QT: INTERNALS AND PRINCIPLES

Get to grips with Qt

Martin Rotter

May 19, 2013

Abstract

Qt is one of the best-known and the mightiest general-purpose libraries available. Its functionality covers each and every thinkable programming area, including threading, graphical interfaces, relational databases, networks, 2D/3D painting and many more. This text aims at one modest task - providing the solid base for learning and understanding Qt by revealing its internals and principles.

Acknowledgements

Who this book is for?

This book is for anyone who is interested in creating dynamic and multi-platform applications using Qt framework. It does not matter if you are experienced software engineer or self-taught enthusiast. Information included in this book can be useful both ways.

What is covered by this book?

Covering all components of Qt framework in one book is impossible task because of massive complexity of these libraries. It's better to focus on certain aspects only. Each Qt-related books begins with graphical interfaces. Probably, that's not the best approach because graphical interfaces represent nontrivial part of any application. You need to be able to manage easy Qt-related tasks first in order be able to master harder ones.

That's why this book starts with fundamental topics, therefore pushing graphical-interfaces-related topics back to further chapters. You learn something about C++programming language, Qt compilation process and Qt framework structure. Meta-object system is discussed too. Understanding meta-object system is one of the main preconditions for building solid Qt-based applications. You learn to use threads to separate logic from user interface.

Finally, newly gained knowledge is used to build applications, which can be easily maintainable, compilable and easy to package and ship to your customers.

This books equips you primarily with principles. Facts (which are unknown to you and are not included in this book) can be found in (Qt-Project, 2012b). Note that in this paper, we discuss relatively new (January, 2013) Qt 5.

What is not covered by this book?

As said earlier, it's not possible to cover all parts of Qt libraries in one book. We will omit some admired Qt features, so that we can concentrate on other ones. Qt Meta Language (QML) will be ignored completely, along with whole QtQuick and other stuff for cell-phones or tablet devices. 2D and 3D painting features won't be described too. Moreover, some other parts of Qt are ignored. You will be informed about some of them throughout the book.

How this book is structured?

As said earlier, there are basically two main stories told by this book. First one lets you know something about Qt, its features and principles. Analogy to this story is called Laboratory Qt and it is the first part of the book.

Second part practically builds on basic Qt knowledge and show the way of complex application construction. This part is called Real-world Qt and it's the second (and more exciting) story.

Are there any preliminaries?

Of course there are. Qt itself is based on the C++programming language and thus C++knowledge is main prerequisite. One could argue that Qt has bindings into many better programming languages and I would respond: "It's true." But C++is core language for Qt and for you, as future Qt developer, using Qt in its native programming language is important.

C++went through massive update recently and we face its eleventh version. So we will use C++11 in this book. You can learn more about C++11 in (Du Toit, 2012) or in section 1.3.

Another prerequisite is basic knowledge of threading terminology.

Text formatting

This book is supplemented with pictures, tables and other fancy elements. There are also source code fragments included as seen in Listing 1.

Listing 1: Sample code fragment

```
1  int main(int argc, char *argv[]) {
2      return EXIT_SUCCESS;
3  }
```

Note that sometimes it is needed to highlight *portion of text* or even make it **really visible**. In some cases, there is a need of providing some extra remark to discussed topic. Typical remark looks similar to one below.

> **One Step Further**
> This is very interesting text here. . .

Source code

Topics of this book are supplemented sample applications to describe the matter. You can find source code in *sources* subdirectory or on `https://github.com/Martin-Rotter/qt-internals-and-principles/tree/master/sources`.

Licensing

Contents

Part I

Laboratory Qt

Chapter 1

Foreword

Qt framework (Qt-Project, 2013b) is one of the greatest libraries ever made. You probably use it and you don't even know about it. If you use Skype (Microsoft, 2013) or K Desktop Environment (KDE) (KDE, 2013), then you use Qt too, because those applications are based on Qt.

Skype uses just graphical interface made in Qt but KDE is totally based on Qt as it uses not just graphical interface from Qt but other components too.

Qt penetrated the world of interactive applications and now it can be found even in devices, where it's not generally expected. First public version of Qt was released in 1995 and huge progress was achieved since that time.

In a matter of time, Qt began to be perceived as very dynamic library which is particularly great for graphical interface design. There was very good reason for such an opinions because KDE was released in 1996, invoking quite a sensation. In short, its desktop environment looked great and overpowered other major environments in this aspect. Qt was pushed forward by those events and became massively popular. The only goal of Qt was to be a good library for anyone who does desktop programming.

As years passed, Qt was more and more robust, KDE made its progress through version 3 and 4, and things have changed. Presently, desktop does not mean everything for application developer. Today cyber-world needs to be interconnected and people want to be mobile. You can't do that with desktop environment running on personal computer. You need cell-phone. Cell-phone with (possibly) good-looking environment and fancy applications. Unfortunately, Qt 4 was not able to offer this kind of functionality to its users – programmers, so they looked at the competition and chose Android as their platform, leaving Qt behind.

Luckily, Qt 5 appeared, bringing us some new exciting features, giving itself a chance to compete its opponents in category of mobile development toolkits. If we add rock-solid desktop features, we have versatile and stable base to build on.

1.1 What is Qt?

As said previously, Qt is framework, toolkit or, simply, set of libraries. It has its very roots in Norway. Original creators are Haavard Nord and Eirik Chambe-Eng. (Wikipedia, 2013c, section History) Basically Qt framework consists of:

- set of libraries written in C++,

- meta-object compiler,

- QtScript interpreter,

- tools for internationalization and Graphical User Interface (GUI) design,

- scripts for various build systems like CMake,

- other tools, e. g. integrated development environment, examples or documentation browser.

So as you see, Qt is not just collection of header/source files. It's completed with a variety of other stuff. You will learn more about Qt structure in Chapter 2.

1.2 Companies behind Qt

Qt lives for more than two decades and its owners changed accordingly. Haavard Nord and Eirik Chambe-Eng assembled themselves in a team and called it Quasar Technologies. Later company was renamed to Trolltech. This company led Qt development for period of 12 years, preffering desktop development.

But as we know, things have changed and smartphones became massively popular in third millennium. That's why Trolltech was acquired by Nokia. It was obvious that Nokia can bring something new to Qt as it is leading company in smartphones world production. Nokia promised that they would keep Qt open-souce and made it available via public Git[1] repository. Nokia somehow was not able to utilize potential of Qt and sold it to another company called Digia. (Wikipedia, 2013c)

---

[1]Git is revision control system originally created to support Linux kernel development. Founding author is well-know Linus Torvalds. Git is multi-platform and is available for Windows, Linux or Mac OS X. It's Portable Operating System Interface (POSIX)-compatible.

1.2.1 Licensing

Qt uses two separate licenses:

1. **Commercial license**, which provides you (as indie developer) with opportunity to produce *closed-source* (proprietary) or *open-source* applications, you can do whatever you want with your copy of Qt. This kind of license is usually sold per particular platform and it is generally rather expensive. This license is usually bought by developers who want to sell their software and/or keep it closed-source, otherwise open-source license is better choice.

   Commercial license grants you even more rights. You can link Qt statically to your application and/or include other proprietary software in it. Technical support is available for commercial users only.

2. **Open-source license**, which provides you (and your users) with much more freedom but forcing you to share source code of your application with the community and allowing anyone to change your application and redistribute it under the same terms. Used license is GNU LGPL license, in version 2.1, and GNU GPL (Stallman, 2007) for your projects.

Licenses have always been quite a problem to Qt framework. Commercial license was fine. However, non-commercial was not. Qt used its own license before GNU LGPL and GNU GPL were chosen as primary ones. Problem was that Q Public License wasn't GPL compatible. This problem became much more obvious when KDE established itself as one the most favored desktop environments, gaining milions of users. They were naturally afraid of KDE becoming the piece of proprietary software, which was more or less possible with Q Public License. Luckily this problem was solved by releasing Qt under GNU GPL.

1.3 C plus plus as base stone

C++ is known as general-purpose programming language, based on C. It was created around 1979 by Bjarne Stroustrup, bringing in many Object-oriented programming (OOP) features such as implementation of classes, polymorphism, entity overloading or inheritance. You can find brief example of basic techniques in Listing 2.

Listing 2: Basic OOP techniques in C++

```
1  /* Base class declaration */
2  class BaseClass {
3      public:
4          BaseClass() {
```

```
 5              cout << "BaseClass␣instance␣constructed." << endl;
 6          }
 7
 8          void BaseClass::whoAmI() const {
 9              cout << "I␣am␣BaseClass." << endl;
10          }
11  };
12
13  /*
14   * Class declaration
15   * This class inherits BaseClass.
16   */
17  class InheritingClass : public BaseClass {
18      public:
19          InheritingClass() : BaseClass() {
20              cout << "InheritingClass␣instance␣constructed." << endl;
21          }
22
23          void InheritingClass::whoAmI() const {
24              cout << "I␣am␣InheritingClass." << endl;
25          }
26  };
27
28  /* Usage of BaseClass and InheritingClass classes. */
29  int main() {
30      BaseClass class_1;
31      InheritingClass class_2;
32      class_1.whoAmI();
33      class_2.whoAmI();
34
35      BaseClass *class_3 = &class_2;
36      class_3->whoAmI();
37
38      ((InheritingClass*) class_3)->whoAmI();
39
40      return 0;
41  }
```

Listing 3: Output of application from Listing 2

```
1  BaseClass instance constructed.
2  BaseClass instance constructed.
3  InheritingClass instance constructed.
4  I am BaseClass.
5  I am InheritingClass.
6  I am BaseClass.
7  I am InheritingClass.
```

C++ has many characteristics – some are bad while other ones may be great.

SYNTAX[bad]

C++ is known to have some oddities rooted in its syntax, e. g. we can be confused by rife usages of `const` keyword. One `const` marks methods which can operate only with constant objects and another distinguishes constant variables from non-constant ones. Even the greatest fan of C++ has to admit uncomfortable usage of this keyword. You can read about this topic in (Prata, 2011, p. 90–92, p. 537).

POINTERS vs. REFERENCES[bad]

This could be one of conventions-related issues. Programmers are not entirely sure whether to use pointers or references for passing values to functions. Generally, terms of references and pointers usage are not strictly set.

MEMORY MANAGEMENT[bad, good]

This is very discussed topic these years as many programmers transitioned to programming languages which produce *managed code* (see explanation below). Nowadays programmers heavily depend on managed code and they have troubles with manual object deletion and other related actions.

> **One Step Further**
> Term *managed code* means that all resources (usually called *objects* in the object-oriented programming) generated by code execution are maintained and managed by an external entity. This entity is often called *a virtual machine* and usually includes sophisticated garbage collector, which is responsible for freeing needless resources from memory.

C++ is considered to be a fairly low-level programming language. Its "*low-levelness*" applies to the way the memory is managed. In this case, no automatic memory management is implemented, yielding responsibility to the programmer. He has to take care of memory allocation and deallocation. There is certainly quite big pronenes to errors in this approach. Programmers simply forgets to free allocated memory space and memory leak occurs.

On the other, manual management of allocated objects gives programmer bigger power to control application memory usage and that's perfect on devices with limited system memory. Manual control of object life can be also much faster than automatic resource management provided by *garbage collectors*.

Neither virtual machine nor complex runtime environment supports execution of C++ application, thus "nobody" supervises actions of your application,

except operating system. Your application is left alone with its segment of primary memory and your application is entrusted with everything, including memory management.

THREADING[bad]

C++ doesn't contain unified interface for threading.[2] That could make pure C++ poorly usable for developing more complex applications if no 3$^{\text{rd}}$-party threading library is available.

FAST CODE EXECUTION[great]

C++ code execution is amazingly fast compared to other modern programming languages. Direct compilation (see more in Chapter 4) into machine code is the cause here. Other favorite languages are compiled into bytecode, thus they have to be compiled just-in-time by virtual machine and that is time consuming job, thus making application execution slow.

Let's make a little test and compare C++ with C#. C# code is known to be compiled into Intermediate Language (IL), which is bytecode, and ran by special runtime.

One of the simplest tasks to compare these two languages could be simple integer array sorting. Quicksort algorithm will do that. Consider implementations in C++ (Listing 4) and C# (Listing 5). Furthermore, we can use try to maximally optimize C# code execution speed by allowing "unsafe code" and using pointers instead of references. This approach is shown in Listing 6.

Series of sample sortings was made with each implementation. Subject of sorting was array filled with descending integers. Such an array can be denoted as $Array = \{x, x - 1, x - 2, \ldots, 0\}$. Series contains 20 these arrays. Results of comparison are displayed in Figure 1.1.

Listing 4: Quicksort implementation in C++

```cpp
void QuickSort::quickSort(int *array, int p, int r) {
    int q;
    if (p < r) {
        q = partition(array, p, r);
        quickSort(array, p, q - 1);
        quickSort(array, q + 1, r);
    }
}

int QuickSort::partition(int *array, int p, int r) {
    int x = array[r];
```

---

[2]Threading is supported in new C++ 11 standard. You can read about threading inclusion in (Du Toit, 2012, p. 1114-1160).

```
12      int i = p - 1;
13      int j;
14      for (j = p; j < r; j++) {
15          if (array[j] <= x) {
16              i += 1;
17              swap(&array[i], &array[j]);
18          }
19      }
20      swap(&array[i + 1], &array[r]);
21      return i + 1;
22  }
23
24  void QuickSort::swap(int *lhs, int *rhs) {
25      int temp = *lhs;
26      *lhs = *rhs;
27      *rhs = temp;
28  }
```

Listing 5: Quicksort implementation in C#

```
1   static void quickSort(int[] array, int p, int r) {
2       int q;
3       if (p < r) {
4           q = partition(array, p, r);
5           quickSort(array, p, q - 1);
6           quickSort(array, q + 1, r);
7       }
8   }
9
10  static int partition(int[] array, int p, int r) {
11      int x = array[r];
12      int i = p - 1;
13      int j;
14      for (j = p; j < r; j++) {
15          if (array[j] <= x) {
16              i += 1;
17              swap(ref array[i], ref array[j]);
18          }
19      }
20      swap(ref array[i + 1], ref array[r]);
21      return i + 1;
22  }
23
24  static void swap(ref int lhs, ref int rhs) {
25      int temp = lhs;
26      lhs = rhs;
27      rhs = temp;
28  }
```

Listing 6: Quicksort implementation in "unsafe" C#

```
1  static unsafe void quickSort(int* array, int p, int r) {
2      int q;
3      if (p < r) {
4          q = partition(array, p, r);
5          quickSort(array, p, q - 1);
6          quickSort(array, q + 1, r);
7      }
8  }
9
10 static unsafe int partition(int* array, int p, int r) {
11     int x = array[r];
12     int i = p - 1;
13     int j;
14     for (j = p; j < r; j++) {
15         if (array[j] <= x) {
16             i += 1;
17             swap(&array[i], &array[j]);
18         }
19     }
20     swap(&array[i + 1], &array[r]);
21     return i + 1;
22 }
23
24 static unsafe void swap(int* lhs, int* rhs) {
25     int* temp = lhs;
26     lhs = rhs;
27     rhs = temp;
28 }
```

We can see that C++ outperformed classic C# implementation, while being aproximately 3 times faster. Even "unsafe" C# implementation got beaten, although the difference was tiny. So we can state that C++ is faster than C# even in fairly simple task. You may think about performance difference if hugely complex computation (e.g. rendering of 3D scene) is needed to be done.

