1

Qt: Overview

1.1 The Qt library

Qt is an open-sourced, cross-platform application framework that is perhaps best known for its widget toolkit. The features of Qt are divided into about a dozen modules. We highlight some of the more important and interesting ones:
Core Basic utilties, collections, threads, I/O, ...
Gui Widgets, models, etc for graphical user interfaces
OpenGL Convenience layer (e.g., 2D drawing API) over OpenGL
Webkit Embeddable HTML renderer (shared with Safari, Chrome)
Other modules include functionality for networking, XML, SQL databases, SVG, and multimedia. However, R packages already provide many of those features.

The history of Qt begins with Haavard Nord and Eirik Chambe-Eng in 1991 and follows with the Trolltech company, until 2008. It is now owned by Nokia, a major cell-phone producer. While it was originally unavailable as open-source on every platform, version 4 was released universally under the GPL. With the release of Qt 4.5, Nokia additionally placed Qt under the LGPL, so it is available for use in proprietary software, as well. Popular software developed with Qt include the communication application Skype and the KDE desktop for Linux.

Qt is developed in C++ with extensions that require a special preprocessor called the *Meta Object Compiler* (`MOC`). The `MOC` allows for convenient syntax in the definition of signals, slots (signal handlers), and properties, which behave very similarly to those of GTK+.

There are many languages with bindings to Qt, and R is one such language. The `qtbase` package interfaces with every module of the library. As its name suggests, `qtbase` forms the base for a number of R packages that provide high-level special-purpose interfaces to Qt. The `qtpaint` package extends the `QGraphicsView` canvas to better support interactive statistical graphics. Features include: a layered buffering strategy, efficient spatial queries for mapping user actions to the data, and an OpenGL renderer

optimized for statistical plots. An interface resembling that of the `lattice` package is provided for `qtpaint` by the `mosaiq` package. The `cranvas` package builds on `qtpaint` to provide a collection of high-level interactive plots in the conceptual vein of GGobi. A number of general utilities are implemented by `qtutils`, including an object browser widget, an R console widget, and a conventional R graphics device based on `QGraphicsView`.

While `qtbase` is not yet as mature as `tcltk` and `RGtk2`, we include it in this book, as Qt compares favorably to GTK+ in terms of GUI features and excels in several other areas, including its fast graphics canvas and integration of the WebKit web browser. [1] In addition, Qt, as a commercially supported package, has thorough documentation of its API, including many C++ examples. However, the complexity of C++ and Qt may present some challenges to the R user. In particular, the developer should have a strong grounding in object-oriented programming and have a basic understanding of memory management.

The development of `qtbase` package is hosted on Github (`http://github.com/ggobi/qtbase`). It depends on the Qt framework, available as a binary install from `http://qt.nokia.com/`.

1.2 An introductory example

As a synopsis for how one programs a GUI using `qtbase`, we present a simple dialog allows the user to input a date. A detailed introduction to these concepts will follow this example.

After ensuring that the underlying libraries are installed, the package may be loaded like any other R package:

```
require(qtbase)
```

Constructors As with all other toolkits, Qtwidgets are objects, and the objects are created with constructors. 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.

```
window <- Qt$QWidget()
label <- Qt$QLabel("Date:")
edit <- Qt$QLineEdit()
button <- Qt$QPushButton("Ok")
```

The constructors are not found in the global environment, but rather in the `Qt` environment, an object exported from the `qtbase` namespace. As such, the `$` lookup operator is used.

[1] There is a GTK+ WebKit port, but it is not included with GTK+ itself.

Widgets in Qt have various properties that specify the state of the object. For example, the `windowTitle` property controls the title of a top-level widget:

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

Qt objects are represented as extended R environments, and every property is a member of the environment. The `$` function called above is simply that for environments.

Method calls tell an object to perform some behavior. Like properties, methods are accessible from the instance environment. For example, the `QLineEdit` widget supports an input mask that constrains user input to a particular syntax. 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 set an input mask we have:

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

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 Chapter 2, 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(label, row=0, column=0, rowSpan=1, columnSpan=1)
lyt$addWidget(edit, 0, 1, 1, 1)
lyt$addWidget(button, 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`:

```
window$setLayout(lyt)
```

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

```
window$show()
```

Callbacks As with other GUI toolkits, we add interactivity to our GUI by binding callbacks to certain signals. To react to the clicking of the button, the programmer attaches a handler to the `clicked`QAbstractButton signal using the `qconnect` function. The function requires the object, the signal name and the handler. Here we print the value stored in the "Date" field.

```
handler <- function(checked)  print(edit$text)
qconnect(button, "pressed", handler)
```

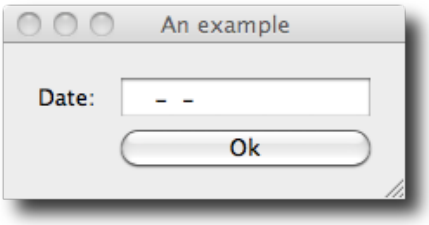

Figure 1.1: Screenshot of our sample GUI to collect a date from the user.

We will discuss callbacks more completely in Section 1.6.

Object-oriented support `QLineEdit` can validate text input, and we would like to validate the entered date. There are a few built-in validators, and for this purpose the regular expression validator could be used, but it is would be difficult to write a sufficiently robust expression. Instead we attempt to coerce the string value to a date via R's `as.Date` function with a format of `"%Y-%m-%d"`. In GTK+, validation would be implemented by a signal handler or other callback. However, as C++ is object-oriented, Qt expects the programmer to derive a new class from `QValidator` and pass an instance to the `setValidator` method on `QLineEdit`.

It is possible to define R subclasses of C++ classes with `qtbase`. More details on working with classes and methods are provided in Section 1.8. For this task, we need to extend `QValidator` and override its `validate` virtual method. The `qsetClass` function defines a new class:

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

To override `validate`, we call `qsetMethod`:

```
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)
})
```

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 indicates a state of "Acceptable", "Invalid", or, if neither can be determined, "Intermediate." These values are listed in an enumeration in the `Qt$QValidator` class (cf. Section 1.7 for more on enumerations).

The class object, which doubles as the constructor, is defined in the current top-level environment as a side effect of `qsetMethod`. We call it to construct an instance, which is passed to the edit widget:

```
validator <- DateValidator()
edit$setValidator(validator)
```

1.3 Classes and objects

The `qtbase` package exports very few objects. The central one is an environment, `Qt`, that represents the Qt library in R. [2] The components of this environment are `RQtClass` objects that represent an actual C++ class or namespace. For example, the `QWidget` class is represented by `Qt$QWidget`:

```
Qt$QWidget
```

```
Class 'QWidget' with 320 public methods
```

An `RQtClass` object contains methods in the class scope (*static* methods in C++), enumerations defined by the class, and additional `RQtClass` objects representing nested classes or namespaces. Here we list some of the components of `QWidget` and access one of the enumeration values:

```
head(names(Qt$QWidget), n = 3)
```

```
[1] "connect"              "DrawChildren"         "DrawWindowBackground"
```

```
Qt$QWidget$DrawChildren
```

```
Enum value: DrawChildren (2)
```

Most importantly, however, an instance of `RQtClass` is in fact an R `function` object, and serves as the constructor of instances of the class. For example, we could construct an instance of `QWidget`:

```
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"
```

[2] The `Qt` object is an instance of `RQtLibrary`. The `qtbase` package provides infrastructure for binding any conventional C++ library, even those independent of Qt. Third party packages can define their own `RQtLibrary` object for some other library.

The base class, `RQtObject`, is an environment containing the properties and methods of the instance. For `w`, we list the first few using `ls`:

```
head(ls(w))
```

```
[1] "setHidden"          "setWindowModified" "setSizeIncrement"
[4] "winId"              "window"            "setStyleSheet"
```

Properties and methods are accessed from the environment in the usual manner. The most convenient extractor is the `$` operator, but `[[` and `get` will also work. (With the `$` operator at the command line, completion works.) For example, a `QWidget` has a `windowTitle` property which is used when the widget draws itself with a window:

```
w$windowTitle                              # initially NULL
```

```
NULL
```

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

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

Although Qt defines methods for accessing properties, the R user will normally invoke methods that perform some action. For example, we could show our widget:

```
w$show()
```

The environment structure of the object masks the fact that the properties and 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()
```

`QPushButton` extends `QWidget` and thus inherits the properties like `windowTitle`:

```
is(b, "QWidget")
```

```
[1] TRUE
```

```
b$windowTitle
```

```
NULL
```

It is important to realize this distinction when referencing the documentation. As with GTK+, the methods are documented with the class that declares the method.

1.4 Methods and dispatch

In C++, it is possible to have multiple methods and constructors with the same name, but different signatures. This is called *overloading*. An overloaded method is roughly similar to an S4 generic, save the obvious difference that an S4 generic does not belong to any class. The selected overload is that with the signature that best matches the types of the arguments. The exact rules of overload resolution are beyond our scope.

It is particularly common to overload constructors. For example, a simple push button can be constructed in several different ways. Here again is the invocation of the `QPushButton` constructor with no arguments:

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

By convention, all classes derived from `QObject`, including `QWidget`, provide a constructor that accepts a parent `QObject`. This has important consequences that are discussed later. We demonstrate this for `QPushButton`:

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

An alternative constructor for `QPushButton` accepts the text for the label on the button:

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

Buttons may also have icons, for example

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

We have passed three different types of object as the first argument to `Qt$QPushButton`: a `QWidget`, a string, and finally a `QIcon`. The dispatch depends only on the type of argument, unlike the constructors in `RGtk2`, which dispatch based on which arguments are specified. (In particular, dispatch is based on position of argument, but not on names given to arguments. We use names only for clarity in our examples.)

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, to learn the methods for a simple button, we would call:

