9

CHAPTER 9
QObject

An important class to become familiar with is the one from which all Qt Widgets are derived: `QObject`.

We will refer to any object of a class derived from `QObject` as a `QObject`. Here is an abbreviated look at its definition.

```
class QObject {
  public:
    QObject(QObject* parent=0);
    QObject * parent () const;
    QString objectName() const;
    void setParent ( QObject * parent );
    const ObjectList & children () const;
    // ... more ...
};
```

The first interesting thing that we observe is that `QObject`'s copy constructor is not `public`. `QObject`s are not meant to be copied. In general, `QObject`s are intended to represent unique objects with identity; that is, they correspond to real-world things that also have some sort of persistent identity. One immediate consequence of not having access to its copy constructor is that a `QObject` can never be passed by value to any function. Copying a `QObject`'s data members into another `QObject` is still possible, but the two objects are still considered unique.

One immediate consequence of not having access to its copy constructor is that `QObject`s can never be passed by value to any function.

Each `QObject` can have (at most) one **parent** object and an arbitrarily large container of `QObject*` children. Each `QObject` stores pointers to its children in a `QObjectList`.[1] The list itself is created in a lazy-fashion to minimize the overhead for objects which do not use it. Since each child is a `QObject` and can have an arbitrarily large collection of children, it is easy to see why copying `QObject`s is not permitted.

The notion of children can help to clarify the notion of identity and the no-copy policy for `QObject`s. If you represent individual humans as `QObject`s, the idea of a unique identity for each `QObject` is clear. Also clear is the idea of children. The rule that allows each `QObject` to have at most one parent can be seen as a way to simplify the implementation of this class. Finally, the no-copy policy stands out as a clear necessity. Even if it were possible to "clone" a person (i.e., copy

[1] `QObjectList` is a `typedef` (i.e., an alias) for `QList<QObject*>`.

its data members to another `QObject`), the question of what to do with the children of that person makes it clear that the clone would be a separate and distinct object with a different identity.

Each `QObject` parent *manages* its children. This means that the `QObject` destructor automatically destroys all of its child objects.

The child list establishes a **bidirectional**, one-to-many association between objects. Setting the parent of one object implicitly adds its address to the child list of the other, for example

```
objA->setParent(objB);
```

adds the `objA` pointer to the child list of `objB`. If we subsequently have

```
objA->setParent(objC);
```

then the `objA` pointer is removed from the child list of `objB` and added to the child list of `objC`. We call such an action **reparenting**.

Parent Objects versus Base Classes

Parent objects should not be confused with **base classes**. The parent-child relationship is meant to describe containment, or management, of objects at runtime. The base-derived relationship is a static relationship between classes determined at compile-time.

It is possible that a parent can also be an instance of a base class of some of its child objects. These two kinds of relationships are distinct and must not be confused, especially considering that many of our classes will be derived directly or indirectly from `QObject`.

It is already possible to understand some of the reasons for not permitting `QObject`s to be copied. For example, should the copy have the same parent as the original? Should the copy have (in some sense) the children of the original? A shallow copy of the child list would not work because then each of the children would have two parents. Furthermore, if the copy gets destroyed (e.g., if the copy was a value parameter in a function call), each child needs to be destroyed too. Even with resource sharing methods, this approach would introduce some serious difficulties. A deep copy of the child list could be a costly operation if the number of children were large and the objects pointed to were large. Since each child could also have arbitrarily many children, this questionable approach would also generate serious difficulties.

9.1 QObject's Child Managment

Example 9.1 shows a `QObject` derived class.

EXAMPLE 9.1 src/qobject/person.h

```
[ . . . . ]
class Person : public QObject {

 public:
    Person(QObject* parent, QString name);
    virtual ~Person();
};
[ . . . . ]
```

The complete implementation is shown in Example 9.2 to show that there is no explicit object deletion done in `~Person()`.

EXAMPLE 9.2 src/qobject/person.cpp

```
#include "person.h"
#include <QTextStream>
static QTextStream cout(stdout, QIODevice::WriteOnly);

Person::Person(QObject* parent, QString name)
          : QObject(parent) {
    setObjectName(name);
    cout << QString("Constructing Person: %1").arg(name) << endl;
}

Person::~Person() {
    cout << QString("Destroying Person: %1").arg(objectName()) <<
    endl;
}
```

`main()`, shown in Example 9.3, creates some objects, adds them to other objects, and then exits. All heap objects were implicitly destroyed.