HUGE COMMUNITYgreat

Plenty of world-renowned software is written using C++, including many 3D games, almost all programs from Adobe as well as Chromium web browser. Many C++ books are available, which makes it easier to learn.
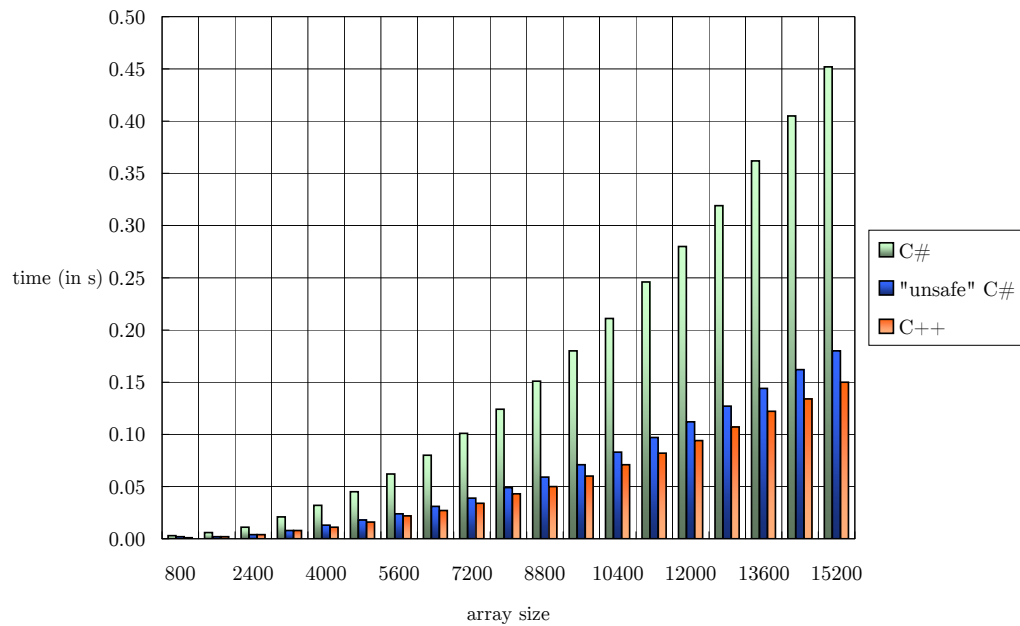
Figure 1.1: C++ vs. C# comparison – Quicksort algorithm

MEMORY CONSUMPTION^{good}

C++ applications, as stated, need no virtual machine for their execution. They just load standard C++ library and extra libraries if needed. Such approach is significantly different from robust and memory greedy runtime environments of some high-level languages. We can mention particularly .NET Framework and Java Runtime Environment (JRE).

CODE PORTABILITY^{bad}

When it comes to *code portability* (which is just another term for *multi-platformity*), C++ leaves its users uncertain. They can be sure about portability of C++ standard library but that's all. Standard library is not trully packed with stunning features, forcing you to use 3rd-party libraries for advanced functionality. Those libraries don't have to be multi-platform, which can be problematic if you want to port your application to another platform.

MISSING CONSTRUCTS^{bad}

Indeed, C++ might be missing some very useful language constructs, which are quite common in other (e.g. functional, logic or even declarative) programming paradigms. Many features emerged in C++ 11 revision, however.

1.3.1 Version 11 and its enhancements

C++ programming language was standardized in 1998. This version is known as C++ 98 and if someone talks about C++, he probably has this version in mind. Programming was changing in time. So, C++ had to change to catch new trends and demands of its users.

   C++ 11 brought many new features, eliminating some of its annoyances. You can read about C++ 11 in (Du Toit, 2012) or go through its finest new properties right now. It is recommended to know something about C++ 11 because Qt 5 uses its enhancements on some compilers.

1.3.1.1 Basic C plus plus 11 information

C++ 11 is source code compatible with C and with C++ 98. It means that valid C (or C++ 98) source code is valid C++ 11 code too. Inprovements in C++ were done in two categories: language core and standard library.

1.3.1.1.1 Language core improvements

Syntax of C++ was always considered to be bad, which is partially true, because C++ offers huge collection of syntactical constructs and sugar, compared to other well-known programming languages. Moreover, C++ 11 adds new language constructs.

Compile time constants

In C++ 98, you cannot write this piece of code:

```
int happy_number() {
    return 7;
}
int *array = new array[10];
array[happy_number()] = happy_number();
```

In this case, compilation ends with error saying: "Function `happy_number()` is not a constant expression." But you think it is. It returns 7 everytime it's called, so it actually is constant expression. That's true but compiler is not aware of it. Keyword `constexpr` tells the compiler to regard `happy_number()` as constant expression, resulting in this code:

```
constexpr int happy_number() {
    return 7;
}
int *array = new array[10];
array[happy_number()] = happy_number();
```

22

Initializer lists

Consider the custom class which encapsulates `std::list` and perhaps adds some functionality:

```cpp
class CustomList {
    private:
        std::list<int> m_list;

    public:
        CustomList() { }

        void insert(int i) {
            m_list.push_back(i);
        }
};
```

Such an implementation allows you to instantiate empty `CustomList` and fill it with values one by one via `insert(int i)` method. What if you know all values in compile time? In older C++ you would have to insert all values one by one.

It would be nice if one could use better syntax for constructing `CustomList` instances, array brace initializer syntax would be great. Its example usage looks like this: `CustomList my_list_instance = {1, 2, 3, 4, 5, 6, 7};`.

C++ 11 allows you to use this syntax via initializer list:

Listing 7: Initializer list usage

```cpp
class CustomList {
    private:
        std::list<int> m_list;

    public:
        CustomList() { }
        CustomList(std::initializer_list<int> values) {
            for (int &value : values) {
                m_list.push_back(value);
            }
        }

        void insert(int i) {
            m_list.push_back(i);
        }
};

// Creating CustomList instance and filling it with values.
CustomList my_list_instance = {1, 2, 3, 4, 5, 6, 7};
```

Clever for-loops

Careful reader certainly noticed strange notation of for-loop in Listing 7 on line 8. This new for-loop syntax is known as *range-based for-loop*.[3] It's just syntactical sugar. This loop works for all containers in standard library as well as for classic C-style arrays. Furthermore, all custom containers defining its iterators are supported.

Type deduction

C++ is statically typed language. So, programmers have to know and mark the type of each and every variable you declare. You basically write:

```
int variable_1 = 15;
std:list<int> *variable_2 = new std::list<int>();
```

or something similar. C++ 11 allows you to omit type of variable with the `auto` keyword:

```
auto variable_1 = 15;
auto *variable_2 = new std::list<int>();
```

Compiler deduces type of each "automatic" variable during compilation. This feature is useful when that particular type is hard to write.[4] You can use `auto` in every thinkable situation as compiler does type checking anyway. Automatic type deduction works for pointer types too.

Lambda expressions

Lambda expressions were the most expected feature for C++ 11. Known from functional languages (e. g. Common Lisp, Scheme), they rapidly found their way into object-oriented programming. Lambda expressions are basically function objects. They can have input parameters and return values.

Lambdas are functions which are defined within another function, thus having no identifier. Typical lambda expression looks like this:

```
[] (int input_1, int input_2) -> int {
    return input_1 * input_2;
}
```

Tricky thing is that lambda expression is able to use variables from the "outside" of its body. Lambdas can be assigned to automatic variable and user can even decide if he wants to allocate lambda expression on the stack or on the heap:

---

[3]This kind of for-loop is available in Qt too, as we will see later.

[4]C++ programmers used to use `typedef` to "clone" types and assign shorter names to them.

```
1  auto twice_function_stack = [] (double input_1) -> double {
2      return input_1 * double;
3  };
4
5  auto twice_function_heap = new auto ([] (double input_1) -> double {
6      return input_1 * double;
7  });
```

Lambda expression can be used as function parameter too. Very simple implementation of in-place map function can look like the one in Listing 8.

Listing 8: Lamda expression as function parameter

```
1   #include <iostream>
2   #include <functional>
3   #include <list>
4
5
6   // In-place map function.
7   // Executes func for each member of input_list
8   void map(const std::function<void(double&)> &func, std::list<double>
        &input_list) {
9
10      // Range-based for-loop.
11      for (double &value : input_list) {
12          func(value);
13      }
14  }
15
16  int main(int argc, char *argv[]) {
17      // Create simple list using list initializer.
18      std::list<double> my_list = {1.7, 2.8, 4.9, 5.9, 0.0};
19
20      // Instantiate lambda expression (anonymous function).
21      auto func_twice = [&] (double &input) {
22          input *= 2;
23      };
24
25      // Use lambda expression as function parameter.
26      map(func_twice, my_list);
27
28      for (double value : my_list) {
29          std::cout << value << "␣";
30      }
31      return 0;
32  }
```

Lambda expressions make huge impact on Qt 5.

Null pointers

It is quite common to type something like `int *variable = NULL` in C++ 03[5]. Let's expand `NULL`. In most cases, the result is `#define NULL ((void *)0)`. So `NULL` is literally "the pointer pointing to nothing of any possible type."

Problem occurs if `NULL` is defined as 0. Troubles might appear when overloaded function gets called with such a `NULL`. It's not obvious if `function(int)` or `function(int*)` gets called by `function(variable)`. It might be the first function on one system or second one on another system. C++ 11 implements new keyword `nullptr` wich is always evaluated to correct value.

1.3.1.1.2 Standard library improvements

Standard library used to be very tiny. It included just necessary classes, nothing special. But time goes forward, so that standard library must go too. Number of fine classes were added. You can find information about some improvements below. Complete list can be found in (Du Toit, 2012).

Threading

Finally, threading was introduced within the standard library. This threading subsystem should not depend on operating system threading implementation. But threading-related stuff in Qt depends on specific classes from operating system (pthreads on Linux) and work really fine. So these standardized threading facilities are not so important for ordinary Qt user.

Tuples (pairs)

Tuples are good bonus for every C++ programmer. So, there is no need to use 3$^{\text{rd}}$ party tuple implementations.

1.4 Qt components

Qt consists of libraries, tools and other supplemental software. You have already seen very brief list of Qt components in Chapter 1.1. Libraries themselves are divided into so-called *modules*.[6] You can learn more about modules in Chapter 2. Let's look into Qt library collection more thoroughly. Qt library collection contains these main components:

---

[5]C++ 03 is just one of many C++ standard revisions. This one was approved in 2003.

[6]You are not familiar with modules yet. Module is simply collection of related classes.

1. Tools for GUI design and implementation consisting of user interface designer (QtDesiger) and user interface classes (known as QtWidgets and QtGui).

2. Painting system which is accessory for GUI design. It can serve as the main force for creating graphics-related software, e. g. painting applications, video editors or perhaps chart designer programs.

3. Testing facilities which enable you to use test-driven development model. Unit-testing is extremely useful for large-scaled projects.

4. Complete thread subsystem that allows you to split your application computations among several threads of execution, making your program more robust and versatile.

5. Networking machinery for swift network communication between workstations and even among processes or threads.

6. Model-view-controller (MVC) architecture for binding your data to GUI or for structuring your data for further usage via abstraction layer (data model).

7. Resource system which allows you to embed any file directly into executable file, including pictures, music files or text files.

8. Facilities for Extensible Markup Language (XML) manupilation, web services integration, integrated help mechanisms, printing support, OpenGL wrapper, vector graphics classes, . . .

Some parts from this (not-so-complete) list will be examined deeply, some won't.

1.4.1 Supported platforms

Qt 5 is multi-platform framework and the support of various operating systems and platforms is one if its key features. Supported operating systems are:

- Windows (+ Windows Embedded Compact),

- Linux,

- Mac OS X,

- OS/2 (eComStation)[7],

---

[7]Qt ports for eComStation are usually released with delay from the original release date of the particular Qt version.

- Android (via Necessitas port).

Qt is ported to even more operating systems but those ports lack quality and completeness.

1.4.2 Qt 5 additions

Qt 5 concetrates on using modern technologies for painting user interfaces and introduces many other tweaks and improvements:

- Qt 5 is neither binary nor source code compatible with previous Qt releases, resulting in need of refactoring and recompilation of your Qt 4-based applications. Qt 3 support was dropped too.

- Brand new Qt component named Qt Platform Abstraction (QPA), allowing you to port Qt 5 easily to new platform and operating systems.

- All classes were update to conform to Unicode 6.2 standard.

- Quite important change happened in the QtGui module as all widget classes got moved into newly established QtWidgets module.

- QtQuick made it to version 2. QtQuick is module for writing applications using QML.

- QtWebKitWidgets now includes rewritten Webkit-based html rendering engine. Html 5, Canvas and WebGL are supported and web pages are now fetched asynchronously.

- C++ 98 – 11 compilers are supported.[8] Meta-object system was improved too.

- New multi-platform user interface style called Fusion is available (Figure 1.2).

1.5 Getting and installing Qt

There are basically three ways of obtaining Qt framework:

1. You have bought commercial Qt license so you can use specific Qt packages provided by Digia.

---

[8] Note that some compilers (e. g. Microsoft Visual C++ (MSVC) compiler) do not support all C++ 11 features yet. Use acclaimed GNU Compiler Collection (GCC) in case of problems. Even if your compilers supports C++ 11 you might have to use some compiler switch to turn C++ 11 support on.
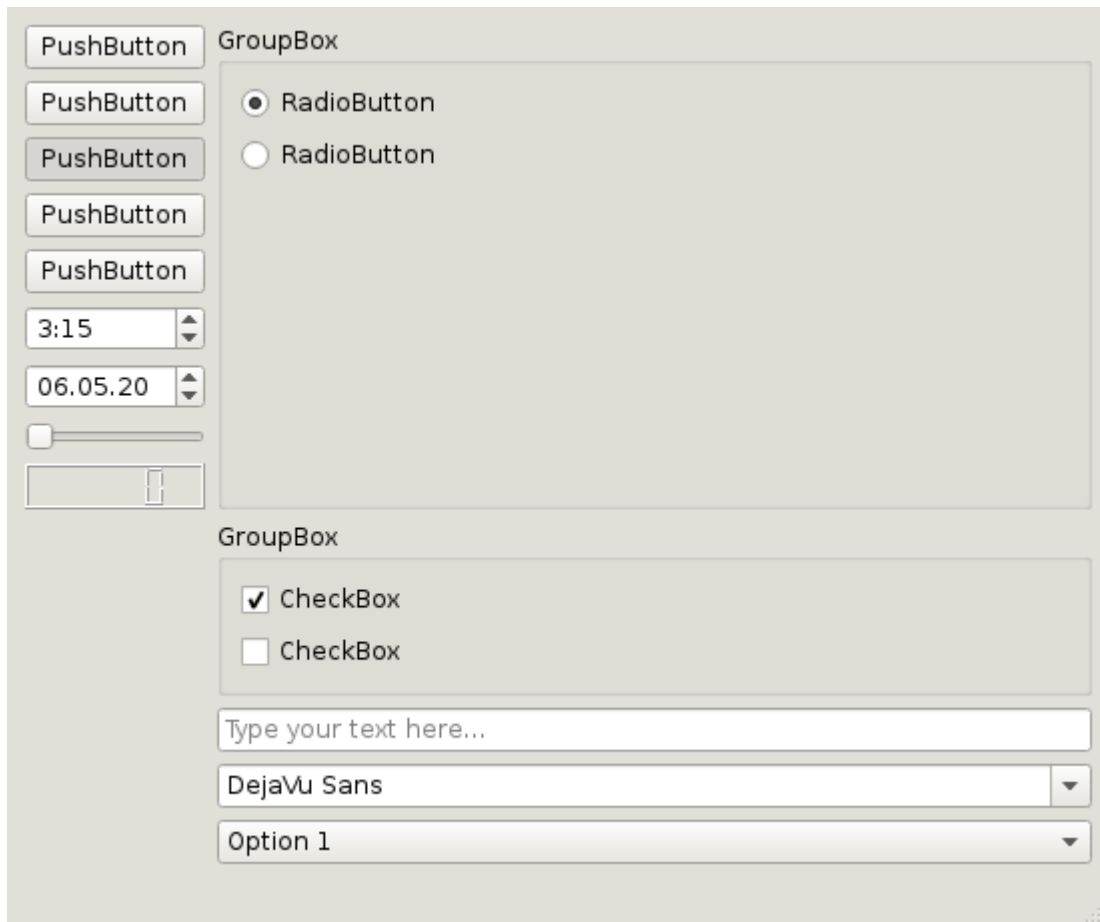
Figure 1.2: Qt Fusion style example

2. You can download open-source Qt framework directly from www.qt-pro-ject.org.

3. You use a Linux distribution which can equip you with Qt framework via its packaging system.

Qt can be downloaded as executable installer file which containes binaries pre-compiled for you as well as documentation and other needed tools. Sometimes manual compilation is needed.[9]

Precompiled Qt libraries are released for certain compilers only. Linux releases are should work with GCC, Windows releases are usually precompiled by MSVC. Qt framework with MinGW support is released from time to time too.

---

[9]It is commonly known that Qt compilation can last for several hours. Thus, consider getting precompiled binaries instead of compiling those by yourself.

1.5.1 Installing Qt on Windows

Qt installation on Windows is fairly straightforward if precompiled Qt binaries are available. All you need to do is obtain the setup executable file and follow instructions. Some troubles might occur, though. Let's assume that Qt was installed in `c:\Qt\Qt5.0.0\`.

You need to change your PATH environment variable in order to be able to run Qt tools from command line. Qt Creator will work even without proper PATH because it does all necessary settings for itself automatically. You can setup PATH variable in Windows 7 as follows:

1. From the desktop, right-click (by mouse) "My Computer" and then click "Properties".

2. Choose "Advanced System Settings" from the list.

3. In the "System Properties" window, hit the "Environment Variables" button.

4. Locate "System variables" group, select PATH variable and hit "Edit" button.

5. Move to the end of the string and add folowing paths:
   ```
   1  c:\Qt\Qt5.0.0\5.0.0\msvc2010\bin\
   2  c:\Qt\Qt5.0.0\Tools\QtCreator\bin\
   ```

   Paths within the PATH variable are separated by semicolons. Typical content of PATH variable may look like the one in Listing 9.