```
out <- qmethods(Qt$QPushButton)
dim(out)
```

```
[1] 436   5
```

```
head(out)
```

```
         name       return                                signature protected
1 QPushButton QPushButton*                            QPushButton()
FALSE
2 QPushButton QPushButton*                   QPushButton(QWidget*)
FALSE
3 QPushButton QPushButton*             QPushButton(QIcon, QString)
FALSE
4 QPushButton QPushButton* QPushButton(QIcon, QString, QWidget*)
FALSE
5 QPushButton QPushButton*                    QPushButton(QString)
FALSE
6 QPushButton QPushButton*          QPushButton(QString, QWidget*)
FALSE
  static
1  FALSE
2  FALSE
3  FALSE
4  FALSE
5  FALSE
6  FALSE
```

1.5 Properties

Every `QObject`, which includes every widget, may declare a set of properties that represent its state. We list some of the available properties for our button:

```
head(qproperties(b))
```

```
                             type readable writable
objectName                 QString     TRUE     TRUE
modal                         bool     TRUE    FALSE
windowModality Qt::WindowModality     TRUE     TRUE
enabled                       bool     TRUE     TRUE
geometry                     QRect     TRUE     TRUE
frameGeometry                QRect     TRUE    FALSE
```

As shown in the table, every property has a type and logical settings for whether the property is readable and/or writeable. Virtually every property value may be read, and it is common for properties to be read-only. For example, we can fully manipulate the `objectName` property, but our attempt to modify the `modal` property fails:

```
b$objectName <- "My button"
b$objectName
```

```
[1] "My button"
```

```
b$modal
```

```
[1] FALSE
```

```
try(b$modal <- TRUE)
```

Qt provides accessor methods for getting and setting properties. The getter methods have the same name as the property, so they are masked at the R level. Setter methods are available and are typically named with the word "set" followed by the property name:

```
b$setObjectName("My button")
```

However, it is recommended to use the replacement syntax shown in the previous example, for the sake of symmetry.

1.6 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 define a special type of method known as a slot in another component (or the same) that can be connected to the signal as the handler. The two components are decoupled as the emitter does not need to know about the receiver except through the signal connection. In R, any function can be treated as a slot and connected as a signal handler. This is similar to the signal handling in `RGtk2`. The function `qconnect` establishes the connection of an R function to a signal. For example

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

Signals are defined by a class and are inherited by subclasses. Here, we list some of the available signals for the `QPushButton` class:

```
tail(qsignals(Qt$QPushButton))
```

```
                         name                              signature
3 customContextMenuRequested customContextMenuRequested(QPoint)
4                     pressed                           pressed()
5                    released                          released()
6                     clicked                       clicked(bool)
7                     clicked                           clicked()
8                     toggled                       toggled(bool)
```

The signal definition specifies the callback signature, given in the `signature` column. Like other methods, signals can be overloaded so that there are multiple signatures for a given signal name. Signals can also have

default arguments, and arguments with a default value are optional in the signal handler. We see this for the `clicked` signal, where the `bool` (logical) argument, indicating whether the button is checked, has a default value of `FALSE`. The `clicked` signal is automatically overloaded with a signature without any arguments.

The `qconnect` function attempts to pick the correct signature by considering the number of formal arguments in the callback. Rarely, two signatures will have the same number of arguments, in which case one will be chosen arbitrarily. To connect to a specific signature, the full signature, rather than only the name, should be passed to qconnect. For example, the following two calls are equivalent:

```
qconnect(b, "clicked", function(checked) print(checked))
qconnect(b, "clicked(bool)", function(checked) print(checked))
```

Any object passed to the optional argument `user.data` is passed as the last argument to the signal handler. The user data serves to parameterize the callback. In particular, it can be used to pass in a reference to the sender object itself, although we encourage the use of closures for this purpose.

The `qconnect` function returns a dummy `QObject` instance that provides the slot that wraps the R function. This dummy object can be used with the `disconnect` method on the sender to break the signal connection:

```
proxy <- qconnect(b, "clicked", function() print("ouch"))
b$disconnect(proxy)
```

```
[1] TRUE
```

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.

Unlike GTK+, Qt widgets generally do not emit hardware events, such as a mouse press, via signals. Instead, a method in the widget is invoked upon receipt of an event. The developer is expected to extend the widget and override the method to catch the event. The apparent philosophy of Qt is that hardware events are low-level and thus should be handled by the widget, not some other instance. We will discuss extending classes in Section **??**. Example 1.2 demonstrates handling widget events.

1.7 Enumerations and flags

Often, it is useful to have discrete variables with more than two states, in which case a logical value is no longer sufficient. For example, the label widget has a property for how its text is aligned. It supports the alignment styles left, right, center, top, bottom, etc. These styles are enumerated by

integer values and Qt defines these by name within the relevant class or, for global enumerations, in the `Qt` namespace. Here are examples of both:

```
Qt$Qt$AlignRight
```

```
Enum value: AlignRight (2)
```

```
Qt$QSizePolicy$Exanding
```

```
NULL
```

The first is the value for right alignment from the `Alignment` enumeration in the `Qt` namespace, while the second is from the `Policy` enumeration in the `QSizePolicy` class.

Although these enumerations can be specifed directly as integers, they are given the class `QtEnum` and have the overloaded operators | and & to combine values bitwise. This makes the most sense when the values correspond to bit flags, as is the case for the alignment style. For example, aligning the text in a label in the upper right can be done through

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

To check if the alignment is to the right, we could query by:

```
as.logical(l$alignment & Qt$Qt$AlignRight)
```

```
[1] FALSE
```

1.8 Extending Qt Classes from R

As Qt is implemented in an object-oriented language, C++, the designers of the API expect the developer to extend Qt classes, like `QWidget`, during the normal course of GUI development. This is a significant difference from GTK+, where it is only necessary to extend classes when one needs to fundamentally alter the behavior of a widget. The `qtbase` package allows the R user to extend C++ classes in order to enhance the features of Qt. The `qtbase` package includes functions `qsetClass` and `qsetMethod` to create subclasses and their methods. Methods may override virtual methods in an ancestor C++ class, and C++ code will invoke the R/ implementation when calling the overridden virtual. Properties may be defined with a getter and setter function. If a type is specified, and the class derives from `QObject`, the property will be exposed by Qt. It is also possible to store arbitrary objects in an instance of an R/ class; we will refer to these as dynamic fields. They are private to the class but are otherwise similar to attributes on any R object. Their type is not checked, and they are useful as a storage mechanism for implementing properties.

Defining a Class

Here, we extend `QMessageBox` to create a dialog, shown when the application is closing, that asks the user whether a document should be saved:

```
qsetClass("SubClass", Qt$QWidget)
```

This creates a variable named `SubClass` in the workspace:

```
SubClass
```

```
Class 'R::.GlobalEnv::SubClass' with 319 public methods
```

Its value is an `RQtClass` object that behaves like the `RQtClass` for the built-in classes, such as `Qt$QWidget`. There are no static methods or enumerations in an `R/` class, so the class object is essentially the constructor:

```
instance <- SubClass()
```

By default, the constructor delegates directly to the constructor in the parent class. A custom constructor is often useful, for example, to initialize fields or to make a compound widget. The function implementing the constructor should be passed as the `constructor` argument. By convention, `QObject` subclasses should provide a `parent` constructor argument, for specifying the parent object. A typical usage would be

```
qsetClass("SubClass2", Qt$QWidget, function(title, prop, parent=NULL) {
  super(parent)
  this$property <- prop
  setWindowTitle(title)
})
```

Within the body of a constructor, the `super` variable refers to the constructor of the parent class, often called the "super" class. In the above, we call `super` to delegate the registration of the parent to the `QWidget` constructor. Another special symbol in the body of a constructor is `this`, which refers to the instance being constructed. We can set and implicitly create fields in the instance by using the same syntax as setting properties.

Defining Methods

One may define new methods, or override methods from a base class through the `qsetMethod` function. For example, accessors for a field may be defined with

```
qsetMethod("field", SubClass, function() field)
qsetMethod("setField", SubClass, function(value) {
  this$field <- value
})
```

For an override of an existing method to be visible from C++, the method must be declared virtual in C++. The `access` argument specifies the scope of the method: `"public"`, `"protected"`, or `"private"`. These have the same meaning as in C++.

As with a constructor, the symbol `this` in a method definition refers to the instance. There is also a `super` function that behaves similarly to the `super` found in a constructor: it searches for an inherited method of a given name and invokes it with the passed arguments:

```
qsetMethod("setVisible", SubClass, function(value) {
  message("Visible: ", value)
  super("setVisible", value)
})
```

In the above, we intercept the setting of the visibility of our widget. If we hide or show the widget, we will receive a notification to the console:

```
instance$show()
```

This is somewhat similar to the behavior of `callNextMethod`, except `super` is not restricted to calling the same method.

Defining Signals and Slots

Two special types of methods are slots and signals, introduced earlier in the chapter. These exist only for `QObject` derivatives. Most useful are signals. Here we define a signal:

```
qsetSignal("somethingHappened", SubClass)
```

```
[1] "somethingHappened"
```

If the signal takes an argument, we need to indicate that in the signature:

```
qsetSignal("somethingHappenedAtIndex(int)", SubClass)
```

```
[1] "somethingHappenedAtIndex"
```

Writing a signature requires some familiarity with C/C++ types and syntax, but this is concise and consistent with how Qt describes its methods. Although almost always public, it is possible to make a signal protected or private, via the `access` argument.

Defining a slot is very similar to defining a signal, except a method implementation must be provided as an R function:

```
qsetSlot("doSomethingToIndex(int)", SubClass, function(index) {
  # ....
})
```

```
[1] "doSomethingToIndex"
```

The advantage of a slot compared to a method is that a slot is exposed to the Qt metaobject system. This means that a slot could be called from another dynamic environment, like from Javascript running in the

`QScript` module or via the D-Bus through the `QDBus` module. It is also necessary to use slots as signal handlers for a GUI built with `QtDesigner`, if one is using the automated connection feature, see Section **??**.

Defining Properties

A property, introduced earlier, is a self-describing field that is encapsulated by a getter and a setter. We can define a property on any class using the `qsetProperty` function. Here is the simplest usage:

```
qsetProperty("property", SubClass)
```

```
[1] "property"
```

We can now access `property` like any other property; for example:

```
instance <- SubClass()
instance$property
```

```
NULL
```

```
instance$property <- "value"
instance$property
```

```
[1] "value"
```

However, the property is not actually exposed by the Qt meta object system. That requires specifying the `type` argument, which we will cover later.

By default, the property value is actually stored as a (private) field in the object, called `".property"`. One can override the default behavior by specifying a function for the getter and/or the setter:

```
qsetProperty("checkedProperty", SubClass, write = function(x) {
  if (!is(x, "character"))
    stop("'checkedProperty' must be a character vector")
  this$.checkedProperty <- x
})
```

```
[1] "checkedProperty"
```

We have taken advantage of the setter override to check the validity of the incoming value. If `"NULL"` is passed as the `write` argument, the property is read-only. One might also want to override the `read` function, for cases

where a property depends only on other properties or on some external resource.

To automatically emit a signal whenever a property is set, one can pass the name of the signal as the `notify` of `qsetProperty`:

```
qsetSignal("propertyChanged", SubClass)
```

```
[1] "propertyChanged"
```

```
qsetProperty("property", SubClass, notify = "propertyChanged")
```

```
[1] "property"
```

If a class derives from `QObject`, as does any widget, we can specify the C++ type of the property to expose it to the Qt meta object system:

```
qsetProperty("typedProperty", SubClass, type = "QString")
```

```
tail(qproperties(SubClass()), 1)
```

```
                 type readable writable
typedProperty QString     TRUE     TRUE
```

We see that the type is now exposed via the general `qproperties` function. Specifying the type enables all of the features of a Qt property.

Example 1.1: A model for workspace objects
In this example, we revisit creating a backend storage model for a workspace browser (cf. Example **??**). With `gWidgets`, we needed to define an Observable class for our model. With Qt, we can leverage the existing signal framework to notify views of changes to the model. This example demonstrates only the model; implementing a view is left to the reader.

Our basic model subclasses `QObject`, not `QWidget`, as it has no graphical representation – a job left for its views:

```
qsetClass("WSModel", Qt$QObject, function(parent=NULL) {
  super(parent)
  updateVariables()
})
```

```
Class 'R::.GlobalEnv::WSModel' with 50 public methods
```

We have two main properties: a list of workspace objects and a digest hash for each, which we use for comparison purposes. The digest is generated by the `digest` package, which we load:

```
library(digest)
```

We store the digest in a property:

```
qsetProperty("digest", WSModel)
```

```
[1] "digest"
```

When a new object is added, an object is deleted, or an object is changed, we wish to signal that occurence to any views of the model. For that purpose, we define a new signal below:

```
qsetSignal("objectsChanged", WSModel)
```

```
[1] "objectsChanged"
```

We then specify this signal name to the `notify` argument when defining the `objects` property, so that assignment will emit the signal:

```
qsetProperty("objects", WSModel, notify="objectsChanged")
```

```
[1] "objects"
```

Our class has a method to update the variable list. This simply compares the digest of the current workspace objects with a cached list. If there are differences, we update the objects, which in turn signals a change.

```
qsetMethod("updateVariables", WSModel, function() {
  x <- sort(ls(envir=.GlobalEnv))
  objs <- sapply(x, function(i)
              digest(get(i, envir=.GlobalEnv)))

  if((length(objs) != length(digest)) ||
     length(digest) == 0 ||
     any(objs != digest)) {
    this$digest <- objs       # update cache
    this$objects <- x         # emit signal
  }
  invisible()
})
```

We assign a callback to notify us that objects were changed:

```
m <- WSModel()
qconnect(m, "objectsChanged", function() message("objects were updated"))
```

Finally, we arrange to call the update function as needed. If the workspace size is modest, using a task callback is a reasonable strategy:

```
addTaskCallback(function(expr, value, ok, visible) {
  m$updateVariables()
  TRUE
})
```

1.9 QWidget Basics

The widgets we discuss in the next section 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. It implements the event processing and property systems. The QWidget class is the base class for all widgets and implements their shared functionality.

Upon construction, widgets are invisible, so that they may be configured behind the scenes. The visible property controls whether a widget is visible.

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

```
[1] FALSE
```

```
w$visible <- TRUE
w$visible
```

```
[1] TRUE
```

The show and hide methods are the corresponding convenience functions for making a widget visible and invisible, respectively.

```
w$show()
```

```
NULL
```

```
w$visible
```

```
[1] TRUE
```

```
w$hide()
```

```
NULL
```

```
w$visible
```

```
[1] FALSE
```

There is an S3 method for print on QWidget that invokes show. Whenever a widget is shown, all of its children are also made visible. The method raise will raise the window to the top of the stack of windows.

Similary, the property enabled controls whether a widget is sensitive to user input, including mouse events.

```
b <- Qt$QPushButton("button")
b$enabled <- FALSE
b$enabled
```

```
[1] FALSE
```

Only one widget can have the keyboard focus at once. The user shifts the focus by tab-navigation or mouse clicks (unless customized, see `focusPolicy`). When a widget has the focus, its `focus` property is `TRUE`. The property is read-only; the focus may be shifted to a widget by calling its `setFocus` method.

Qt has a number of mechanisms for the user to query a widget for some description of its purpose and usage. Tooltips, stored as a string in the `toolTip` property, may be shown when the user hovers the mouse over the widget. Similarly, the `statusTip` property holds a string to be shown in the statusbar instead of a popup window. Finally, Qt/ provides a "What's This?" tool that will show the text in the `whatsThis` property in response to query, such as pressing SHIFT+F1 when the widget has focus.

Except for top-level windows, the position and size of a widget are determined automatically by a layout algorithm; see Chapter 2. To specify the size of a top-level window, manipulate the `size` property, which holds a `QSize` object:

```
w$size <- qsize(400, 400)
## or
w$resize(400, 400)
```

```
NULL
```

```
w$show()
```

```
NULL
```

We create the `QSize` object with the `qsize` convenience function implemented by the `qtbase` package. The `resize` method is another convenient shortcut. One should generally configure the size of a window before showing it. This helps the window manager optimally place the window.

Fonts

Fonts in Qt are represented by the `QFont` class. The `qtbase` package defines a convenience constructor for `QFont` called `qfont`. The constructor accepts 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 font should be italicized, as a logical. Defaults are obtained from the application font, returned by `Qt$QApplication$font()`.

For example, we could create a 12 point, bold, italicized font from the helvetica family:

```
f <- qfont(family="helvetica", pointsize=12,  weight=Qt$QFont$Bold,
           italic=TRUE)
```

The font for a widget is stored in the font property. For example, we change the font for a label:

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

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

Palette Every platform has its own distinct look and feel, and an application should conform to platform conventions. Qt hides these details from the application. Every widget has a palette, stored in its paletteroperty and represented by a QPalette object. A QPalette maps the state of a widget to a group of colors that is used for painting the widget. The possible states are active, inactive and disabled. Each color within a group has specific role, as enumerated in QPalette::ColorRole. Examples include the color for background (Window), the foreground (WindowText) and the selected state (Highlight). Qt chooses the correct default palette depending on the platform and the type of widget. One can change the colors used in rendering a widget by manipulating the palette, 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. It is possible to customize the rendering of a widget using CSS syntax. The supported syntax is described in the overview on stylesheets provided with Qt documentation and is not summarized here, as it is quite readable.

The style sheet for a widget is stored in its styleSheet property, as a string. For example, for a button, we could set the background to white and the foreground to red:

```
b <- Qt$QPushButton("Style sheet example")
b$show()
```

```
NULL
```

```
b$styleSheet <- "QPushButton {color: red; background: white}"
```

The CSS syntax may be unfamiliar to R/ programmers, so the qtbase package provides an alternative interface that is reminiscent of the par function. We specify the above stylesheet in this syntax:

```
qsetStyleSheet(color = "red", background = "white", what = "QPushButton",
               widget = b)
```

The `widget` argument defaults to `NULL`, which applies the stylesheet application-wide through the `QApplication` instance. The default for `what` is `"*"`, meaning that the stylesheet applies to any widget class. The following would cause all widgets in the application to have the same colors as the button:

```
qsetStyleSheet(color = "red", background = "white")
```

Example 1.2: A "error label"

This example extends the line edit widget to display an error state via an icon embedded within the entry box. Such a widget might prove useful when one is validating entered values. Our implementation uses a stylesheet to place the icon in the background and to prevent the text from overlapping the icon.

To indicate an error, we will add an icon and set the tooltip to display an informative message. The constructor will be the default, so our class is defined with:

```
qsetClass("LineEditWithError", Qt$QLineEdit)
```

The main method sets the error state. We use stye sheets to place an image to the left of the entry message and set the tooltip.

```
qsetMethod("setError", LineEditWithError, function(msg) {
  f <- system.file("images/cancel.gif", package="gWidgets")
  qsetStyleSheet("background-image" = sprintf("url(%s)", f),
                 "background-repeat" = "no-repeat",
                 "background-position" = "left top",
                 "padding-left" = "20px",
                 widget = this)
  setToolTip(msg)
})
```

```
[1] "setError"
```

We can clear the error by resetting the properties to `NULL`.

