Table of Contents

1
Miscellaneous and Advanced Concepts

In this chapter, we will present you with some additional topics that didn't fit into the main part of the book.

First, we'll take a look at teaching you programs to communicate with the user in different languages. Then, we will see how to establish communication channels between processes. Next, it will be explained how to make your applications multithreaded to leverage the growing number of computing cores in modern machines.

Handling media data is also an important topic. Thus, we will try to show you the basics of using multimedia with Qt in the following sections of the chapter. Finally, we will look into debugging and testing. This will cover how to print simple debug messages as well as how GDB or CDB are integrated into Qt Creator and made usable through this IDE. At the end, Qt Test will be introduced to show you how you can realize unit tests with Qt.

By the end of this chapter, you will have a broader perspective of the technologies available as a part of Qt. You will be able to use a wider range of tools and technologies in your games, making them better and richer in terms of features.

The main topics covered in this chapter are the following:

- Internationalization
- Inter-process communication and multithreading
- Audio and video
- Debugging and testing

Internationalization

Qt is an application programming framework used all over the world. Just as it can handle many text encodings, it also supports many languages. With Qt, you can easily teach your application to communicate with the user using their native language. The font engine used by Qt can render fonts that use a variety of different writing systems and all standard text widgets can accept input in any language, and Qt's text drawing routines try to take into consideration text rendering rules present in particular languages. Still, you as the application programmer have to perform some additional work to ensure that your application is correctly internationalized and localized.

Since `QString` is based on Unicode, it can hold and express text in virtually any language in the world. Thus, it is important that you use `QString` for every user-visible text in your program. However, other text data that the user can't see (for example, object names, debug messages, and text format definitions) can still be held using the traditional `const char *` and `char` types.

Producing translations

Before you can translate your application, you need to mark texts that should be translated in the source code. We described how to do that in `Chapter 6`, *Qt Core Essentials*. Basically, any string literal visible in the user interface should be passed through the `tr()` or `QCoreApplication::translate()` functions. It may be quite time-consuming to do that if your application wasn't originally written with internationalization in mind, so marking the text for translation as you write the code is strongly recommended.

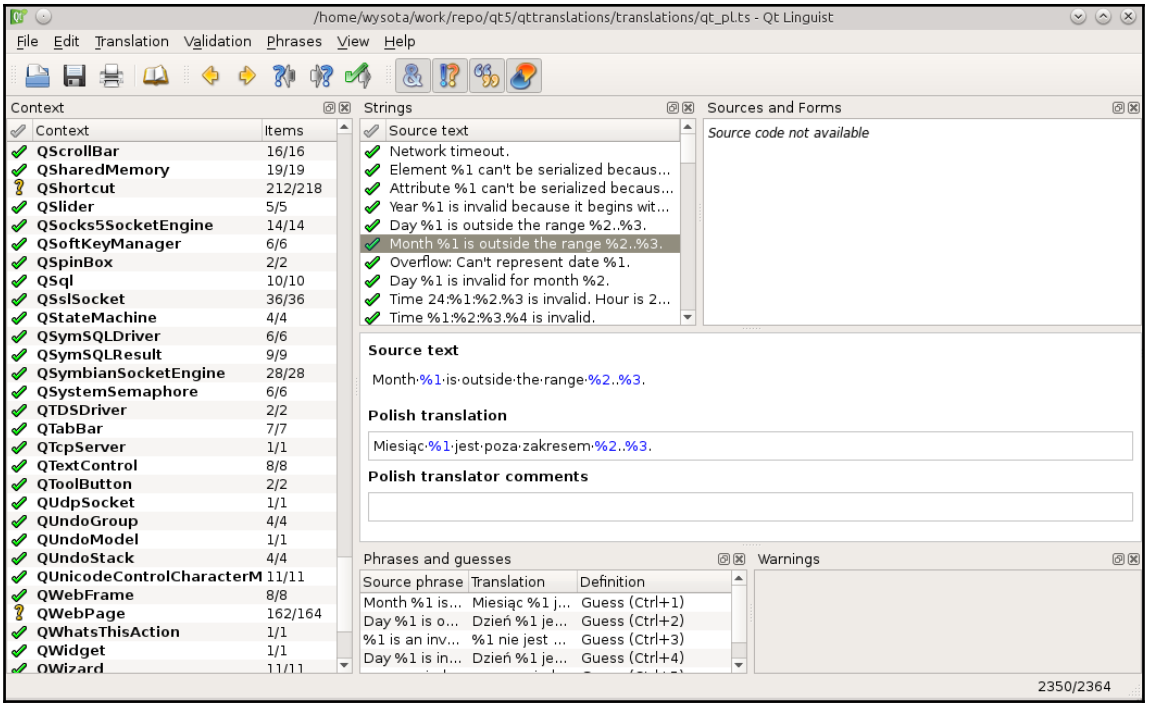
Once the source code is complete, the next step is to extract all messages and produce message catalogs. This is done by declaring translations in the project file. To declare translations, you need to fill the `TRANSLATIONS` variable with paths to catalogs that need to be maintained by the project:

```
TRANSLATIONS += translations/myapp_en.ts \
                translations/myapp_fr.ts \
                translations/myapp_pl.ts
```

The convention is to name the files with your application name, followed by an underscore, the international symbol of the language (for example, `en` for English, `fr` for French, and `pl` for Polish), and a `.ts` file extension. If a language is used in more than one country, you can follow the language name with a country symbol forming the full language variant (for example, `en_us` or `en_au`).

When the list is ready, you should run the `lupdate` tool on the project file. You can either do this manually or from Qt Creator by opening the **Tools** menu, going to **External** and then **Linguist**, and finally choosing **Update translations (lupdate)**. This will produce files declared in the `TRANSLATIONS` variable that contain all the messages from your project. Whenever you change your source code, you can run `lupdate` again to update message catalogs without overwriting the already present translations.

The next step is to give those files to people who can act as translators for the application. Qt contains a dedicated tool called **Linguist** that can help with the task. You can see it in the following screenshot:



The Linguist tool is very easy to use, so we will not discuss it in detail here. Two important things to note are to set the language details for the catalog as you start working with a file by opening the **Edit** menu and choosing **Translation File Settings**. The other important thing to remember is to mark your translations as "done" if you want them to be used. This is done with the *Ctrl + Enter* keyboard shortcut (or by choosing **Done** and **Next** from the **Translation** menu) so that a green tick appears next to the source text in the list of strings.

When translations are ready, you can call the `lrelease` tool (either from Qt Creator or from Qt Linguist) to produce compressed message catalogs (with the `.qm` file extension).

Applying translation

The final step is to enable the translation process for your application. Create an instance of `QTranslator` and load a message catalog (`.qm`) into it. You can specify a file path in the real filesystem or even put the message catalog in the resource file and specify a resource path. Each `QTranslator` object can only contain content of a single message catalog file.

To apply a translation to the application, pass the `QTranslator` object to the `QCoreApplication::installTranslator()` method. From now on, all calls to `QObject::tr()` will result in substituting original texts with their translated versions. If a translation for any message cannot be found, the translator uses the original text as a fallback.

Time for action – Translating an application

Create a new **Qt Widgets Application** in Creator. Choose to create a widget class called `Widget`. Place two buttons on the form and name them `left` and `right`. Set their text to `Left` and `Right`. Connect the `clicked` signal of each button to a slot in the `Widget` class:

```
ui->setupUi(this);
connect(ui->left, &QPushButton::clicked,
        this, &Widget::leftClicked);
connect(ui->right, &QPushButton::clicked,
        this, &Widget::rightClicked);
```

The slots will simply display message boxes:

```
void Widget::leftClicked() {
    QMessageBox::information(this, "", tr("You clicked left!"));
}
void Widget::rightClicked() {
    QMessageBox::information(this, "", tr("You clicked right!"));
}
```

Now, let's say that we have to translate our application into German and Russian. Edit the project file to add the `TRANSLATIONS` variable:

```
TRANSLATIONS += translations/myapp_ru.ts translations/myapp_de.ts
```

Save the project file. Next, open Creator's **Tools** menu, choose **External**, **Linguist** and finally, **Update Translations (lupdate)**. You will see `lupdate` crawl your source files and report the number of message entries found. All specified files will be created and, on subsequent runs, they will be updated with new text for translation.

Open each file in Qt Linguist application and input the translations. You will note that both texts from the form and texts from the message boxes appear in the file. Linguist even displays a translated version of the form, allowing you to check that the translated text fits into the form correctly.

If the same message in one language can have more than one meaning represented by different messages in another language, as we described earlier, to make the translator's work easier, the programmer can set a second argument to `tr()`, which is text visible for the translator in Linguist as a **Programmer comments** field near the message to be translated.

From the **File** menu of Linguist, choose **Save** and then **Release**. This will create message catalogs (binary translation files with the `.qm` extension) in the same directory.

Add a new Qt resource file to the project and add all the `.qm` files to the resources. You don't need to add the `.ts` files, since they are not needed for running the application.

The last part is to apply the translations. Add the following code to the `main()` function:

```
QApplication app(argc, argv);
QTranslator translator;
if (!translator.load(QLocale::Russian, ":/translations/myapp_")) {
    qWarning() << "failed to load translation";
}
app.installTranslator(&translator);
```

As you can see, we've specified a part of the resource path to the translation files. Instead of specifying the language suffix (such as `ru` or `de`) explicitly, we specify the value of the `QLocale::Language` enum. Qt will automatically use the correct suffix and find the corresponding file. Note that we have to name our translation files using the same pattern so that when they are converted to message catalogs, `QTranslator::load` will be able to find them. Alternatively, if you used another naming scheme, you could pass the full path to the file to `QTranslator::load`.

The second installation step involves calling `installTranslator` on the application object, passing it a pointer to the translator object that is to be added to a list of objects responsible for showing our user interfaces in the proper language. Remember to ensure that the translator object lives as long as you want it to perform translations. Do not create the object on stack unless you are in the `main()` function.

When you run the application, you should see the translated user interface now. Try to replace `QLocale::Russian` with `QLocale::German` and see the other translation.

What just happened?

Each user-visible string is wrapped in a `tr()` function call. If a translator is installed on the application object, a call to `tr()` results in calling `QTranslator::translate()` for each installed translator that performs a lookup in its message catalog. If a match is found, the translation is returned and the search stops. Otherwise, an empty string is returned and the next translator is tried. If no match is found in any of the catalogs, the original (untranslated) message is used.

`QTranslator::translate()` is a virtual method; thus, by subclassing `QTranslator` and reimplementing `translate()`, you can provide a custom translation mechanism for your application.