   Listing 9: Setting PATH environment variable for Qt on Windows
   ```
   1  %SystemRoot%\system32;%SystemRoot%;c:\Qt\Qt5.0.0\5.0.0\msvc2010\
          bin\;c:\Qt\Qt5.0.0\Tools\QtCreator\bin\
   ```

1.5.2 Installing Qt on Linux

As stated, Qt can be installed on Linux in two ways:

1. Linux distribution *package manager* offers it as the package. This is the case of many major distributions.

2. Classical installation via executable file:

   (a) Obtain installation file from (Qt-Project, 2013b).

(b) Open terminal and navigate to folder containing obtained installation file.

(c) Change permissions on the file:

```
1  sudo chmod +x ./qt-5-installation-file.run
```

You need to run `chmod` as superuser (root) if you want to install Qt into system-wide location.

(d) Install Qt by executing `./qt-5-installation-file.run`, follow on-screen instructions. It's good to install Qt into separate folder structure to keep system structure clean. Using `/opt/qt5` as base installation directory is generally good idea.

(e) There is no need of editing PATH environment variable if you use Qt Creator for development. Otherwise, make sure you set correct values to environment variables (see Listing 10).

Listing 10: Setting environment variables for Qt on Linux

```
1  QTDIR=/opt/qt5/5.0.0/gcc
2  PATH=$PATH:$QTDIR/bin
3  QMAKESPEC=$QTDIR/mkspecs/linux-g++
```

`QTDIR` variable contains path to root qt directory. This is the directory which contains subdirectories `bin`, `include`, `lib`, ...

1.5.3 Compilling Qt

Sometimes, you may need to compile Qt on your own. Compilation allows you to throw away features you do not like, resulting in smaller dynamic/static libraries sizes.

Qt sources are always contained within compressed file. All you need to do is to have correctly installed C++ compiler[10]. Basic compilation steps are quite similar for each operating system:

1. Decompress source package and navigate to its root folder using terminal (command prompt).

2. Run `./configure -opensource -nomake examples -nomake tests`.

3. Now, run `make` (on Linux), `nmake` (on Windows with Visual Studio) or `mingw32 -make` (on Windows with MinGW).

---

[10]Personally, I prefer GCC-based compilers for Qt development.

Compilation process can be complicated, as many problems can occur. See (Qt-Project, 2012b) for more information.

Chapter 2

Qt framework structure

Qt framework itself is a huge software collection and needs to be divided into logical units. Two main units are *libraries* and *additional software*.

Additional software includes compilers, tools for internationalization and tens of other tools. Some of them will be described in Chapter 4.

Qt offers very rich and diverse functionality (see Chapter 1.4), ranging from network communication to painting vector pictures.

This chapter introduces some basic concepts of Qt: *modules* and *tree-like* structure.

2.1 Modules

Each unit of related functionality is called *module*. Module is set of classes which is contained within a single (static or dynamic, see Chapter 4) library file. If you want to use this module in your code, you have to include appropriate header files and link your binary against the library file. Higher number of used modules results in more linked libraries and (possibly) bigger output binaries. Choice of Qt modules for application programming is therefore important.

2.1.1 Linking

Each module usually depends on QtCore module, including QtWidgets module. QtWidgets module is used for building classical widget-based GUIs. Moreover, QtWidgets module depends on QtGui module. QtGui module contains So each Qt-based application with user interface has to be linked against at least three modules.

Consider elementary GUI application with main window. You can find source in `sources/laboratory/04-guiapp` subdirectory. Application is compiled with modules

QtCore and QtWidgets. You can use GNU *ldd* application to list all dynamic libraries required for running the executable file. Output for our sample application looks very similar to the on in Listing 11.

Listing 11: Libraries needed for GUI application

```
1  [root@arch-linux 04-guiapp]# ldd -d -r 04-guiapp
2  linux-gate.so.1 (0xb77c7000)
3  libQt5Widgets.so.5 => /opt/qt5/5.0.0/gcc/lib/libQt5Widgets.so.5 (0
     xb719f000)
4  libQt5Gui.so.5 => /opt/qt5/5.0.0/gcc/lib/libQt5Gui.so.5 (0xb6d89000)
5  libQt5Core.so.5 => /opt/qt5/5.0.0/gcc/lib/libQt5Core.so.5 (0xb693f000
     )
6  libGL.so.1 => /usr/lib/libGL.so.1 (0xb6833000)
7  libpthread.so.0 => /usr/lib/libpthread.so.0 (0xb6817000)
8  libstdc++.so.6 => /usr/lib/libstdc++.so.6 (0xb672e000)
9  libm.so.6 => /usr/lib/libm.so.6 (0xb66eb000)
10 libgcc_s.so.1 => /usr/lib/libgcc_s.so.1 (0xb66ce000)
11 libc.so.6 => /usr/lib/libc.so.6 (0xb651d000)
12 libgobject-2.0.so.0 => /usr/lib/libgobject-2.0.so.0 (0xb64cd000)
13 libglib-2.0.so.0 => /usr/lib/libglib-2.0.so.0 (0xb63d2000)
14 libX11.so.6 => /usr/lib/libX11.so.6 (0xb629c000)
15 libicui18n.so.49 => /opt/qt5/5.0.0/gcc/lib/libicui18n.so.49 (0
     xb6084000)
16 libicuuc.so.49 => /opt/qt5/5.0.0/gcc/lib/libicuuc.so.49 (0xb5f0a000)
17 libdl.so.2 => /usr/lib/libdl.so.2 (0xb5f05000)
18 libgthread-2.0.so.0 => /usr/lib/libgthread-2.0.so.0 (0xb5f01000)
19 librt.so.1 => /usr/lib/librt.so.1 (0xb5ef8000)
20 /lib/ld-linux.so.2 (0xb77c8000)
21 libXext.so.6 => /usr/lib/libXext.so.6 (0xb5ee5000)
22 libpcre.so.1 => /usr/lib/libpcre.so.1 (0xb5e7d000)
23 libffi.so.6 => /usr/lib/libffi.so.6 (0xb5e76000)
24 libxcb.so.1 => /usr/lib/libxcb.so.1 (0xb5e53000)
25 libicudata.so.49 => /opt/qt5/5.0.0/gcc/lib/libicudata.so.49 (0
     xb4d32000)
26 libXau.so.6 => /usr/lib/libXau.so.6 (0xb4d2e000)
27 libXdmcp.so.6 => /usr/lib/libXdmcp.so.6 (0xb4d27000)
```

Pay attention to lines 3–5. Typical program with user interface needs to be linked against QtCore, QtGui and QtWidgets. Console applications need just QtCore. You can list unused (but linked) libraries too as seen in Listing 12.

Listing 12: Unused (but linked) libraries for GUI application

```
1  [root@arch-linux 04-guiapp]# ldd -d -r -u 04-guiapp
2  Unused direct dependencies:
3  /opt/qt5/5.0.0/gcc/lib/libQt5Gui.so.5
4  /usr/lib/libGL.so.1
```

34

```
5  /usr/lib/libpthread.so.0
6  /usr/lib/libm.so.6
```

Threading library (pthread) is used by QtCore on Linux. LibGL is 3D graphics library. LibGL is unused because no OpenGL-related function was explicitly called in our sample application.

Number of chosen modules affects memory consumption of each executable file. So, pick a reasonable subset of available Qt modules to make your application thin and fit.

2.2 Tree-like class structure

Wisely developed library has smart class structure which makes that library easily maintainable and expandable. *Class inheritance* is used very extensively if library design is something we need to deal with. Read (Prata, 2011, p. 708-783) to get more familiar with C++class inheritance if you are not so far. Class inheritance says that if one class is inheritor of another class, then it inherits parent's *data* and *methods*.

It's good habit to have some properties available in all classes of the library. Such a property could be e. g. *id*, the textual (or perhaps numerical) identification of each object (instantiated class) within the library. You would have to define what *id* means in each and every of your classes manually without inheritance usage. With inheritance, everything you must do, is to define *id* in exactly one of your classes, promoting this class to *root* class and make the rest of classes to inherit the new *library base class*.

This approach is solid base for having library with the tree-like structure (see Figure 2.1) where classes are structured according to their natural relationship.

So, as we found out, there is exactly one class that sits above other classes, which share its data and methods. Qt disposes this kind of top-level class too, it's called QObject.

One Step Further

Many well-known libraries follow root class idea and tree-like class structure. One example is .NET Framework. Its base class is called System.Object and provides some basic functionality (shared by all .NET classes via class inheritance) such as method providing basic string representation of each object. You can find more about .NET base class in (Nigel, 2010, p. 84). Java follows very similar class hierarchy ideas.
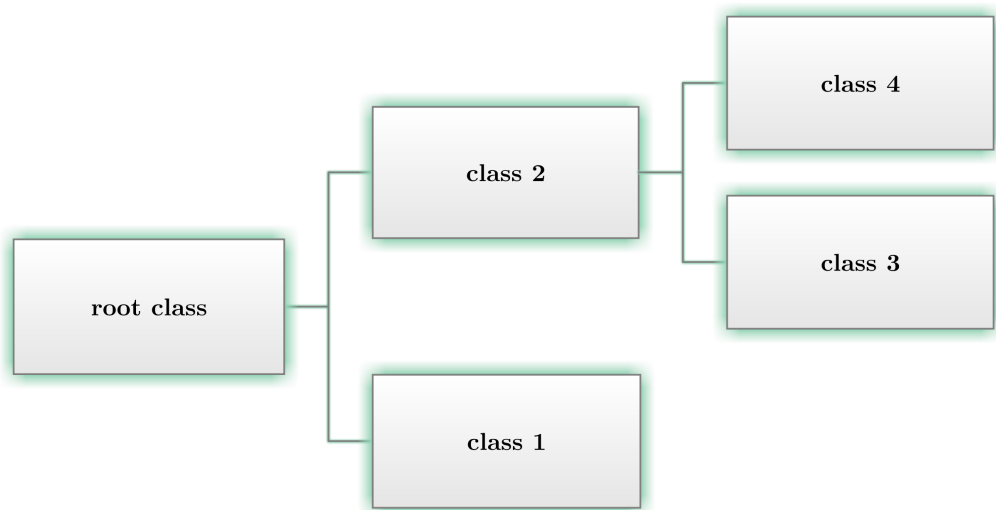

Figure 2.1: Typical library tree-like class structure

Chapter 3

Using Qt framework

Basic Qt library structure is known to us but we need to know something about Qt-related development environments and tools. You can develop Qt applications in any development environment, including Microsoft Visual Studio. Some tools are part of Qt framework, however. This includes young and thriving Qt Creator development environment.

Every Qt/C++ programmmer should be aware of existence of certain rules concerning source code appearance. These rules are called *conventions* and you will learn about them too.

3.1 Qt Creator

Qt Creator (see Figure 3.1) is fully-featured C++ & JavaScript development environment. It is suitable for plain C++ development as well as for Qt development. Qt Creator is part of Qt SDK, thus can be installed along with Qt libraries. Qt Creator supports big collection of features:

- multiple build systems (CMake, Autotools and QMake),

- syntax highlighting for more than one hundred programming languages,

- auto-completion for variables, functions and macros (see Figure 3.2),

- consistent look on every supported operating system,

- many plugins,

- refactoring facilities,

- tools for debugging,

- cooperation with Android SDK,

- dynamic keyboard shortcuts,

- integrated Qt help system (see Figure 3.4),

- context-aware help (see Figure 3.3),

- support for simultaneously installed Qt frameworks,

- integrated Qt Designer for GUI design,

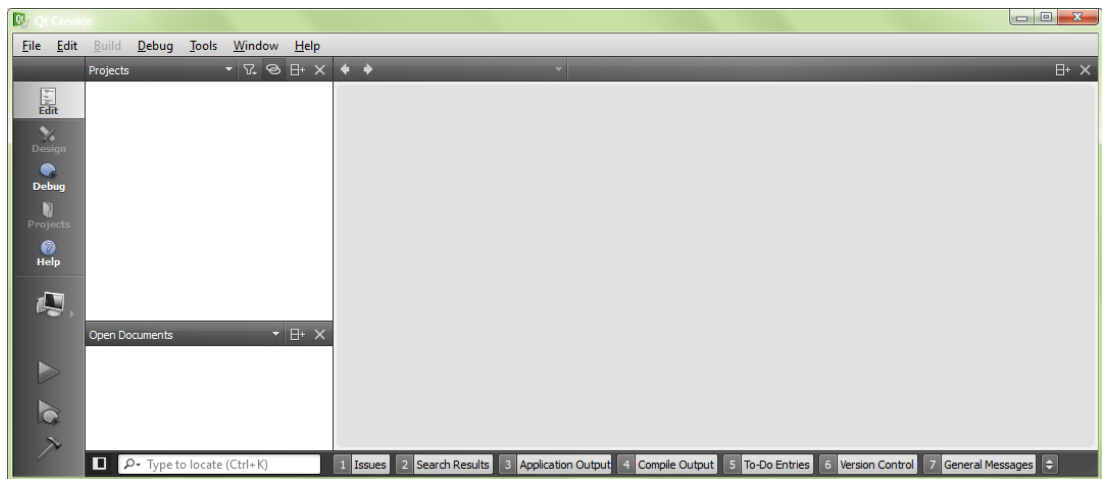- sharing source code via online services and many other features.

Figure 3.1: Qt Creator empty environment

3.1.1 Speeding-up Qt Creator

You can make Qt Creator less memory-hungry by disabling some unneeded plugins. You can disable plugins in `About -> About Plugins...` menu. Minimal setup for desktop Qt development using CMake may look like the one in Table 3.1.

3.2 Qt Designer

Qt Designer (see Figure 3.5) is a tool for user interface design. It is standalone application which is integrated into Qt Creator too.
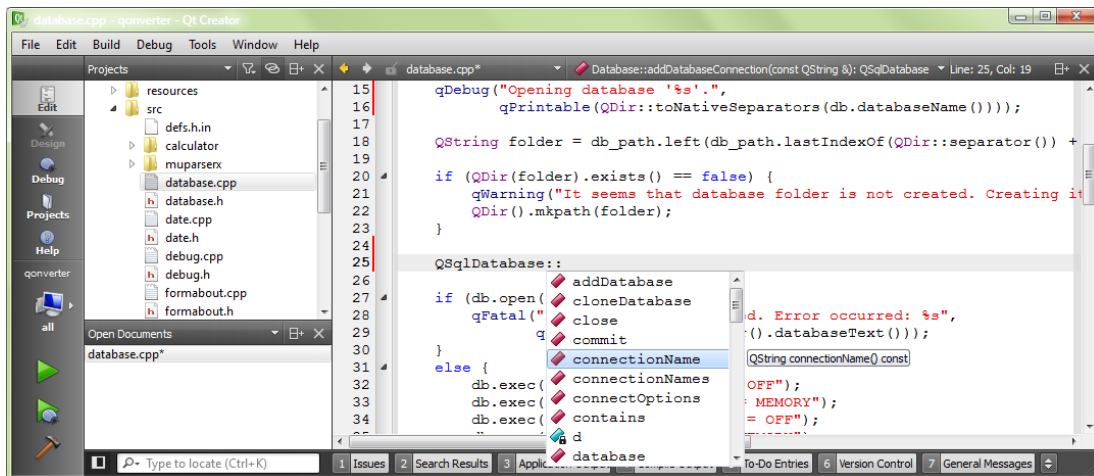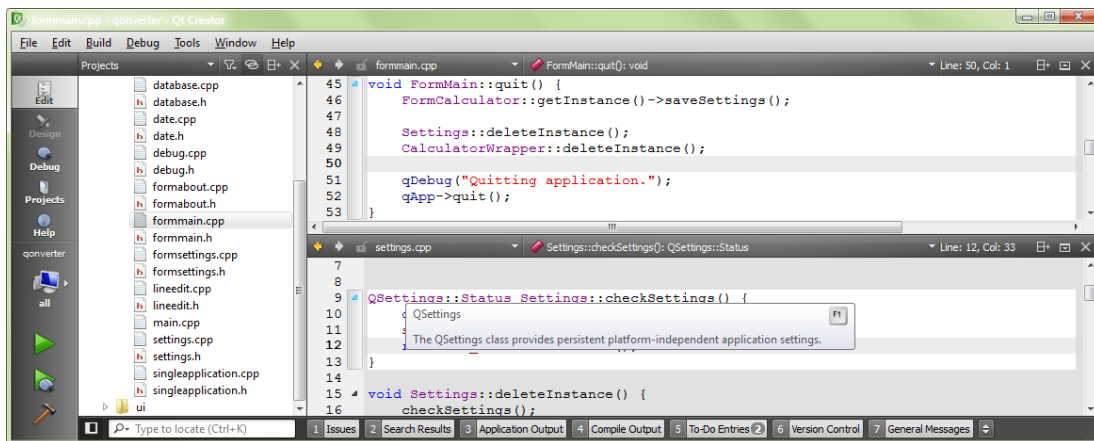
Figure 3.2: Qt Creator auto-completion