EXAMPLE 9.3 src/qobject/main.cpp

```
#include <QTextStream>
#include "person.h"

static QTextStream cout(stdout, QIODevice::WriteOnly);
```

```
int main(int , char**) {
    cout << "First we create a bunch of objects." << endl;
    Person bunch(0, "A Stack Object");  ❶
    /* other objects are created on the heap */
    Person *mike = new Person(&bunch, "Mike");
    Person *carol = new Person(&bunch, "Carol");
    new Person(mike, "Greg");           ❷
    new Person(mike, "Peter");
    new Person(mike, "Bobby");
    new Person(carol, "Marcia");
    new Person(carol, "Jan");
    new Person(carol, "Cindy");
    new Person(0, "Alice");             ❸
    cout << "\nDisplay the list using QObject::dumpObjectTree()"
              << endl;
    bunch.dumpObjectTree();
    cout << "\nProgram finished - destroy all objects." << endl;
    return 0;
}
```

❶ not a pointer
❷ We do not need to remember pointers to children, since we can reach them via object navigation.
❸ Alice has no parent—memory leak?

Here is the output of this program:

```
First we create a bunch of objects.
Constructing Person: A Stack Object
Constructing Person: Mike
Constructing Person: Carol
Constructing Person: Greg
Constructing Person: Peter
Constructing Person: Bobby
Constructing Person: Marcia
Constructing Person: Jan
Constructing Person: Cindy
Constructing Person: Alice

Display the list using QObject::dumpObjectTree()
QObject::A Stack Object
    QObject::Mike
        QObject::Greg
        QObject::Peter
        QObject::Bobby
    QObject::Carol
        QObject::Marcia
        QObject::Jan
        QObject::Cindy
```

continued

```
Program finished - destroy all objects.
Destroying Person: A Stack Object
Destroying Person: Mike
Destroying Person: Greg
Destroying Person: Peter
Destroying Person: Bobby
Destroying Person: Carol
Destroying Person: Marcia
Destroying Person: Jan
Destroying Person: Cindy
```

Notice that Alice is not part of the `dumpObjectTree()` and does not get destroyed.

EXERCISE: QOBJECT'S CHILD MANAGMENT

Add the function

```
void showTree(QObject* theparent)
```

to main.cpp. The output of this function, after all objects have been created, should look like this:

```
Member: Mike - Parent: A Stack Object
Member: Greg - Parent: Mike
Member: Peter - Parent: Mike
Member: Bobby - Parent: Mike
Member: Carol - Parent: A Stack Object
Member: Marcia - Parent: Carol
Member: Jan - Parent: Carol
Member: Cindy - Parent: Carol
```

9.2 Composite Pattern: Parents and Children

According to [Gamma95], the **Composite pattern** is intended to facilitate building complex (composite) objects from simpler (component) parts by representing the part-whole hierarchies as tree-like structures. This must be done in such a way that clients do not need to distinguish between simple parts and more complex parts that are made up of (i.e., contain) simpler parts.

In Figure 9.1 there are two distinct classes for describing the two roles.

- A **composite object** is something that can contain children.
- A **component object** is something that can have a parent.

In Figure 9.2, we can see that `QObject` is both composite *and* component. We can express the whole-part relationship as a parent-child relationship between `QObject`s. The highest level (i.e., most "composite") `QObject` in such a tree