Have-a-go hero – Selecting the appropriate language

Of course, specifying the language in the application source code is not really acceptable. The simplest solution is to use the `QLocale::system()` function that returns the system locale. If a translation of your application is present for the system locale, Qt will load it. Otherwise, the UI will not be translated.

Alternatively, you can allow the user to pass the desired language using command-line arguments. Even better, you can create a dialog that allows the user to select one of the available translations and save the selected language using the `QSettings` class.

Translating user-visible content in QML

All internationalization functionality we described so far applies also to QML. All the mentioned C++ functions have their equivalents in QML. The following table shows the corresponding mappings:

C++	QML
`tr()`	`qsTr()`
`translate()`	`qsTranslate()`
`QT_TR_NOOP()`	`QT_TR_NOOP()`

All other aspects remain the same—you use the `lupdate` and `lrelease` tools for producing catalogs, and Qt Linguist for doing the translations. Also, you still need to install a `QTranslator` object on the application.

Inter-process communication

More often than not, one program will need to cooperate with another program running on the same machine. Sometimes, one of them is providing services to the other; another time, they are exchanging data or maybe one process is providing a monitoring interface that another process can attach to and read statistics. Qt handles many standard ways of communication between processes, thereby facilitating communication with programs not only based on Qt. In this chapter, we will take a look at some of the IPC mechanisms commonly used, namely, shared memory and local sockets.

The shared memory

Usually, each process has its own memory space reserved for it by the operating system kernel, and the process has exclusive access to each allocated block. The **shared memory** IPC mechanism works by creating a memory block in the operating system kernel that can be attached to more than one process so that one process can modify a memory area, while another process can read the data stored within. It is very important to remember that only this particular block of memory is shared between processes. An implication of this is that it does not make sense to store pointers or any complex structures (such as class instances) in a shared memory block. You should always treat shared memory as an opaque block of bytes, similar to a file on the disk.

To create a shared memory block, we need to ask the system to create it for us. Qt handles shared memory via the `QSharedMemory` class that represents a single shared memory block. Before we can create the block, we have to decide what key we will use to identify this particular block (as there can be many shared memory blocks in the system) so that other processes can attach to it using the same key. This is done either by passing the key to the constructor of `QSharedMemory`, or by using the dedicated `setKey()` call and passing a string containing the key.

Keys used by `QSharedMemory` differ from the representation used by the platform Qt is running on. To be able to talk with a non-Qt application, you have to use `setNativeKey()` instead.

Then, you can `create()` a new block (passing the size of the block that is to be created) or `attach()` to an existing one. Optionally, you can tell `QSharedMemory` whether you want `ReadOnly` access to the block or `ReadWrite` access. Both of these methods return a Boolean value, telling us whether the operation was successful or not. To access the shared block, you can call `data()`, which returns a pointer to the shared block (or null if the previous operation had failed). Now you can start using the shared block by reading from it or writing to it. Since more than one process can try to access the block at the same time, it is important to protect it from concurrent access. Qt provides a way to do this using the `lock()` and `unlock()` methods.

Once you are done with using the block, call `detach()` on it to release it from the current process. Once all processes using the block call `detach()`, the block is destroyed and can no longer be accessed. When a `QSharedMemory` object is destroyed, it's automatically detached if it wasn't already.

As an exercise, we will create a stub of a game engine that reports game statistics through shared memory. For that, we will create two programs—the engine itself that will create a shared memory block and modify it according to what happens in the engine, and a client program that will attach to the block and read it.

Time for action – Game engine

Our game engine will really be more of a mock-up rather than a real engine. Create a **Qt Widgets Application** project and get rid of the widget Creator created for you as we will not need it. Now add a new header file to the project and call it `gamestats.h`.

It will contain the definition of the data structure that we will expose through shared memory:

```
#include <QtGlobal>
struct GameStats {
  quint32 playerCount = 0;
  quint32 uptime = 0;
};
```

Next, add a new class called `GameServer` and derive it from `QObject`. We'll need a slot that will write the game state to shared memory and a few private fields:

```
class GameServer : public QObject {
    Q_OBJECT
public:
    GameServer();
    ~GameServer();
private slots:
    void writeState();
private:
    QSharedMemory m_sharedMemory;
    QElapsedTimer m_elapsedTimer;
    QSlider *m_slider;
};
```

Initialize the `m_sharedMemory` object in the constructor:

```
m_sharedMemory.setKey("gameStats");
if(!m_sharedMemory.create(sizeof(GameStats))) {
    qFatal("failed to create shared memory");
    return;
}
```

Since we intend to write the content of the `GameStats` structure to the shared memory chunk, we pass the size of the structure in bytes to the `QSharedMemory::create()` function. If the operation fails for some reason, we abort the application.

We've encountered the `QElapsedTimer` class before. Basically, it's a helper class that remembers its starting time and can calculate how much time has passed since then. We start the timer in the constructor:

```
m_elapsedTimer.start();
```

Next, since we want to write the game's uptime to the shared memory, we need our `writeState()` slot to be called repeatedly each second. We can achieve this by setting up a `QTimer` in the constructor:

```
QTimer *timer = new QTimer(this);
connect(timer, &QTimer::timeout,
        this,  &GameServer::writeState);
timer->start(1000);
```

We don't have an actual game engine here, so let's use a slider as the data source for the `playerCount` field. Create and show the slider in the constructor as well:

```
m_slider = new QSlider();
connect(m_slider, &QSlider::valueChanged,
        this, &GameServer::writeState);
m_slider->show();
```

Unlike as is the case with the timer, we can't set `this` as the parent of the slider because widgets can only have other widgets as parents. This means that you'll need to delete the slider in the destructor to avoid a memory leak.

All the preparations are complete, and we can now implement the `writeState()` slot:

```
void GameServer::writeState() {
    GameStats stats;
    stats.playerCount = m_slider->value();
    stats.uptime = m_elapsedTimer.elapsed() / 1000;
    if(!m_sharedMemory.lock()) {
        qFatal("memory lock failed");
        return;
    }
    qDebug() << "writing to shared memory";
    std::memcpy(m_sharedMemory.data(), &stats, sizeof(GameStats));
    m_sharedMemory.unlock();
}
```

First, we initialize a `GameStats` object. Next, we try to lock the shared memory and abort the application if there was an error. Finally, we use `memcpy()` to copy the binary content of the structure to the shared memory chunk. After we `unlock()` the memory, other processes will be able to read this content.

The `main()` function simply creates a `GameServer` object:

```
int main(int argc, char *argv[]) {
    QApplication a(argc, argv);
    GameServer server;

    return a.exec();
}
```

What just happened?

We implemented a really simple mechanism for periodically pushing two values to the outside world using shared memory. Now, we need the other end of the inter-process communication system—an application that can read those values and do something meaningful with them.

Time for action – Client application

Start by creating a new project and choosing **Qt Console Application** as its type. Add the `gamestats.h` file to the project and include it in `main.cpp`.

This time, we won't need to create extra classes. Let's put all the code into the `main()` function. First, we create a `QSharedMemory` object and use the `attach()` function to try to attach it to an existing shared memory chunk:

```
QSharedMemory sharedMemory("gameStats");
if(!sharedMemory.attach(QSharedMemory::ReadOnly)) {
    qDebug() << sharedMemory.errorString() << endl;
    return -1;
}
```

Since we pass the same key (`gameStats`) in both the server and the client, we can be sure that both applications will access the same chunk of memory. If the operation is successful, we can lock the shared memory and read the content:

```
GameStats stats;
if (!sharedMemory.lock()) {
    qFatal("memory lock failed");
    return -1;
}
std::memcpy(&stats, sharedMemory.data(), sizeof(GameStats));
sharedMemory.unlock();
```

Again, we use `memcpy()` to copy the bytes from the shared memory chunk to the content of our `stats` variable. It's up to you how to use the obtained value. For example, you can simply print it to the output:

```
qDebug() << "Players online:" << stats.playerCount;
qDebug() << "Engine up time:" << stats.uptime << "seconds";
```

To test our monitor, run the server application and then, after some time has lapsed execute the newly created monitor tool. Modify the slider value and execute the tool again to see the changed result.

What just happened?

The client application attaches to an existing shared memory block by passing a key shared with the server. From that moment on, the block of memory is visible from both applications. However, it is not safe when one process reads the shared segment and another modifies it at the same time, as memory can be modified in the middle of the reading process, giving the reader a mix of old and new data. To prevent such situations, any access to the memory has to be synchronized. Every read and write operation to the block is prepended with obtaining an exclusive lock on the segment. It is important to release the lock as soon as you don't need it, as holding the lock for any extended period may result in performance degradation of other processes trying to obtain access to the shared segment.

Passing complex data

The approach we just used will only work for structures that are **trivially copyable**, that is, structures that can be moved around by copying bytes. While basic integer types and fixed sized arrays of those types are safe to use, there are plenty of classes that are not trivially copyable. For example, `QString`, `QVector`, or any other class that allocates memory is not safe to copy in that way. Even copying these values within one process will lead to a double freeing of the same buffer, and passing it to another process will result in trying to access a pointer that is not accessible in the other process.

This means that you can't use `memcpy` to put such a value into the shared memory object. However, you still can serialize the value using one of the many techniques discussed in `Chapter 6`, *Qt Core Essentials*. The resulting array of bytes can be safely written to the shared memory and then deserialized back in the other process.

However, note that serialization generally gives you byte arrays of variable size, so ensure that you allocate enough shared memory for the allowed range of values. You will also need to pass the actual size of the byte array along with the content to successfully decode it on the other end.

Local sockets

Shared memory is most useful when there is a group (usually more than two) of processes that need to use the same data, and the data is either not very complex nor very large. However, there are other situations where processes might want to exchange a lot of data over a long period of time. An example of such a situation is an external game console sending requests to a game engine and receiving some data it needs in response. For such use cases, an ideal solution is to use a local socket (implemented in Qt with `QLocalServer` and `QLocalSocket`), which is usually a special file in the filesystem that acts as a pipe between two or more processes, where one process streams data into one end of the pipe and another process reads data from the other end. From the developer's perspective, local sockets are very similar to network sockets that use TCP protocol (they were described in `Chapter 7`, *Networking*).

A system equivalent of a local socket is a named pipe or a Unix domain socket, depending on the operating system. Thus, you can use your operating system's calls to communicate with a Qt application that makes use of `QLocalSocket`.

Local socket communication uses the client-server pattern. One process creates a server socket and listens to incoming connections. Other processes (clients) connect to that server socket and can perform streaming two-way communication.