Figure 3.3: Qt Creator context-aware help

3.3 Tools and chains

Qt Creator supports multiple compilers and multiple Qt libraries installed side by side. You can choose any installed compiler or Qt library to build your projects. Qt Creator uses special terminology for groups of compilers and Qt libraries called *kits*. Kit (see typical kit setup in Figure 3.6) is virtual container for one compiler and one specific Qt library (for example Qt 5 library). Moreover, kit specifies primary debugger and other stuff needed to compile your projects. Kit (or *toolchain*) says what is used to compile your project. You can manage compilers and Qt versions in `Build & Run` section of Qt Creator settings dialog.
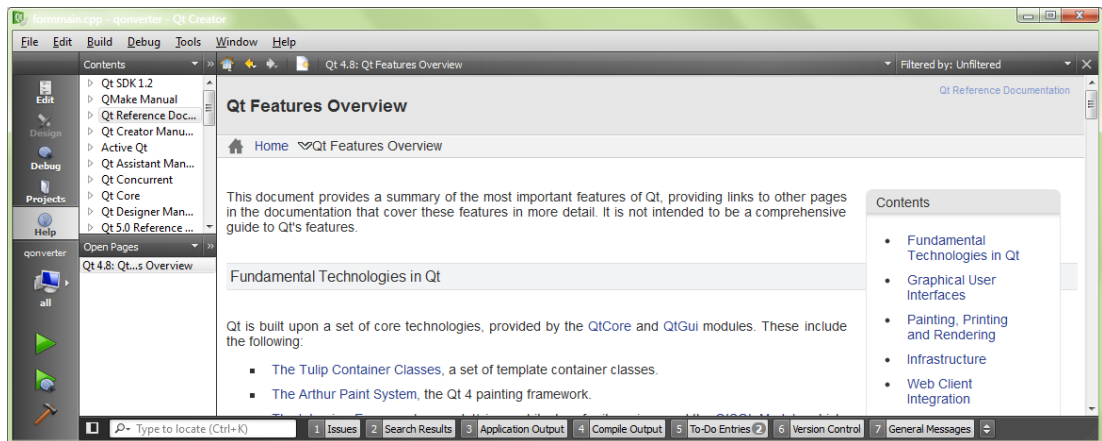
Figure 3.4: Qt Creator full reference documentation

3.4 Qt Framework Source Code

As we know, Qt offers free source code. It is available on-line via (Qt-Project, 2013a). Qt libraries consist of many classes. Some of them are not part of the public Application Programming Interface (API) and are not documented. You can discover these classes by browsing the source code. Available source code allows you to check logic and correctness of each Qt class you use in your Qt-based applications. Moreover, you can seek for bugs and possible improvements in Qt.

Qt source code is great reference for writing custom components which are based on classes from the Qt.

3.5 Code conventions

Writing working application is not enough in Qt world. Qt itself is huge library and rules are more important here than everywhere. These rules we talk about apply primarily to our source code. Code must be self-descriptive. This can be achieved by following certain code conventions. We are talking here about source code "typography".[1]

3.5.1 What are conventions?

Programming can be very difficult job sometimes. That's why programmers do need to make their work easier. Conventions are tools to achieve that. It's always

---

[1]You should assure yourself you have some base to build on before you proceed. (McConnell, 2004, Chap. Layout and Style) is the good starting point.

Table 3.1: Qt Creator minimal plugins setup

enabled	disabled
CMakeProjectManager	AutotoolsProjectManager
GenericProjectManager	ClassView
Qt4ProjectManager	CodeAnalyzer
QtSupport	DeviceSupport
CppEditor	GLSL
CppTools	BinEditor
Debugger	Bookmarks
Designer	ImageViewer
Help	Macros
ProjectExplorer	UpdateInfo
ResourceEditor	Welcome
CodePaster	QtQuick
Todo	FakeVim
	HelloWorld
	TaskList
	VersionControl

useful to know what certain source code does simply by taking a brief look at it. If this happens, then code is written well, it is readable, understandable and it is actually joy to browse it.

Let's compare source code to formatted text (e.g. text of a book). Good book does have its contents formatted so that reader can read it smoothly and comfortably. It is supplemented with pictures, tables, charts and diagrams. Text itself is separated into paragraphs (usually one paragraph per idea), which can have indented first line or can be separated by vertical white space. In addition, important words can be highlighted with color or *emphasised* alternatively. You can disagree with claims in this paragraph but these claims can be summed up into one thing called *style*. Your potential disagreement signs that there are many various styles. Some apply to books, other ones to cars. Styles are not perfect.

If majority of interested people likes these adjustments (e.g. visual adjustments of source code), then we call these adjustments *conventions*. Almost everything can have its style and convention. Style makes things usable and predictable.

Good programmer has to realise he produces source code not for him but for other programmers in a community or in a team. Aware programmer produces code styled the way a team (or a community) likes, not the way that he wants.
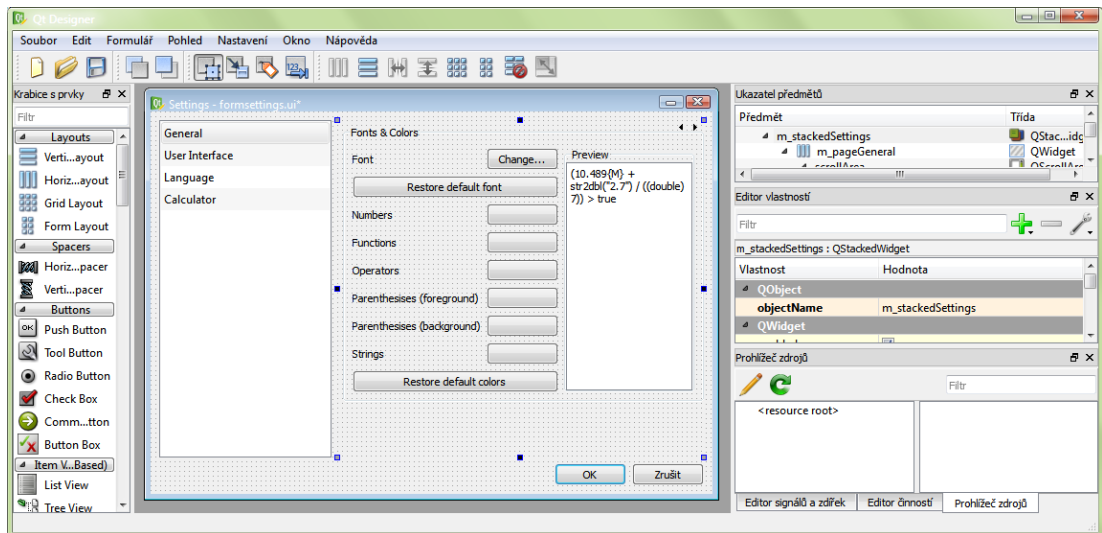
Figure 3.5: Qt Designer environment

Figure 3.6: Qt Creator kit setup

3.5.2 Applying Qt conventions

Compare Listing 13 and Listing 14. Just a brief look can advise you what is meant by code readability.

Listing 13: Bad code style

```
1  #ifndef CAR_H
2  #define CAR_H
3
4  #include <QtCore/QDebug>
5
6
7  class Car {
8      private:
9          unsigned int NumberOfWheels;
10     public:
```

```
11          Car(int w);
12          void showMeThisParticularCar();
13      private:
14          bool owner;
15  };
16  #endif // CAR_H
```

Listing 14: Good code style

```
1   #ifndef CAR_H
2   #define CAR_H
3
4
5   class Car {
6       public:
7           // Creates new car.
8           Car(int number_of_wheels, bool has_owner = true);
9
10          // Displays information about this car.
11          void showCar();
12
13      private:
14          int m_numberOfWheels; // Stores count of wheels of this car.
15          bool m_hasOwner; // True if this cas is owned by someone.
16  };
17
18  #endif // CAR_H
```

There are considerable differences between these two code fragments. Importance of conventions is not probably obvious because of sample code shortness but it grows up rapidly with regard to code complexity and length. We should examine lines of Listing 14 now.

3.5.3 Elements of good Qt code style

Base for Listing 14 is sample application in `sources/laboratory/06-good-car`. Let's explore both file `sources/laboratory/06-good-car/car.h` and `sources/laboratory/06-good -car/car.cpp`.

File car.h

Using two blank lines between header file last `#define` macro and class declaration beginning is generally good habit. Some programmers use just one blank line.

```
1   #ifndef CAR_H
```

```
2  #define CAR_H
3
4
5  class Car {
```

Blank-lining is good in many constructs of C++ language. One blank line should appear before each class section except the first one.

```
5  class Car {
6      public:
```

Comment your code. Always comment parts of code which are not straightforward.

```
7      // Creates new car.
```

Use lower-cased names for variables in functions (methods) with undersore as delimiter for words in a variable name. Boolean variables should include verb in its name.

```
8      Car(int number_of_wheels, bool has_owner = true);
```

Note that method should contain verb in its name because method always "does" something. Sometimes, single verb as name is pretty enough to describe what method does or how it works. Use Camel notation for method names. In Camel notation, all words in a compound (except for a first word) begin with upper-cased letter and are not separated by spaces or any other character.

```
11     void showCar();
```

Use Hungarian notation for class data members. In Hungarian notation, all variable names have prefix which signals data type or purpose of the variable. It is recommended to separate prefixes from the rest of the variable name by underscore character. Members prefixed with m_ are simply instance data members, while members starting with s_ are static data members of a class. Use this customized Hungarian notation along with Camel notation.

```
14     int m_numberOfWheels; // Signs count of wheels of this car.
15     bool m_hasOwner; // Does this car have an owner?
```

File car.cpp

Include Qt header files first, since they may include system-based header files, so that you have less inclusions in your source after all. Your own header files should be included as last ones. Leave two blank lines (sometimes one blank line is enough) between headers inclusions and the rest of your source code.

```
1  #include <QDebug>
2
3  #include "car.h"
4
5
6  Car::Car(int number_of_wheels, bool has_owner) {
```

Don't use `s_` or `m_` prefixes for variables with method scope (the ones declared inside method) because those are not data members.

```
6      m_numberOfWheels = number_of_wheels;
7      m_hasOwner = has_owner;
```

> **One Step Further**
>
> Inclusion of header files is a matter that should attract our attention. It is **highly** recommended to avoid typing used Qt module into header file path, for example writing `#include <QtCore/QDebug>` is not good idea.
>
> `QDebug` class could be removed from QtCore module and moved into newly forged one in the future.[2] The code from Listing 13 won't work with that hypothetical Qt build. Include Qt stuff in a simpler way instead, for example `#include <QDebug>` is much better. Don't include entire Qt modules either. Reason is the same. Moreover, including a whole module may result in inclusion of class headers you would never use in your application.

As we said, code style is unique for each and every programmer in some detail. If you program an application, try to stop sometimes and imagine that you are someone who sees your piece of code for the first time and try to think about goal of the code. Or even better, send your code to someone else for review.

---

[2]This has actually happened with Qt 5 release. Module QtWidgets was created and some classes from QtGui were moved into it.

Chapter 4

Compilation process

Compilers pursue programming languages since the time languages arised. Modern compilers are often written in quite high level programming languages. Compiler of Scheme can be (and for learning purposes is) written in Scheme itself. Smart people see here typical instance of "the chicker or the egg" problem. What if we have just invented new programming language and we need to program its compiler? We need to use another programming language to program its very first compiler.

Same problem was faced by developers in the times of programming antiquity. Perfect example is the C compiler. When C was invented, there was (naturally) no compiler for it. It had to be programmed from scratch using another programming language. Assembly language was chosen. That led to quite high code complexity and very huge effort by the programmers had to be spent, so that compiler could be finished.

4.1   Compilers, linkers, assemblers, ...

Compiler is generally a converter. It does certain kind of conversion. If we talk about programming language compiler, then we naturally expect to convert textual *source code* into executable form. Output of a compiler (of object-oriented language) is sometimes called *object code*. Transformation from source code into object code is not straightforward and needs to be done in several steps in majority of C++ compilers.

In fact, compiler doesn't do source code to object code transformation. It does transformation from source code to *assembly language*. Assembly language is then assembled into object code by *assembler*.

Object code itself can be directly executable but, in most cases, it is not. C++ offers many functions and features via embedded standard library. All functions

are placed is separate dynamic-link library. Object code contains just signatures of used functions. Function bodies are stored in library and object code needs to be told where library is located, so that called standard library functions can find their bodies and execute successfully. Process of connecting library functions signatures in source code to library function bodies in library file is called *linking* and tool which performs such a process is called *linker*. Linking can be divided into static and dynamic, see below for more information.

One Step Further

There are two types of library linkage:

DYNAMIC LINKAGE

Is very popular for its usefulness. Dynamic linking means that executable file (operating system more precisely) seeks for needed libraries in certain predefined paths in run time. Usually one version of each library is placed somewhere in well-known folder structure and each executable is linked against it. So more running executables can actually use the same library file. This saves memory and is very popular within Unix-like operating systems but it can bring certain level of disorder into poorly designed operating system. This has something to do with Windows because many applications doesn't link with libraries stored in system path and use varying versions of the same library sometimes, duplicating library presence in memory and increasing memory usage.

STATIC LINKAGE

Not so favourite kind of linkage. Library is packed into executable file and linked in compile time. This makes executable file (sometimes considerably) larger but no additional dependencies (in form of external dynamic libraries) are required. GNU GPL Qt libraries **cannot** be linked statically.

4.2   Executable file and its structure

Final output of C++ code compilation is executable file or library file. Structure of executable file differs from platform to platform. Linux uses Executable and Linkable Format (ELF) and Windows uses Portable Executable (PE).

Both executable file formats differ in details but they follow the same idea. Executable file is divided into header and body in this idea. Header usually contains table with information about placement of linked libraries. This table is filled with actual information when exectuble file launches. Body of executable file includes object code.[1]

---

[1]Information in this paragraph is intentionally simplified for our purposes.

Let's look at compilation process of plain C++ application and Qt-based application. There are many differences as different entities take part in the process.

4.3   Classic C plus plus compilation process

Experienced C++ programmer is probably familiar with standard compilation process (Figure 4.1). This process consists of four main steps:

1. Makefile generation utility generates desired kind of makefiles. This step is fairly optional and is not needed for small applications.

2. Preprocessor examines input source code, replaces all occurrences of preprocessor definitions and expand macros. Files produced by preprocessor are ready to be processed by compiler.

3. Compiler checks syntactical correctness of input C++ code. If code is correct, then compilation goes on, otherwise procedure halts and error is displayed to the user. Compiler produces assembly code which is accepted by assembler.

4. Assembler takes assembly code and produces machine code (object code) for target architecture.

5. Linker accepts compiled machine code as its input and produces executable file by linking machine code against needed libraries and adding necessary metadata and headers.