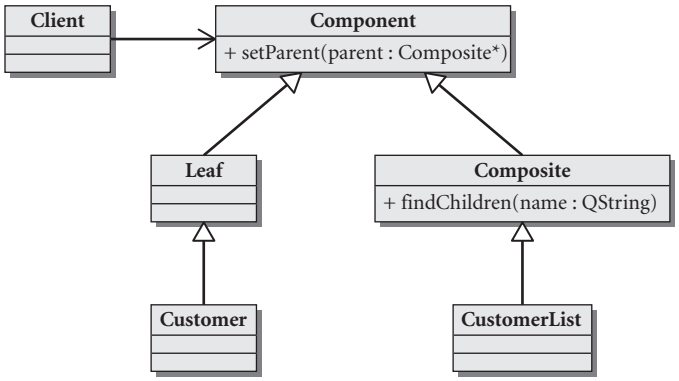


FIGURE 9.1 Components and composites

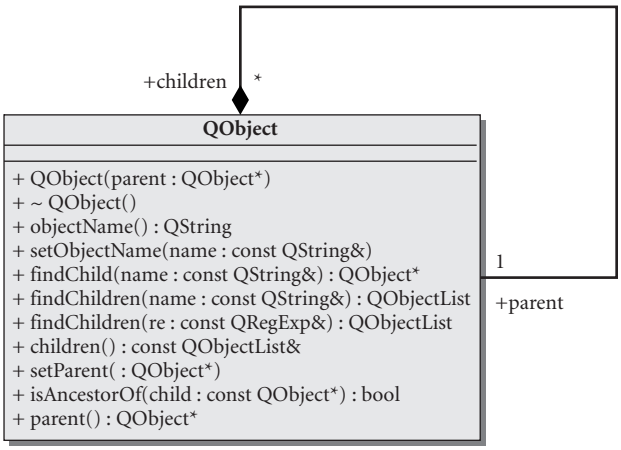


FIGURE 9.2 QObject

(i.e., the **root** of the tree) will have lots of children but no parent. The simplest `QObject`s (i.e., the **leaf nodes** of this tree) will each have a parent but no children. Client code can recursively deal with each node of the tree.

For an example of how this pattern might be used, let's look at Suffolk University. In 1906 the founder, Gleason Archer, decided to start teaching the principles of law to a small group of tradesmen who wanted to become lawyers. He was assisted by one secretary and, after a while, a few instructors. The organizational chart for this new school was quite simple: a single office consisting of several employees with various tasks. As the enterprise grew, the chart gradually became more complex with the addition of new offices and departments. Today, 100 years

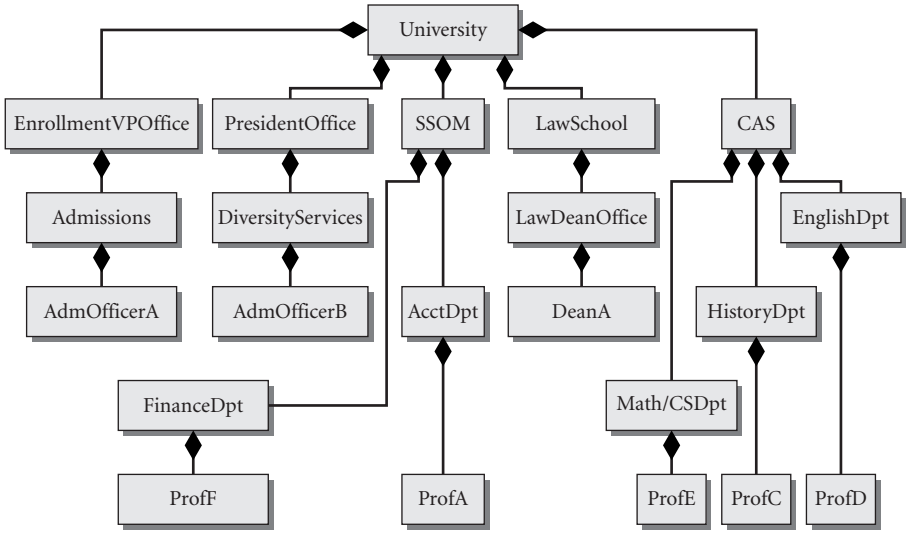


FIGURE 9.3 Suffolk University organizational chart

later, the Law School has been joined with a College of Arts and Sciences, a School of Management, a School of Art and Design, campuses abroad, and many specialized offices so that the organizational chart has become quite complex and promises to become more so. Figure 9.3 shows an abbreviated and simplified subchart of today's Suffolk University.

Each box in the chart is a component. It may be composite and have subcomponents which, in turn, may be composite or simple components. For example, the PresidentOffice has individual employees (e.g., the President and his assistants) and sub-offices (e.g., DiversityServices). The leaves of this tree are the individual employees of the organization.

We can use the Composite pattern to model this structure. Each node of the tree can be represented by an object of

```
class OrgUnit : public QObject {
  public:
    QString getName();
    double getSalary();
  private:
    QString m_Name;
    double m_Salary;
};
```

The `QObject` public interface allows us to build up a tree-like representation of the organization with code that instantiates an `OrgUnit` and then calls `setParent()` to add it to the appropriate child list.

