drop sources. Each source has its method `setAcceptDrops` called with a `TRUE` value. In addition, one must implement several methods so again, a subclass of the desired widget is needed. Typically one implements at a minimum a `dropEvent` method. This method has an `QDropEvent` instance passed in which has the method `mimeData` to get the data from the `QDrag` instance. This data has several methods for querying the type of data, as illustrated in the example. If everything is fine, one calls the even'ts `acceptProposedAction` method to set the drop action. One can also specify other drop actions.

Additionally, one can implement methods for `dragMoveEvent` and `dragLeaveEvent`. In the example, the move and leave event adjust proporties of the widget to indicate it is a drop target.

Example 1.2: Drag and drop

We will use sub classes of the label class to illustrate how one implements basic drag-and-drop functionality. Out treatment follows the Qt tutorial on the subject. We begin by setting up a label to be a drag target.

```
qsetClass("DragLabel", Qt$QLabel, function(text="", ...) {
  parent(...)
  setText(text)
})
```

```
Class 'R:.GlobalEnv::DragLabel' with 372 public methods
```

The main method to implement for the sub class is `mousePressEvent`. The argument `e` contains event information for the mouse, we don't need it here. We have the minimal structure here: implement mime data to pass through, set up a `QDrag` instance for the data, then call the `exec` method to initiate.

```
qsetMethod("mousePressEvent", DragLabel, function(e) {
  md <- Qt$QMimeData()
  md$setText(text)

  drag <- Qt$QDrag(this)
  drag$setMimeData(md)
```

```
  drag$exec(Qt$Qt$CopyAction | Qt$Qt$MoveAction, Qt$Qt$CopyAction)
})
```

```
[1] "mousePressEvent"
```

Implementing a label as a drop target is a bit more work, as we customize its appearance. Our basic constructor follows:

```
qsetClass("DropLabel", Qt$QLabel, function(text="", ...) {
  parent(...)

  setText(text)
  setAcceptDrops(TRUE)

  this$bgrole <- backgroundRole()
  setMinimumSize(200, 200)
  setAutoFillBackground(TRUE)
  clear()
})
```

```
Class 'R::.GlobalEnv::DropLabel' with 371 public methods
```

We wish to override the call to `setText` above, as we want to store the original text in a property of the widget. Note the use of `super` with a method definition below to call the next method.

```
qsetMethod("setText", DropLabel, function(str) {
  this$orig_text <- str
  super("setText", str)                 # next method
})
```

The `clear` method is used to restore the label to an initial state. We have saved the background role and original text as properties.

```
qsetMethod("clear", DropLabel, function() {
  setText(this$orig_text)
  setBackgroundRole(this$bgrole)
})
```

```
[1] "clear"
```

The enter event notifies the user that a drop can occur on this target by changing the text and the background role.

```
qsetMethod("dragEnterEvent", DropLabel, function(e) {
  super("setText", "<Drop Text Here>")
  setBackgroundRole(Qt$QPalette$Highlight)

  e$acceptProposedAction()
})
```

The move and leave events are straightforward. We call `clear` when the drag leaves the target to restore the widget.

```
qsetMethod("dragMoveEvent", DropLabel, function(e) {
  e$acceptProposedAction()
})
qsetMethod("dragLeaveEvent", DropLabel, function(e) {
  clear()
  e$acceptProposedAction()
})
```

Finally, the important drop event. The following shows how to implement this in more generality than is needed for this example, as we only have text in our mime data. The `setPixmap` and `setTextFormat` methods for labels will be discussed in Section.

```
qsetMethod("dropEvent", DropLabel, function(e) {
  md <- e$mimeData()

  if(md$hasImage()) {
    setPixmap(md$imageData())
  } else if(md$hasHtml()) {
    setText(md$html)
    setTextFormat(Qt$Qt$RichText)
  } else if(md$hasText()) {
    setText(md$text())
    setTextFormat(Qt$Qt$PlainText)
  } else {
    setText("No match")                 # replace...
  }

  setBackgroundRole(this$bgrole)
  e$acceptProposedAction()
})
```

2

Layout managers

Qt provides a set of classes to faciliate the layout of child widgets of a component. These layout managers, derived from the `QLayout` class, are tasked with positioning of the child widgets, allocation of size to the child widgets, updating size when the parent is resized or when child widgets are hidden or removed. Unlike GTK+, where this management is tied to a container object, Qt decouples the layout from the widget.

In this chapter we discuss how to program GUI layouts. An alternative would be to use the Qt Designer application to specify the layout. We begin with an example that shows many of the different types of layouts.

Example 2.1: Using layout managers to mock up an interface

This example illustrates how to layout a somewhat complicated GUI by hand using a combination of different layout managers provided by Qt. Figure 2.1 shows a screenshot of the finished layout.

Qt provides the standard layouts for box layouts and grid layouts, in addition there are notebook containers and special layouts for forms, as seen in the following. Our GUI is layed out from the outside in. The first layout used is a grid layout which will hold three main areas: one for table (we use a label for now), one for a notebook, a layout to hold some buttons.