4.4   Qt-way C plus plus compilation process

Qt/C++ compilation process (Figure 4.2) differs from classic C++ code compilation process because Meta-object compiler (moc) takes part in the compilation process. moc one of fundamental basic stones of Qt itself. It is just more sophisticated preprocessor tool and source code generator. You will learn about moc later because it is essential part of Qt Meta-object system (mos).
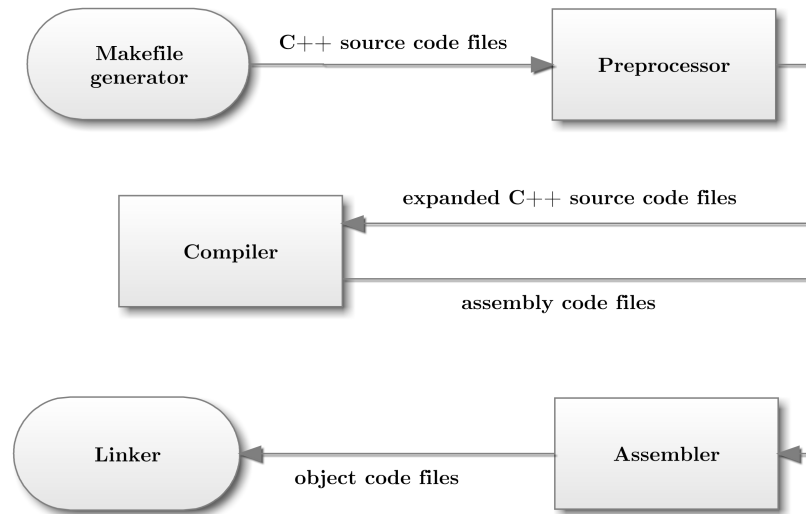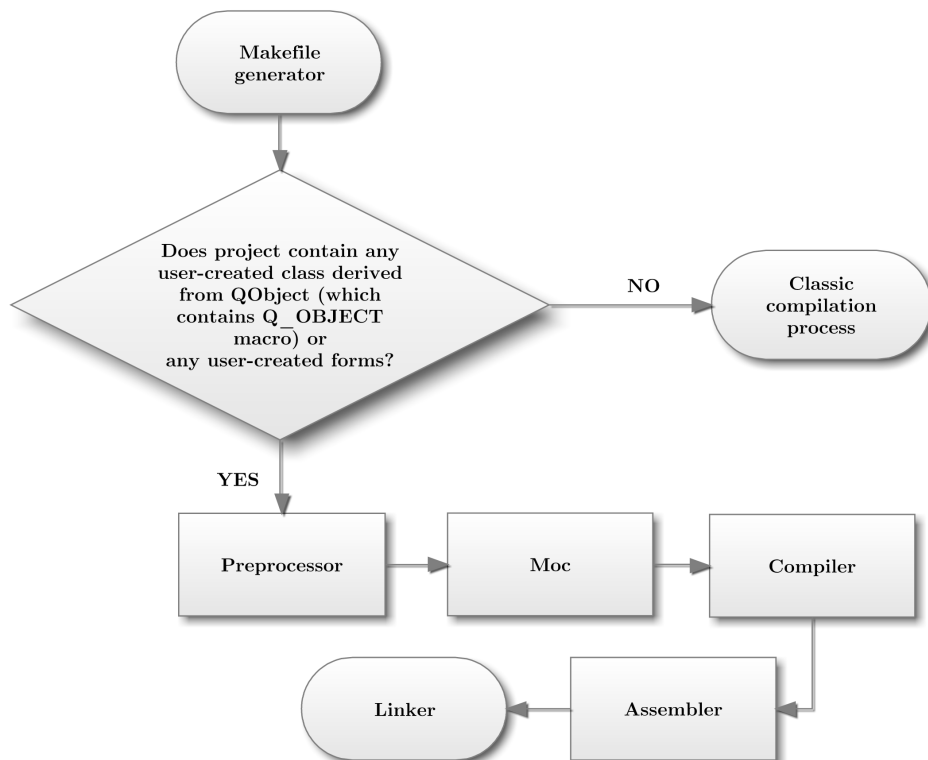
Figure 4.1: Classic C++ code compilation process

Figure 4.2: Qt-way C++ code compilation process

50

Chapter 5

Global Qt functions and macros

Qt has its functionality separated into modules. There is one special module (incarnated in QtCore module) called QtGlobal. QtGlobal is contained within single header file `qglobal.h` and is the part of QtCore library file. QtGlobal contains following:

- type clones (Table 5.1) for every standard C++ type,

- functions,

- macros.

Table 5.1: C++ type aliases in Qt

orignal type name	Qt type name
signed char	qint8
unsigned char	quint8
short	qint16
unsigned short	quint16
int	qint32
unsigned int	quint32
qint64	qlonglong
quint64	qulonglong
unsigned char	uchar
unsigned short	ushort
unsigned int	uint
unsigned long	ulong
double	qreal

5.1 Fundamental functions

QtGlobal offers very fundamental functions for value-based comparing and other basic tasks. These functions (often) wrap similar functions from standard C/C++ library. Most used functions are:

T qAbs(const T & value)
> Returns absolute value of input parameter.

const T & qBound(const T & min, const T & value, const T & max)
> Returns input value "rounded" to fit within bounds.

double qInf()
> Returns value which represents infinity.

qint64 qRound64(qreal value) and int qRound(qreal value)
> Mathematically rounds input paramater either to 64/32 bit integer.

All functons can be found in `/qt-root-directory/include/QtCore/qglobal.h`. Some other functions will be used throughout the rest of the book.

5.2 Producing console outputs with Qt

Qt offers better way to produce console printing for debugging purposes via `QDebug` class and `qInstallMessageHandler` function. You can always use traditional `std::cout` for console printing but `QDebug` way is much better. Basic syntax for using `QDebug` is fairly simple (Listing 15) as it implements `<<` operator and can act as `printf(...)` function.

Listing 15: Basic QDebug usage

```
qDebug() << "Print␣number␣" << 10 << ".\n";
qDebug("Print␣number␣%d.\n", 10);
```

First `qDebug` usage requires explicit `<QDebug>` inclusion. Second usage acts as wrapper for the `printf` function from standard C library. You can use also `qWarning`, `qCritical` or `qFatal` functions according to importance of message.

Default implementation halts an application if `qFatal` is called and uses `std::cerr` output for printing messages. You can implement custom behavior for previous functions very simply:

1. You need to implement global function (or static method) with signature `void (*function)(QtMsgType, const QMessageLogContext &, const QString &)`. Typical implementation may look like Listing 16.

2. You need to assign handler to this function via `qInstallMessageHandler` function.

Listing 16: Typical printing handler for QDebug

```cpp
void debug_handler(QtMsgType type, const QMessageLogContext &
    placement, const QString &message) {
    switch (type) {
    case QtDebugMsg:
        fprintf(stderr, "[%s]_INFO_(%s,_line_%d)_:_%s\n",
            APP_LOW_NAME,
            placement.file,
            placement.line,
            qPrintable(message));
        break;
    case QtWarningMsg:
        fprintf(stderr, "[%s]_WARNING_(%s,_line_%d)_:_%s\n",
            APP_LOW_NAME,
            placement.file,
            placement.line,
            qPrintable(message));
        break;
    case QtCriticalMsg:
        fprintf(stderr, "[%s]_CRITICAL_(%s,_line_%d)_:_%s\n",
            APP_LOW_NAME,
            placement.file,
            placement.line,
            qPrintable(message));
        break;
    case QtFatalMsg:
        fprintf(stderr, "[%s]_FATAL_(%s,_line_%d)_:_%s\nApplication_
            is_halting_now.\n",
            APP_LOW_NAME,
            placement.file,
            placement.line,
            qPrintable(message));
        qApp->exit(EXIT_FAILURE);
    }
}
```

Calling `qFatal` function results in error dialog in Windows operating system (Figure 5.1).

One Step Further
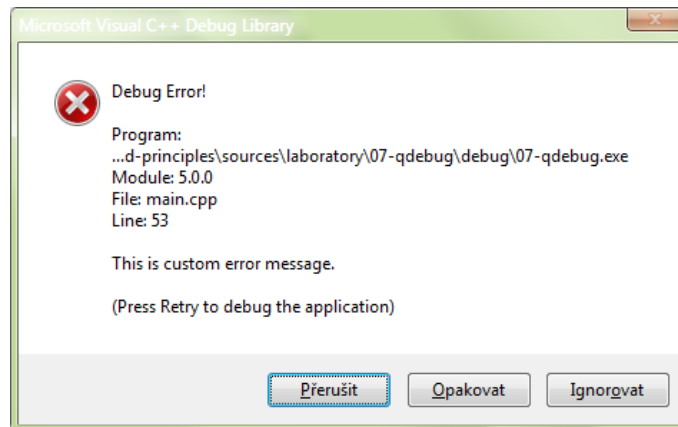Debugging outputs should always be written in English language with ASCII character set.

Figure 5.1: Application crash dialog in Windows

This implementation prints out extra information which is extremely important for debugging. However, you can do whatever you want in your implementation. Storing outputs to database or sending them over network are just some possible enhancements.

There is another (simpler) way of forcing Qt to format console outputs. You may tweak QT_MESSAGE_PATTERN build environment variable to display application name or other useful information. Head to (Qt-Project, 2012b) for more information or browse source code in sources/laboratory/07-qdebug.

Chapter 6

Meta-object system

Meta-object system forms substantial part of Qt functionality, providing majority of Qt classes with ability to asynchronously report its state when something happens. Furthermore, you can equip even your custom classes with extra textual information, fetch names of your objects at run time or make your classes use of the custom *property system* that provides faster and syntactically unified access to your class member data.

One Step Further

The word "meta" (which is originally Greek preposition, in Greek written as "μετα") was for the first time used by *Aristotle*, the great Greek philosopher.

   Aristotle wrote plenty of writings, covering poetry, music or politics. His creations needed to be sorted later so that they could be interpreted correctly. Writings got sorted and scholars realized that there is one book with no name. It was placed *after* Aristotle's great work *Physics*. That's why that mysterious paper was named *Metaphysics*, literally "the paper after Physics."

6.1   What is meta-object?

Generally, meta-object is an object that extends another object, providing us certain kind of information hidden "behind" that particular object or set of objects. Meta-objects lie beyond actual objects, forming "higher" abstraction layer of any Qt application.

   Each class instance exposes its private data through *methods* to its users – other classes. Publicly available class members (methods or *properties*[1]) form class interface, the only way to control class data and class behavior. This data is the only classical way to "see" the object from the view of its purpose but it

---

[1]Property can be understood as a private data member plus accessing getter/setter functions.

says completely nothing about object inner structure and representation, e. g. it doesn't expose type of an object (at run time) or count of its methods. Classic class methods do not provide us with *meta-information*. Meta-objects do that.

6.2   Reflection

Ability to obtain and perhaps modify meta-information of any object is an action called *introspection* (or *reflection*). We can distinguish two kinds of reflection:

RUN TIME REFLECTION

> This is the superior way of reflection. Introspection of meta-information of certain object is possible at run time but with one important addition. Compiler which supports run time reflection has absolutely no need to know the basis for meta-information construction at compile time. It does not need to add any extra data to the output code to allow reflection. Reflection is natural part of the language.

> This kind of reflection is supported primarily by languages that profit from using virtual machine and special output executable file structure. Both Java and .NET-based languages (e. g. C# or Visual Basic) (Nigel, 2010, p. 333) provide this.

COMPILE TIME REFLECTION

> Compiler of compile time reflection supported language has to do extra work to make reflection available. It usually produces extra code that grabs all (or most of) meta-information at compile time by going through the source code and extracting property names, method names, class names and other needed information. Extracted information is then formed into certain aggregations that are available as meta-objects at run time.

> This approach makes the compilation little slower because an extra tool has to be executed to do the job. This concerns Qt. Qt uses meta-object compiler to produce meta-objects.

6.3   Qt meta-object system

Qt uses compilation-based reflection due to C++ language limitations. Each object created within Qt meta-object system is automatically equipped with shadow meta-object. This meta-object allows you to do amazing things with that particular object. You can obtain its class name, check if this object's class inherits another class, get name of the superclass or names of its methods. You can even

call methods by their names stored in a string ([Listing 17](#))! Complete example can be found in `sources/laboratory/08-invoke` directory.

Listing 17: String-based method invokation in Qt meta-object system

```cpp
#include <QDebug>

#include <iostream>

#include "myapplication.h"


int main(int argc, char *argv[]){
    MyApplication a(argc, argv);

    std::string input;
    qDebug("Type name of method to be executed: ");
    std::cin >> input;
    QMetaObject::invokeMethod(&a, input.c_str());

    return a.exec();
}
```

6.4 Enabling meta-objectivity for custom classes

Not all classes in a Qt-based application take part in the meta-object system. You need to to several steps to make sure that objects of your class will be accompanied with corresponding meta-objects:

1. Your class needs to inherit `QObject`. Public inheritance is recommended. `QObject` class is fantastic base stone for any custom classes in Qt applications. You will learn about it in the next chapter.

2. Your class needs to contain `Q_OBJECT` macro in its private section, best way is right under class name. This macro adds several methods to your class, one of them is all important `QMetaObject *metaObject()` method. Moreover, dynamic translation system is enabled by this macro too. You will learn about Qt applications translation later.

6.5 QObject class

`QObject` class is the very base class of Qt which provides many marvelous features. It is good to use `QObject` as the base class even for your custom classes within any

Qt application because there is one particularly amazing feature – the automatic memory management provided by Qt object trees.

6.5.1 Qt object trees

There are some rules that apply to the way `QObject` should be inherited. Copy constructor and assignment operator mustn't be implemented in inheriting class. Reasons are very simple:

1. Each and every `QObject` instance stores pointer to its parent `QObject` instance. This results in instance tree hierarchy (Figure 6.1). Should copy of `QObject` instance point to the parent of the original `QObject` instance?

   Consider situation in the Figure 6.2. "George" instance was cloned and placed in the hierarchy. In general, there is no rule on where to place new copy in the tree hierarchy. It could be positioned as the sibling of the original object. New "George" is the sibling of the original "George". Problem becomes clear when the original "George" instance is freed from memory. All its children are removed too, in other words, whole subtree with "George" as the root gets cleared from application memory but another (cloned) "George" remains untouched. Is this desired behavior? In some situations it could be but sometimes it's not.

2. Each `QObject` instance has certain properties and those can be unique. Example of such a property is instance name (can be set by `void QObject::setObjectName(const QString & name)` function) which should be unique for each `QObject` instance. The same name could be automatically assigned to the new copy of the instance but that results in two instances with the same name (Figure 6.2) and that's the problem because you may want to search for one particular object by name which is possible in Qt. Two objects with the same name make search ambiguous.

Every complex Qt-based application usually contains several `QObject` tree hierarchies. These trees are disjunct. Example of typical tree hierarchy can be application main window. It usually contains menu bar, status bar, bunch of buttons, some text boxes and other visual elements. Naturally all these elements are owned by main windows. Thus, main window is the root of the main window elements tree hierarchy. If main window is cleared from memory, all its children are cleared from memory too which is desired behavior. That makes memory management more automatic.

`QObject`-based object can be deleted from memory by calling `this->deleteLater()` method or by using classic `delete` operator. See more about deleting objects in Qt in Chapter 8. Existence of tree hierarchies impacts positively on several topics:
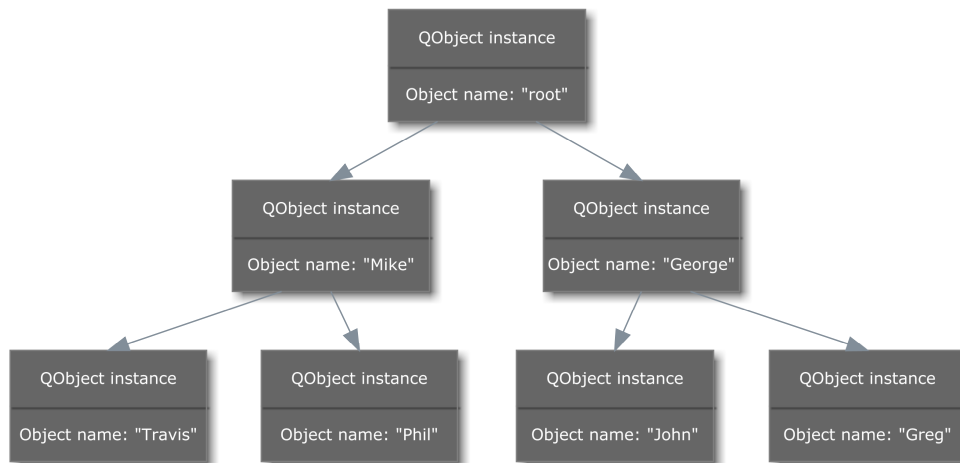
Figure 6.1: QObject instances tree hierarchy

- more sophisticated memory management,[2]

- better track of the number of objects in application component,

- better debugging.

6.5.2 Subclassing QObject

You have already read something about `QObject` class in previous paragraphs. You know that `QObject` instances form tree hierarchy. Subclassing `QObject` is similar to standard C++ class subclassing but you need to include `Q_OBJECT` macro and you should instantiate `QObject` with correct parent object, except some rare cases. Inheriting `QObject` is very simple, just see Listing 18.

You see that `QObject` constructor accepts pointer to parent object which is used to construct `QObject` base for `MyQObject` instances.