For each `OrgUnit` pointer `ouptr` in the tree, we initialize its `m_Salary` data member as follows:

- If `ouptr` points to an individual employee, we use that employee's actual salary.
- Otherwise we initialize it to `0`.

We can implement the `getSalary()` method somewhat like this:

```
double OrgUnit::getSalary() {
  QList<OrgUnit*> childlst = findChildren<OrgUnit*>();
  double salaryTotal(m_Salary);
  if(!childlst.isEmpty())
    foreach(OrgUnit* ouptr, childlst)
      salaryTotal += ouptr->getSalary();
  return salaryTotal;
}
```

A call to `getSalary()` from any particular node returns the total salary for the part of the university represented by the subtree whose root is that node. For example, if `ouptr` points to `University`, `ouptr->getSalary()` returns the total salary for the entire university. But if `ouptr` points to `EnglishDpt`, then `ouptr->getSalary()` returns the total salary for the English Department.

9.2.1 Finding Children

`QObject` provides convenient and powerful functions named `findChildren()` for finding children in the child list. The signature of one of its overloaded forms looks like this:

```
QList<T> parentObj.findChildren<T> ( const QString & name ) const
```

If `name` is an empty string, `findChildren()` works as a class filter by returning a `QList` holding pointers to all children, which can be typecast to type `T`.

To call the function, you must supply a template parameter after the function name, as shown in Example 9.4.

EXAMPLE 9.4 src/findchildren/findchildren.cpp

```
[ . . . . ]
/* Filter on Customer* */
     QList<Customer*> custlist = parent.findChildren<Customer*>();
     foreach (Customer* current, custlist) {
          qDebug() << current->toString();
     }
[ . . . . ]
```

9.3 QApplication and the Event Loop

Interactive Qt applications with GUI have a different control flow from console applications and filter applications[2] because they are event-based, and often multi-threaded. Objects are frequently sending messages to each other, making a linear hand-trace through the code rather difficult.

Observer Pattern

When writing event-driven programs, GUI views need to respond to changes in the state of data model objects, so that they can display the most recent information possible.

When a particular subject object changes state, it needs an *indirect* way to alert (and perhaps send additional information to) all the other objects that are listening to state-change events, known as observers. A design pattern that enables such a message-passing mechanism is called the **Observer pattern**, sometimes also known as the **Publish-Subscribe pattern.**

There are many different implementations of this pattern. Some common characteristics that tie them together are

1. They all enable concrete subject classes to be decoupled from concrete observer classes.
2. They all support broadcast-style (one to many) communication.
3. The mechanism used to send information from subjects to observers is completely specified in the subject's base class.

Qt's approach is very different from Java's approach, because signals and slots rely on generated code, while Java just renames observer to listener.

The Qt class `QEvent` encapsulates the notion of an event. `QEvent` is the base class for several specific event classes such as `QActionEvent`, `QFileOpenEvent`, `QHoverEvent`, `QInputEvent`, `QMouseEvent`, and so forth. `QEvent` objects can be created by the window system in response to actions of the user (e.g., `QMouseEvent`) at specified time intervals (`QTimerEvent`) or explicitly by an application program. The `type()` member function returns an `enum` that has nearly a hundred specific values that can identify the particular kind of event.

A typical Qt program creates objects, connects them, and then tells the application to `exec()`. At that point, the objects can send information to each other in a

[2] A filter application is not interactive. It simply reads from standard input and writes to standard output.

variety of ways. `QWidgets` send `QEvents` to other `QObjects` in response to user actions such as mouse clicks and keyboard events. A widget can also respond to events from the window manager such as repaints, resizes, or close events. Furthermore, `QObjects` can transmit information to one another by means of signals and slots.

Each `QWidget` can be specialized to handle keyboard and mouse events in its own way. Some widgets will `emit` a signal in response to receiving an event.

An **event loop** is a program structure that permits events to be prioritized, enqueued, and dispatched to objects. Writing an event-based application means implementing a **passive interface** of functions that only get called in response to certain events. The event loop generally continues running until a terminating event occurs (e.g., the user clicks on the QUIT button).

Example 9.5 shows a simple application that initiates the event loop by calling `exec()`.