```
qsetMethod("clearError", LineEditWithError, function() {
  setStyleSheet(NULL)
  setToolTip(NULL)
})
```

Finally, to test this out, we have the following

```
e <- LineEditWithError()
e$text <- "The quick brown fox..."
e$setError("Replace with better boilerplate text")
```

```
NULL
```

1.10 Importing a GUI from QtDesigner

`QtDesigner` is a tool for graphical, drag-and-drop design of GUI forms. Although this book focuses on constructing a GUI by programming in R, we recognize that a graphical approach may be preferable in some circumstances. `QtDesigner` outputs a GUI definition as an XML file in the `"UI"` format. The `QUiLoader` class loads a `"UI"` definition through its `load` method:

```
loader <- Qt$QUiLoader()
widget <- loader$load(Qt$QFile("designer.ui"))
```

The `widget` object could be shown directly; however, we first need to implement the behavior of the GUI by connecting to signals. Through the `QtDesigner` GUI, the user can connect signals to slots on built-in widgets. This works for some trivial cases, but in general one needs to handle signals with R code. There are two ways to accomplish this: manual or automatic.

To manually connect an R handler to a signal, we first need to obtain the widget with the signal. Every widget in a `UI` file is named, so we can call the `qfindChild` utility function to find a specific widget. Assume we have a button named "findButton" and corresponding text entry "findEntry" in our UI file, then we retrieve them with

```
findButton <- qfindChild(widget, "findButton")
findEntry <- qfindChild(widget, "findEntry")
```

Then we connect to the `clicked`QPushButton signal:

```
qconnect(findButton, "clicked", function() {
  findText(findEntry$text)
})
```

Alternatively, we could establish the signal connections automatically. This requires defining each signal handler to be a slot in the parent object, which will need to be of a custom class:

```
qsetClass("MyMainWindow", Qt$QWidget, function() {
  Qt$QMetaObject$connectSlotsByName(this)
})
```

```
Class 'R:.GlobalEnv:MyMainWindow' with 318 public methods
```

The constructor calls `connectSlotsByName` to automatically establish the connections. For a slot to be connected to the correct signal, it must be named according to the convention `"on_[objectName]_[signalName]"`. For example,

```
qsetSlot("on_findButton_clicked", MyMainWindow, function() {
  findText(findEntry$text)
})
```

In the case of a large, complex GUI, this automatic approach is probably more convenient than manually establishing the connections.

2

Qt: Layout Managers and Containers

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 determining the geometry of child widgets, according to a specific layout algorithm. Layout managers will generally update the layout whenever a parameter is modified, a child widget is added or removed, or the size of the parent changes. Unlike GTK+, where this management is tied to a container object, Qt decouples the layout from the widget.

This chapter will introduce the available layout managers, of which there are three types: the box (`QBoxLayout`), grid (`QGridLayout`) and form (`QFormLayout`). Widgets that function primarily as containers, such as the frame and notebook, are also described here.

Example 2.1: Synopsis of Layouts in Qt

This example uses a combination of different layout managers to organize a reasonably complex GUI. It serves as a synopsis of the layout functionality in Qt. A more gradual and detailed introduction will follow this example. Figure 2.1 shows a screenshot of the finished layout.

First, we need a widget as the top-level container. We assign a grid layout to the window for arranging the main components of the application:

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

There are three objects managed by the grid layout: a table (we use a label as a placeholder), a notebook, and a horizontal box layout for some buttons. We construct them

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

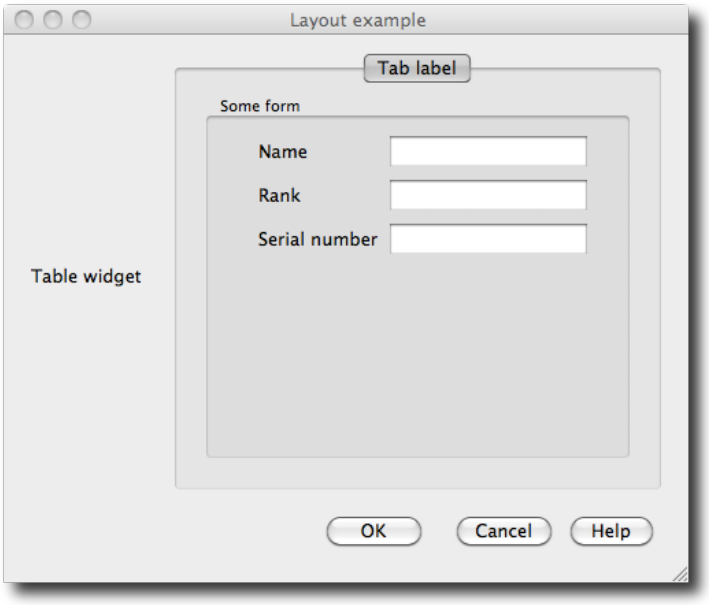

Figure 2.1: A mock GUI illustrating various layout managers provided by Qt.

and add them to the grid layout:

```
gridLayout$addWidget(tableWidget, row=0, column=0,
                     rowspan=1, colspan=1)
```

```
NULL
```

```
gridLayout$addWidget(nbWidget, 0, 1)
```

```
NULL
```

```
gridLayout$addLayout(buttonLayout, 1, 1)
```

```
NULL
```

We add construct our buttons and add them to the box:

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

We add a stretch, which acts much like a spring, to pack our buttons against the right side of the box. A fixed space of 12 pixels is inserted between the "Help" and "Cancel" buttons.

The notebook widget is constructed next, with a single page:

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

The form layout allows us to create standardized forms where each row contains a label and a widget. Although this could be done with a grid layout, using the form layout is more convenient, and allows Qt to style the page as appropriate for the underlying operating system. We place a form layout in the notebook page and populate it:

```
formLayout <- Qt$QFormLayout()
nbPage$setLayout(formLayout)
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)
```

Each `addRow` call adds a label and an adjacent input widget, in this case a text entry.

This includes our cursory demonstration of layout in Qt. We have constructed a mock-up of a typical application layout using the box, grid and form layout managers.

2.1 Layout Basics

Adding and Manipulating Children

We will demonstrate the basics of layout in Qt with a horizontal box layout, `QHBoxLayout`:

```
layout <- Qt$QHBoxLayout()
```

`QHBoxLayout`, like all other layouts, is derived from the `QLayout` base class. Details specific to box layouts are presented in Section **??**.

A layout is not a widget. Instead, a layout is set on a widget, and the widget delegates the layout of its children to the layout object:

```
wid <- Qt$QWidget()
wid$setLayout(layout)
```

```
NULL
```

Child widgets are added to a container through the `addWidget` method:

```
layout$addWidget(Qt$QPushButton("Push Me"))
```

```
NULL
```

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

Internally, layouts store child components as items of class `QLayoutItem`. The item at a given zero-based index is returned by `itemAt`. We get the first item in our layout:

```
item <- layout$itemAt(0)
```

The actual child widget is retrieved by calling the `widget` method on the item:

```
button <- item$widget()
```

Qt provides the methods `removeItem` and `removeWidget` to remove an item or widget from a layout:

```
layout$removeWidget(button)
```

```
NULL
```

Although the widget is no longer managed by a layout, its parent widget is unchanged. The widget will not be destroyed (removed from memory) as long as it has a parent. Thus, to destroy a widget, one should set the parent of the widget `NULL` using `setParent`:

```
button$setParent(NULL)
```

```
NULL
```

Size and Space Negotiation

The allocation of space to child widgets depends on several factors. The Qt documentation for layouts spells out the steps: [1]

1. All the widgets will initially be allocated an amount of space in accordance with their sizePolicy and sizeHint.
2. If any of the widgets have stretch factors set, with a value greater than zero, then they are allocated space in proportion to their stretch factor.
3. If any of the widgets have stretch factors set to zero they will only get more space if no other widgets want the space. Of these, space is allocated to widgets with an Expanding size policy first.

[1] `http://doc.qt.nokia.com/4.6/layout.html`

Table 2.1: Possible size policies

Policy	Meaning
Fixed	to require the size hint exactly
Minimum	to treat the size hint as the minimum, allowing expansion
Maximum	to treat the size hint as the maximum, allowing shrinkage
Preferred	to request the size hint, but allow for either expansion or shrinkage
Expanding	to treat like `Preferred`, except the widget desires as much space as possible
MinimumExpanding	to treat like `Minimum`, except the widget desires as much space as possible
Ignored	to ignore the size hint and request as much space as possible

4. Any widgets that are allocated less space than their minimum size (or minimum size hint if no minimum size is specified) are allocated this minimum size they require. (Widgets don't have to have a minimum size or minimum size hint in which case the strech factor is their determining factor.)
5. Any widgets that are allocated more space than their maximum size are allocated the maximum size space they require. (Widgets do not have to have a maximum size in which case the strech factor is their determining factor.)

Every widget returns a size hint to the layout from the the `sizeHint` method/property. The interpretation of the size hint depends on the `sizePolicy`roperty. The size policy is an object of class `QSizePolicy`. It contains a separate policy value, taken from the `QSizePolicy` enumeration, for the vertical and horizontal directions. The possible size policies are listed in Table **??**.

tab:qt:size-policies As an example, consider `QPushButton`. It is the convention that a button will only allow horizontal, but not vertical, expansion. It also requires enough space to display its entire label. Thus a `QPushButton` instance returns a size hint depending on the label dimensions and has the policies `Fixed` and `Minimum` as its vertical and horizontal policies respectively. We could prevent a button from expanding at all:

```
b <- Qt$QPushButton("No expansion")
b$setSizePolicy(vertical=Qt$QSizePolicy$Fixed,
                horizontal=Qt$QSizePolicy$Fixed)
```

Thus, the sizing behavior is largely inherent to the widget, rather than any layout parameters. This is a major difference from GTK+, where a

widget can only request a minimum size and all else depends on the parent container widget. The Qt approach seems better at encouraging consistency in the layout behavior of widgets of a particular type.

Most widgets attempt to fill the allocated space; however, this is not always appropriate, as in the case of labels. In such cases, the widget will not expand and needs to be aligned within its space. By default, the widget is centered. We can control the alignment of a widget via the `setAlignment` method. For example, we align the label to the left side of the layout:

```
label <- Qt$QLabel("Label")
layout$addWidget(label)
```

```
NULL
```

```
layout$setAlignment(label, Qt$Qt$AlignLeft)
```

```
[1] TRUE
```

Alignment is also possible to the top, bottom and right. The alignment values are flags and so many be combined to specify both vertical and horizontal alignment.

It is also possible to specify an amount of spacing between every cell of the layout. Here, we request 5 pixels of space:

```
layout$spacing <- 5
```

2.2 Box Layouts

Box layouts arrange child widgets as if they were packed into a box in either the horizontal or vertical orientation. The `QHBoxLayout` class implements a horizontal layout whereas `QVBoxLayout` provides a vertical one. Both of these classes extend the `QBoxLayout` class, where most of the functionality is documented. We create a horizontal layout:

```
hb <- Qt$QHBoxLayout()
```

Child widgets are added to a box container through the `addWidget` method:

```
sapply(1:3, function(i) hb$addWidget(Qt$QPushButton(i)))
```

```
[[1]]
NULL

[[2]]
NULL
```

```
[[3]]
NULL
```

The `direction` property specifies the direction in which the widgets are added to the layout. By default, this is left to right (top to bottom for a vertical box).

The `addWidget` method for a box layout takes two optional parameters: the stretch factor and the alignment. Stretch factors proportionally allocate space to widgets when they expand. For those familiar with GTK+, the difference between a stretch factor of 0 and 1 is roughly equivalent to the difference between `"FALSE"` and `"TRUE"` for the value of the `expand` parameter to `gtkBoxPackStart`. However, recall that the widget size policy and hint can alter the effect of a stretch factor. After the child has been added, the stretch factor may be modified with `setStretchFactor`:

```
hb$setStretchFactor(wid, 2.0)
```

```
[1] FALSE
```

Spacing There are two types of spacing between two children: fixed and expanding. Fixed spacing between any two children was already described. To add a fixed amount of space between two specific children, call the `addSpacing` method while populating the container. The following line is from Example 2.1:

```
buttonLayout$addSpacing(12L)
```

```
NULL
```

An expanding, spring-like spacer between two widgets is known as a *stretch*. We add a stretch with a factor of 2.0 and subsequently add a child button that will be pressed against the right side of the box:

```
g$addStretch(2)
```

```
NULL
```

```
g$addWidget(Qt$QPushButton("Help..."))
```

```
NULL
```

This is just a convenience for adding an invisible widget with some stretch factor.

Struts It is sometimes desirable to restrict the size of a layout in the perpendicular direction. For a horizontal box, this is the height. The box layout provides the *strut* for this purpose:

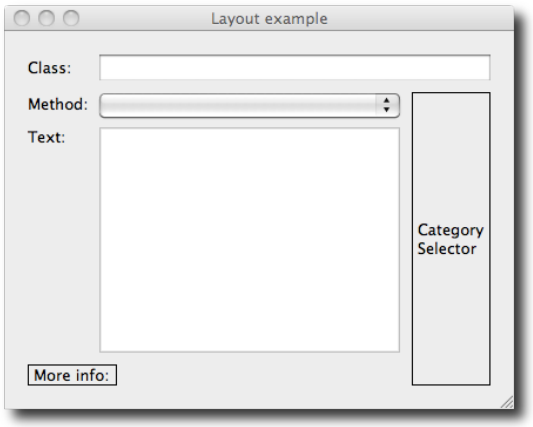

Figure 2.2: A mocked up layout using the `QGridLayout` class.

```
g$addStrut(10)
```

```
NULL
```

We have limited the height of the box to 10 pixels.

2.3 Grid Layouts

The `QGridLayout` class provides a grid layout for aligning its child widgets into rows and columns. To illustrate grid layouts we mock up a GUI centered around a text area widget (Figure 2.2). To begin, we create the window with the grid layout:

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

When we add a child to the grid layout, we need to specify the zero-based row and column indices:

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

In the second call to `addWidget`, we pass values to the optional arguments for the row and column span. These are the numbers of rows and columns, respectively, that are spanned by the child. For our second row, we add a labeled combo box:

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

The bottom three cells in the third column are 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:

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

Since this widget will expand, we align the label to the top of its cell. Otherwise, it will be centered in the vertical direction.

Finally we add a space for information on the fourth row:

```
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)
```

Again we draw a frame around the label. By default the box would expand to fill the space of the two columns, but we prevent this through a `"Fixed"` size policy.

There are a number of parameters controlling the sizing and spacing of the rows and columns. The concepts apply equivalently to both rows and columns, so we will limit our discussion to columns, without loss of generality. A minimum width is set through `setColumnMinimumWidth`. The actual minimum width will be increased, if necessary, to satisfy the minimal width requirements of the widgets in the column. If more space is available to a column than requested, the extra space is apportioned according to the `stretch` factors. A column stretch factor is set by calling the `setColumnStretch` method.

Since there are no stretch factors set in our example, the space allocated to each row and column would be identical when resized. To allocate extra space to the text area, we set a positive stretch factor for the third row and second column:

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

As it is the only item with a positive stretch factor, it will be the only widget to expand when the parent widget is resized.

The spacing between widgets can be set in both directions via the `spacing` property, or set for a particular direction with `setHorizontalSpacing` or `setVerticalSpacing`. The default values are derived from the style.

The method `itemAtPosition` returns the `QLayoutItem` instance corresponding to the specified row and column:

```
lineEdit <- lyt$itemAtPosition(0, 1)$widget()
```

The item method `widget` returns the corresponding widget. Removing a widget is similar to a box layout, using `removeItem` or `removeWidget`. The methods `rowCount` and `columnCount` return the dimensions of the grid.

2.4 Form layouts

Forms can easily be arranged with the grid layout, but Qt provides a convenient high-level form layout (`QFormLayout`) that conforms to platform-specific conventions. A form consists of a number of rows, where each row has a label and an input widget. We create a form and add some rows for gathering parameters to the `dnorm` function:

```
w <- Qt$QWidget()
w$setWindowTitle("Wrapper for 'dnorm' function")
w$setLayout(flyt <- Qt$QFormLayout())
flyt$addRow("quantile", Qt$QLineEdit())
flyt$addRow("mean", Qt$QLineEdit())
flyt$addRow("sd", Qt$QLineEdit())
flyt$addRow(Qt$QCheckBox("log"))
```

```
NULL
```

```
NULL
```

The first three calls to `addRow` take a string for the label and a text entry for entering a numeric value. Any widget could serve as the label. A field may be any widget or layout. The final call to `addRow` places only a single widget in the row. As with other layouts, we could call `setSpacing` to adjust the spacing between rows.

To retrieve a widget from the layout, call the `itemAt` method, passing the zero-based row index and the role of the desired widget. Here, we obtain the edit box for the quantile parameter:

```
quantileEdit <- flyt$itemAt(0, Qt$QFormLayout$FieldRole)
```

2.5 Frames

The frame widget, `QGroupBox`, groups conceptually related widgets by drawing a border around them and displaying a title. `QGroupBox` is often used to group radio buttons, see Section 3.5 for an example. The title, stored in the `title`roperty, may be aligned to left, right or center, depending on the `alignment`roperty, see Figure **??**. If the `checkable`roperty is `"TRUE"`, the frame contents can be enabled or disabled by clicking a check button.

2.6 Separators

Like frames, a horizontal or vertical line is also useful for visually separating widgets into conceptual groups. There is no explicit line or separator widget in Qt. Rather, one configures the more general widget `QFrame`, which draws a frame around its children. Somewhat against intuition, a frame can take the shape of a line:

```
separator <- Qt$QFrame()
separator$frameShape <- Qt$QFrame$HLine
```

This yields a horizontal separator. A frame shape of `"Qt$QFrame$VLine"` would produce a vertical separator.

2.7 Notebooks

A notebook container is provided by the widget `QTabWidget`:

```
nb <- Qt$QTabWidget()
```

To create a page, one needs to specify the label for the tab and the widget to display when the page is active:

```
nb$addTab(Qt$QPushButton("page 1"), "page 1")
```

```
[1] 0
```

```
icon <- Qt$QIcon("small-R-logo.jpg")
nb$addTab(Qt$QPushButton("page 2"), icon,  "page 2")
```

```
[1] 1
```

As shown in the second call to `addTab`, one can provide an icon to display next to the tab label. We can also add a tooltip for a specific tab, using zero-based indexing:

```
nb$setTabToolTip(0, "This is the first page")
```

```
NULL
```

The `currentIndex`roperty holds the zero-based index of the active tab. We make the second tab active:

```
nb$currentIndex <- 1
```

The tabs can be positioned on any of the four sides of the notebook; this depends on the `tabPosition`roperty. By default, the tabs are on top, or `"North"`. We move them to the bottom:

```
nb$tabPosition <- Qt$QTabWidget$South
```

Other features include close buttons, movable pages by drag and drop, and scroll buttons for when the number of tabs exceeds the available space. We enable all of these:

```
nb$tabsClosable <- TRUE
qconnect(nb, "tabCloseRequested", function(index) {
  nb$removeTab(index)
})
```

```
QObject instance
```

```
nb$movable <- TRUE
nb$usesScrollButtons <- TRUE
```

We need to connect to the `tabCloseRequested` signal to actually close the tab when the close button is clicked.

General Widget Stacking It is sometimes useful to have a widget that only shows one of its widgets at once, like a `QTabWidget`, except without the tabs. There is no way to hide the tabs of `QTabWidget`. Instead, one should use `QStackedWidget`, which stacks its children so that only the widget on top of the stack is visible. There is no way for the user to switch between children; it must be done programmatically. The actual layout is managed by `QStackedLayout`, which should be used directly if only a layout is needed, e.g., as a sub-layout.

2.8 Scroll Areas

Sometimes a widget is too large to fit in a layout and thus must be displayed partially. Scroll bars then allow the user to adjust the visible portion of the widget. Widgets that often become too large include tables, lists and text edit panes. These inherit from `QAbstractScrolledArea` and thus natively provide scroll bars without any special attention from the user. Occasionally, we are dealing with a widget that lacks such support and thus need to explicitly embed the widget in a `QScrollArea`. This often arises when displaying graphics and images. To demonstrate, we will create a simple zoomable image viewer. The user can zoom in and out and use the scroll bars to pan around the image. First, we place an image in a label and add it to a scroll area:

```
image <- Qt$QLabel()
image$pixmap <- Qt$QPixmap("someimage.png")
sa <- Qt$QScrollArea()
sa$setWidget(image)
```

Next, we define a function for zooming the image:

```
zoomImage <- function(x = 2.0) {
  image$resize(x * image$pixmap$size())
  updateScrollBar <- function(sb) {
    sb$value <- x * sb$value + (x - 1) * sb$pageStep / 2
  }
  updateScrollBar(sa$horizontalScrollBar())
  updateScrollBar(sa$verticalScrollBar())
}
```

Of note here is that we are scaling the size of the pixmap using the `*` function, which `qtbase` is forwarding to the corresponding method on the `QSize` object. Updating the scroll bars is also somewhat tricky, since their value corresponds to the top-left, while we want to preserve the center point. We leave the interface for calling the `zoomImage` function as an exercise for the interested reader.

The geometry of a scroll area is such that there is an empty space in the corner between the ends of the scroll bars. To place a widget in the corner, pass it to the `setCornerWidget` method.

2.9 Paned Windows

`QSplitter` is a split pane widget, a container that splits its space between its children, with draggable separators that adjust the balance of the space allocation.

Unlike `GtkPaned` in GTK+, there is no limit on the number of child panes. We add three and change the orientation from horizontal to vertical:

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

We can adjust the sizes programmatically:

```
sp$setSizes(c(100L, 200L, 300L))
```

```
NULL
```

3

Qt: Widgets

This chapter covers some of the basic dialogs and widgets provided by Qt. Together with layouts, these form the basis for most user interfaces. The next chapter will introduce the more complex widgets that typically act as a view for a separate data model.

3.1 Dialogs

Qt implements the conventional high-level dialogs, including those for printing, selecting files, selecting colors, and, most usefully, sending simple messages and input requests to the user. We first introduce message and input dialogs. This is followed by a discussion of the infrastructure in Qt for implementing custom dialogs and wizards. Finally, we briefly introduce some of the remaining high-level dialogs, such as the file selector.

Message Dialogs

All dialogs in Qt are derived from `QDialog`. The message dialog, `QMessageBox`, communicates a textual message to the user. At the bottom of the dialog are a set of buttons, each representing a possible response. Normally, the type of message is indicated by an icon. If extra details are available, the dialog provides the option for the user to view them.

Qt provide two ways to create a message box. The simplest approach is to call a static convenience method for issuing common types of messages, including warnings and simple questions. The alternative, described later, involves several steps and offers more control at a cost of convenience. Here we take the simple path for presenting a warning dialog:

```
response <- Qt$QMessageBox$warning(parent = NULL, title = "Warning!",
                                   text = "Warning message...")
```

This blocks the flow of the program until the user responds and returns the standard identifier for the button that was clicked. Each type of button corresponds to a fixed type of response. The standard button/response

codes are listed in the QMessageBox::StandardButton enumeration. In this case, there is only a single button, "QMessageBox$Ok". The dialog is *modal*, meaning that the user cannot interact with the "parent" window until responding. If the "parent" is "NULL", as in this case, input to all windows is blocked. The dialog is automatically positioned near its parent, and if the parent is destroyed, the dialog is destroyed, as well. Additional arguments specify the available buttons/responses and the default response. We have relied on the default values for these.

For more control over the appearance and behavior of the dialog, we will take a more gradual path. First, we construct an instance of QMessageBox. It is possible to specify several properties at construction. Here is how one might construct a warning dialog:

```
dlg <- Qt$QMessageBox(icon = Qt$QMessageBox$Warning,
                      title = "Warning!",
                      text = "Warning text...",
                      buttons = Qt$QMessageBox$Ok,
                      parent = NULL)
```

This call introduces the icon property, which is a code from the QMessageBox::Icon enumeration and identifies a standard, themeable icon. The icon also implies the message type, just as a button implies a response type. We also need to specify the possible responses with the "buttons" argument.

Our dialog is already sufficiently complete to be displayed. However, we have the opportunity to specify further properties. Two of the most useful are informativeText and detailedText:

```
dlg$informativeText <- "Less important warning information"
dlg$detailedText <- "Extra details most do not care to see"
```

Both provide additional textual information at an increasing level of detail. The informativeTextQMessageBox will be rendered as secondary to the actual message text. To display the detailedText, the user will need to interact with a control in the dialog. An example is a stack trace for the warning.

After specifying the desired properties, the dialog is shown. The approach to showing the dialog depends on whether the dialog should be modal. A modal dialog is displayed with the exec method.

```
dlg$exec()
```

```
[1] 1024
```

As its name implies, exec executes a loop that will block until the user responds. As with the static convenience methods, the return value indicates the button/response.

To present a non-modal dialog, we first need to register a response listener, as the response will arrive asynchronously:

```
qconnect(dlg, "finished", function(response) {
  ## handle response
  ## dlg$close() necessary?
})
```

```
QObject instance
```

There are several signals that indicate user response, including `"finished"`, `"accepted"`, and `"rejected"`. The most general is `"finished"`, which passes the button/response code as its only argument.

Then we show, raise and activate the dialog:

```
dlg$show()
dlg$raise()
dlg$activateWindow()
```

Modal dialogs may be window modal (`Qt$Qt$WindowModal`), where the dialog blocks all access to its ancestor windows, or application modal (`Qt$Qt$ApplicationModal`) (the default) where all windows are blocked. To specify the type of modality, call `setWindowModality`.

To summarize, we present a general message box, supporting multiple responses:

```
dlg <- Qt$QMessageBox()
dlg$windowTitle <- "[This space for rent]"
dlg$text <- "This is the main text"
dlg$informativeText <- "This should give extra info"
dlg$detailedText <- "And this provides\neven more detail"
dlg$icon <- Qt$QMessageBox$Critical
dlg$standardButtons <- Qt$QMessageBox$Cancel | Qt$QMessageBox$Ok
## 'Cancel' instead of 'Ok' is the default
dlg$setDefaultButton(Qt$QMessageBox$Cancel)
if(dlg$exec() == Qt$QMessageBox$Ok)
  print("A Ok")
```

Input dialogs

The `QInputDialog` class provides a convenient means to gather information from the user and is in a sense the inverse of `QMessageBox`. Possible input modes include selecting a value from a list or entering text or numbers. By default, input dialogs consist of an input control, an icon, and two buttons: “Ok” and “Cancel”.

Like `QMessageBox`, one can display a `QInputDialog` either by calling a static convenience method or by constructing an instance and configuring it before showing it. We demonstrate the former approach for a dialog that requests textual input:

```
text <- Qt$QInputDialog$getText(parent = NULL,
```

```
                          title = "Gather text",
                          label = "Enter some text")
```

The return value is the entered string, or `"NULL"` if the user cancelled the dialog. Additional parameters allow one to specify the initial text and to override the input mode, e.g., for password-style input.

We can also display a dialog for integer input. Here, we ask the user for an even integer between 1 and 10:

```
num <- Qt$QInputDialog$getInt(parent = NULL, title = "Gather integer",
                        label = "Enter an integer from 1 to 10",
                        value = 0, min = 2, max = 10, step = 2)
```

The number is chosen using a bounded spin box. To request a real value, call `Qt$QInputDialog$getDouble` instead.

The final type of input is selecting an option from a list of choices:

```
item <- Qt$QInputDialog$getItem(parent = NULL, title = "Select item",
                         label = "Select a letter",
                         items = LETTERS, current = 17)
```

The dialog contains a combo box filled with the capital letters. The initial choice is 0-based index 17, or the letter "R". The chosen string is returned.

`QInputDialog` has a number of options that cannot be specified via one of the static convenience methods. These option flags are listed in the `QInputDialog$InputDialogOption` enumeration and include hiding the "Ok" and "Cancel" buttons and selecting an item with a list widget instead of a combo box. If such control is necessary, we must explicitly construct a dialog instance, configure it, execute it and retrieve the selected item.

```
dlg <- Qt$QInputDialog()
dlg$setWindowTitle("Select item")
dlg$setLabelText("Select a letter")
dlg$setComboBoxItems(LETTERS)
dlg$setOptions(Qt$QInputDialog$UseListViewForComboBoxItems)
```

```
if (dlg$exec())
  print(dlg$textValue())
```

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

Button boxes

Before discussing custom dialogs, we first introduce the `QDialogButtonBox` utility for arranging dialog buttons in a consistent and cross-platform manner. Dialogs often have a standard button placement that varies among desktop environments. `QDialogButtonBox` is a container of buttons that arranges its children according to the convention of the platform. We place some standard buttons into a button box:

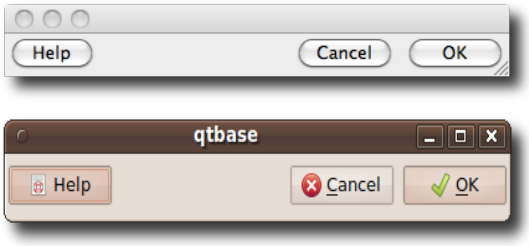

Figure 3.1: Dialog button boxes and their implementation under Mac OS X and Linux.