To use local sockets, you have to enable the `QtNetwork` module for your project by adding a `QT += network` line to the project file.

To set up a server, you need to use an instance of the `QLocalServer` class. To start the server, you will call the `listen()` method and pass it a file path that is to be created as your socket. At this point, if someone connects to the socket, a `newConnection()` signal will be emitted. You can catch it and call `nextPendingConnection()` to obtain a socket handler for communication with the client.

Handling the client end is also easy—you create an instance of `QLocalSocket` and call `connectToServer()`, passing it the file path to the server socket. When the connection is successful, the socket will emit a `connected()` signal.

From now on, the two sides behave the same way as both use `QLocalSocket` for communication. Since it's derived from `QIODevice`, you can use `read()` and `write()` to perform operations on the socket. Connection is terminated upon calling `close()`, in which case the socket will emit a `disconnected()` signal.

As local sockets are very similar to TCP sockets, you can refer to `Chapter 7`, *Networking*, for a more complete description of socket communication.

Web sockets

QML does not support constructs such as local sockets or shared memory out of the box. If you want to use them, you have to provide your own QML wrappers for them. However, QML does have an IPC mechanism in the form of web sockets. This protocol is often used in web applications for instantiating sockets from within a JavaScript environment and using it to connect over TCP to external servers. While the QML engine doesn't act as a complete web browser, you can still use web sockets in your QML application, thanks to the `QtWebSockets` QML module.

To use web sockets, you need to import the right module in your document:

```
import QtWebSockets 1.0
```

Then, you can start declaring socket servers:

```
WebSocketServer {
    id: socketServer
    listen: true
    port: 8080
    onClientConnected: {
        webSocket.onTextMessageReceived.connect(function(message) {
            console.log("Server received message: ", message);
            webSocket.sendTextMessage("Server says 'Hello!'");
        });
    }
}
```

The server object has an API similar to `QTcpServer`. It exposes host and port properties that define where the server socket is created. By default, the socket is bound to the localhost and the port is chosen automatically.

When a client connects to the server, the `onClientConnected` handler is invoked and passed a socket through which the server communicates with the client. In the preceding example, we connect a function to the `textMessageReceived` signal, which prints the message to the console and sends a textual response back to the client.

A client socket can be created by declaring the `WebSocket` instance:

```
WebSocket {
    id: socket
    url: "ws://localhost:8080"
    active: true
    onStatusChanged: {
        console.log("Client status changed: ", status);
        if(status == WebSocket.Open) {
            sendTextMessage("Hello!");
        }
    }
    onTextMessageReceived: {
        console.log("Client received message: ", message);
    }
}
```

The socket is connected to a remote server defined by the `url` property (you can query the earlier created socket server for its URL). When the connection is established, we use `sendTextMessage` to transmit data to the server. The response is intercepted in the `onTextMessageReceived` handler and printed to the console just like before.

You can use web sockets to fetch data from remote locations or to trigger actions in a web service. You can also establish a communication channel to a process running on the same machine, just like you would when using local sockets.

Equivalent API for C++ is also available through classes called `QWebSocket` and `QWebSocketServer`.

Multithreading

Multithreading allows simultaneous execution flows within a single process. The key difference between multithreading and multiprocessing is that all threads within a process have access to the same memory space.

Qt provides its own abstraction to threading, which covers all the implementation details that are platform-dependent. While C++11 introduced threading support in the language, Qt applications still greatly benefit from using the Qt threading API. The most significant benefit is the ability to safely use signals and slots across threads.

Avoiding long operations in the main thread

When developing a game or any application with graphical user interface, you are often limited by the rule that you must not perform long calculations in the main thread. If you run a slow function in a click handler, your interface will simply freeze until the function returns. Sometimes, you can avoid it by splitting a long operation into a sequence of short operations and running them periodically, allowing Qt to update the user interface in between. However, sometimes it's considerably easier to spawn a new thread and run a slow function in that thread where it won't interfere with the user interface at all.

However, it's not that simple. There is another rule—you must never access the user interface directly from other threads, because the implementation of the user interface is not thread safe. For example, you can't add or remove widgets or change text on a label. Well, since you did something useful in a background thread, you almost certainly want to pass the result back to the UI thread, but how can we achieve this? In Qt, it's pretty easy.

Each `QObject` has **thread affinity**, that is, it belongs to a particular thread. Any event handlers and slots of the object will be executed in that thread (the only exception is a signal-slot connection made with the `Qt::DirectConnection` type). By default, an object belongs to the thread where it was created.

You can learn about the object's thread affinity using the `thread()` function. It returns a `QThread` object representing the thread this object belongs to. You can also change the object's thread affinity by calling the `moveToThread()` function, passing a `QThread` object representing the target thread. Note that you can't change the object's thread affinity if it has a parent or if you try to do it from the thread the object doesn't belong to originally.

All UI objects naturally belong to the main thread because they were created in it. While you can't call slots of UI objects directly from other threads, you can create a signal and connect it to a UI object slot. When a signal is emitted, Qt will execute the slot in the thread of its owner, so it will be safe. Signals themselves are thread safe, so you can call them from any thread.

Time for action – Creating a background worker thread

Create a new **Qt Widgets Application** project with a widget-based form. Add a button called `start` that will start the calculation and a label called `result` for displaying the result. Next, create a class named `Worker` that inherits `QObject`. This class will perform a long calculation in its `calculate()` public slot:

```
void Worker::calculate() {
    float x = 0;
    for(int i = 0; i < 100000000; i++) {
        x += sin(i);
    }
    emit finished(x);
}
```

After the calculation is done, the object emits the `finished()` signal that we also need to create. That's all we need for the worker. Let's start setting up the thread communication.

First, add the following private fields to the widget class:

```
QThread* m_workerThread;
Worker* m_worker;
```

Initialize these fields in the widget's constructor:

```
m_workerThread = new QThread(this);
m_workerThread->start();
m_worker = new Worker();
m_worker->moveToThread(m_workerThread);
connect(m_worker, &Worker::finished,
        this, &Widget::workerFinished);
```

First, we create a new `QThread` object and call its `start()` function. This tells Qt to begin a new native thread and start an event loop in it. The loop will keep running until we call the `exit()` or `quit()` functions of the thread object, in which case the event loop will stop once it has processed the current event. At the moment, the new thread's event loop has nothing to do, but we'll make it run the `Worker::calculate()` function later. Next, we create a `Worker` object and move it to the new thread. This operation ensures that whenever a slot of this object needs to be called, it will be called by this thread's event loop. Finally, we connect the thread's `finished()` signal to our `workerFinished()` slot. Since our widget object belongs to the main thread, the `workerFinished()` slot will be called in the main thread, even though the `finished()` signal is emitted from the worker thread.

When the start button is pressed, we should run the following code to start the calculation:

```
void Widget::start()
{
    ui->result->setText(tr("Calculating..."));
    QTimer::singleShot(0, m_worker, &Worker::calculate);
}
```

The `QTimer::singleShot()` function is a shortcut for creating a `QTimer`, connecting its `timeout()` signal to a slot and starting it with the specified delay. We already know that the `calculate()` slot will be called in the worker thread, and passing `0` as the delay ensures that the slot will be called as early as possible.

This approach won't work if you need to call the worker's slot with arguments. In this case, create a signal with the same arguments in the widget class and connect it to the worker's slot. When you need the slot to start running, simply emit the widget's signal, and it will work just as in this example.

In the `workerFinished()` slot, we can safely update the UI:

```
void Widget::workerFinished(float result) {
    ui->result->setText(tr("Result: %1").arg(result));
}
```

Finally, we need to perform some cleanup in the widget's destructor:

```
Widget::~Widget() {
    m_worker->deleteLater();
    m_workerThread->quit();
    m_workerThread->wait();
    delete ui;
}
```

The `deleteLater()` call will schedule the deletion of the worker, but it will happen in the worker's thread. The `quit()` call instructs the thread to stop its event loop. Once the event loop is stopped, the native thread will terminate. The `wait()` function will block the current loop (in our case, the main loop) until the thread is terminated. If calculation is not running at the moment, the termination should happen almost instantly. If the operation is running, however, the current thread will be blocked until it finishes.

It's also possible to execute operations in another thread by subclassing `QThread` and overriding its `run()` virtual function. However, this is usually not advised, as this approach is not really easier than the method described earlier, and doing so will leave the thread without an event loop, preventing you from using slots in the thread and forcing you to implement your own logic for stopping the thread.

Thread synchronization

Threading in Qt is governed by the same rules as with every other technology, including the fact that access to a resource shared by more than one thread needs to be synchronized to avoid race conditions. Qt offers a number of mechanisms for thread synchronization, the simplest one being `QMutex`, which implements a classic mutex primitive with `lock()` and `unlock()` operations, ensuring that only one thread can execute in a critical section. This class is usually used in conjunction with the `QMutexLocker` class that locks the mutex when the locker is created and unlocks it when the locker goes out of scope. This ensures that the mutex will be properly unlocked even if an exception is thrown in the protected section.

The mutex is usually stored in a private field of the shared object. In each public method that accesses the object's data, the mutex must be locked:

```
void Object::setX(int x) {
    QMutexLocker(&m_mutex);
    m_x = x;
}
```

Threading in QML

In QML, you can execute scripts in a separate thread with the use of the `WorkerScript` element. Its API is very simple—it accepts a URL of a script to execute via the `source` property. Once the script is started, you can communicate with it using messages. `WorkerScript` objects have a `sendMessage()` function that accepts a JavaScript object. The object is transmitted to the worker thread where it can be intercepted by connecting a function to `WorkerScript.onMessage`:

```
WorkerScript.onMessage = function(message) {
  ...
}
```

The script can also use `sendMessage` to transmit data to the main thread where again, it can be read by connecting to `onMessage` and querying the `reply` property of the received object:

```
WorkerScript {
    source: "script.js"
    onMessage: {
        console.log(messageObject.reply)
    }
}
```

Since the script is executed in a thread different from the main QML thread, it cannot access any Qt Quick items or their properties. It cannot even access properties of the `WorkerScript` object itself or any object exposed to QML from C++.