EXAMPLE 9.5 src/eventloop/main.cpp

```
[ . . . . ]

int main(int argc, char * argv[]) {
    QApplication myapp(argc, argv);   ❶

    QWidget rootWidget;
    setGui(&rootWidget);

    rootWidget.show();                ❷
    return myapp.exec();              ❸
};
```

❶ Every GUI, multithreaded, or event-driven Qt Application must have a QApplication object defined at the top of main().
❷ Show our widget on the screen.
❸ Enter the event loop.

When we run this app, we first see a widget on the screen as shown in the following figure.

We can type in the `QTextEdit` on the screen, or click on the Shout button. When Shout is clicked, a widget is superimposed on our original widget as shown in the next figure.

This message dialog knows how to self-destruct, because it has its own buttons and actions.

9.3.1 Layouts: A First Look

Whenever more than a single widget needs to be displayed, they must be arranged in some form of a **layout** (see Section 11.5). Layouts are derived from the abstract base class, `QLayout`, which is derived from `QObject`. Layouts are geometry managers that fit into the composition hierarchy of a graphical interface. Typically, we start with a widget that will contain all of the parts of our graphical construction. We select one or more suitable layouts to be children of our main widget (or of one another) and then we add widgets to the layouts.

It is important to understand that widgets in a layout are not children of the layout—they are children of the widget that owns the layout. Only a widget can be the parent of another widget. It may be useful to think of the layout as an older sibling acting as the nanny of its widgets.

In Example 9.6, we are laying out widgets in a vertical fashion with `QVBoxLayout`.

EXAMPLE 9.6 src/eventloop/main.cpp

```
[ . . . . ]

QWidget* setGui(QWidget *box) {
     QLayout* layout = new QVBoxLayout;
     box->setLayout(layout);                    ❶

     QTextEdit *te = new QTextEdit;             ❷
     layout->addWidget(te);                     ❸
```

```
    te->setHtml("Some <b>text</b> in the <tt>QTextEdit</tt>"
         "edit window <i>please</i>?");

    QPushButton *quitButton=new QPushButton("Quit");
    layout->addWidget(quitButton);

    QPushButton *shoutButton = new QPushButton("Shout");
    layout->addWidget(shoutButton);

    Messager *msgr = new Messager("This dialog will self-
    destruct.", box);

    QObject::connect(quitButton, SIGNAL(clicked()),
                 qApp, SLOT(quit()));   ❹

    qApp->connect(shoutButton, SIGNAL(clicked()), msgr, SLOT(shout()));
    return box;
}
```

❶ box is the parent of layout.
❷ This is the window for qDebug messages.
❸ te is the child of layout.
❹ qApp is a global variable that points to the current QApplication object.

The widgets are arranged vertically in this layout, from top to bottom, in the order that they were added to the layout.

9.3.2 Connecting to Slots

In Example 9.7, we saw the following connections established:

```
QObject::connect(quitButton, SIGNAL(clicked()), qApp, SLOT(quit()));
qApp->connect(shoutButton, SIGNAL(clicked()), msgr, SLOT(shout()));
```

`connect()` is actually a static member of `QObject` and can be called with any `QObject` or, as we showed, by means of its class scope resolution operator. `qApp` is a global pointer that points to the currently running `QApplication`.

The second `connect` goes to a `slot` that we declare in Example 9.7.

EXAMPLE 9.7 src/eventloop/messager.h

```
#ifndef MESSAGER_H
#define MESSAGER_H

#include <QObject>
#include <QString>
#include <QErrorMessage>
```

continued

```
class Messager : public QObject {
    Q_OBJECT
 public:
    Messager (QString msg, QWidget* parent=0);

 public slots:
    void shout();

 private:
    QWidget* m_Parent;
    QErrorMessage* message;
};

#endif
```

Declaring a member function to be a `slot` enables it to be connected to a `signal` so that it can be called passively in response to some event. For its definition, shown in Example 9.8, we have kept things quite simple: the `shout()` function simply pops up a message box on the screen.

EXAMPLE 9.8 src/eventloop/messager.cpp

```
#include "messager.h"

Messager::Messager(QString msg, QWidget* parent)
        : m_Parent(parent) {
    message = new QErrorMessage(parent);
    setObjectName(msg);
}

void Messager::shout() {
   message->showMessage(objectName());
}
```

9.3.3 Signals and Slots

When the main thread of a C++ program calls `qApp->exec()`, it enters into an event loop, where messages are handled and dispatched. While `qApp` is executing its event loop, it is possible for `QObject`s to send messages to one another.