Listing 18: Subclassing QObject

```
/* header file (myqobject.h) */
class MyQObject : public QObject {
    Q_OBJECT

    public:
        explicit MyQObject(QObject *parent = 0);
};

```

---

[2]Tree hierarchies form just part of Qt memory management as you will see in Chapter 7.
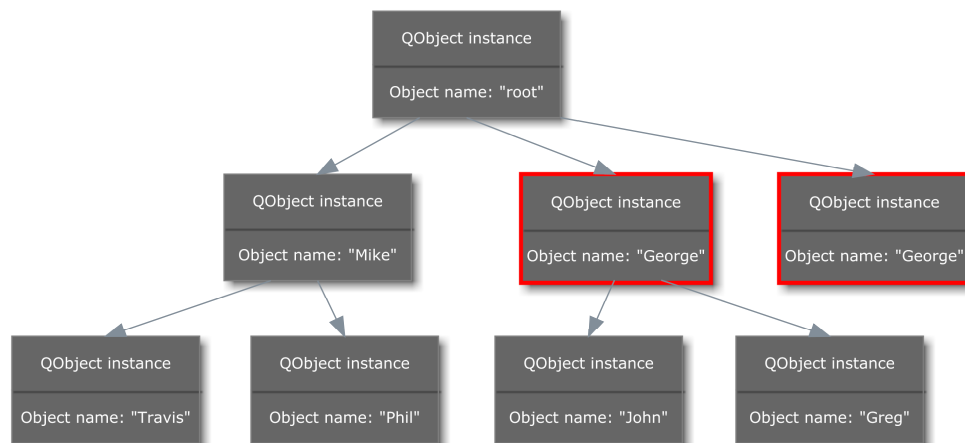
Figure 6.2: Broken QObject instances tree hierarchy

```
9  /* source file (myqobject.cpp) */
10 #include "myqobject.h"
11
12
13 MyQObject::MyQObject(QObject *parent) : QObject(parent) {
14 }
```

6.5.3 Signal – slot mechanism

Signal – slot mechanism is the main tool to interconnect two `QObject`-based objects, allowing thread-safe communication between them. Signals and slots form the alternative to the callback mechanism.

What is signal?
Signal is sign, which signs occurrence of specific event that happened during method execution of particular `QObject`-based class. In fact, signal is specially formed method with arbitrary number of arguments.

What is slot?
Slot is method in the same or another class which represents natural reaction to particular signal occurrence.

Imagine typical *textbox* control (Figure 6.3). This textbox could offer many *signals* which are usual for this kind of control. Every textbox should emit appropriate signal if its content changes or if ENTER key is pressed by application user.

60
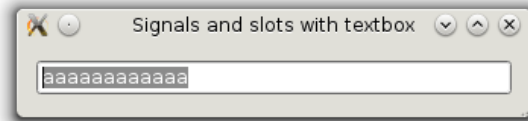
Figure 6.3: Typical textbox example

So there is textbox which emits signals. We need to have receiver of signals too. Receiver of signals from textbox could be for example the application itself. Application can perform some action (for example exit) if ENTER key is pressed inside textbox. Such an action is called *slot*. If there exists slot in one particular entity that reacts to signal emitted by another entity, then we say that there is *signal – slot connection* between those two entities.

One signal (of one entity) can be connected to several slots (contained in several entities), this behavior represents 1:N relationship. Several signals can be connected to one slot (M:1 relationship). Existence of 1:1 relationship is obvious. Moreover, arbitrary number of signals can be connected to arbitrary number of other signals so you can forward signals very easily.

Signal – slot mechanism does not apply just to GUI elements. Every `QObject` subclass can take advantage of it. Simple example can be file downloader class which signals progress of download.

One Step Further

Tiny definitions of signal and slot on page 60 correspond with terms and known from .NET-based languages. (Nigel, 2010, p. 200-202)

6.5.3.1 Using signal – slot mechanism

We are familiar with basic terms now. We are also able to subclass QObject. Let's implement very simple bank account representation. We expect that account provides us with possibility to save/withdraw money, check its status or make payments to another account.

Accounts are usually managed by bank. Bank ensures us that our payments are sent to the correct target accounts. Sample application can be found in `sources/laboratory/11-bank` subdirectory. Let's dig into it.

Application contains two primary classes: `Account` and `Bank`. Let's start with `Account` class (see Listing 19). This class inherits `QObject` (lines $1-2$), thus, meta-object features (including signal – slot mechanism) are available. Section for slot declarations is preceded by `slots` keyword (line 15) with optional modifier. You can have a public slot as well as a private slot. It's just a matter of a situation.

As stated earlier, slot is just method with special behavior, it's able to react to signal occurrence.

Class `Account` contains two signals. Their purpose is to inform connected `QObject`-based instances (for example `Bank` instances) about money flow within the account. Signals are placed in their own section (line 24) preceded by `signals` keyword.

> **One Step Further**
> Quick look inside the Qt `qglobal.h` file tells us that `signals` keyword is synonym for `public` keyword. Thus, signals are just public methods. This observation gets clarified on page 65 in chapter Signals and slots with regard to meta-object compiler.

Listing 19: Account class design

```
1  class Account : public QObject {
2      Q_OBJECT
3
4      public:
5          // No copy constructor or assignment operator is declared,
6          // just simple constructor.
7          explicit Account(const QString &owner,
8                           int deposit,
9                           Bank *parent);
10
11         // These are NOT slots.
12         void status();
13         QString name();
14
15     public slots:
16         // Used by customer who requests money from his account.
17         // Customer can be either bank or account owner.
18         void withdrawMoney(int sum);
19
20         // Used by customer to save money to this account.
21         // Customer can be either bank or account owner.
22         void saveMoney(int sum);
23
24     signals:
25         // Emitted if money is withdrawn successfully from this
26             account.
27         void withdrawn(int sum);
28
29         // Emitted if money is saved successfully into this account.
30         void saved(int sum);
31
32     private:
```

62

```
32        QString m_owner;
33        int m_deposit;
34 };
```

Second biggest class is the `Bank` class (see Listing 20). Primary purpose of this class is to manage underlying accounts and make sure money transfers among them are okay.

Listing 20: Bank class design

```
1  class Bank : public QObject {
2      Q_OBJECT
3
4    public:
5      explicit Bank(QObject *parent = 0);
6      void printAccounts();
7      void transfer(const QString &from, const QString &to, int sum);
8
9    signals:
10     // Emitted if both accounts are ready for money transfer and
11     // money should be withdrawn from the first one.
12     void withdrawalWanted(int sum);
13
14   protected:
15     // Seeks for accounts in children of this bank.
16     void checkAccounts();
17
18     // Returns account with given name or nullptr if no
19     // such account exists.
20     Account *getAccountByName(const QString &name);
21
22   protected slots:
23     // Adds given sum to account and disconnects
24     // account.
25     void serveAccount(int sum);
26
27   private:
28     QList<Account*> m_accounts;
29
30     Account *m_sendingAccount;
31     Account *m_waitingAccount;
32 };
```

Money transfer is managed by `void transfer(const QString &from, const QString &to, int sum)` method (Listing 21). Method checks for existence of both accounts and some other tasks and, finally, establishes two signal – slot connections (lines 23 – 24).

First connection says: "If the bank wants to withdraw the money from the first account, then account must really withdraw the money." Second connection says: "If the money was withdrawn from the first account, then the bank should finalize money transfer by sending the money to the second account."

Bank starts the money transfer procedure by trying to withdraw the money from the first account (line 26).

Listing 21: Money transfer between accounts

```cpp
void Bank::transfer(const QString &from, const QString &to, int sum)
 {
   if (sum <= 0) {
       qDebug("You cannot transfer sum %d USD.", sum);
       return;
   }

   checkAccounts();

   Account *acc_from;
   Account *acc_to;
   if ((acc_from = getAccountByName(from)) == nullptr) {
       qDebug("Source account is not registered in this bank.");
       return;
   }
   if ((acc_to = getAccountByName(to)) == nullptr) {
       qDebug("Destination account is not registered in this bank.")
           ;
       return;
   }

   m_sendingAccount = acc_from;
   m_waitingAccount = acc_to;

   connect(this, &Bank::withdrawalWanted, acc_from, &Account::
       withdrawMoney);
   connect(acc_from, &Account::withdrawn, this, &Bank::serveAccount)
       ;

   emit withdrawalWanted(sum);
}
```

If money are withdrawn from source account, they are added to target account via void serveAccount(int sum) method. Then, all established connections are destroyed.

6.5.3.2 Explanations

We saw typical `connect(....)` method usage on line 23 of Listing 21. It corresponds with this generic notation:

```
1  connect(source-object, source-signal, target-object, target-slot);
```

This is basic syntax for connecting two objects. First and third arguments are pointers to both objects. Second argument is signature of the source signal in the form `&Class::signal`. Last argument is slot signature in the same form. Both signal and slot have to have the same number of mutually compatible arguments. Otherwise, connection is impossible.

We see that bank emits signal on line 26 of Listing 21. Value of argument `sum` is then delivered to all slots of objects which have active connection with this signal.

There are other kinds of signal–slot connection. You can connect signal to another signal too:

```
1  connect(source-object, source-signal, target-object, target-signal);
```

Target signal is emitted when source signal is emitted. You can forward signals this way. This is primarily used in classes which form some kind of sublayer between other layers which usually run in different threads.

6.5.3.3 Signals and slots with regard to meta-object compiler

There is a need of extra meta-object compiler work before the actual C++ code is compiled into assembly code (see Figure 4.2 on page 50). Meta-object compiler goes through every `QObject`-based class in project and seeks signals and slots. If it founds any, then:

1. New source file is created and it is named `moc_original-file-name.cpp`. This file contains meta-information about all Qt entities from the original source file.

2. All signals are supplied with method bodies which are written to the `moc_original-file-name.cpp` file.

3. Each slot/signal obtains unique integer id, starting from 0. So if class contains two signals and two slots, then their ids are $0, 1, 2, 3$.

4. Some extra methods are added, see below for more details.

Meta-object compiler creates new meta-method which maps external method calls to signal/slot calls. This method has signature `int Account::qt_metacall( QMetaObject::Call _c, int _id, void **_a)` and its typical body looks like the one in Listing 22. This code comes from Bank example, see `sources/laboratory/11-bank.`

Listing 22: Signal/slot call entry point

```cpp
int Account::qt_metacall(QMetaObject::Call _c, int _id, void **_a)
{
    _id = QObject::qt_metacall(_c, _id, _a);
    if (_id < 0)
        return _id;
    if (_c == QMetaObject::InvokeMetaMethod) {
        if (_id < 4)
            qt_static_metacall(this, _c, _id, _a);
        _id -= 4;
    } else if (_c == QMetaObject::RegisterMethodArgumentMetaType) {
        if (_id < 4)
            *reinterpret_cast<int*>(_a[0]) = -1;
        _id -= 4;
    }
    return _id;
}
```

This method is called if signal or slot of particular `Account` instance should be invoked. Note that `Account` offers two signals and two slots. Wanted operation is determined according to variable `_c`. Possible operations include signal invocation, slot invocation and other actions. In our case, let's suppose that signal/slot should be called. Execution processes to line 7. Id of wanted element is checked and if id is within range, then slot or signal is executed by `void Account::qt_static_metacall (QObject *_o, QMetaObject::Call _c, int _id, void **_a)` method.

Signals and slots are entities created at compile time but connections are not. There is no need to dig into technical aspects of connections. Generally, connection consists of two pointers (source object and target object) and names of signals or slots. Signals and slots are invoked by name at run time. This kind of invocation was mentioned few pages back. Static method `bool QMetaObject::invokeMethod(....)` is used to do that. (Qt-Project, 2012b, QMetaObject class) You will learn something more about connections in chapter Threading (page 77).

6.5.4 QObject instance lifetime

It's good to know something about events that happen during the life cycle of each and every `QObject`-based instance.

Geniture

`QObject`-based object is born on the stack or on the heap. There is no real difference between stacked and heaped object from Qt perspective. New objects are added to one of trees (see Qt object trees on page 58) if they are created with valid parent

pointer.

Lifetime

Object is used as any other C++ object/class. It can take part in many signal/slot connections.

Forfeiture

Stacked objects are freed automatically if they get out of scope. This usually happens if method returns. Heaped objects are freed manually or by object tree destruction. All established connections are disconnected and signal `void destroyed (QObject *obj = 0)` is emitted just before object gets deleted from memory. You can connect other objects to this signal and delete them if signals occurs. You can chain deletion of several object trees this way.

Chapter 7

Memory management

Qt brings many new features into standard C++ memory management. We can stick with plain `new` and `delete` operators. Qt features are, however, highly addictive and useful. We have already heard about object trees (page 58) which are the real base for Qt memory management because tree structure is quite natural. Therefore, object trees are used much and many memory leaks get fixed with them.

Each object tree has root object and we (as programmers) are responsible for deleting this root from memory in the right time as there is no other parent object which gets this done for us.

7.1 Copy-on-write

Qt uses copy-on-write technique for managing Qt classes data. If you copy `QString` instance, then these two instances point to the same textual data, unless you try to modify contents of any instance. Shared data are then copied to new place and each instance has its own data. Modifications are done after.

7.2 Safe pointers

Pointers are both great and evil. They offer us great possibilities, e.g. they are used as thin method parameters. Pointers are also cause of the most of application crashes. We often use pointers which point to 0. Qt offers better pointers. They deal with their invalidity and ensuring that pointer is used only if it points to existing object. Lets introduce `QPointer` class. (Qt-Project, 2012b, QPointer class) `QPointer` does one simple job, it sets pointer to 0 if it's deleted. This helps a lot because you can check if pointer is valid by comparing it to 0. This class is pretty straightforward, look at documentation for more information.

Chapter 8

Event system

Events are reactions for some other actions. In software, actions can be divided into two categories:

HUMAN-TRIGGERED ACTIONS
> This type of actions represents natural notion of what events are. Mouse button clicks, keyboard key presses or perhaps cursor movements are human-triggered events. Each of them disposes certain properties, e.g. mouse-click produces coordinates of mouse when its button was clicked and key push produces the pressed character or perhaps set of characters if key sequence is used.

APPLICATION-TRIGGERED ACTIONS
> Are not triggered directly by application user. Typical application-triggered event is painting event which is responsible for drawing GUI elements on the screen. User starts this event indirectly by manipulating application user interface.

Event is consequence of occurrence of certain action in running application.

Each `QObject` subclass has ability to send events. Events are usually distributed by one entity which manages whole event process. This entity "sits" on the top of event loop. Event loop is part of application execution and it encapsulates all events sent by objects inside the loop. Loop goes from time to time through all raised events from its underlying objects and delivers those events to target objects. Event loop structure looks similar to one in Listing 24. Event loop exits if "exit" signal occurs.

Each Qt application contains one global event loop with `QApplication` instance as managing entity (entity which caused the creation of the event loop). Consider Listing 23. Call on line 8 results in entering to the global event loop, so that events from application objects, e.g. from main application window, can be processed.

Listing 23: Global event loop

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

    // Display main application window here.
    .............