A `QWidget` instance can hold one layout specified by the `setLayout` method. We use widget for a top-level window and specify a grid layout.

```
w <- Qt$QWidget()
w$setWindowTitle("Layout example")
gridLayout <- Qt$QGridLayout()
w$setLayout(gridLayout)
```

Here we define the two main widgets and the layout for our buttons.

```
tableWidget <- Qt$QLabel("Table widget") # for now
nbWidget <- Qt$QTabWidget()
buttonLayout <- Qt$QHBoxLayout()
```

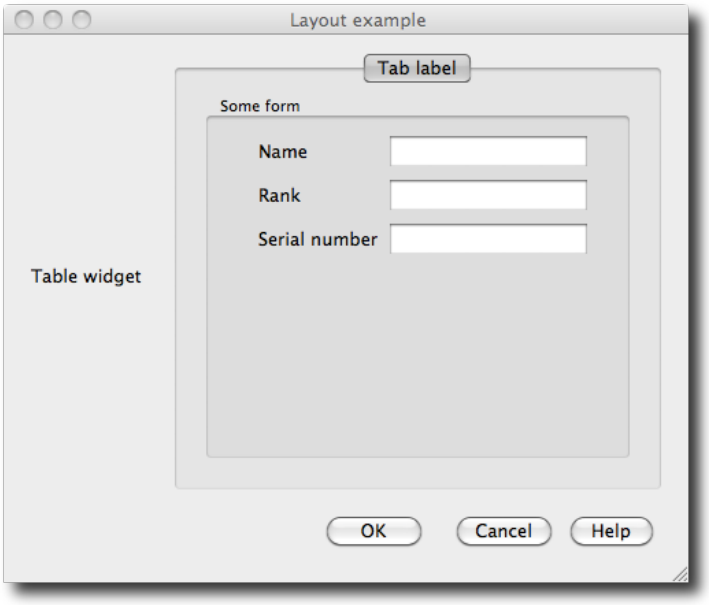

Figure 2.1: A mock GUI illustrating various layout managers provided by Qt.

Grid layouts have two main methods, `addWidget` which is inherited from a base `QLayout` class, and `addLayout`. In addition, we specify what part of the grid the child widget or layout will occupy through a 0-based row and column and spanning directions for both row and columns.

```
gridLayout$addWidget(tableWidget, row=0, column=0,
                     rowspan=5, colspan=1)
gridLayout$addWidget(nbWidget, 0, 1, 5, 4)
gridLayout$addLayout(buttonLayout, 5, 0, 1, 5)
```

Buttons are added to our box layout through the `addWidget` method. In this case, we want to push the buttons to the right side of the GUI, so we first add a stretch. Stretches are specified by integers. Unallocated space is given first to widgets that have a non-zero stretch factor. We also set spacing of 12 pixels between the "OK" and "Cancel" buttons.

```
b <- sapply(c("OK", "Cancel", "Help"), function(i) Qt$QPushButton(i))
buttonLayout$addStretch(1L)
buttonLayout$addWidget(b$OK)
buttonLayout$addSpacing(12L)
buttonLayout$addWidget(b$Cancel)
buttonLayout$addWidget(b$Help)
```

For our notebook widget we add pages through the `addTab` method. We pass in the child widget and a label below. In addition, we set a tooltip for the tab label to give more feedback to the user.

```
nbPage <- Qt$QWidget()
nbWidget$addTab(nbPage, "Tab label")
nbWidget$setTabToolTip(0, "A notebook page with a form")
```

We wish to layout a form inside of the notebook tab, but first create a framed widget using a `QGroupBox` widget. This widget allows us to easily specify a title. We addd this to the page using a box layout.

```
f <- Qt$QGroupBox()
f$setTitle("Some form")
lyt <- Qt$QHBoxLayout()
nbPage$setLayout(lyt)
lyt$addWidget(f)
```

The form layout allows us to layout standardized forms where each row contains a label and a widget. This could be done with a grid layout, but using the form layout makes it easier for Qt to style the page as appropriate for the underlying operating system. For illustration purposes, we set the horizontal alignment of the widget and the left align the label.

```
formLayout <- Qt$QFormLayout()
f$setLayout(formLayout)
## can override defaults:
formLayout$setFormAlignment(Qt$Qt$AlignHCenter)
formLayout$setLabelAlignment(Qt$Qt$AlignLeft)
```

Our form template just uses 3 line edit widgets. The `addRow` method makes it easy to specify the label and the widget.

```
l <- sapply(c("name", "rank", "snumber"), function(i) Qt$QLineEdit())
formLayout$addRow("Name", l$name)
formLayout$addRow("Rank", l$rank)
formLayout$addRow("Serial number", l$snumber)
```

Finally, we set the minimum size for our GUI and call `show` on the top-level widget.

```
w$setMinimumSize(width=500, height=400)
w$show()
```

2.1 Box layouts

Box layouts arrange child widgets by packing in values horizontally (from left to right although right to left is possible) or vertical. The `QHBoxLayout` constructor provides a horizontal layout whereas `QVBoxLayout` provides a

vertical one. Both of these classes subclass the `QBoxLayout` class where most of the functionality is documented. The `direction` property specifies how the layout is done. By default, this is left to right or top to bottom, but can be set (e.g., using `Qt$Qt$LeftToRight`).