A **signal** is a message that is presented in a class definition like a `void` function declaration. It has a parameter list but no function body. A signal is part of the interface of a class. It looks like a function but it cannot be called—it must be *emitted* by an object of that class. A signal is implicitly `protected`, and so are all the identifiers that follow it in the class definition until another access specifier appears.

A **slot** is a `void` member function. It can be called as a normal member function.

A signal of one object can be *connected* to the slots of one or more[3] other objects, provided the objects exist and the parameter lists are assignment compatible[4] from the signal to the slot. The syntax of the connect statement is:

```
bool QObject::connect(senderqobjptr,
                      SIGNAL(signalname(argtypelist)),
                      receiverqobjptr,
                      SLOT(slotname(argtypelist))
                      optionalConnectionType);
```

Any `QObject` that has a signal can emit that signal. This will result in an indirect call to all connected slots.

QWidgets already `emit` signals in response to events, so you only need to make the proper connections to receive those signals. Arguments passed in the `emit` statement are accessible as parameters in the slot function, similar to a function call, except that the call is indirect. The argument list is a way to transmit information from one object to another.

Example 9.9 defines a class that uses signals and slots to transmit a single `int` parameter.

EXAMPLE 9.9 src/widgets/sliderlcd/sliderlcd.h

```
[ . . . . ]
class QSlider;
class QLCDNumber;
class LogWindow;
class QErrorMessage;

class SliderLCD : public QMainWindow {
    Q_OBJECT
 public:
    SliderLCD(int minval = -273, int maxval = 360);
    void initSliderLCD();

  public slots:
    void checkValue(int newValue);
    void showMessage();

 signals:
    void toomuch();
 private:
    int m_Minval, m_Maxval;
    LogWindow* m_LogWin;
    QErrorMessage *m_ErrorMessage;
```

continued

[3] Multiple signals can be connected to the same slot also.

[4] Same number of parameters, each one being assignment compatible.

```
    QLCDNumber* m_LCD;
    QSlider* m_Slider;
};
#endif
[ . . . . ]
```

In Example 9.10, we can see how the widgets are initially created and connected.

EXAMPLE 9.10 src/widgets/sliderlcd/sliderlcd.cpp

```
[ . . . . ]

SliderLCD::SliderLCD(int min, int max) : m_Minval(min),
m_Maxval(max) {
    initSliderLCD();
}

void SliderLCD::initSliderLCD() {
    m_LogWin = new LogWindow();                                    (1)
    QDockWidget *logDock = new QDockWidget("Debug Log");
    logDock->setWidget(m_LogWin);
    logDock->setFeatures(0);                                       (2)
    setCentralWidget(logDock);

    m_LCD = new QLCDNumber();
    m_LCD->setSegmentStyle(QLCDNumber::Filled);
    QDockWidget *lcdDock = new QDockWidget("LCD");
    lcdDock->setFeatures(QDockWidget::DockWidgetClosable);         (3)
    lcdDock->setWidget(m_LCD);
    addDockWidget(Qt::LeftDockWidgetArea, lcdDock);

    m_Slider = new QSlider( Qt::Horizontal);
    QDockWidget* sliderDock = new QDockWidget("How cold is it
today?");
    sliderDock->setWidget(m_Slider);
    sliderDock->setFeatures(QDockWidget::DockWidgetMovable);
    /* Can be moved between doc areas */
    addDockWidget(Qt::BottomDockWidgetArea, sliderDock);

    m_Slider->setRange(m_Minval, m_Maxval);
    m_Slider->setValue(0);
    m_Slider->setFocusPolicy(Qt::StrongFocus);
    m_Slider->setSingleStep(1);                                    (4)
    m_Slider->setPageStep(20);                                     (5)
    m_Slider->setFocus();                                          (6)

    connect(m_Slider, SIGNAL(valueChanged(int)),  /*SliderLCD is a
    QObject so
        connect does not need scope resolution. */
            this, SLOT(checkValue(int)));
```

```
    connect(m_Slider, SIGNAL(valueChanged(int)),
            m_LCD, SLOT(display(int)));

    connect(this, SIGNAL(toomuch()),
            this, SLOT(showMessage()));                    (7)
    m_ErrorMessage = NULL;
}
```

(1) a class defined in the utils library
(2) cannot be closed, moved, or floated
(3) can be closed
(4) Step each time left or right arrow key is pressed.
(5) Step each time PageUp/PageDown key is pressed.
(6) Give the slider focus.
(7) Normally there is no point in connecting a signal to a slot on the same object, but we do it for demonstration purposes.