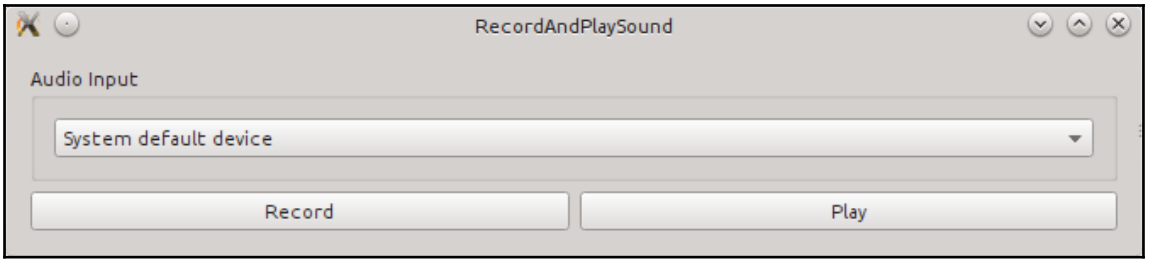
Audio and video

In this section, we will take a look at how to play sounds and videos. This is crucial for successfully programming games, or can you think of a modern game without music or videos? Normally, you have background music and, additionally, sound effects when a player jumps, kicks, shoots a gun, or performs similar actions. In some games, there are also short videos introducing a level or telling parts of the game's story.

Before we start to play with Qt's Multimedia module, keep in mind that you have to enable it. In qmake-based projects, add `QT += multimedia`. If you are using QML, you'll also need to add `import QtMultimedia 5.0` to your QML file. Also be aware that Qt Multimedia uses the media framework of the underlying operating system. This means that depending on the machine your game is running on, specific audio or video codecs may or may not be available.

Time for action – Recording audio

To demonstrate how to record and play audio, we'll create a little program called `RecordAndPlaySound`:



As always, you'll find the sources for this example attached to this book. The graphical user interface is—as you can see—very minimalistic. The window is implemented as a designer form class. At the top, there is a combobox named `ui->input`. Its purpose is to select the audio input. Beneath, we find two checkable buttons: one for recording audio (`ui->record`) and one for playing the last recorded audio (`ui->play`). Since you already know how to build such a simple GUI, we skip this part and directly take a look at the relevant code that is located in the main window's constructor:

```
RecordAndPlaySound::RecordAndPlaySound(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::RecordAndPlaySound),
    m_recorder(new QAudioRecorder(this)),
    m_player(new QMediaPlayer(this))
{
    ui->setupUi(this);
    m_file = QDir(QCoreApplication::applicationDirPath())
        .absoluteFilePath("test.ogg");
    m_recorder->setOutputLocation(m_file);
    const QStringList inputs = m_recorder->audioInputs();
    for(const QString &input: inputs) {
        ui->input->addItem(m_recorder->audioInputDescription(input),
                           input);
    }
    m_recorder->setAudioInput(ui->input->currentData().toString());
    qDebug() << "supportedAudioCodecs: "
             << m_recorder->supportedAudioCodecs();
    qDebug() << "supportedContainers: "
             << m_recorder->supportedContainers();
    QAudioEncoderSettings settings;
    settings.setCodec("audio/x-vorbis");
    m_recorder->setEncodingSettings(settings, QVideoEncoderSettings(),
                                    "audio/ogg");
    connect(m_player, &QMediaPlayer::stateChanged,
            this, &RecordAndPlaySound::playbackStateChanged);
}
```

Now start the program, select the **Audio Input**, and check the **Record** button. Once you say your test phrase, uncheck the record button to stop the recording. Now hit the **Play** button. In theory, you should then hear what you last recorded.

What just happened?

In the initializer list, we initialize some member variables: `m_file` of the `QString` type holds the file path that will be used to store the recorded audio; `m_recorder` of the `QAudioRecorder` type is responsible for recording and `m_player` of the `QMediaPlayer` type is responsible for playing the audio. In this example, `m_file` is set to a file named `test.ogg`, which will be placed right beside the program's executable. In a real game, you will have to determine the filename in a more sophisticated way.

In the constructor's body, after setting up the UI, we define where `m_recorder` should save the recorded audio. This is done by passing the previously defined `m_file` to `QMediaRecorder::setOutputLocation()`. Actually, `setOutputLocation()` expects a `QUrl`, but thanks to implicit type conversion, we can also pass a `QString`.

Since `setOutputLocation()` takes `QUrl`, it is not restricted to local files. You can also use network files. Even if we did not check it in the example, `setOutputLocation()` returns `false` if the passed location can't be used to store media recording.

Next, we want to provide the ability to choose the audio input. For devices that have several audio inputs, this is a must. By using `QAudioRecorder::audioInputs()`, we get a list containing all inputs that are available on the device. Unfortunately, the returned identifiers are not really readable, especially not for most users of your game. Alternatively, do you expect your customer to know what, `alsa:front:CARD=Creative,DEV=0` means, for example? People will probably run away screaming if they saw this kind of text, so we need a better way of dealing with the audio inputs. Thanks to the `QAudioRecorder::audioInputDescription()` function that takes the identifier as an argument, we can convert the cryptic strings into a better readable format. This, way we get a slightly nicer stringm such as **HDA Creative, CA0110-IBG Analog Front speakers**. So we loop through the list of audio inputs, generate the readable description, and put them to the combobox. Note that we also pass the original audio input identifier to the items as the so-called user data. Thus, we are able to provide the following slot:

```
void RecordAndPlaySound::on_input_currentIndexChanged(int index)
{
    m_recorder->setAudioInput(ui->input->itemData(index).toString());
}
```

Every time the current index of the checkbox changes, we fetch the user data, transform it to `QString`, and pass it to `QAudioRecorder::setAudioInput()` so that `m_recorder` will use this audio input for recording sounds. To start the program in a consistent state, we set the recorder's audio input to the user value of the combobox's currently selected item right after we have left the loop for setting up the box. If you do not define an audio input, Qt will take the default one.

Next, we define the audio's encoding that is handled entirely by a class called `QAudioEncoderSettings`. Remember, however, that `QAudioEncoderSettings`—or better, it's base class, `QMediaRecorder`—can only handle encodings that are supported by the current operating system, and the exact names of codecs and containers can vary. To gather the available codecs and options, `QMediaRecorder` offers `supportedAudioCodecs()`, `supportedAudioSampleRates()`, and `supportedContainers()`. In our example, we canonically dictate to use the free audio format Ogg Vorbis:

```
QAudioEncoderSettings settings;
settings.setCodec("audio/x-vorbis");
m_recorder->setEncodingSettings(settings, QVideoEncoderSettings(),
                                "audio/ogg");
```

Here, after creating an object of `QAudioEncoderSettings`, we set the codec with `setCodec()` to `audio/vorbis`. For all other available options that `QAudioEncoderSettings` provides, we are fine with their default values.

If you like to change other options, these self-explanatory functions would be of interest: `setBitRate()`, `setChannelCount()`, `setEncodingMode()`, `setQuality()`, and `setSampleRate()`. If you like to add encoder-specific options, that are not covered by Qt Multimedia's API, use `setEncodingOption()` and `setEncodingOptions()`.

Lastly, we need to set the chosen codec setting to `m_recorder` using `QMediaRecorder::setEndocingSettings()`. As the first argument, the audio setting is passed, while the video setting is passed in the second argument—yes, with Qt Multimedia, you also can record videos—and the container format is passed as the third. Here, we choose Ogg.

At the end of the constructor, we establish a connection that informs us about a state change of `m_player`. For now, do not worry about that; we will get to this in detail a little later. Rather, let's take a look how the actual recording is started and stopped:

```
void RecordAndPlaySound::on_record_toggled(bool checked)
{
    if (checked) {
```

```
            m_recorder->record();
        } else {
            m_recorder->stop();
        }
    }
```

This implementation is pretty simple—if the `ui->record` button gets checked, we start the recording by calling `record()`, and as soon as the button gets unchecked, we stop the recording with `stop()`. There is nothing more to do since we already set up the recorder in the constructor.

Have-a-go hero – Making the audio settings fail safe

Our example will fail terribly if Ogg Vorbis is not supported, or if its codec name or container name is different on the current platform. So go ahead and make it fail safe! To do so, you have to check the available codecs with `supportedAudioCodecs()` and then fall back to another (supported) codec if Ogg Vorbis is not available.

Time for action – Playing sound

Now, let's complete our little sample program and take a look at how to play back the recorded file:

```
void RecordAndPlaySound::on_play_toggled(bool checked)
{
    if (checked) {
        m_player->setMedia(QUrl::fromLocalFile(m_file));
        m_player->play();
    } else {
        m_player->stop();
    }
}
```

The overall logic for playing sound is the same as it is for recording: when the button `ui->play` gets checked, we call `play()` to start the playback. If the button gets unchecked, we stop the music with `stop()`. There is, of course, also a function named `pause()` to pause the playback, but for the sake of simplicity, we haven't implemented it here. Right before we call `play()`, we specify which file should be played; therefore, we call `setMedia()` and pass the `m_file` local variable, holding the file path to the recorded file. You do not have to call `setMedia()` each time before calling `play()`. Defining the media once is enough.

At this point, we owe you an explanation for the connection statement in the constructor. In this statement, the music player's `stateChanged()` signal was connected to this slot:

```
void RecordAndPlaySound::playbackStateChanged(QMediaPlayer::State state)
{
    if (state == QMediaPlayer::StoppedState) {
        ui->play->setChecked(false);
    }
}
```

The whole purpose of this slot is to uncheck the `ui->play` button when the music stops playing so that you have the chance to start the playback again. Hence, we uncheck the button when the player enters `StoppedState`. This state signifies that currently, no music is being played and that calling `play()` will play the media from the start. Apart from this state, `QMediaPlayer` can be in two more states: `QMediaPlayer::PlayingState` and `QMediaPlayer::PausedState`. The former indicates that the player is currently playing, while the latter indicates that it was stopped, although playing can be resumed from where it was paused.

To sum up music playing with Qt Multimedia, it's a three liner:

```
QMediaPlayer *player = new QMediaPlayer();
player->setMedia(QUrl::fromLocalFile(pathToFile));
player->play();
```

Have a go hero – Making a voice chat

In the chapter about networking, you saw how to make a simple text-based chat. With the knowledge of recording and playing sounds, let's extend that example by sending small audio messages. The steps you need to carry out are to record an audio message, read the content of the recorded file, send it, receive it, save it to a temporary file, and play it back.

Of course, a real voice chat application doesn't write each message to a file on disk and read it right back. That would be too slow. You would much rather read the audio stream to a byte array and send it over the network. The `QAudioRecorder` class cannot be used for that. You would need to use the `QAudioInput` class that provides a `QIODevice` for reading the raw audio data from the microphone.