Child widgets are added to a box container through the `addWidget` method. The basic call specifies just the child widget, but one can specify an integer value for `stretch` and an alignment enumeration.

How size gets allocated to the widgets managed by a layout depends on many things. The Qt documentation for layouts described this as follows. A widget has methods `sizeHint` to provide the needed size for the widget. The default policy is that this is the preferred size, but it can be resized. One can change the policy through the `setSizePolicy` method. One sets the value for the horizontal and vertical sizing. Values are taken from the `QSizePolicy` enumeration with possible values of `"Fixed"` (`Qt$QSizePolicy$Fixed`) for using the size hint; `"Minimum"`, for size hint being minimal and sufficient; `"Maximum"`, for the size hint being the maximal size; `"Preferred"`; `"Expanding"`, for able to expand to fill the space; `"Ignored"`, to ignore the size hint and give the widget all the possible space. For exmaple, a button, by default, will expand in the horizontal direction unless the size policy is adjusted:

```
b <- Qt$QPushButton("Stop the expansion")
b$setSizePolicy(Qt$QSizePolicy$Fixed, Qt$QSizePolicy$Fixed)
```

When more than one widget can expand, how the space gets allocated is determined by their stretch factors (set by the `addWidget` method). Widgets with a stretch factor of 0 only get space if no other widgets want the space. Of those, one with an expanding size policy get allocated space first.

If after this allocation, widgets haven't received their minimial size request, then space is allocated to do so. Then is a widget is allocated more than its maximal size, the allocation is shrunk.

Stretch factors are used to proportionally allocate space to widgets when they expand.

When a widget is allocated additional space, the child can have its alignment adjusted. The enumeration `Alignment` specifies values `"Left"` (`Qt$Qt$AlignLeft`), `"Right"`, `"HCenter"` (horizontal center), `"Justify"`, `"Top"`, `"Bottom"`, `"VCenter"` (vertical center), and `"Center"`. Values may be "or"ed together (e.g. `Qt\$Qt\$AlignLeft \| Qt\$Qt\$`

In addition to adding child widgets, one can nest child layouts through `addLayout`.

The child widgets are indexed (0-based). A count of child widgets is returned by `count`, and individual widget is retrieved by calling the `widget` method on the return value of the layout's `itemAt` method. The `insertWidget` can be used to insert a widget, with arguments similar to `addWidget`. Its initial argument is an integer specifying the `index`. All child widgets with this index or higher have their index increased by 1.

Removing a child Qt provides the methods `removeItem` and `removeWidget` to remove an widget from a layout. Once removed from one layout, these may be reparented if desired, or destroyed. This is done by setting the widget's parent to `NULL` using `setParent`.

Spacing The space between two children is controlled through the `setSpacing` method. This sets the common width in pixels, which can be adjusted individually through the `addSpacing` method. The margin area around all the children can be adjusted with the `setContentsMargins` method, although this is often specified through the style.

Springs and Struts A stretchable blank widget can be added through the `addStretch` method, where an integer is specified to indicate the stretch factor. If no other widgets have a stretch specified then this widget will take all the non-requested space. A strut (`addStrut`) can be specified to restrict to a minimum height.

```
w <- Qt$QWidget()
w$setWindowTitle("Box container example")
g <- Qt$QHBoxLayout()
w$setLayout(g)
g$addWidget(Qt$QLabel("left"))
g$addStretch(1L)
g$addWidget(Qt$QLabel("middle"))
g$addSpacing(12L)
g$addWidget(Qt$QPushButton("right"))
g$count()
```

Scrolling layouts

It may be desirable to constrain the size of a layout and allow the user to pan through its children with scrollbars. The `QScrollArea` class makes this very straightforward. The method `setWidget` is used to add the child widget. To ensure that a given widget is visible, the method `ensureWidgetVisible` is available, where the widget is passed as the argument.

```
sa <- Qt$QScrollArea()
w <- Qt$QWidget()
w$setMinimumSize(400,400)
lyt <- Qt$QVBoxLayout()
w$setLayout(lyt)
for(i in rownames(state.x77)) {
  msg <- sprintf("%s had a population of %s thousand in 1977",
                 i, state.x77[i,"Population"])
  lyt$addWidget(Qt$QLabel(msg))
}
```

```
sa$setWidget(w)
sa$show()
sa$raise()
```

To ensure the value for New York (row 32 in the data set) is visible, we have:

```
widget <- lyt$itemAt(32 - 1)$widget()
sa$ensureWidgetVisible(widget)
```

Framed Layouts

A frame with a title is a common decoration to a container often utilized to group together widgets that are naturally related. In Qt this layout is implemented through the `QGroupBox` widget. The method `setTitle` can be used to set the title, or it can be passed to the constructor. If the standard position of the title determined from the style is not to the likeing, it can be adjusted through the `setAlignment` method. This method takes an enumerated value from `Qt$Qt$AlignLeft`, `Qt$Qt$AlignHCenter` or `Qt$Qt$AlignRight`. The property `flat` can be set to `TRUE` to minimize the allocated space.