Only the argument types belong in the `connect` statement; for example, the following is not legal:

```
connect( button, SIGNAL(valueChanged(int)), lcd, SLOT(setValue(3)))
```

Example 9.11 defines the two slots, one of which conditionally `emits` another signal.

EXAMPLE 9.11 src/widgets/sliderlcd/sliderlcd.cpp

```
[ . . . . ]

void SliderLCD::checkValue(int newValue) {
    if (newValue> 120) {
        emit toomuch();                          (1)
    }
}

/* This slot is called indirectly via emit because
   of the connect */
void SliderLCD::showMessage() {
    if (m_ErrorMessage == NULL) {
        m_ErrorMessage = new QErrorMessage(this);
    }
    if (!m_ErrorMessage->isVisible()) {
        QString message("Too hot outside! Stay in. ");
        m_ErrorMessage->showMessage(message);    (2)
    }
}
```

(1) Emit a signal to anyone interested.
(2) This is a direct call to a slot. It's a member function.

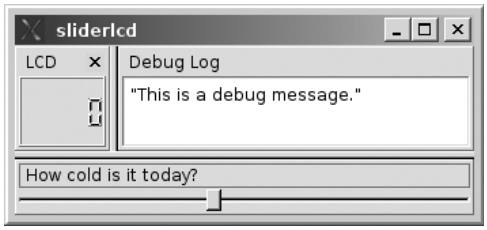
Example 9.12 contains client code to test this class.

EXAMPLE 9.12 src/widgets/sliderlcd/sliderlcd-demo.cpp

```
#include "sliderlcd.h"
#include <QApplication>
#include <QDebug>

int main(int argc, char ** argv) {
    QApplication app(argc, argv);
    SliderLCD slcd;
    slcd.show();
    qDebug() << QString("This is a debug message.");
    return app.exec();
}
```

Whenever the `slider` produces a new value, that value is transmitted as an argument from the `valueChanged(int) signal` to the `display(int) slot` of the `lcd`.



Synchronous or Asynchronous?

In single-threaded applications, or in multithreaded applications where the emitting and receiving `QObjects` are in the same thread, signals are sent in a **synchronous** manner. This means the thread blocks (suspends execution) until the code for the slots has completed execution (see Section 12.2).

In multi-threaded applications, where signals are emitted by an object in one thread and received by an object in another, it is possible to have signals queued, or executed in an **asynchronous** way, depending on the optional `Qt::ConnectionType` passed to `connect()`.

EXERCISES: SIGNALS AND SLOTS

1. Modify the `sliderlcd` program as follows:

- Make the lcd display show the temperatures as hexadecimal integers.
- Make the lcd display characters have a different ("flat") style.

- Give the slider a vertical orientation.
- Give the slider and the lcd display more interesting colors.
- Add a push button that the user can click to switch the lcd display from decimal mode to hexadecimal mode.
- Make the push button into a toggle that allows the user to switch back and forth between decimal and hexadecimal modes.

2. Write an application, similar to the one in Section 9.3, but that has four buttons. The first one, labeled Advice, should be connected to a slot that randomly selects a piece of text (such as a fortune cookie) and displays it in the `QTextEdit` window. The second one, labeled Weather, randomly selects a sentence about the weather and displays it in the `QTextEdit` window. The third one, labeled Next Meeting, pops up a message dialog with a randomly generated (fictitious) meeting time and descriptive message in it. The fourth one, labeled Quit, terminates the program. Use signals and slots to connect the button clicks with the appropriate functions.

9.4 Q_OBJECT and moc: A Checklist

`QObject` supports features not normally available in C++ objects.

- Children (see the first two sections of Chapter 9)
- Signals and slots (see Section 9.3.3)
- MetaObjects, metaproperties, metamethods (see Chapter 15)
- `qobject_cast` (see Section 15.3)

These features are only possible through the use of generated code. The Meta Object Compiler, `moc`, generates additional functions for each `QObject`-derived class that uses the macro. Generated code can be found in files with names `moc_filename.cpp`.

This means that some errors from the compiler/linker may be confuscated[5] when `moc` is not able to find or process your classes. To help ensure that `moc` processes each of your `QObject`-derived classes, here are some guidelines for writing C++ code and `qmake` project files.