Playing a sound in QML

Unfortunately, there is currently no built-in support for recording sound with QML. Playing sound, however, is straightforward and if you have studied the C++ part, there will be no surprises:

```
import QtQuick 2.9
import QtQuick.Window 2.2
import QtMultimedia 5.0
Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Hello World")
    Text {
        id: button
        text: qsTr("Play");
        anchors.centerIn: parent
    }
    Audio {
        id: music
        source: "example.ogg"
    }
    MouseArea {
        anchors.fill: parent
        onPressed: {
            music.play();
        }
    }
}
```

What just happened?

Well, let's skip the basic QML code for creating a button labeled `Play` and concentrate on the highlighted parts. First, we have the `Audio` type, and we give that object an ID that we can refer to later, and with `source`, we define the file that should be played. Finally, we start the playback by calling `play()` on the `Audio` object referenced by the `music` identifier when the rectangle gets clicked on. It's like the three liner of C++: creating `Audio`, defining the `source`, and playing it. The right codec will be chosen automatically if the media framework supports it.

If you like to implement a play/stop button, just alter `onPressed` to the following:

```
onPressed: {
    if (music.playbackState == Audio.StoppedState) {
        music.play();
        button.text = qsTr("Stop");
    } else {
        music.stop();
        button.text = qsTr("Play");
    }
}
```

We have simply inserted a check of audio's `playbackState`, which can be `StoppedState`, `PausedState`, and `PlayingState`. The meaning is the same as `QMediaPlayer::State`. If the player is currently in the stopped state, we start the music and change the text of the button to "stop". Otherwise, we stop the music and set the button's text to "play".

By the way of an alternative to `Audio`, we could also have used the more feature-rich `MediaPlayer` type.

Playlists and loops

If you have a background music theme for your game, it surely does not consist of only one file. It's more likely that you have multiple tracks that you want to play in a loop. Alternatively, maybe you really only have one file, but need it to be played in a loop. This can easily be achieved with `QMediaPlaylist`. First, you have to create an instance of this class:

```
QMediaPlaylist *playlist = new QMediaPlaylist();
```

Then, you need to add music or soundtracks to the list. This is done with `addMedia()`. As a parameter, you can either pass a single `QMediaContent` or a `QList` of `QMediaContent`. Since `QMediaContent` provides a default constructor taking `QUrl`, we can write this:

```
playlist->addMedia(QUrl::fromLocalFile("/path/to/firstFile.mp3"));
playlist->addMedia(QUrl::fromLocalFile("/path/to/secondFile.mp3"));
```

With `clear()`, on the other hand, one can remove all items from the list, but we do not want to do this right now. We would rather like to define how the sounds should be played. This can be done with `setPlaybackMode()`. Looping endlessly through the list can be done with the following:

```
playlist->setPlaybackMode(QMediaPlaylist::Loop);
```

Passing the playlist to the player and starting it can be done with this:

```
QMediaPlayer *player = new QMediaPlayer();
player->setPlaylist(playlist);
player->play();
```

Be aware that by calling `setPlaylist()`, the parentship of `playlist` is not passed to `player`, so we do need to take care of freeing the memory ourselves. If we use the player as the parent of the list—by passing it as an argument to the constructor, the playlist gets automatically deleted as soon as the player is deleted.

With `save()` and `load()`, `QMediaPlaylist` provides the ability to save and load playlists.

You can also use playlists from QML. Create a `Playlist` object and put multiple `PlaylistItem` objects into it to create a playlist. Then, pass the playlist to the `playlist` property of a `MediaPlayer`, `Audio`, or `Video` object.

Playing sound effects

`QMediaPlayer` can surely be used to play sound effects. For sound effects, however, it is better to use `QSoundEffect`, as this class is especially designed to play low latency sound effects. Its usage is similar to `QMediaPlayer`. Let's take a look at the following example:

```
QSoundEffect *effect = new QSoundEffect();
effect->setSource(QUrl::fromLocalFile("soundeffect.wav"));
effect->play();
```

First, we create an instance of `QSoundEffect`, then we set the file that should be played with `setSource()`, and finally, we start the effect with `play()`. If you want to play the effect multiple times, use `setLoopCount()`. As a special parameter, `QSoundEffect::Infinite` can be passed, resulting in a repeated playback of the effect. Since `QSoundEffect` inherits `QObject`, it can be conveniently used within signal and slot connections:

```
connect(someObject, SIGNAL(someSignal()), effect, SLOT(play()));
```

With this statement, every time `someObject` emits `someSignal()`, our `effect` plays the `soundeffect.wav` sound file. This approach is very light-weighted and should be preferred over `QMediaPlayer` for simple sound effects.

Playing videos in widget-based applications

The good thing about playing videos is that it is almost the same as playing sounds, so you do not have to make yourself familiar with a new syntax.

Let's take a look at the following three lines of code:

```
QMediaPlayer *player = new QMediaPlayer();
player->setMedia(QUrl::fromLocalFile("/path/to/movie.ogg"));
player->play();
```

Do you recognize these lines? They are the same as those we showed you for playing sounds and in fact, if you run the code, you will hear the sound of the video but won't see anything. First, you have to tell `QMediaPlayer` where to show the video. This is where a new module comes into play: Qt Multimedia widgets. To enable Qt Multimedia widgets, add `QT += multimediawidgets` to the project file. Then, we can use `QVideoWidget` as a canvas for the video:

```
QVideoWidget *videoWidget = new QVideoWidget();
player->setVideoOutput(videoWidget);
```

After we create an instance of `QVideoWidget`, we pass it to the player by calling its `setVideoOutput()` function. This instructs the media player to use `videoWidget` as a canvas. Note that you can only set one video output to a player. Displaying a movie on different widgets with only one player is not possible.

`QVideoWidget` also provides the `setBrightness()`, `setContrast()`, `setHue()`, and `setSaturation()` functions to alter the visible properties of the video.

Like any other `QWidget`, `QVideoWidget` can be part of a bigger widget-based form. If your game is based on Graphics View, though, you can use `QGraphicsVideoItem` instead:

```
QGraphicsVideoItem *item = new QGraphicsVideoItem();
player->setVideoOutput(item);
```

`QGraphicsVideoItem` inherits `QGraphicsObject`, which inherits `QGraphicsItem`, so you can use `QGraphicsVideoItem` just like any other item in Graphics View. Also, as we did with `QVideoWidget`, we set the instance of `QGraphicsVideoItem` as the player's video output. You do not have to do anything else.

If you want to play a video in a loop or play multiple videos one after the other, you can use `QMediaPlaylist`, just like you have used it for sounds.

Playing videos in QML

The only real type you have to know for playing videos in QML is `Video`:

```
Video {
    id: video
    source: "/path/to/video.ogg"
}
```

With `source`, you define the video file and, with the definition of an ID, you can start the video by calling `video.play()`. `Video` inherits the `Item` type so that it behaves like any other QML item. `Video`, however, is a convenience type that behaves like the combination of `MediaPlayer` and `VideoOutput`:

```
MediaPlayer {
    id: mediaPlayer
    source: "/path/to/video.ogg"
}
VideoOutput {
    source: mediaPlayer
}
```

`MediaPlayer` defines the file to play, and `VideoOutput` is the canvas for displaying the video, similar to `QMediaPlayer` and `QVideoWidget` in the C++ world. Using `source` of `VideoOutput`, you define which player's video should be shown.

Debugging output

A common and simple way of debugging the application is to print some values to the application output and examine them. While this approach certainly has a lot of limitations, it's still convenient in simple cases. You have surely noted that in the previous chapters, we used `qDebug()` quite often. You already know that it prints out messages. However, this is only a small part of the extensive message handling system Qt offers to us. In this section, we'll discuss its usage possibilities in more depth.

Using qDebug() and friends

The easiest possibility to debug an application is to write messages to `std::cout`, `std::cerr`, or `std::clog`, which may help you understand what your code is actually doing. Instead of these streams, you can use the Qt message system that offers more features. You can use the following functions to access Qt message streams:

Function	Purpose
`qDebug()`	Debug messages
`qInfo()`	Informational messages
`qWarning()`	Warnings
`qCritical()`	Critical errors
`qFatal()`	Fatal errors that will terminate your application

All these functions are defined in the `QtGlobal` header file, but to use their full capabilities, you should include the `QtDebug` header.

The simplest way of printing a message is to pass a string literal as the argument of one of the functions listed earlier:

```
int someInt = 5;
if (someInt % 2 == 0) {
    qDebug("Entering scope for even integers.");
} else {
    qDebug("Entering scope for odd integers.");
}
```

If you run this code in Qt Creator, you will see **Entering scope for odd integers** in the **Application Output** panel. In this specific case, we already know that `someInt` is `5`, but what if `someInt` was an argument of a function? We may then want to know for which value the debug message was printed, so we can extend the example code with the following line:

```
qDebug("Entering scope for odd integers: someInt = %d", someInt);
```

In this case, the function works similar to the `printf()` C function. Now we will see the actual value of `someInt` in the output. Even if this is a practical solution for such a short message with only one argument, this will quickly become hard to read for complex messages with multiple arguments. A much more preferred syntax is available once you include the `QtDebug` header:

```
#include <QtDebug>
//...
qDebug() << "Entering scope for odd integers: someInt ="
         << someInt;
```

This will produce the output of `Entering scope for odd integers: someInt = 5`. If you take a close look, you will note that a space was inserted after the equal sign. This is because every `<<` expands to a space. After a complete instruction, a new line is added automatically, so the following code will result in two lines on the output panel:

```
qDebug() << "Line 1";
qDebug() << "Line 2";
```

The `QLoggingCategory` class allows you to define categories for your messages. To use a category object, you need to pass it to `qCDebug()`, `qCInfo()`, `qCWarning()`, or `qCCritical()` functions as the first argument instead of using regular functions such as `qDebug()`.

You may ask yourself, however, if that is all there is to it. It is nice, but what's the real advantage? The advantage comes with Qt's container classes and types. Imagine that you want to debug the content of a list or a hash. To do that, you would have to loop through their elements and print each single element. If you use the `<<` operator instead, Qt will do the work for you:

```
QList<qreal> list;
list << 1.0 << 1.2 << 1.4;
qDebug() << list;
QHash<QString, int> hash;
hash.insert("Apples", 1);
```

```
hash.insert("Bananas", 3);
qDebug() << hash;
```

The output will then look well formatted, like this: `(1, 1.2, 1.4)` and `QHash(("Bananas", 3)("Apples", 1))`.