    // Trigger global QApplication event loop.
    return a.exec();
}
```

Listing 24: Typical event loop scheme

```
while() {
    check_queue_of_pending_events;
    process_all_pending_events;
    remove_processed_events_from_the_queue;
    if (exit) {
        return status;
    }
}
```
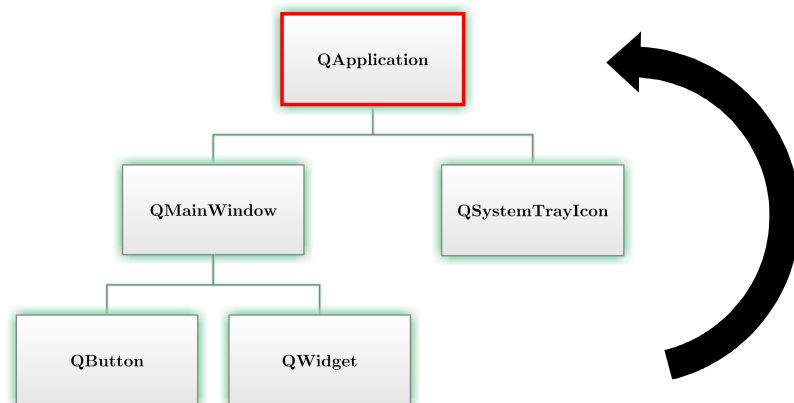

Figure 8.1: Typical event loop

Let's take a look at Figure 8.1 which displays typical structure of Qt application. Arrow indicates position of supervising entity (entity which triggered event loop). If QSystemTrayIcon notices that certain event happened in it, QApplication

instance is notified about the situation and is given information about:

- type of event,

- event properties,

- sender.

Event is posted by QSystemTrayIcon and appended to event loop queue for further processing and QSystemTrayIcon possibly waits for backward event delivery from the event loop.

QApplication's event loop iterates and finds new unsolved events which are eventually removed from the queue and sent to their senders. QSystemTrayIcon then *handles* the event and responds with expected behavior which finally concludes life of this event.

Handling event means calling *event handler* – some kind of special method. If you handle some events in one object, sometimes you need to *propagate* the same event in another object too. For example if you move your cusor over QPushButton (ordinary button) instance, paint-event is raised, because QPushButton needs to be repainted on the screen, and this event is propagated to parent object which is usually application window which eventually needs to redraw itself too. Some events are propagated and other ones are not.

Easiest way to handle events in Qt is reimplementing available event handlers. Each QObject-based Qt class offers specific handlers. All handlers are declared as protected methods. See Listing 25 (example sources/laboratory/12-child-event) for typical extension of event handler. In this case, we reimplemented behavior of child-event which is triggered if child is added (removed) to (from) another QObject instance. Event handler of superclass is called on line 35 because we want to only extend the original handler and not to replace its behavior completely. This approach is common in Qt and is used especially for GUI-related events.

Listing 25: Reimplementing event handler

```
1  // myqobject.h
2  class MyQObject : public QObject{
3      Q_OBJECT
4
5      public:
6          explicit MyQObject(QObject *parent = 0);
7
8      protected:
9          void childEvent(QChildEvent *event);
10 };
11
12 // myqobject.cpp
```

73

```
13  #include <QChildEvent>
14
15  #include "myqobject.h"
16
17
18  MyQObject::MyQObject(QObject *parent) : QObject(parent) {
19  }
20
21  void MyQObject::childEvent(QChildEvent *event) {
22      if (event->added()) {
23          qDebug("Child␣%s␣(%s)␣was␣added␣to␣%s.",
24              event->child()->metaObject()->className(),
25              qPrintable(event->child()->objectName()),
26              qPrintable(objectName()));
27      }
28      else if (event->removed()) {
29          qDebug("Child␣%s␣(%s)␣was␣removed␣from␣%s.",
30              event->child()->metaObject()->className(),
31              qPrintable(event->child()->objectName()),
32              qPrintable(objectName()));
33      }
34
35      QObject::childEvent(event);
36  }
```

8.1   Event filters

Classic event handlers are unusable if you want to handle all events in one place. One solution is to use generic `bool` `QObject::event(QEvent*)` event handler which catches all events. Another solution is event filtering. Event filter is ordinary method which accepts destination `QObject` instance and event object. Listing 26 shows the approach quite clearly.

Listing 26: Using event filter

```
1   // myqobject.h
2   class MyQObject : public QObject {
3       Q_OBJECT
4
5       public:
6           explicit MyQObject(QObject *parent = 0);
7
8       protected:
9           bool eventFilter(QObject *object, QEvent *event);
10  };
```

```cpp
// myqobject.cpp
#include <QEvent>

#include "myqobject.h"


MyQObject::MyQObject(QObject *parent) : QObject(parent) {
    installEventFilter(this);
}

bool MyQObject::eventFilter(QObject *object, QEvent *event) {
    qDebug("Event_happened_in_%s.", qPrintable(object->objectName()))
        ;

    if (event->type() == QEvent::ChildAdded) {
                qDebug("Observing_child-event_for_%s_and_child_%s.",
            qPrintable(object->objectName()),
            qPrintable(static_cast<QChildEvent*>(event)->child()->
                objectName()));
    }

    return QObject::eventFilter(object, event);
}
```

8.2   Events and QObject instance lifetime

QObject instances are capable of sending and receiving events. One instance can be addressed by hundreds of pending events. What if that QObject instance is scheduled for deletion and some events aren't processed yet?

Remaining events need to be processed before objects is deleted. This is done automatically by parent event loop. Listing 27 displays enhanced event loop. Objects from current context are checked and if some of them are scheduled for deletion, then they are freed. That happens after all events are consumed.

Listing 27: Enhanced event loop scheme

```
while() {
    check_queue_of_pending_events;
    process_all_pending_events;
    remove_processed_events_from_the_queue;

    foreach (QObject object in event_loop_context) {
        if (object.is_scheduled_for_deletion) {
            object.delete;
```

```
 9              }
10          }
11          if (exit) {
12              return status;
13          }
14 }
```

This approach for deleting `QObject`-based instances is used automatically and the only thing the programmer needs to do is to call `object.deleteLater()` with particular object.

Programmer can use classic `delete` operator too but that's risky. If there are pending events with deleted object as recipient, then these events are delivered to unallocated memory and segmentation fault occurs.

Chapter 9

Threading

Every modern and flawless application needs to superimpose its functionality into several layers. One layer often represents GUI, while another takes care of storing application data in a database or provides some kind of extensive mathematical computations. Dividing application functionality into separated blocks is a good approach. Paramount technique for this approach is called *threading*.

9.1 What is thread?

Operating system has only one duty – it must allow client programs to run. But programs cannot run simultaneously, so only one program is run at a time and other programs are forced to wait until it's their turn. They are patiently waiting in the queue for their chance to become active and do the job they are asked to do.

Operating system allocates very tiny amount of time for each program and switches among them very swiftly. Each program usually lives for several milliseconds in one round. This circling among programs (sometimes called *processes*) is called *multitasking*.

Moreover, every program can be divided into several parts which can run independently. Operating system, in fact, cycles among these program parts instead of whole programs. Independently-runnable part of the program can be called *thread of execution*. Each and every computer program contains at least one thread. Important thing about threads is that all threads of one process share its resources – joint data. Shared data allows you to make your threads cooperate with each other. Threads are used for many purposes:

COMMUNICATION

Threads are used to handle communication among entities like web servers or other remote devices.

INTERFACE LATENCY

If you do not separate extensive computations from your GUI then GUI may be blocked when those computations are performed. Threads are used to separate computations from GUI. Interface thread is no longer blocked by computations and no freezing occurs.

PARALLEL COMPUTING

Certain formulas are very difficult to solve in reasonable time and the only way to make computations faster is to make them running collaterally.

GOOD HABIT

Experienced programmer always tries to divide application functionality into well-formed and rationally-built parts. Threading is great and powerful technique to achieve that.

9.2   Threading and operating system

Threading was added to some operating systems additionally. The only way to do that was via library. This was the case of POSIX Threads (PThreads) on Linux. Threads were not part of the official Linux concept and became available later.

9.3   Threading in Qt

PThreads library is used as the backend for Qt threading support on Linux and Windows. Threading is very complicated and complex subject but at least basic usage (which is fine for most users) can be introduced.

Base class for working with threads is `QThread`. (Qt-Project, 2012b, QThread class) `QThread` should **never** be subclassed. If you want to run code of any class in separated thread, then you need to:

1. Derive your class from `QObject` and add signals/slots of your desire to it.

2. Create new `QThread` instance.

3. Call `void QObject::moveToThread(QThread *targetThread)` with your class as `this` pointer.

There is no need to do this procedure for static methods or global functions. You can use `QFuture<T> QtConcurrent::run(void *function)` to run those. (Qt-Project, 2012b, QtConcurrent) See Listing 28 (example `sources/laboratory/14-threading`) for more.

Listing 28: QtConcurrent usage for global or static functions

```cpp
void faktorial(int input) {
    qDebug().nospace() << "Factorial from thread " << QThread::
        currentThreadId() << ".";

    if (input < 0) {
        return;
    }

    int result = 1;
    for (int i = 1; i <= input; i++) {
        result *= i;
    }

    qDebug() << result;
}


int main(int argc, char *argv[]) {
    QCoreApplication a(argc, argv);

    // Number of main thread handle?
    qDebug() << "Main thread is " << QThread::currentThreadId() << ".
        ";

    // Use QtConcurrent for glogal function.
    QFuture<void> result = QtConcurrent::run(faktorial, int(5));
    result.waitForFinished();

    return a.exec();
}
```

Advance usage of threading is introduced in part Real-world Qt.

Part II

Real-world Qt

Chapter 1

Foreword

1.1 What is covered by this part?

This part is different from the previous one. In this part, you will be guided through development of more complex Qt-based application. Important parts of development process will be discovered, including initial ideas, solving dependencies, licensing, programming the application, translating or publishing.

Someone could state that licensing issues or publishing the application are not important topics. Programmer's main task is to program the application correctly. That's true but programmer needs to take advantage of other tasks too because only if he/she deals well with all of them, good application can emerge.

There is quite good documentation (Qt-Project, 2012b) available for Qt but it lacks some parts which are covered by this part of the book. We will see how to automatically build our software with needed customizations or how to make our software available to users.

1.2 Lifecycle of typical application

Development of typical application starts if someone needs something. Photographer wants special photo-editing software which doesn't seem to be available. Or perhaps teacher needs to have specific agenda application for his particular needs. Programmers often program new software for themselves. So applications are born due to various (and usually rational) reasons.

Before the application is actually written, there must be certain research done by the programmer. See Chapter 2 for more about this topic. Special "50:50" rule applies here. It's unwritten rule that you (as the programmer) should spent at least 50% of your time (dedicated for development of particular software) for research. You should think about problems and choose right algorithms. The

other 50% is meant to be time for writing your ideas in programming language. Most huge failures roots in shoddy research.

Chapter 2

Creating new applications

There is usually need for something useful in the beginning. Sample application for this book is the same case. It is called Qonverter and it is supposed to be the simple calculator with some unusual features.

One Step Further
Obtain the source code of Qonverter from (Rotter, 2013).

Some time ago, I needed calculator with ability to do some more advanced (but still simple) operations. I needed to compute factorials or medians. I also needed to do bitwise operations on numbers such as shifting or logical conjunction and disjunction along with usual mathematical operations. Moreover, I needed to do unit and currency conversions sometimes.

I searched for calculator application which could offer such a functionality to me. No satisfying application was found. Unit (currency) converter was usually missing. Many calculators don't offer specific mathematical functions I like to use from time to time.

So I decided to write my own calculator which provides function I need.

2.1 Choosing programming language

I am not skilled programmer but I know C++ programming language. Choice of programming language is extremely important. You may take a look at chapter C plus plus as base stone to see C++ features. C++ is fairly fast so it is a good choice for this kind of application.

I decided to use newest (as of January 2013) C++ version called C++ 11. It offers many features (some already mentioned in this book) which can bring me many advantages:

- shorter code,

- easier-to-understand code,

- (perhaps) greater performance.

Many disadvantages can arise:

- lack of support by C++ compilers,

- lack of support by libraries.

There are many C++ compilers to choose from. I want to keep my Qonverter multiplatform because I use both Linux and Windows operating systems. GCC is main compiler for Linux. Latest GCC (version 4.8.0, as of March 2013) is known to support all major features of C++ 11.

Several great compilers are available for Windows platform. I could use MSVC 2010 compiler but it lacks some C++ 11 features so latest Minimalistic GNU for Windows (MinGW) compiler was chosen instead. MinGW is GCC-based compiler. Mac OS X operating system uses Clang compiler which is known to support C++ 11 too (Wikipedia, 2013a). So hypothetically, these operating systems might be supported:

- Windows,

- GNU/Linux,

- Mac OS X.

In fact, even more operating systems can be supported in the future, because Qt is ported for example to eComStation (Wikipedia, 2013b) operating system.

2.2   Choosing libraries

I had picked compilers and programming language in five minutes. It wasn't the big deal. I had to choose libraries too and that was the big deal. I knew just few libraries but only one which provides me with ability to build GUI. It was Qt. Latest Qt (Qt 5) is known to work with C++ 11. It just came out when i decided to create Qonverter (spring of 2012) so I decided to use it.

One last piece of puzzle needed to be found – a library for mathematical core of the application. I was searching for the right one for one week and nothing great appeared. Eventually, I discovered muParserX library (Berg, 2013) which is developed by Ingo Berg.

Chapter 3

muParserX library

Library looked great but it had several disadvantages. I compiled it and some features were missing. I sent email with some minor code tweaks to Ingo Berg and he allowed me to cooperate with him on the project. It was great news for me.

muParserX supports all major compilers and is even compilable with C++ 11. Main prerequisites were met and I started to cooperate on the project. Some important attributions done will be mentioned later. muParserX library offers many fancy features:

- support for scalar value types, matrices, booleans and strings,

- ability to write custom operators and functions,

- ability to define variables and constants,

- many other features, including reverse Polish notation (RPN) representation if math formula,

- ability to define functions with varying count of parameters.

These five facts were very important for me as they allowed me to start thinking about brand new calculator application that suits my needs. However, muParserX wasn't finished and rock-solid project. Many things needed to be solved. Ingo Berg led (and still leads) the way of muParserX development but two pairs of eyes see more. Code fixing, new ideas or debugging are important tasks too. Help is always needed.

Some functions (important for me) were missing in the library, so i simply added them. For example factorial function or percentage operator % were added and many other functions were rewritten or tweaked.

I participated in new design for internationalization which is important for every developer. Typical user expects that software "talks" in his native language.

I changed behavior of definition of constants, variables, functions and operators along with other minor things.

Chapter 4

Writing Qonverter

Libraries and compiler are selected. Let's think about application. Let's assume we need to write simple and easy-to-use calculator application, which should run on Windows and Linux. We already know something about Qt and we are able to seek for needed information in (Qt-Project, 2012b). Qonverter should run in single window mode but we may eventually need to hide it into tray area. So tray icon functionality has to be provided.

One Step Further
Use source code of Qonverter tu understand this part of the book. Source code is commented and this part of the book is (intentionally) just collection of hints on how to build your own Qt application.

Qonverter should be translated to English (as it is globally spoken language) and Czech which is the native language of the author of this book. Qonverter should fit into desktop environments so native icon themes with fall-back theme should be used. Installation of Qonverter should be as easy as possible. This means simple ZIP archive for Windows and native package for Linux distributions. More specific feature list for Qonverter could look like this one:

- use muParserX as mathematical core,

- offer unit and currency converters,

- support on-line currency rates synchronization for currency converter,

- allow conversions of mathematical formulas for unit converter (This means that for example formula $5 + 7$ is computed first and result is then converted as needed.),

- calculator is simple,

- most used functions are accessible via calculator keypad,

- input text box for mathematical formulas in calculator is multi-line,

- language of the application can be switched manually,

- free software license is used,

- user can define variables,

- many built-in constants,

- application depends only on Qt and muParserX.

These are primary goals. Optional goals appear during the course of development.

4.1 Qonverter structure

Qonverter application consists of two main parts:

CORE

Is based on muParserX library and is responsible for providing results of input formulas. Logic of currency (unit) converter is separated into core too. Core doesn't depend on graphical interface of the application.

GUI

Forms visual part of the application and provides user with main application window and switchable tray icon.

4.2 Programming application core

We need to wrap muParserX library and make reasonable subset of its functionality available through Qt-friendly interface. This is done by `Calculator` class. One of its main goals is to provide tools for numerical computations. muParserX library originally does computations via `ParserX` class. Its typical usage is shown in Listing 29. We see that we created `ParserX` instance and set the expression. Then we tried to evaluate it. Any exception is caught and error description is written to the standard output.

Listing 29: Basic ParserX class usage

```
ParserX m_parser;
m_parser.SetExpr("5+7");
```

```
 3
 4  Value result;
 5  try {
 6      // Evaluate the expression.
 7      result = m_parser.Eval();
 8  }
 9  catch (ParserError &e) {
10      qDebug("Error_occurred.");
11  }
12  // The 'result' variable contains result of computation.
```

Calculator class should provide enhanced approach for numerical computations and it does as it offers method void calculateExpression(Calculator::CallerFunction function, QString expression) method. This method takes any mathematical expression in textual form and identification of target function.

Note that numerical computations may take some time. Thus, we need to calculate expressions asynchronously.

Calculator class will be used in the *singleton* pattern and will be separated in its own thread by CalculatorWrapper class which manages Calculator singleton instance. This class basically just starts the thread with calculator and quits it if needed (Listing 30).

Listing 30: Running calculator in separated thread

```
 1  // CalculatorWrapper constructor.
 2  CalculatorWrapper::CalculatorWrapper(QObject *parent) : QObject(
      parent) {
 3    // Create calculator.
 4    m_calculator = new Calculator();
 5
 6    // Create separate thread for calculator.
 7    m_thread = new QThread();
 8
 9    // Prepare calculator for usage in separate thread.
10    m_calculator->moveToThread(m_thread);
11
12    // Connect thread to calculator.
13    connect(m_thread, &QThread::started, m_calculator, &Calculator::
        initialize);
14    connect(m_thread, &QThread::finished, m_thread, &QThread::
        deleteLater);
15  }
16
17  // CalculatorWrapper destructor.
18  CalculatorWrapper::~CalculatorWrapper() {
19    qDebug("Deleting_calculator_wrapper.");
20
```

```
21   m_thread->quit();
22   m_thread->wait(1000);
23
24   delete m_calculator;
25 }
26
27 CalculatorWrapper &CalculatorWrapper::getInstance() {
28   if (s_instance.isNull()) {
29     s_instance.reset(new CalculatorWrapper());
30     s_instance.data()->m_thread->start();
31   }
32
33   return *s_instance;
34 }
```

Qonverter builds on muParserX in aspect of variables, functions and constants. Qonverter wraps all these entities in single structure called `MemoryPlace`. It holds information about the actual type of underlying entity.