Group boxes have a `checkable` property that if enabled the widget will be drawn with a checkbox to control whether the children of the group box are sensitive to user input.

Separators

The `QGroupBox` widget provides a framed area to separate off related parts of GUI. Sometimes, just a separating line is all that is desired. There is no separate separator widget in Qt, unlike GTK+. However, the `QFrame` class provides a general method for framed widgets that can be used for this purpose with the appropriate settings.

The frame shape can be a box or others types but for this purpose a line is desired. The enumerations `Qt$QFrame$HLine` or `Qt$QFrame$VLine` can be passed to the method `setFrameShape` to specify a horizontal or vertical line. Its appearance can be altered by `setFrameShadow`. A value of `Qt$QFrame$Sunken` or `Qt$QFrame$Raised` is suitable.

2.2 Grid Layouts

The `QGridLayout` class provides a grid layout for aligning its child widgets into rows and columns.

The `addWidget` method is used to add a child widget to the layout and the `addLayout` method adds a nested layout. They have similar arugments. The widget (or layout) is the first argument followed by either a row and column index, a row and column index with an alignment enumeration, a row and

column index and a row span and column span amount (defaulting to 1), or in addition an alignment enumeration.

The method `itemAtPosition` returns the `QLayoutItem` instance corresponding to the specified row and column. This has a `widget` method to find the corresponding widget attached at that row-column cell. The methods `rowCount` and `columnCount` can be used to find the size of the grid.

Removing a widget is similar for a box layout using `removeItem` or `removeWidget`.

Rows and columns are dual concepts and are so implemented. Consequently, both have similar methods differing only by the use of `column` or `row/`. We discuss columns. A column minimum width can be set through `setColumnMinimumWidth`. If more space is available to a column than requested, then the extra space is apportioned according to the `stretch` factors. This can be set for a column through the `setColumnStretch` method. Taking an integer value 0 or larger.

The spacing between widgets can be set in both directions with `setSpacing`, or fine-tuned with `setHorizontalSpacing` or `setVerticalSpacing`. The style may set these too wide for some tastes.

Example 2.2: Using a grid layout

To illustrate grid layouts we mock up a GUI centered around a central text area widget. If the window is resized, we want that widget to get the extra space allocated to.

We begin by setting a grid layout for our parent widget.

```
w <- Qt$QWidget()
w$setWindowTitle("Layout example")
lyt <- Qt$QGridLayout()
w$setLayout(lyt)
```

We use the default left-alignment for labels in the following. Our first row has a label in column 1 and a text-entry widget spanning two columns.

```
lyt$addWidget(Qt$QLabel("Class:"), 0, 0)
lyt$addWidget(Qt$QLineEdit(), 0, 1, rowspan=1, colspan=2)
```

Our second row strats with a label and a combobox each spanning the default of one column.

```
lyt$addWidget(Qt$QLabel("Method:"), 1, 0)
lyt$addWidget(Qt$QComboBox(), 1, 1)
```

The third column of the second row and rest of the layout is managed by a sublayout, in this case a vertical box layout. We use a label as a stub and set a frame style to have it stand out.

```
lyt$addLayout(slyt <- Qt$QVBoxLayout(), 1, 2, rowspan=3, 1)
slyt$addWidget(l <- Qt$QLabel("Category\nSelector"))
l$setFrameStyle(Qt$QFrame$Box)
```

The text edit widget is added to the third row, second column. Since this widget will expand, we set an alignment for the label. Otherwise, the default alignment will center it in the vertical direction.

```
lyt$addWidget(Qt$QLabel("Text:"), 2, 0, Qt$Qt$AlignTop)
lyt$addWidget(l <- Qt$QTextEdit(), 2, 1)
```

Finally we add a space for information on the 4th row. Again we placd this in a box. By default the box would expand to fill the space of the two columns, but we fix this below as an illustration.

```
lyt$addWidget(l <- Qt$QLabel("More info:"), 3, 0, rowspan=1, colspan=2)
l$setSizePolicy(Qt$QSizePolicy$Fixed, Qt$QSizePolicy$Preferred)
l$setFrameStyle(Qt$QFrame$Box)
```

As it is, since there are no stretch factors set, the space allocated to each row and column would be identical when resized. To force the space to go to the text widget, we set a positive stretch factor for the 3rd row and 2nd column. Since the others have the default stretch factor of 0, this will allow that cell to grow when the widget is resized.

```
lyt$setRowStretch(2, 1)
lyt$setColumnStretch(1,1)
```

2.3 Form layouts

Forms can easily be generated with the grid layout, but Qt provides an even more convenient form layout (`QFormLayout`) that has the added benefit of conforming to the traditional styles for the underlying operating system. This can be used in combination with the `QDialogButtonBox` , which provides a container for buttons that also tries to maintain an appearance consistent with the underlying operating system.