```
db <- Qt$QDialogButtonBox(Qt$QDialogButtonBox$Ok |
                          Qt$QDialogButtonBox$Cancel |
                          Qt$QDialogButtonBox$Help)
```

Figure 3.1 shows how the buttons are displayed on two different operating systems. To indicate the desired buttons, we pass a combination of flags from the `QDialogButtonBox$StandardButton` enumeration. Each standard button code implies a default label and role, taken from the `QDialogButtonBox$ButtonRole` enumeration. In the above example, we added a standard `OK` button, with the label "OK" (depending on the language) and the role `AcceptRole`. The `Cancel` button has the appropriate label and `CancelRole` as its role. Icons are also displayed, depending on the platform and theme. The benefits of using standard buttons include convenience, standardization, platform consistency, and automatic translation of labels.

To respond to user input, one can connect directly to the `clicked` signal on a given button. It is often more convenient, however, to connect to one of the high-level button box signals, which include: `accepted`, which is emitted when a button with the `AcceptRole` or `YesRole` is clicked; `rejected`, which is emitted when a button with the `RejectRole` or `NoRole` is clicked; `helpRequested`; or `clicked` when any button is clicked. For this last signal, the callback is passed the button object.

```
qconnect(db, "accepted", function() message("accepted"))
```

```
QObject instance
```

```
qconnect(db, "rejected", function() message("rejected"))
```

```
QObject instance
```

```
qconnect(db, "helpRequested", function() message("help"))
```

```
QObject instance
```

```
qconnect(db, "clicked", function(button) message(button$text))
```

```
QObject instance
```

The first button added with the `AcceptRole` role is made the default. Overriding this requires adding the default button with `addButton` and setting the `default`troperty on the returned button object.

Custom Dialogs

Every dialog in Qt inherits from `QDialog`, which we can leverage for our own custom dialogs. One approach is to construct an instance of `QDialog` and add arbitrary widgets to its layout. However, we suggest an alternative approach: extend `QDialog` or one of its derivates and implement the custom functionality in a subclass. This more formally encapsulates the state and behavior of the custom dialog. We demonstrate the subclass approach by constructing a dialog that requests a date from the user.

We begin by defining our class and its constructor:

```
qsetClass("DateDialog", Qt$QDialog,
          function(parent = NULL) {
            super(parent=parent)
            setWindowTitle("Choose a date")
            this$calendar <- Qt$QCalendarWidget()
            buttonBox <-
              Qt$QDialogButtonBox(Qt$QMessageBox$Cancel |
                                  Qt$QMessageBox$Ok)
            layout <- Qt$QVBoxLayout()
            layout$addWidget(calendar)
            layout$addWidget(buttonBox)
            setLayout(layout)
          })
```

```
Class 'R::.GlobalEnv::DateDialog' with 336 public methods
```

Our dialog consists of a calendar, implemented by the `QCalendarWidget`, and a set of response buttons, organized by a `QDialogButtonBox`. The calendar is stored as a field on the instance, so that we can retrieve the selected date upon request.

We define a method that gets the currently selected date:

```
qsetMethod("selectedDate", DateDialog, function(x) calendar$selectedDate)
```

```
[1] "selectedDate"
```

`DateDialog` can be executed like any other `QDialog`:

```
dateDialog <- DateDialog()
if (dateDialog$exec())
  message(dateDialog$selectedDate())
```

Wizards

`QWizard` implements a wizard – a multipage dialog that guides the user through a sequential, possibly branching process. Wizards are composed of pages, and each page has a consistent interface, usually including buttons for moving backwards and forwards through the pages. The look and feel of a `QWizard` is consistent with platform conventions.

We create a wizard object and set its title:

```
wizard <- Qt$QWizard()
wizard$setWindowTitle("A wizard")
```

```
NULL
```

Each page is represented by a `QWizardPage`. We create one for requesting the age of the user and add the page to the wizard:

```
getAgePage <- Qt$QWizardPage(wizard)
getAgePage$setTitle("What is your age?")
```

```
NULL
```

```
lyt <- Qt$QFormLayout()
getAgePage$setLayout(lyt)
```

```
NULL
```

```
lyt$addRow("Age", (age <- Qt$QLineEdit()))
```

```
NULL
```

```
wizard$addPage(getAgePage)
```

```
[1] 0
```

Two more pages are added:

```
getToysPage <- Qt$QWizardPage(wizard)
getToysPage$setTitle("What toys do you like?")
```

```
NULL
```

```
lyt <- Qt$QFormLayout()
getToysPage$setLayout(lyt)
```

```
NULL
```

```
lyt$addRow("Toys", (toys <- Qt$QLineEdit()))
```

```
NULL
```

```
wizard$addPage(getToysPage)
```

```
[1] 1
```

```
getGamesPage <- Qt$QWizardPage(wizard)
getGamesPage$setTitle("What games do you like?")
```

```
NULL
```

```
lyt <- Qt$QFormLayout()
getGamesPage$setLayout(lyt)
```

```
NULL
```

```
lyt$addRow("Games", (games <- Qt$QLineEdit()))
```

```
NULL
```

```
wizard$addPage(getGamesPage)
```

```
[1] 2
```

Finally, we run the wizard by calling its `exec` method:

```
ret <- wizard$exec()
```

File and Directory Choosing Dialogs

`QFileDialog` allows the user to select files and directories, by default using the platform native file dialog. As with other dialogs there are static methods to create dialogs with standard options. These are `"getOpenFileName"`, `"getOpenFileNames"`, `"getExistingDirectory"`, and `"getSaveFileName"`. To select a file name to open we would have:

```
fname <- Qt$QFileDialog$getOpenFileName(NULL, "Open a file...", getwd())
```

All take as intial arguments a parent, a caption and a directory. Other arguments allow one to set a filter, say. For basic use, these are nearly as easy to use as R's `file.choose`. If a file is selected, `fname` will contain the full path to the file, otherwise it will be `NULL`.

To allow multiple selection, call the plural form of the method:

```
fnames <- Qt$QFileDialog$getOpenFileNames(NULL, "Open file(s)...", getwd())
```

To select a file name for saving, we have

```
fname <- Qt$QFileDialog$getSaveFileName(NULL, "Save as...", getwd())
```

And to choose a directory,

```
dname <- Qt$QFileDialog$getExistingDirectory(NULL, "Select directory", getwd())
```

To specify a filter by file extension, we use a name filter. A name filter is of the form `Description (*.ext *.ext2)` (no comma) where this would match files with extensions `ext` or `ext2`. Multiple filters can be used by separating them with two semicolons. For example, this would be a natural filter for R users:

```
rfilter <- paste("R files (*.R .RData)",
"Sweave files (*.Rnw)",
"All files (*.*)", sep=";;")
fnames <- Qt$QFileDialog$getOpenFileNames(NULL,
"Open file(s)...", getwd(),
rfilter)
```

Although the static functions provide most of the functionality, to fully configure a dialog, it may be necessary to explicitly construct and manipulate a dialog instance. Examples of options not available from the static methods are history (previously selected file names), sidebar shortcut URLs, and filters based on low-level file attributes like permissions.

Example 3.1: File dialogs
We construct a dialog for opening an R-related file, using the file names selected above as the history:

```
dlg <- Qt$QFileDialog(NULL, "Choose an R file", getwd(), rfilter)
dlg$fileMode <- Qt$QFileDialog$ExistingFiles
dlg$setHistory(fnames)
```

The dialog is executed like any other. To get the specified files, call `selectedFiles`:

```
if(dlg$exec())
  print(dlg$selectedFiles())
```

Other Choosers

Qt/ provides several additional dialog types for choosing a particular type of item. These include `QColorDialog` for picking a color, and `QFontDialog` for selecting a font. These special case dialogs will not be discussed further here.

3.2 Labels

As seen in previous example, basic labels in Qt are instances of the `QLabel` class. Labels in Qt are the primary means for displaying static text and images. Textual labels are the most common, and the constructor accepts

a string for the text, which can be plain text or, for rich text, HTML. Here we use HTML to display red text:

```
l <- Qt$QLabel("<font color='red'>Red</font>")
```

By default, `QLabel` guesses whether the string is rich or plain text. In the above, the rich text format is identified from the markup. The `textFormat` property overrides this.

The label text is stored in the `text` property. Properties relevant to text layout include: `alignment`, `indent` (in pixels), `margin`, and `wordWrap`.

3.3 Buttons

As we have seen, the ordinary button in Qt is called `QPushButton`, which inherits most of its functionality from `QAbstractButton`, the common base class for buttons. We create a simple button "Ok" button:

```
button <- Qt$QPushButton("Ok")
```

Like any other widget, a button may be disabled, so that the user cannot press it:

```
button$enabled <- FALSE
```

This is useful for preventing the user from attempting to execute commands that do not apply to the current state of the application. Qt changes the rendering widget, including that of the icon, to indicate the disabled state.

A push button usually executes some command when clicked, i.e., when the `clicked` signal is emitted:

```
qconnect(button, "clicked", function() message("Ok clicked") )
```

```
QObject instance
```

Icons and pixmaps

A button is often decorated with an icon, which serves as a visual indicator of the purpose of the button. The `QIcon` class represents an icon. Icons may be defined for different sizes and display modes (normal, disabled, active, selected); however, this is often not necessary, as Qt will automatically adapt an icon as necessary. As we have seen, Qt automatically adds the appropriate icon to a standard button in a dialog. When using `QPushButton` directly, there are no such conveniences. For our "Ok" button, we could add an icon from a file:

```
button$icon <- Qt$QIcon(system.file("images/ok.gif", package="gWidgets"))
```

However, in general, this will not be consistent with the current style. Instead, we need to get the icon from the QStyle:

```
style <- Qt$QApplication$style()
button$icon <- style$standardIcon(Qt$QStyle$SP_DialogOkButton)
```

The "QStyle::StandardPixmap" enumeration lists all of the possible icons that a style should provide. In the above, we passed the key for an "Ok" button in a dialog.

We can also create a QIcon from image data in a QPixmap object. QPixmap stores an image in a manner that is efficient for display on the screen [1]. One can load a pixmap from a file or create a blank image and draw on it using the Qt painting API (not discussed in this book). Also, using the qtutils package, we can draw a pixmap using the R graphics engine. For example, the following uses ggplot2 to generate an icon representing a histogram. First, we create the Qt graphics device and plot the icon with grid:

```
require(qtutils)
device <- QT()
grid.newpage()
grid.draw(GeomHistogram$icon())
```

Next, we create the blank pixmap and render the device to a paint context attached to the pixmap:

```
pixmap <- Qt$QPixmap(device$size$toSize())
pixmap$fill()
painter <- Qt$QPainter()
painter$begin(pixmap)
device$render(painter)
painter$end()
```

Finally, we use the icon in a button:

```
b <- Qt$QPushButton("Histogram")
b$setIcon(Qt$QIcon(pixmap))
```

3.4 Checkboxes

The QCheckBox class implements a checkbox. Like the QPushButton class, QCheckBox extends QAbstractButton. Thus, QCheckBox inherits the signals clicked, pressed, and released. We create a check box for our demonstration:

```
checkBox <- Qt$QCheckBox("Option")
```

[1] QPixmap is not to be confused with QImage, which is optimized for image manipulation, or the vector-based QPicture

The checked property indicates whether the button is checked:

```
checkBox$checked
```

```
[1] FALSE
```

Sometimes, it is useful for a checkbox to have an indeterminate state that is neither checked nor unchecked. To enable this, set the tristate property to "TRUE". In that case, one needs to call the checkState method to determine the state, as it is no longer boolean but from the "Qt::CheckState" enumeration.

The stateChanged signal is emitted whenever the checked state of the button changes:

```
qconnect(checkBox, "stateChanged", function(state) {
  if (state == Qt$Qt$Checked)
    message("checked")
})
```

```
QObject instance
```

The argument is from the "Qt::CheckState" enumeration; it is not a logical vector.

Groups of checkboxes

Checkboxes and other types of buttons are often naturally grouped into logical units. The frame widget, QGroupBox, is appropriate for visually representing this grouping. However, QGroupBox holds any type of widget, so it has no high-level notion of a group of buttons. The QButtonGroup object, which is *not* a widget, fills this gap, by formalizing the grouping of buttons behind the scenes.

To demonstrate, we will construct an interface for filtering a data set by the levels of a factor. A common design is to have each factor level correspond to a check button in a group. For our example, we take the cylinders variable from the Cars93 data set of the MASS package. First, we create our QGroupBox as the container for our buttons:

```
w <- Qt$QGroupBox("Cylinders:")
lyt <- Qt$QVBoxLayout()
w$setLayout(lyt)
```

Next, we create the button group:

```
bg <- Qt$QButtonGroup()
bg$exclusive <- FALSE
```

By default, the buttons are exclusive, like a radio button group. We disable that by setting the exclusive property to "FALSE".

We add a button for each level of the "Cylinders" variable to both the button group and the layout of the group box widget:

```
data(Cars93, package="MASS")
cyls <- levels(Cars93$Cylinders)
sapply(seq_along(cyls), function(i) {
  button <- Qt$QCheckBox(sprintf("%s Cylinders", cyls[i]))
  lyt$addWidget(button)
  bg$addButton(button, i)
})
sapply(bg$buttons(), function(i) i$checked <- TRUE)
```

Every button is initially checked.

We can retrieve a list of the buttons in the group and query their checked state:

```
checked <- sapply(bg$buttons(), function(i) i$checked)
if(any(checked)) {
  ind <- Cars93$Cylinders %in% cyls[checked]
  print(sprintf("You've selected %d cases", sum(ind)))
}
```

By attaching a callback to the buttonClicked signal, we will be informed when any of the buttons in the group are clicked:

```
qconnect(bg, "buttonClicked", function(button) {
  message(paste("Level '", button$text, "': ", button$checked, sep = ""))
})
```

3.5 Radio groups

Another type of checkable button is the radio button, QRadioButton. Radio buttons always belong to a group, and only one radio button in a group may be checked at once. Continuing our filtering example, we create several radio buttons for choosing a range for the "Weight" variable in the "Cars93" dataset:

```
l <- list(Qt$QRadioButton("Weight < 3000", w),
          Qt$QRadioButton("3000 <= Weight < 4000", w),
          Qt$QRadioButton("4000 <= Weight", w))
```

The simplest way to group the radio boxes into place them into the same layout:

```
w <- Qt$QGroupBox("Weight:")
lyt <- Qt$QVBoxLayout()
w$setLayout(lyt)
```

```
NULL
```

```
sapply(l, function(i) lyt$addWidget(i))
```

```
[[1]]
NULL

[[2]]
NULL

[[3]]
NULL
```

```
l[[1]]$setChecked(TRUE)
```

```
NULL
```

As with any other derivative of `QAbstractButton`, the checked state is stored in the `checked`:

```
l[[1]]$checked
```

```
[1] TRUE
```

The `toggled` signal is emitted twice when a button is checked or unchecked:

```
sapply(l, function(i) {
  qconnect(i, "toggled", function(checked) {
    if(checked) {
      message(sprintf("You checked %s.", i$text))
    }
  })
})
```

`QButtonGroup` is a useful utility for grouping radio buttons:

```
buttonGroup <- Qt$QButtonGroup()
lapply(l, buttonGroup$addButton)
```

```
[[1]]
NULL

[[2]]
NULL

[[3]]
NULL
```

Since our button group is exclusive, we can query for the currently checked button:

```
buttonGroup$checkedButton()
```

```
QRadioButton instance
```

3.6 Combo Boxes

A combo box allows a single selection from a drop-down list of options. In this section, we describe the basic usage of `QComboBox`. This includes populating the menu with a list of strings and optionally allowing arbitrary input through an associated text entry. For the more complex approach of deriving the menu from a separate data model, see Section **??**.

This example shows how one combobox, listing regions in the U.S., updates another, which lists states in that region. First, we prepare a `data.frame` with the name, region and population of each state and split that `data.frame` by the regions:

```
df <- data.frame(name=state.name, region=state.region,
                 population=state.x77[,'Population'], stringsAsFactors=FALSE)
statesByRegion <- split(df, df$region)
```

We create our combo boxes, loading the `region` combobox with the regions:

```
state <- Qt$QComboBox()
region <- Qt$QComboBox()
region$addItems(names(statesByRegion))
```

The `addItems` accepts a character vector of options and is the most convenient way to populate a combo box with a simple list of strings. The `currentIndex` property indicates the index of the currently selected item:

```
region$currentIndex
```

```
[1] 0
```

```
region$currentIndex <- -1
```

By setting it to -1, we make the selection to be empty.

To respond to a change in the current index, we connect to the `activated` signal:

```
qconnect(region, "activated", function(ind) {
  state$clear()
  state$addItems(statesByRegion[[ind+1]]$name)
})
```

```
QObject instance
```

Our handler resets the state combo box to correspond to the selected region, indicated by `"ind"`.

Finally, we place the widgets in a form layout:

```
w <- Qt$QGroupBox("Two comboboxes")
lyt <- Qt$QFormLayout()
```

```
w$setLayout(lyt)
lyt$addRow("Region:", region)
lyt$addRow("State:", state)
```

To allow a user to enter a value not in the menu, the property `editable` can be set to `TRUE`. This would not be sensible for our example.

3.7 Sliders and spin boxes

Sliders and spin boxes are similar widgets used for selecting from a range of values. Sliders give the illusion of selecting from a continuum, whereas spinboxes offer a discrete choice. However, underlying each is an arithmetic sequence. Our example will include both widgets and synchronize them for specifying a single range. The slider allows for quick movement across the range, while the spin box is best suited for fine adjustments.

Sliders

Sliders are implemented by `QSlider`, a subclass of `QAbstractSlider`. `QSlider` selects only from integer values. We create an instance and specify the bounds of the range:

```
sl <- Qt$QSlider()
sl$minimum <- 0
sl$maximum <- 100
```

We can also customize the step size:

```
sl$singleStep <- 1
sl$pageStep <- 5
```

Single step refers to the effect of pressing one of the arrow keys, while pressing `"Page Up/Down"` adjusts the slider by `pageStep`.

The current cursor position is give by the property `value`; we set it to the middle of the range:

```
sl$value
```

```
[1] 0
```

```
sl$value <- 50
```

A slider has several aesthetic properties. We set our slider to be oriented horizontally (vertical is the default), and place the tick marks below the slider, with a mark every 10 values:

```
sl$orientation <- Qt$Qt$Horizontal
sl$tickPosition <- Qt$QSlider$TicksBelow
sl$tickInterval <- 10
```

The valueChanged signal is emitted whenever the value property is modified. An example is given below, after the introduction of the spin box.

Spin boxes

There are several spin box classes: QSpinBox (for integers), QDoubleSpinBox and QDateTimeEdit. All of these derive from a common base, QAbstract-SpinBox. As our slider is integer-valued, we will introduce QSpinBox here. Configuring a QSpinBox proceeds much as it does for QSlider:

```
sp <- Qt$QSpinBox()
sp$minimum <- sl$minimum
sp$maximum <- sl$maximum
sp$singleStep <- sl$singleStep
```

There is no "pageStep" for a spin box. Since we are communicating a percentage, we specify "%" as the suffix for the text of the spin box:

```
sp$suffix <- "%"
```

It is also possible to set a prefix.

Both QSlider and QSpinBox emit the valueChanged signal whenever the value changes. We connect to the signal on both widgets to keep them synchronized:

```
f <- function(value, obj) obj$value <- value
qconnect(sp, "valueChanged", f, user.data=sl)
qconnect(sl, "valueChanged", f, user.data=sp)
```

We pass the other widget as the user data, so that state changes in one are forwarded to the other.

3.8 Single-line text

As seen in previous examples, a widget for entering or displaying a single line of text is provided by the QLineEdit class:

```
le <- Qt$QLineEdit("Initial Contents")
```

The textroperty holds the current value:

```
le$text
```

```
[1] "Initial Contents"
```

We wish to select the text, so that the initial contents are overwritten when the user begins typing:

```
le$setSelection(start = 0, length = nchar(le$text))
```