- Each class definition should go in its own .h file.
- Its implementation should go in a corresponding .cpp file.
- The header file should be "`#ifndef` wrapped" to avoid multiple inclusion.
- Each source (.cpp) file should be listed in the `SOURCES` variable of the project file, otherwise it will not be compiled.

[5] confusing + obfuscated

- The header file should be listed in the `HEADERS` variable of the .pro file. Without this, `moc` will not preprocess the file.
- The `Q_OBJECT` macro must appear inside the class definition, so that `moc` will know to generate code for it.

MULTIPLE INHERITANCE AND QOBJECT Because each `Q_OBJECT` has signals and slots, it needs to be preprocessed by `moc`. `moc` works under the assumption that you are only deriving from `QObject` once, and further, that it is the first base class in the list of base classes. If you accidentally inherit from `QObject` multiple times, or if it is not the first base class in the inheritance list, you may receive very strange errors from moc-generated code.

9.5 Values and Objects

We can divide C++ types into two categories: value types and object types.

Instances of **value types** are relatively "simple": They occupy contiguous memory space, and can be copied or compared quickly. Examples of value types are `Anything*`, `int`, `char`, `QString`, `QDate`, and `QVariant`.

Instances of **object types**, on the other hand, are typically more complex and maintain some sort of identity. Object types are rarely copied (cloned). If cloning is permitted, the operation is usually expensive and results in a new object (graph) that has a separate identity from the original.

The designers of `QObject` asserted an unequivocal "no copy" policy by designating its assignment operator and copy constructor `private`. This effectively prevents the compiler from generating assignment operators and copy constructors for `QObject`-derived classes. One consequence of this scheme is that any attempt to pass or return `QObject`-derived classes by value to or from functions results in a compile-time error.

EXERCISES: QOBJECT

1. Rewrite the Contact and ContactList from "Exercise: Contact List" in Chapter 4 so that they both derive from `QObject`.

   When a Contact is to be added to a ContactList, make the Contact the child of the ContactList.

2. Port the client code you wrote for "Exercise: Contact List" to use the new versions of Contact and ContactList.

9.6 tr() and Internationalization

If you are writing a program that will ever be translated into another language (**internationalization**), Qt Linguist and Qt translation tools have already solved the problem of how to organize and where to put the translated strings. To prepare our code for translation, we use `QObject::tr()` to surround any translatable string in our application. The `tr()` function is generated for every `QObject` and places its international strings in its own "namespace," which has the same name as the class.

`tr()` serves two purposes:

1. It makes it possible for Qt's `lupdate` tool to extract all of the translatable string literals.
2. If a translation is available, and the language has been selected, the strings will actually be translated into the selected language at runtime.

If no translation is available, `tr()` returns the original string.

It is important that each translatable string is indeed fully inside the `tr()` function and extractable at compile time. For strings with parameters, use the `QString::arg()` function to place parameters inside translated strings. For example,

```
statusBar()->message
(tr("%1 of %2 complete. progress: %3%%")
.arg(processed).arg(total).arg(percent));
```

This way, translations can place the parameters in different order in situations where language changes the order of words/ideas.

For a much more complete guide to internationalization, we recommend [Blanchette06], written by one of Linguist's lead developers.

POINT OF DEPARTURE

There are other open-source implementations of signals and slots, similar to the Qt `QObject` model. One is called **xlobject** (available at http://sourceforge.net/projects/xlobject). In contrast to Qt, it does not require any `moc`-style preprocessing, but instead relies heavily on templates, so it is only supported by modern (post-2002) C++ compilers. The **boost** library (available from http://www.boost.org) also contains an implementation of signals and slots.

REVIEW QUESTIONS

1. What does it mean when object A is the parent of object B?
2. What happens to a `QObject` when it is reparented?
3. Why is the copy constructor of `QObject` not public?
4. What is the composite pattern?
5. How can `QObject` be both composite and component?
6. How can you access the children of a `QObject`?
7. What is an event loop? How is it initiated?
8. What is a signal? How do you call one?
9. What is a slot? How do you call one?
10. How are signals and slots connected?
11. How can information be transmitted from one object to another?
12. Deriving a class from `QObject` more than once can cause problems. How might that happen accidentally?
13. What is the difference between value types and object types? Give examples.