To add a child widget with a label is done through the `addRow` method, where the label, specified first, may be given as a string and the widget is specified second. The first argument can also be a widget to replace the label, and the second a layout for nesting layouts. The `insertRow` method is similar only one first specifies the row number using a 0-based index. The `setSpacing` method can be used to adjust the default spacing between rows.

After construction, the widget may be retrieved through the `itemAt` method. This returns a layout item, to get the widget call its `widget` method. The `setWidget` method can be used to change a widget.

Example 2.3: Simple use of `QFormLayout`
The following illustrates a basic usage where three values are gathered.

```
w <- Qt$QWidget()
w$setWindowTitle("Example of a form layout")
```

```
w$setLayout(flyt <- Qt$QFormLayout())
l <- list()
flyt$addRow("mean", l$mean <- Qt$QLineEdit())
flyt$addRow("sd", l$sd <- Qt$QLineEdit())
flyt$addRow("n", l$n <- Qt$QLineEdit())
w$show(); w$raise()
```

The form style can be overridden using the `setFormAlignment` and `setLabelAlignment` methods. The Mac default is to have center aligned form with right-aligned labels, whereas the Windows default is to have left aligned labels. One can also override the default as to how the fields should grow when the widget is resized (`setFieldGrowthPolicy`).

```
flyt$setFormAlignment(Qt$Qt$AlignLeft | Qt$Qt$AlignTop)
flyt$setLabelAlignment(Qt$Qt$AlignLeft);
```

2.4 Notebooks

The notebook layout is provided by the widget `QTabWidget`. This is not a layout, rather a notebook page consists of a label and widget. Of course, you can use a layout for the widget.

Pages are added through the method `addTab`. One can specify a widget, a widget and label or a widget, icon and label. Pages are inserted by index with the `insertTab` method.

Tabs allow the user to select pages, and in Qt can be customized. The text for a tab is adjusted through `setTabText` and the icon through `setTabIcon`. These use a 0-based index to refer to the tab. A tooltip can be added through `setTabToolTip`. The tabs will have close buttons if the property `tabsClosable` is `TRUE`. One connects to the `tabCloseRequested` signal to close the tab. The tab position is adjusted through the `setTabPosition` method with enumerated values such as `Qt\$QTabWidget\$North`. Call `isMovable` with `TRUE` allows the pages to be reorganized by the user.

When there are numerous tabs, the method `setUsesScrollButtons` can indicate if the widget should expand to accomodate the labels or add scroll buttons.

The current tab is adjusted through the `currentIndex` property. The actual widget of the current tab is returned by `curentWidget`. To remove a page the `removeTab` is used, where tabs are referred to by index.

Example 2.4: A tab widget
A simple example follows. First the widget is defined with several properties set.

```
nb <- Qt$QTabWidget()
nb$setTabsClosable(TRUE)
nb$setMovable(TRUE)
```

```
nb$setUsesScrollButtons(TRUE)
```

We can add pages with a label or with a label and an icon:

```
nb$addTab(Qt$QPushButton("page 1"), "page 1")
icon <- Qt$QIcon("small-R-logo.jpg")
nb$addTab(Qt$QPushButton("page 2"), icon,  "page 2")
for(i in 3:10) nb$addTab(Qt$QPushButton(i), sprintf("path %s", i))
```

The close buttons put out a request for the page to be closed, but do not handle directly. Something along the lines of the following is then also needed.

```
qconnect(nb, "tabCloseRequested", function(index) {
  nb$removeTab(index)
})
```

QStackedWidget

The `QStackedWidget` is provided by Qt to hold several widgets at once, with only one being visible. Similar to a notebook, only without the tab decorations to switch pages. For this widget children are added through the `addWidget` method and can be removed with `removeWidget`. The latter needs a widget reference. The currently displayed widget can be found from `currentWidget` (which returns `NULL` if there are no child widgets). Alternatively, one can refer to the widgets by index, with `count` returning their number, and `currentIndex` the current one and `indexOf` returns the index of the widget specified as an argument.

2.5 Paned Windows

Split windows with handles to allocate space are created by `QSplitter`. There is no limit on the number of child panes that can be created. The default orientation is horizontal, set the `orientation` property using `Qt$Qt$Vertical` to change this.

Child widgets are added through the `addWidget` method. These widgets are referred to by index and can be retrieved through the `widget` method.

The `moveSplitter` method is not implemented, so programattically moving the a splitter handle is not possible.

```
sp <- Qt$QSplitter()
sp$addWidget(Qt$QLabel("One"))
sp$addWidget(Qt$QLabel("Two"))
sp$addWidget(Qt$QLabel("Three"))
sp$show()
sp$raise()
sp$setOrientation(Qt$Qt$Vertical)
```

```
sp$widget(0)$text
```

```
[1] "One"
```

2.6 Main windows

In Qt the `QMainWindow` class provides a widget for use as the primary widget in an interface. Although it is a subclass of `QWidget` – which we have used in our examples so far as a top-level window – the implementation also has preset areas for the standard menu bars, tool bars and status bars. As well, there is a possible dock area for "dockable widgets."