It is likely that you also want to know what location the messages are coming from. This can be done in two ways. The easiest is to use the `Q_FUNC_INFO` macro. A call such as `qDebug() << Q_FUNC_INFO;` inside a `someFunction()` function of the `SomeClass` class will result in `SomeClass::someFunction()`. Even if that should be precise enough to locate the corresponding source code in your game, you also can use the `QT_MESSAGE_PATTERN` environment variable to customize the output. For this variable, different placeholders are defined:

Placeholder	Description
`%{appname}`	Name of the application. The same value as `QCoreApplication::applicationName()` returns it.
`%{category}`	The message category specified via `QLoggingCategory`.
`%{file}`	The name of the source file, including the path relative to the path of the executable.
`%{function}`	Name of the function.
`%{line}`	Line in the source file.
`%{message}`	The actual message.
`%{pid}`	Current process identifier of the application. The same value as `QCoreApplication::applicationPid()` returns it.
`%{threadid}`	Current thread identifier.
`%{type}`	The type of handler, for example, `debug`, `warning`, `critical`, or `fatal`.

If you define `QT_MESSAGE_PATTERN` as `%{message} (%{pid})`, the process identifier, enclosed in braces, will be appended after the actual message. Since the process identifier may be important for debug messages, but not for warning messages, we can specify the output even closer. Everything between `%{if-debug}` and `%{endif}` will only appear if you use `qDebug()`, so we change `QT_MESSAGE_PATTERN` to `%{message}%{if-debug} (%{pid})%{endif}`, and the process identifier will only occur at the end of debug messages. Of course, there are also the conditionals `%{if-warning}`, `%{if-critical}`, and `%{if-fatal}`.

If you want to alter the message pattern during runtime, you can use the `qSetMessagePattern()` function instead of the environment variable:

```
qDebug() << "foo";
qSetMessagePattern("%{message} (%{pid}; %{threadid})");
qDebug() << "bar";
```

This will result in the following output:

```
foo
bar (18357; 0x207a9b0)
```

Time for action – Defining your own message format

The information where exactly a message came from can be very useful, especially for large projects, so try to define `QT_MESSAGE_PATTERN` that fits your needs. You can add this variable locally to each project using Qt Creator's great opportunity to define variables for the run environment or set it globally on your development machine. If you want to use Qt Creator, switch to the **Projects** pane, go to the current kit's **Run** section, and under **Run Environment**, you'll find a nice editor for altering, removing, and adding environment variables that are set before Qt Creator launches the application.

Time for action – Using qDebug() with custom classes

You can already use `qDebug()` with lists, hashes, and almost every other important Qt class. However, you can also support debugging output for your own custom classes. Suppose we have this simple class:

```
class SimpleClass
{
public:
    SimpleClass(int loc) : local(loc) {}
    int local;
};
```

If we would now use `qDebug()` to debug this class, it would result in an error such as `no match for 'operator<<' (operand types are 'QDebug' and 'SimpleClass')`. In order to get it to work, all we have to do is overload the `operator<<` of `QDebug`:

```
QDebug operator<<(QDebug dbg, const SimpleClass &sc)
{
    dbg.nospace() << "(SimpleClass: local = " << sc.local << ")";
    return dbg.space();
}
```

Now, if we write `qDebug() << SimpleClass(5);`, we would get the `(SimpleClass: local = 5)` debug message.

What just happened?

How is that possible? Well, since we overloaded the stream operator of `QDebug`, Qt can now use it and write information about `SimpleClass`. In the operator, we get the debug stream and a constant reference to the object that should be printed out. So we can use `dbg` like we would work with `qDebug()`, for example. The `QDebug::nospace()` function prevents a space from being inserted by the `operator<<`, whereas `QDebug::space()` does the exact opposite.

This solution, however, will fail if you like to print private members, since only class members can access private members, and you can't define this conversion operator as a member of your class. An easy solution would be to declare the operator as a friend by adding `friend QDebug operator<< (QDebug dbg, const SimpleClass &sc);` to `SimpleClass`. A cleaner way is to create a dedicated public method for the output operation and use it in the implementation of the operator:

```
class SimpleClass {
public:
    SimpleClass(int loc) : local(loc) {}
    QDebug toDebug(QDebug dbg) const {
        dbg.nospace() << "(SimpleClass: local = " << local << ")";
        return dbg.space();
    }
private:
    int local;
};
QDebug operator<<(QDebug dbg, const SimpleClass &sc) {
    return sc.toDebug(dbg);
}
```

If you have overloaded the stream operator for a custom class, you can use it for `qDebug()`, `qInfo()`, `qWarning()`, and `qCritical()`. This is because each function returns a `QDebug` object that contains information about the requested logging channel. Only `qFatal()` can't make use of the stream operator.

Depending on how excessive you are using `qDebug()` and `qWarning()`, it can become quite noisy. To turn all debug and warning messages off, you can define `QT_NO_DEBUG_OUTPUT` and/or `QT_NO_WARNING_OUTPUT` in the project file. Thus, all messages will be suppressed.

Time for action – Redirecting the stream of QDebug

By default, messages are printed to `stderr` or sent to the debugger, depending on what platform you are on. With `qInstallMessageHandler()`, however, you can yourself intervene and define what should be done with the messages. Therefore, you first have to define your own message handler of the `QtMessageHandler` type, which is defined as `void myMessageHandler(QtMsgType, const QMessageLogContext &, const QString &);`. For an example, let's put all the messages in a file:

```
void myMessageHandler(QtMsgType type, const QMessageLogContext
    &context, const QString &message)
{
    Q_UNUSED(context)
    QString text;
    text += "[" + QDateTime::currentDateTime().toString() + "] ";
    switch (type)
    {
    case QtDebugMsg:
        text += "Debug: ";
        break;
    // ...
    }
    text += message;
    QFile log("myApp.log");
    log.open(QIODevice::WriteOnly | QIODevice::Append);
    QTextStream stream(&log);
    stream << text << "\n";
}
```

What just happened?

First, we define `myMessageHandler` of the `QtMessageHandler` type. Then, inside that function, we create a `QString` called `text`. You can construct this string however you want. In the example, we wrote the time, the type of the error, and the actual message to the output string. Then, we open a file where we specified `QIODevice::Append` so that the text is appended to the content of the file rather than replacing it. Finally, we write the text to the file.

To open and close a file each time, `qWarning()`, `qDebug()`, and so on is called, which is not an ideal solution. In real code, open the file only once and close it when leaving the application. If you want to avoid the temporarily created string, you can stream the information directly to the file.

Lastly, we only have to tell Qt to use our message handler. Ideally, this is done before `QApplication` is constructed so that possible warnings of `QApplication` are also written to the file. So, at the first line of the `main()` function, we write the following:

```
qInstallMessageHandler(myMessageHandler);
```

If we call `qDebug() << SimpleClass(5);` again, `[Di. Nov. 19 21:10:30 2013] Warning: (SimpleClass: local = 4)` will be appended to the file called `myApp.log`.

In the example, we did not have to use `QMessageLogContext`, but you might want to use it since it holds information about the path of the file, the function's name, and the line number.

Have a go hero – Redirecting the messages to QTextEdit

You have seen how to redirect the messages to a file; now it might be handy to see the messages beside your game in a widget. So have a try and alter the previous example to show the messages in a text edit.

Using a debugger

The `qDebug()` is nice for a first approach, but it cannot replace a real debugger such as GDB or CDB. If you can't figure out the problem in a short period of time, better switch to a real debugger. Qt Creator offers you plenty of options for efficiently using a debugger. It supports GDB, LLDB, and CDB, so you can use at least one of them under Windows, Linux, and macOS. Help on how to install the debugger on your platform can be found in Qt Creator's documentation under the **Setting Up Debugger** section.

Time for action – Debugging a sample code block

To get an impression of the integration in Qt Creator, let's try to debug the `MyScene::movePlayer()` function of our elephant game from `Chapter 5`, *Animations in Graphics View*. This function does the logic for the parallel scrolling via a calculation of medium complexity.

The code of interest is listed as follows, and we want to check if, and when, the highlighted code is executed:

```
const int visiblePlayerPos = m_currentX - m_worldShift;
const int newWorldShiftRight = visiblePlayerPos - rightShiftBorder;
if (newWorldShiftRight > 0) {
    m_worldShift += newWorldShiftRight;
}
const int newWorldShiftLeft = shiftBorder - visiblePlayerPos;
if (newWorldShiftLeft > 0) {
    m_worldShift -= newWorldShiftLeft;
}
```

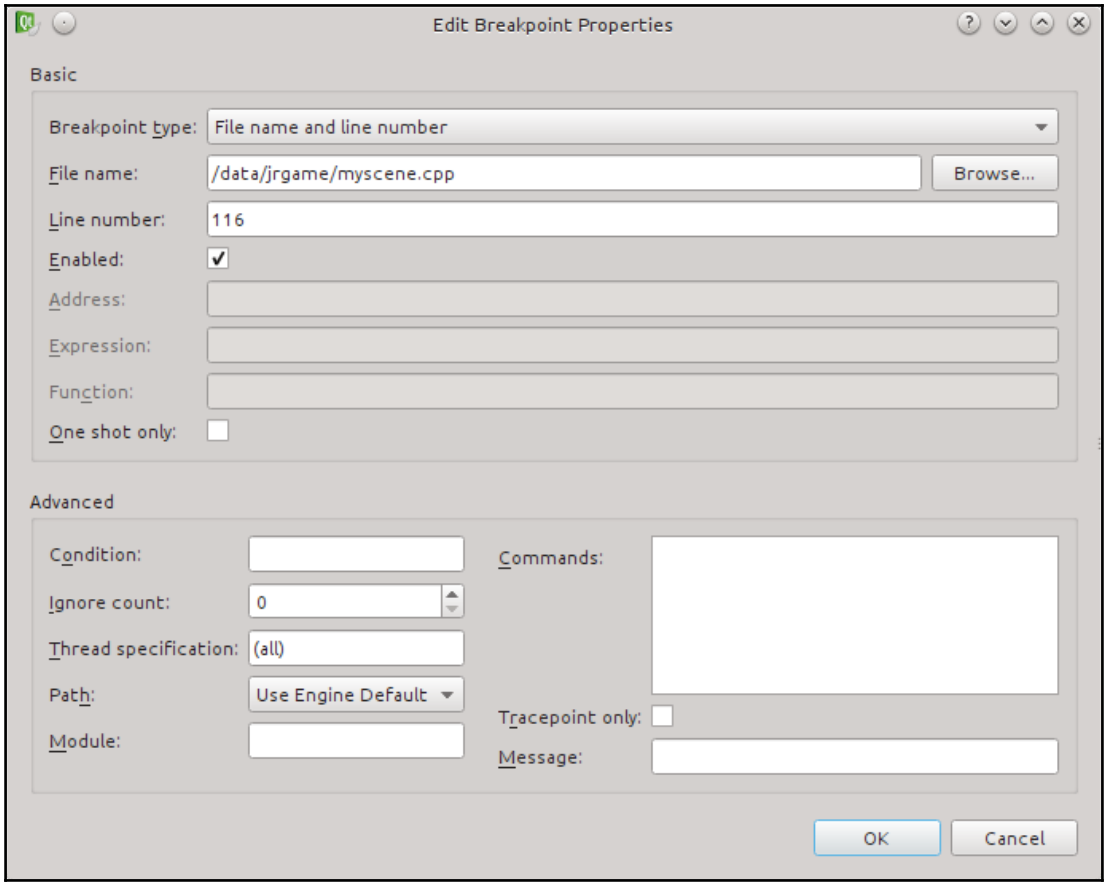
So we need to set a break point to cause a halt of execution at this point. The break point can be set by either clicking to the left of the line number or by pressing *F9* while the cursor is on that line.