A main window has just one central widget that is specified through the `setCentralWidget` method. The central widget can then have a layout and numerous children. The title of the window is set from the inherited method `setWindowTitle`.

Actions

The menubars and toolbars are representations of collections of actions, defined through the `QAction` class. As with other toolkits, an action encapsulates a command that may be shared among parts of a GUI, in this case menu bars, toolbars and keyboard shortcuts. In Qt the action can hold the label (`setText`), an icon (`setIcon`), a status bar tip (`setStatusTip`), a tool tip (`setToolTip`), and a keyboard shortcut (`setToolTip`). The text and icon may be set at construction time, in addition to using the above methods.

Actions inherit the `enabled` method to toggle whether an action is sensitive to user input.

Keyboard shortcuts A keyboard shortcut use a `QKeySequence` to bind a key sequence to the command. Key sequences can be found from the standard shortcuts provided in the enumeration `Qt$QtKeySequence`, for example

```
Qt$QKeySequence$Cut
```

```
Cut
  8
attr(,"class")
[1] "QtEnum"
```

This value (or more simply the "Cut" string) can be passed to the constructor to create the shortcut.

Using these standard shortcuts ensures that the keyboard shortcut is the standard one for the underlying operating system. Alternatively, custom shortcuts can be used, such as

```
Qt$QKeySequence("Ctrl-X, Ctrl-C")
```

```
QKeySequence instance
```

This shows how a modifier can be specified (from `"Ctrl"`) a key (case insensitive), and how a comma can be used to create multi-key shortcuts.

For buttons and labels, a shortcut key can be specified by prefixing the text with an ampersand `&`, such as

```
button <- Qt$QPushButton("&Save")
```

Then the shortcut will by Alt-S.

The shortcut event occurs when the shortcut key combination is pressed in the appropriate context. The default context is when the widget is a child of the parent window, but this can be adjusted through the method `setShortcutContext`.

Checkable actions, and action groups Actions can be set checkable through the `setCheckable` method. When in a checked state, the `checked` property is `TRUE`. When a checkable action is checked the `toggled` signal is emitted, the argument `checked` passes in the state.

A `QActionGroup` can be used to group together checkable actions so that only one is checkable (like radio buttons). To use this, you create an instance and use the `addAction` to link the actions to the group.

Example 2.5: Creating an action
To create an action, say to "Save" an object requires a few steps. It is recommended that the main constructor is passed the parent widget the action will apply within.

```
parent <- Qt$QMainWindow()
saveAction <- Qt$QAction("Save", parent)
```

We could also pass the icon to the constructor, but instead set the icon, a shortcut, a tooltip, and a statusbar tip through the action's methods.

```
saveAction$setShortcut(Qt$QKeySequence("Save"))
iconFile <- system.file("images/save.gif", package="gWidgets")
saveAction$setIcon(Qt$QIcon(iconFile))
saveAction$setToolTip("Save the object")
saveAction$setStatusTip("Save the current object")
```

The action encapsulates a command, in this case we have a stub:

```
qconnect(saveAction, "triggered", function(checked) print("Save object"))
```

Menubars

Main windows may have a menubar. This may appear at the top of the window, or the menubar area on Mac OS X. Menubars are instances of `QMenuBar` which provides access to list of top-level submenus.

These submenus are added through `addMenu`, where a string with a possible shortcut are specied to label the menu. A `QMenu` instance is returned. To submenus one can add

1. Nested submenus through the `addMenu` method,
2. An action through the `addAction` method, or
3. a separator through `addSeparator`.

Actions may be removed from a window through `removeAction`, but usually menu items are just disabled if their command is not applicable.

Example 2.6: Menu items
In a data editor application, one might imagine a menu item for coercion of a chosen column from one type to the next. In the following, we assume we have a function `colType` that returns the column type of the selected column or `NA` if no column is selected. We begin by making a menu bar, and a "Data" menu item. To this we add a few actions, and then a "Coerce" submenu. In the submenu, we use an action group so that only one type can be checked at a time. Actions must be added to both the action group and the submenu.

```
mb <- Qt$QMenuBar() ## or parent$menuBar()
menu <- mb$addMenu("Data")
#
menu$addAction(a <- Qt$QAction("Apply Function...", parent))
qconnect(a, "triggered", function() cat("apply ..."))
menu$addAction(a <- Qt$QAction("Relevel Factors...", parent))
qconnect(a, "triggered", function() cat("relevel ..."))
#
menu$addSeparator()
#
cmenu <- menu$addMenu("Coerce")
aList <- sapply(c("character", "factor", "numeric"), function(i) {
  a <- Qt$QAction(i, parent)
  a$setCheckable(TRUE)
  qconnect(a, "toggled", function(checked) print(i)) ## stub
  a
})
actionGroup <- Qt$QActionGroup(w)
QT <- sapply(aList, function(i) actionGroup$addAction(i))
QT <- sapply(aList, function(i) cmenu$addAction(i))
```

In the application, we might include logic to update the menu items along the lines of the following. If no column type is available (no column is selected) we disable the sub menu, otherwise we set the check accordingly. Of course, in the application we would ensure that checking the menu item updates the state in the data model through the `triggered` handler.

```
updateMenus <- function() {
  val <- colType()
```

```
  cmenu$setEnabled(!is.na(val))
  if(!is.na(val)) {
    aList[[val]]$setChecked(TRUE)
  }
}
```

Context menus

Context menus can be added to widgets using the same `QMenu` widget (not `QMenuBar`). The `popup` method will cause the menu to popup, but it needs to be told where. (The `exec` method will also popup a menu, but blocks other input.) The location of the popup is specified in terms of global screen coordinates, but typically the location known is in terms of the widgets coordinates. (For example, the point $(0,0)$ being the upper-left corner of the widget.) The method `mapToGlobal` will convert for you. Position is in terms of a `QPoint` instance, which can be constructed or may be returned by an event handler. We illustrate both in the example.

Initiating the popup menu can be done in different ways. In the example below, we first show how to do it when a button is pressed. More natural ways are to respond to right mouse clicks, say. These events may be found within event handlers, say the `mousePressEvent` event. (The `QMouseEvent` object passed in has a `button` method that can be checked.) However, the operating system may provide other means to initiate a popup. Rather than program these, Qt provides the `contextMenuEvent`. We can override that in a subclass, as illustrated in the example.

Example 2.7: Popup menus
We imagine a desire to popup possible function names that complete a string. Such suggestions are computed from a function in the `utils` package. We show how to offer these in a popup menu we do this for a button press (not the most natural case):

```
b <- Qt$QPushButton("Completion example")
qconnect(b, "pressed", function(...) {
  ## compute popup
  popup <- Qt$QMenu()
  comps <- utils:::matchAvailableTopics("mean")
  l <- sapply(comps, function(i) popup$addAction(Qt$QAction(i, b)))
  popup$popup(b$mapToGlobal(Qt$QPoint(0,0)))
})
```

More naturally, we might want this menu to popup on a right mouse click in a line edit widget. To implement that, we define a subclass and reimplement the `contextMenuEvent` method. We use the `globalPos` method of the passed through event to get the appropriate position.

```
qsetClass("popupmenuexample", Qt$QLineEdit,
          function(parent=NULL) super(parent) )
#
qsetMethod("contextMenuEvent", popupmenuexample, function(e) {
  popup <- Qt$QMenu()
  comps <- utils:::matchAvailableTopics(this$text)
  if(length(comps) > 10)
    comps <- comps[1:10]                # trim if large
  sapply(comps, function(i) {
    a <- Qt$QAction(i, this)
    qconnect(a, "triggered", function(...) this$setText(i))
    popup$addAction(a)
  })
  popup$popup(e$globalPos())
})
b <- popupmenuexample()
```

Toolbars

Toolbars, giving easier access to a collection of actions or even widgets, can be readily added to a main window. The basic toolbar is an instance of the `QToolBar` class. Toolbars are added to the main window through the `addToolbar` method.

To add items to a toolbar we have:

1. `addAction` to add an action,
2. `addWidget` to embed a widget into the toolbar,
3. `addSeparator` to place a sepearator between items.

Actions can be removed through the `removeAction` method. The method `actions` will return a list of actions.

Toolbars can have a few styles. The orientation can be horizontal (the default) or vertical. The `setOrientation` method adjusts this with values specified by `Qt$Qt$Horizontal` or `Qt$Qt$Vertical`. The toolbuttons can show a combination of text and/or icons. This is specified through the method `setToolButtonStyle` with values taken from the `ToolButtonStyle` enumeration. The default is icon only, but one could use, say, `Qt$Qt$ToolButtonTextUnderIcon`.

Example 2.8: A simple toolbar
The following illustrates how to put in toolbar items to open and save a file. We suppose we have a function `getIcon` that returns a `QIcon` instance from a string.

We define a top-level window to hold our toolbar and be the parent for our actions that will be placed in the toolbar. We store them in a list for ease in manipulation at a later time in the program.

```
w <- Qt$QMainWindow()
```

```
a <- list()
a$open <- Qt$QAction("Open", w)
a$open$setIcon(getIcon("open"))
a$save <- Qt$QAction("Save", w)
a$save$setIcon(getIcon("save"))
```

We define our toolbar, set its button style and then add to top-level window in the next few commands.

```
tb <- Qt$QToolBar()
tb$setToolButtonStyle(Qt$Qt$ToolButtonTextUnderIcon)
w$addToolBar(tb)
```

Finally, we add the actions to the toolbar.

```
sapply(a, function(i) tb$addAction(i))
```

Statusbars

Main windows have room for a statusbar at the bottom of the window. The status bar is used to show programmed messages as well as any status tips assigned to actions.

A statusbar is an instance of the `QStatusBar` class. One may be added to a main window through the `setStatusBar` method. For some operating systems, a size grip is optional and its presence can be adjusted through the `sizeGripEnabled`roperty.