Instead of *F9*, you can, of course, define a custom shortcut. If you want to do this, take a look at **Tools - Options - Environment - Keyboard**.

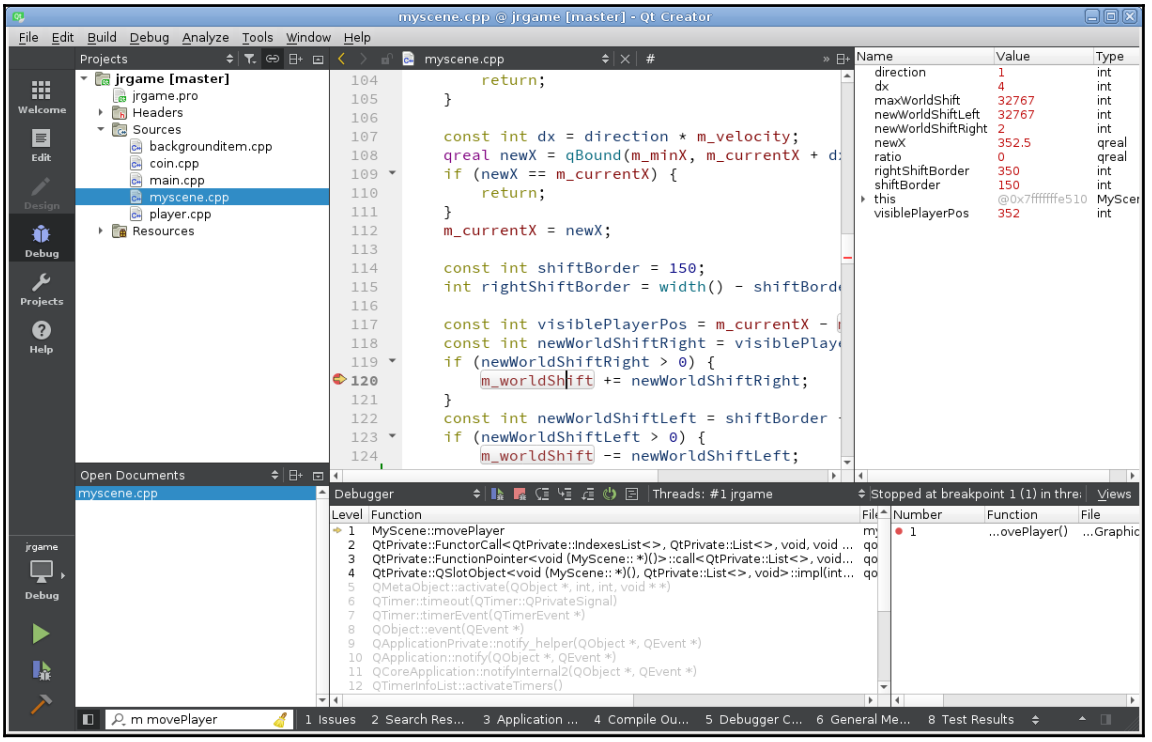
After you have toggled the break point, a red dot with an hourglass will show up:

```
119     if (newWorldShiftRight > 0) {
120         m_worldShift += newWorldShiftRight;
121     }
```

If you change the creator's edit mode to the debug mode (*Ctrl* + *4*), you can find a view called **Breakpoints**, which lists all break points you have set so far. If you wish to alter a break point, for example, to change the line number or its type, right-click on either the icon next to the line number or the list item on the view and choose **Edit breakpoint**. The following dialog will pop up and allow you to easily edit this break point:



Now start debugging the game by pressing *F5*. The game will be started, and the debugger will attach to the game's process. As soon as the player approaches the right-hand border of the window, Qt Creator will be activated and show that the application was stopped on a break point:



What just happened?

As you triggered the camera movement, line 120 of the `MyScene::movePlayer()` function was reached. Thus, the break point causes the execution to halt. In debug mode, you then have different views holding various pieces of information. We will focus on the most important two views: the **Stack** and **Locals and Expressions** views. Under **Windows - Views**, you can display other views as you wish.

In the bottom left-hand corner of the picture, you see the so called **Stack** view. There, as soon as the game is interrupted, you will find all nested function calls that lead to the current state. This is called a call stack trace. The first element of the list contains a `MyScene::movePlayer` function. This was expected, since we set the break point there. The next line indicates the function that was executed before and the function that called `MyScene::movePlayer()`. If you double-click on that line, Qt Creator opens the corresponding file at the given line number. This way, you can easily and quickly go through the sources that lead to the function call that you were interested in.

Second, take a look at the top-right. Here, you'll see the **Locals and Expressions** view. It outlines information about local and member variables. There, you can see that, for example, the `rightShiftBorder` local variable has a value of `350`. The value of `direction` is `1`, as you can see from the first entry. You can also see that this variable is an integer type. On the first line of the function, we have defined this variable as `int direction = m_player->direction();`. So if you want to double-check—assuming that you know that `Player::direction()` returns the value of the private member called `m_direction`—you can do that too. Just expand the section of the tree labeled `this` and then expand `m_player`. As you see, you can get a pretty clear picture of the value of almost every variable of your game.

If this information already satisfies you, hit *F5* to continue the execution. If you want to know what happens next, you can **step over**, **step into**, or **step out** of the function. This way, you can keep going step by step and investigate how your game evolves. You can use the buttons right above the **Stack** view for navigation inside the code.

Have a go hero – Familiarizing yourself with the debugger

Go ahead and make yourself familiar with the debugger and also take a look at the other views and their information. The better you know the debugger environment, the faster you can hunt down any causes of erroneous behavior.

Wait, however, you may ask yourself how to debug a Qt Quick application. The answer is pretty easy—just as you would debug a C++ application. You only have to ensure that in Qt Creator's run settings, the checkbox for QML debugging in the **Debugger Settings** page is enabled. If so, simply set your break points and start debugging with *F5*. That's all. A great feature when debugging QML applications is that you can change properties on the fly. To do that, either open the QML/JS Console and type in JavaScript commands to alter variables, or start editing variables in the **Locals and Expressions** tab by simply double-clicking on them.

If you use C++ as well as QML in your application, ensure that C++ debugging is also enabled in the **Debugger Settings** page so that you can debug both components without any limitation.

The Qt Creator Manual in general is very good and self-explanatory. In its documentation, you will also find a chapter about debugging C++ and QML applications. Take a look at the **Qt Creator Manual** documentation page.

Testing

A very important and crucial point during software development is testing the application. You have to ensure and check that your game is working as you—and your users—expect it to. Software testing is, of course, a wide area with different topics and different view points. In our case, we want to focus on two aspects: assertions and unit testing.

Testing assertions

During development, it is useful to also test pre- and post-conditions or—in a more general sense—to test the assumptions you have made. Therefore, Qt offers three macros to help you with that.

If you have to check simple conditions, use `Q_ASSERT`. To check at a certain point whether the `count` variable equals `5`, you would write the following:

```
int count = 4;
Q_ASSERT(count == 5);
```

Since `count` isn't `5` in this example, `Q_ASSERT` will use `qFatal()` to print an error message such as `ASSERT: "count == 5" in file ../Debugging/main.cpp, line 12` and will then abort.

To customize and pass additional information to the error message generated by `Q_ASSERT`, you can use `Q_ASSERT_X` instead:

```
Q_ASSERT_X(count == 5, "Core Computation", "List size is not 5");
```

If the condition is `false`, the last two arguments are used to form the error message, which, in the example, would look like `ASSERT failure in Core Computation: "List size is not 5", file ../Debugging/main.cpp, line 12`. If the test condition is not met, `Q_ASSERT_X` will also abort the execution of the program.

The `Q_CHECK_PTR` convenience macro is designed for testing whether a pointer is null:

```
char *c = 0;
Q_CHECK_PTR(c);
```

This code will throw the `std::bad_alloc` exception because the passed pointer is null.

All three macros are useful to spot errors during the development, but they have no effect on the release mode. This is because, during compilation in release mode, the `QT_NO_DEBUG` variable is set, preventing all the checks from being performed. Thus, the program will continue to run even if the checked conditions fail. Note that any computations done directly in a macro invocation will not be executed in the release mode, so you don't need to worry about the performance impact of your assertions.

Unit testing with Qt Test

Qt Test is a simple unit testing framework designed for easy use in Qt applications. Here's how it works. First, you have to create a class that inherits `QObject`. Next, you have to define at least one private slot; each slot is later treated as a unit test. If you add more than one slot, they get invoked in the order in which you have declared them. Finally, create an object of your class and pass it to the `QTest::qExec()` function. Qt Test then executes all the tests and prints the result to `stderr` by default.

Even though you can use the Qt Test framework to test the GUI of your application or your game, that is not its strength as it only offers rudimentary tools to do so. So you might want to use another tool for GUI testing. For unit testing, however, Qt Test is just perfect.

Time for action – Writing a first unit test

So let's make our first, minimalistic unit test. Create a new project using the **Qt Unit Test** template from the **Other Project** section. After selecting project name, location, and kits, you'll see a few specialized pages of the wizard. First, you can select the Qt modules that you'll need. Next, you'll be able to configure the generated test class. Input `FirstTest` as the class name and leave the rest with the default values.

The generated project will contain a single file called `tst_firsttest.cpp` with the following content:

```
#include <QString>
#include <QtTest>
class FirstTest : public QObject {
  Q_OBJECT
public:
    FirstTest();
private Q_SLOTS:
    void testCase1();
};
FirstTest::FirstTest()
{
}
void FirstTest::testCase1() {
  QVERIFY2(true, "Failure");
}
QTEST_APPLESS_MAIN(FirstTest)
#include "tst_firsttest.moc"
```

There is not even a `main.cpp` file! The `QTEST_APPLESS_MAIN` macro generates a `main()` function for you.