Messages may be placed in the status bar of three types: *temporary* where the message stays briefly, such as for status tips; *normal* where the message stays, but may be hidden by temporary messagesl and *permanent* where the message is never hidden. In addition to messages, one can embed widgets into the statusbar.

The `showMessage` method places a temporary message. The duration can be set by specifying a time in milliseconds for a second argument. Otherwise, the message can be removed through `clearMessage`.

Use `addWidget` with a label to make a normal message, use `addPermanentWidget` to make a permanent message.

Dockable widgets

In Qt main windows have dockable areas where one can anchor widgets that can be easily detached from the main window to float if the user desires. An example use might be a toolbar or in a large GUI, a place to dock a workspace browser. The main methods are `addDockWidet` and `removeDockWidget`. Adding a dock widget requires first creating an instance of `QDockWidget` and then setting the desired widget through the dock widgets `setWidget`. Widgets may go on any side of the central widget. The position is specified through the `DockWidgetArea` enumeration, with values such as `Qt$Qt$LeftDockWidgetArea`.

Dock widgets can be stacked or arranged in a notebook like manner. The latter is done by the `tabifyDockWidget`, which moves the second argument (a dock widget) on top of the first with tabs, like a notebook, for the user to select the widget.

Floating a dock widget is intitiated by the user through clicking an icon in the title bar or programatically through the `floating` property.

Example 2.9: Using a main widget for the layout of an IDE

This example shows how to mock up a main window similar to the one presented by the web application, R-Studio. (`rstudio.org`). We begin by setting a minimum size and a title for the main window.

```
w <- Qt$QMainWindow()
w$setMinimumSize(800, 500)
w$setWindowTitle("Rstudio-type layout")
```

We add in a menu bar and toolbar. The actions are minimal, not including icons, commands etc. We show the file menu definitions.

```
l <- list()
mb <- Qt$QMenuBar()
w$setMenuBar(mb)
fmenu <- mb$addMenu("File")
fmenu$addAction(l$new <- Qt$QAction("New", w))
fmenu$addSeparator()
fmenu$addAction(l$open <- Qt$QAction("Open", w))
fmenu$addAction(l$save <- Qt$QAction("Save", w))
fmenu$addSeparator()
fmenu$addAction(l$quit <- Qt$QAction("Quit", w))
```

The toolbar has just a few actions added.

```
tb <- Qt$QToolBar()
w$addToolBar(tb)
tb$addAction(l$new)
tb$addSeparator()
tb$addAction(l$quit)
tb$addSeparator()
tb$addAction(l$help <- Qt$QAction("Help", w))
```

Our central widget holds two main areas: one for editing files and one for a console. As we may want to edit multiple files, we use a tab widget for that. A `QSplitter` is used to divide the space between the two main widgets.

```
centralWidget <- Qt$QSplitter()
centralWidget$setOrientation(Qt$Qt$Vertical)
w$setCentralWidget(centralWidget)
```

```
fileNotebook <- Qt$QTabWidget()
fileNotebook$addTab(Qt$QLabel("File notebook"), "About")
fileNotebook$setTabsClosable(TRUE)
centralWidget$addWidget(fileNotebook)
```

Our console widget is just a stub.

```
consoleWidget <- Qt$QLabel("Console goes here")
centralWidget$addWidget(consoleWidget)
```

On the right side of the layout we will put in various tools for interacting with the R session. We place these into dock widgets, in case the user would like to place them elsewhere on the screen. Defining dock widgets is straightforward. We show a stub for a workspace browser.

```
workspaceBrowser <- Qt$QLabel("Workspace browser goes here")
wbDockWidget <- Qt$QDockWidget("Workspace")
wbDockWidget$setWidget(workspaceBrowser)
```

The workspace and history browser are placed in a notebook to conserve space. We add the workspace browser on the right side, then tabify the history browser.

```
w$addDockWidget(Qt$Qt$RightDockWidgetArea, wbDockWidget)
w$tabifyDockWidget(wbDockWidget, hbDockWidget)
```

We next place a notebook to hold any graphics produced in a dock widget. This one occupies its own space.

```
plotNotebook <- Qt$QTabWidget()
pnDockWidget <- Qt$QDockWidget("Plots")
w$addDockWidget(Qt$Qt$RightDockWidgetArea, pnDockWidget)
```

Finally, we make status bar and add a transient message.

```
sb <- Qt$QStatusBar()
w$setStatusBar(sb)
sb$showMessage("Mock-up layout for an IDE", 2000)
```

3

Widgets

3.1 Labels

3.2 Buttons

actions icons signals default
`QDialogButtonBox`

3.3 single-line text

3.4 Checkboxes

3.5 Radio groups

3.6 Comboboxes

3.7 Sliders and spinboxes

3.8 List views

3.9 Table widget

4

Widgets using an MVC framework

Example: model for workspace browser

4.1 Model-View-Controller in Qt

model, delegate, ...

4.2 Comboboxes

4.3 Table views

4.4 Tree views

5

Qt paint