If you run this example, you'll get the given output:

```
********* Start testing of FirstTest *********
Config: Using QtTest library 5.10.1, Qt 5.10.1 (x86_64-little_endian-lp64
shared (dynamic) release build; by GCC 5.3.1 20160406 (Red Hat 5.3.1-6))
PASS : FirstTest::initTestCase()
PASS : FirstTest::testCase1()
PASS : FirstTest::cleanupTestCase()
Totals: 3 passed, 0 failed, 0 skipped, 0 blacklisted, 0ms
********* Finished testing of FirstTest *********
```

What just happened?

Qt Creator created a `QObject`-derived class called `FirstTest` and added a private slot named `testCase1()` to it. The class is passed to the `QTEST_APPLESS_MAIN` macro, so Qt will use it as the source of test cases. Each slot of the class (with a few exceptions that we'll discuss as we move on) is considered to be a test case and will be executed. You'll find the information about failed and succeeded tests in the output, along with the information about the current Qt version. However, what about `initTestCase()` and `cleanupTestCase()`? We have not declared these slots.

Well, before Qt starts the test on a class, it tries to invoke a slot called `initTestCase()` if it exists. When all tests on a class are finished, the test framework will call `cleanupTestCase()`. You can use these two functions to initialize and clean up the entire test. They are comparable with a constructor and a destructor. To initialize or to clean up before and after a single test, Qt will automatically call the `init()` and `cleanup()` functions. So if you want to use them, just define—and implement—these four functions as slots. Qt will do the rest.

There are three variations of the macro that generates the `main()` function:

- `QTEST_MAIN` will create a `QApplication`, `QGuiApplication`, or `QCoreApplication` object (depending on the modules enabled in the project file) before running tests
- `QTEST_GUILESS_MAIN` will create a `QCoreApplication` object before running tests
- `QTEST_APPLESS_MAIN` won't create an application object and will only run tests

Since we used the `Q_OBJECT` macro in a `.cpp` file (`tst_simpletest.cpp`), we have to include the `.moc` file by hand. This way, **qmake** will generate the instructions required for the meta-object compiler.

Time for action – Writing a real unit test

So far, our test case was empty; let's change that and add a real test. In general, you can do whatever you want inside a test slot. There are no special limitations. At one point, however, you have to check whether a condition is met. For this task, the following macros come in handy.

The most brutal way to force a test to fail is to use `QFAIL`, as it stops the execution of the test immediately and prints the error message you can define as a parameter. Consider that you ran a test case such as the following:

```
class FirstTest : public QObject {
//...
private Q_SLOTS:
    void firstTest() {
        QFAIL("No way!");
    }
};
```

You would get as a report (shortened, as the following reports will also be) the following:

```
FAIL! : FirstTest::firstTest() No way!
   Loc: [../unit_test/tst_firsttest.cpp(14)]
Totals: 3 passed, 1 failed, 0 skipped, 0 blacklisted, 0ms
```

The report states that one test failed. Due to the `FAIL!` prefix, you can spot the troublemaker easily. Additionally, the file and the line number are mentioned.

If a test requires special preconditions—for example, the existence of certain modules—but they are not met, you can skip a test with `QSKIP`. Here's an example:

```
void areMathematiciansCrazy() {
    if (1 != 2) {
        QSKIP("Thank goodness! 1 still isn't 2.");
    }
    //...
}
```

The following code results in:

```
SKIP : FirstTest::areMathematiciansCrazy() Thank goodness! 1 still isn't 2.
   Loc: [../unit_test/tst_firsttest.cpp(18)]
Totals: 2 passed, 0 failed, 1 skipped
```

The `QCOMPARE` macro is used to compare two values; the actual to an expected one. Consider this example:

```
QCOMPARE(1 + 1, 11);
```

This code will compare whether *1+1* equals *11*. Note that `QCOMPARE` is very strict on the data types, so both values have to be of the same type. If this test fails, the output informs you about the actual value `2` and what you would have expected `11`. For example, you can see an output such as the following:

```
FAIL! : FirstTest::firstTest() Compared values are not the same
   Actual (1 + 1): 2
   Expected (11) : 11
```

With `QVERIFY`, you can check a condition. If it is `false`, the test will fail and print an error. `QVERIFY(1 == 2);` will cause a log such as `FAIL! : SimpleTest::simpleTest() '1 == 2' returned FALSE`. If you like to add a customized, more descriptive message text, use the `QVERIFY2` macro. The message that can be defined as a second parameter will be printed after the error message, as seen in the preceding code:

```
QVERIFY2(1 == 1, "Well, now mathematicians are crazy.");
```

Lastly, we would like to mention `QTRY_COMPARE_WITH_TIMEOUT` and `QTRY_VERIFY_WITH_TIMEOUT`. Both behave like `QCOMPARE` and `QVERIFY`, except that as a second or third parameter, a timeout can be specified. The macros then compare the values or check the conditions until they become `true` or the timeout is reached. In the latter case, the test will fail. For convenience, there are also `QTRY_COMPARE` and `QTRY_VERIFY`, which have a timeout of 5 seconds defined.

Time for action – Running a unit test that uses a data function

If you have a unit test that needs to test a function with different values, you can separate the testing logic from the data. For this, create a private slot named in a similar manner to the slot of the actual test and add `_data` to it:

```
void test_data() {
    QTest::addColumn<QString>("string");
    QTest::addColumn<int>("count");
    QTest::newRow("simple string") << "Hello" << 5;
    QTest::newRow("with interpunctation") << "world!" << 6;
}
void test() {
    QFETCH(QString, string);
    QFETCH(int, count);
    QCOMPARE(string.count(), count);
}
```

If you run this code, it will result in the following:

```
PASS   : FirstTest::test(simple string)
PASS   : FirstTest::test(with interpunctation)
```

What just happened?

In the `test_data()` function, we declared that we need two values for each test run of `test()` by calling `Qtest::addColumn()` twice. Of course, the number of variables is not limited. Each call of `addColumn()` first defines the type of the new variable, `QString` and `int`, and then the name of this variable, `string` and `count`. With `QTest::newRow()`, we define the values of these variables. The parameter of `newRow()` should be a short descriptive text of what will be tested. Then, the values are defined with the stream operator. As the names of the `addColumn()` and `newRow()` functions suggest, think of these variable definitions as a simple table of test values.

The `test()` function then loops through the rows of this imaginary table. Each time, we fetch the values out of the cells using `QFETCH`. Here, the arguments of `QFETCH` have to match the previous declaration made by `QTest::addColumn()`. After the fetch is done, we can use `string` and `count` as a normal `QString` and integer.

If you take a look at the output, you will spot the use case of the `QTest::newRow()` argument. It is printed after the test name so that you can see which data pair was processed and passed, skipped, or failed.

Useful command-line arguments

To complete this part about the Qt Test framework, let's emphasize on two useful capabilities of the test's command-line options.

Suppose you have defined multiple time-consuming tests and the last one fails. Now, after you have tried to fix the cause of the failure, you have to perform all tests again to notice that the fix does not work. This is quite an unsatisfying workflow. You have to check functions that run over and over again only to test a specific one. Of course, you could move the declaration of that test to the top to force it to be executed at the beginning, but that is not an ideal solution. Fortunately, this issue can be solved with a command-line option.

When you invoke the test program, simply pass the name of the slot you want to execute. It will then be the only test that is performed. For example, if your unit test creates the executable named `unittest.exe` and you would like to test `firstTest()`, you can call `unittest.exe firstTest`. If you like to execute more than one specific test, just add the names of the other tests separated by a space. If you do not pass any specific test at all, all the tests will be performed.

If a test uses the data functionality as described earlier, you can specify the row by adding the row's name right after the slot's name, separated by a colon. Enclose the entire name in quotes whenever the row's name contains spaces, for example, `unittest.exe test:"simple string"`.

If you're using Qt Creator to run your tests, you can edit the command-line arguments of your executable on the **Projects** pane.

The second useful capability comes into play when you work with multiple test classes, each holding a lot of unit tests. Then, you most likely need to process the results in a more efficient way than going through the output in the Qt Creator's application output panel. With the `-o` command-line option, you can define a file in which the results should be stored. This file can then be further processed, for example, by a report tool or something similar. The default format of the file is a simple text file holding the messages as you have seen them in the output panel. Since this format may not be the best for further automatic processing, you can pass `-xml` as an additional command-line argument causing the output to be formatted as XML. Besides `-txt`, the default format, and `-xml`, you can also pass `-lightxml` to get a stream of XML tags, or `-xunitxml` to get an Xunit XML document, to the command line to choose the format of the output.

Pop quiz

Q1. What are the files that must be passed to a `QTranslator` object?

1. UI files (`*.ui`)
2. Translation source files (`*.ts`)
3. Message catalog files (`*.qm`)

Q2. How can you ensure that two `QSharedMemory` objects refer to the same memory buffer?

1. Use the copy constructor to create a new `QSharedMemory` object for the same buffer
2. Pass the same key to both `QSharedMemory` objects
3. All `QSharedMemory` objects created by the same executable share the same buffer

Q3. What stream operator do you have to overload to comfortably use your own class with `qDebug()`?

1. `QDebug& operator<<()`
2. `QDebug& operator+()`
3. `QDebug operator||()`

Q4. What does the `Q_ASSERT(condition)` macro do?

1. It always aborts the execution of the program if the condition is `false`
2. It always aborts the execution of the program if the condition is `true`
3. It aborts the execution of the program if the condition is `false`, but only if the program was built in debug mode

Summary

In the first part of this chapter, you familiarized yourself with a number of frameworks to make your games take advantage or multiprocess and multithreaded architectures and also to reach out to a larger number of end users by making your games talk to them in their own language. The Qt Multimedia framework we presented to you allows you to record and output multimedia content in both the widgets and Qt Quick worlds.

Since testing and debugging are essential parts of software development and should take as much time as inventing and coding a game, we took a look at these too. First, you saw how to use `qDebug()` and how you could redirect its output. In situations where this technique does not help you spot the error, you saw how elegantly Qt Creator integrates a native debugger. With Qt Creator, you can easily set and manage break points as well as inspect the call stack.

At the end, you finally learned how to create unit tests using Qt Test. This way, you can automatically guarantee that your game will function well. With the knowledge about debugging and testing, you will be able to create code of exceptional quality.

Index