```
1 enum Type {
2   CONSTANT = 0,
3   IMPLICIT_VARIABLE = 1,
4   EXPLICIT_VARIABLE = 2,
5   SPECIAL_VARIABLE = 3,
6   FUNCTION = 4
7 };
```

Structure `MemoryPlace` (Listing 31) defines properties used for constants, variables or functions, including name and description. `MemoryPlace` can encapsulate these kinds of entities:

CONSTANTS

Constants are named values. Values of constants do not change. Constants are special numbers which are interesting in some way and are used extensively in mathematical computations.

FUNCTIONS

Functions are stored as `MemoryPlace` instances too. Each function is described by its name and description.

VARIABLES

We can divide variables into three categories:

1. IMPLICITLY-CREATED VARIABLES

   Those are created during evaluation of mathematical expression by calculator engine.

2. EXPLICITLY-CREATED VARIABLES

Explicitly-created variables are defined manually by application user.

3. CRITICAL VARIABLES

Critical variables are just ordinary variables with one exception – they cannot be deleted. They are used for storing last two successfully calculated results (variables `ans` and `ansx`) and for main memory variable `m`.

Listing 31: Declaration of MemoryPlace class

```cpp
// Represents entity which as able to hold value.
// This includes calculator variables and constants.
struct MemoryPlace {
    QString m_name;
    QString m_description;
    Value *m_value;
    Type m_type;

    // This is used only by variables, each constant has m_variable
    //    equal to nullptr.
    // So there is way to distinguish variables from constants.
    Variable *m_variable;

    // Constructs "empty" variable.
    // This constructor is used for constructing
    // "shallow" clones of implicitly-created variables.
    MemoryPlace(const QString &name);

    // Creates new variable or constant
    MemoryPlace(const QString &name, const QString &description,
                const Value &value, const Type &type);

    // Destructor.
    ~MemoryPlace();
};
```

Declaration of this class is straighforward but let's take a look at destructor implementation (Listing 32).

Listing 32: MemoryPlace class destructor

```cpp
MemoryPlace::~MemoryPlace() {
  // Free resources of this object if:
  if (m_type == CONSTANT || m_type == FUNCTION) {
    delete m_value;
    qDebug("Constant '%s' deleted.", qPrintable(m_name));
```

```
 6    }
 7    else if (m_type == EXPLICIT_VARIABLE || m_type == SPECIAL_VARIABLE)
          {
 8      delete m_value;
 9      delete m_variable;
10      qDebug("Variable_'%s'_deleted.", qPrintable(m_name));
11    }
12    else {
13      qDebug("Implicitly-created_variable_'%s'_deleted.", qPrintable(
          m_name));
14    }
15 }
```

We see that destructor looks complicated because freeing of `m_value` and `m_variable` properties is conditional. In fact, constants and functions do not use `m_variable` property which is not freed in those cases. Also note that nothing is freed from the memory in case of implicit variables. Properties `m_value` and `m_variable` of implicit variables are controlled and freed by automatic pointers.[1]

4.2.1 Discovering implicitly-created variables

New variables can be defined during evaluation of any mathematical expression. This variable is stored in calculator engine and is automatically deleted when engine gets freed from the memory. We need to track these implicitly-variables down too because we want to work with them. The only way to discover new variables is to go through all variables from muParserX engine when any computation finishes. `Calculator` class uses method `calculateExpression(...)` to do computations. This method computes input formula and then scans for new variables. It calls method shown in Listing 33 to do the job. So, implicitly-created variables are not lost and user can interact with them.

Listing 33: Scanning for implicitly-created variables

```
1 void Calculator::consolidateMemoryPlaces() {
2   var_maptype vmap = m_parser->GetVar();
3   QString variable_name;
4
5   // Go through all defined variables.
6   for (var_maptype::iterator item = vmap.begin(); item!=vmap.end();
       ++item) {
7     Variable &var = (Variable&) *(item->second);
```

---

[1]Automatic pointer is clever object which encapsulates ordinary pointer in C++. Instance of automatic pointer calls `delete` operator with the underlying pointer if that particular instance goes out of the execution scope. (Prata, 2011, p. 969-978)

```
 8
 9     variable_name = QString::fromStdWString(item->first);
10
11     if (!m_memoryPlaces.contains(variable_name)) {
12       // We found variable which exists in calculator engine,
13       // but is not in external calculator list, ergo, this variable
             was
14       // implicitly created during calculator engine lifetime.
15       m_memoryPlaces.insert(variable_name, new MemoryPlace(
           variable_name));
16       m_memoryPlaces[variable_name]->m_value = (Value*) var.GetPtr();
17       m_memoryPlaces[variable_name]->m_variable = &var;
18       m_memoryPlaces[variable_name]->m_type = MemoryPlace::
           IMPLICIT_VARIABLE;
19     }
20   }
21 }
```

4.2.2   Model for collection of constants, variables and functions

`Calculator` class keeps track of all constants, variables and functions and offers this collection through custom *model*. Take a look at (Qt-Project, 2012b, keyword Model/View Programming) before you proceed.

Model is contained within the class `ConstantsModel`. Name of the model is misleading, it contains variables, functions and constants. This model implements standard interface from `QAbstractListModel` which contains following methods:

```
1 int rowCount(const QModelIndex &parent) const;
2 int columnCount(const QModelIndex &parent) const;
3 QVariant data(const QModelIndex &index, int role) const;
4 QVariant headerData(int section, Qt::Orientation orientation, int
    role) const;
5 QModelIndex index(int row, int column = 0, const QModelIndex &parent
    = QModelIndex()) const;
```

Row count equals to sum of counts all variables, constants and functions. Column count equals to 4 because model provides name, description, value and value type for each `MemoryPlace` instance. This model forms the only interface which can be used by other Qonverter components to gather information about variables, constants or functions. It is used by auto-completion feature and by overview dialog as you will see in Chapter 4.3.5.

The most important part of this model is the `data(...)` method. This method (Listing 34) returns data elements for each column/row.

Listing 34: Implementation of data provider in ConstantsModel

```
1  QVariant ConstantsModel::data(const QModelIndex &index, int role)
       const {
2    switch (role) {
3      case Qt::ToolTipRole:
4      case Qt::DisplayRole:
5      case Qt::EditRole:
6        switch (index.column()) {
7          case (int) ConstantsModel::DESCRIPTION: {
8            // Return description.
9          }
10         case (int) ConstantsModel::VALUE_TYPE : {
11           // Return type of variable/constant value.
12         }
13         case (int) ConstantsModel::VALUE: {
14           // Return value of variable/constant.
15         }
16         default:
17           // Return other needed data.
18       }
19       break;
20     // This role is used to return raw variable/constant/function
           information.
21     case Qt::UserRole:
22       // Return raw (unmodified) data of variable/constant/function.
23     default:
24       return QVariant();
25   }
26 }
```

We can see that method is divided into several parts. All roles are used to return human-readable representations of a variable/constant/function. `Qt::UserRole` is different. It is used to return original raw data which are not formatted.

4.2.3   Programming unit/currency converter

Unit converter and currency converters use very similar approach with one exception. Logic of converters is not separated in threads. It is not needed because calculations done in them are very simple and fast. Currency converter is represented by `CurrencyConverter` class and unit converter is represented by `UnitConverter` class. Both classes encapsulate list of coefficients for currencies/units and then do various multiplications to produce desired results. Check out the source code for more information.

4.3 Programming GUI

Let's focus on GUI of Qonverter in high detail. Everyone agrees that calculator application requires quite specific user interface. Many calculators offers skinnable look with "better" user experience but most of those calculator applications don't offer native look on each and every supported platform.

There was one simple task to be done in the area of skins and styles. Qonverter should support some ways of skinning but native looks & feels should be available too.

4.3.1 Qt style sheets

Qonverter uses Qt style sheets (Qt-Project, 2012b, style sheets) along with dynamic QStyle-based styles loading. Qt style sheets follow Cascading Style Sheets (CSS), specification 2.1. Qonverter uses specifically tweaked style sheets which are parsed in run time to allow loading of images from relative paths.

Listing 35: Loading and parsing of skin file

```
QTextStream str(&skin_file_name_full_path);
QString skin_data;

// Read skin data from file and close it.
skin_data.append(str.readAll());
skin_file_full_path.close();
skin_file_full_path.deleteLater();

// Here we use "/" instead of QDir::separator() because CSS2.1 url
    field
// accepts '/' as path elements separator.
skin_data = skin_data.replace("##",
            APP_SKIN_PATH + "/" + skin_folder + "/images");

// Set skin to application.
qApp->setStyleSheet(skin_data);
```

Skin file is loaded and its content is stored in single QString instance. Then parsing is done. All references to external files are refreshed to point to correct files. Typical Qonverter style sheet fragment looks like the one in Listing 36.

Listing 36: Typical Qonverter style sheet

```
...
QLineEdit[readOnly="true"] {
    color: gray;
    font-weight: lighter;
```

```
 5  }
 6
 7  /* spin boxes and other stuff */
 8  QDoubleSpinBox {
 9      background-color: qlineargradient(x1: 0, y1: 0, x2: 0, y2: 1,
           stop: 0 #ffffff, stop: 1 #f7f7f7);
10      border-radius: 1px;
11      border: 1px solid gray;
12  }
13
14  QCheckBox::indicator:checked {
15      image:url(##/checkbox.png);
16  }
17  ...
```

"##" is special mark which represents absolute path to global Qonverter skins storage which is set in compile time and is platform-dependent as seen in Listing 37. For more inspiration, check collection of skins included in Qonverter source code.

Listing 37: Paths to global skins storage

```
 1  #if defined(Q_OS_LINUX)
 2  #define APP_SKIN_PATH APP_PREFIX + QString("/share/qonverter/skins")
 3  #elif defined(Q_OS_MAC)
 4  #define APP_SKIN_PATH QApplication::applicationDirPath() + "/../
       Resources/skins"
 5  #elif defined(Q_OS_WIN) || defined(Q_OS_OS2)
 6  #define APP_SKIN_PATH QApplication::applicationDirPath() + QString("/
       skins")
 7  #endif
```

Qonverter supports dynamic loading of external Qt plug-ins. They can be loaded from platform-dependent directory path. On Windows, this path equals to executable file path. Linux uses default system paths for finding Qt plug-ins. On Mac OS X, however, extra path is provided (Listing 38).

Listing 38: Settings up path for dynamic plug-ins loading

```
 1    // Add 3rd party plugin directory to application PATH variable.
 2    // This is useful for styles, encoders, ...
 3    // This is probably not needed on Windows or Linux, not sure about
         Mac OS X.
 4  #if defined(Q_OS_MAC)
 5    QApplication::addLibraryPath(APP_PLUGIN_PATH);
 6  #endif
```

4.3.2 Calculator button layout

User interface layout represents critical point of calculator user interface. Many calculators support so-called "modes". Calculators do include "scientific" mode or perhaps "basic" mode. Each mode display different set of buttons in the calculator window which is not good. I prefer static interfaces, so that user memorizes exactly one mode. All buttons have fixed position. The only thing, that changes, is appearance of buttons.

As we know, Qonverter supports skins. Some skins represent pure native look & feel. Other ones may tweak user interface to bring new features and one of them is graphical distinction of calculator buttons. Each button provides specific functionality but some buttons do very similar jobs. For example calculator buttons "5" and "6" do almost the same thing, thus, they are related to each other. On the other hand, "max" button does completely different job than does the "=" button.

Buttons can be separated into groups and they really are in Qonverter. Each calculator button carries special flag, which exposes purpose of the button (Listing 39). Type of button is available via dynamic property.

Listing 39: Types of calculator buttons

```
// Here are possible types of each CalculatorButton instance.
enum Type {
    NUMBER   = 0,
    OPERATOR    = 1,
    FUNCTION    = 2,
    SOLVER   = 3,
    COMPARE = 4,
    CONTROL = 5,
    BIT      = 6
};

// Marking some buttons as "numeric" buttons.
QList<CalculatorButton*> but_numbers;
but_numbers <<  m_ui->m_btnOne << m_ui->m_btnTwo <<
                m_ui->m_btnThree << m_ui->m_btnFour <<
                m_ui->m_btnFive << m_ui->m_btnSix <<
                m_ui->m_btnSeven << m_ui->m_btnEight <<
                m_ui->m_btnNine << m_ui->m_btnZero <<
                m_ui->m_btnDot;

// Setting property for each button in "numeric" button group.
foreach (CalculatorButton *btn, but_numbers) {
    btn->setProperty("type", (int) CalculatorButton::NUMBER);
}
```

Qonverter style sheet can (but doesn't have to) take advantage of calculator buttons resolution and highlight each group of buttons differently. That's what "Modern" skin does. Modern skin file can be found in `resources/skins/base` subdirectory of Qonverter source code tree. This skin contains just basic enhancements for user interface plus distinctive coloring (Listing 40) for calculator buttons.

Listing 40: Calculator button coloring style sheet

```
1  /* some code here */
2  .....
3
4  /* colors for calculator buttons */
5  CalculatorButton[type="2"] {
6      background-color: rgb(245, 245, 245);
7  }
8  CalculatorButton[type="2"]:hover {
9      background-color: qlineargradient(x1: 0, y1: 0, x2: 0, y2: 1,
           stop: 0 #d7d7d7, stop: 1 #e6e6e6);
10 }
11 CalculatorButton[type="3"] {
12     background-color: rgb(251, 153, 14, 240);
13 }
14 CalculatorButton[type="3"]:hover {
15     background-color: qlineargradient(x1: 0, y1: 0, x2: 0, y2: 1,
           stop: 0 #e89116, stop: 1 #fb9d18);
16 }
17
18 /* some code here */
19 .....
```

We see that each group of buttons can be highlighted differently in user interface. Figure 4.1 display Qonverter application with native (Windows 7) skin applied and with Modern skin applied. There is noticeable difference between these two skins. Left one looks natively while right one does not. It's matter of taste. Anyone with basic knowledge of CSS can write custom skins.

4.3.3 Advanced text boxes

There exists one Qt class called `QLineEdit`. It is used for single-line text boxes. However, I needed two new categories of advanced text boxes in Qonverter:

TEXT BOX WITH CLEAR BUTTON
>   This text box provides additional button. Contents of a text box is cleared when that button is pressed.

TEXT BOX WITH STATUS REPORTING

(a) Native skin   (b) Modern skin
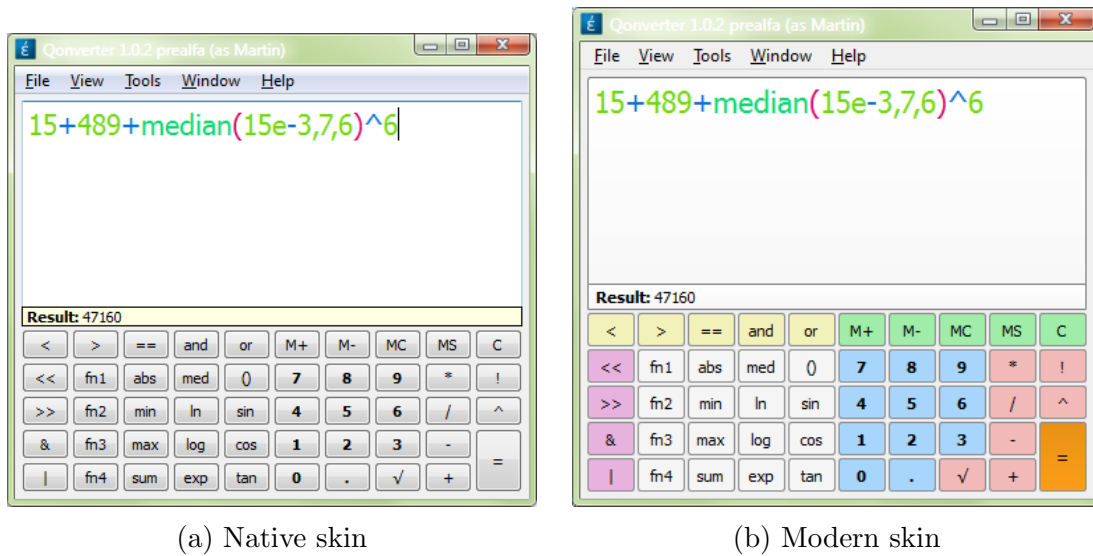
Figure 4.1: Skinned Qonverter

This text box offers extra icon. This icon indicates if contents of text box is valid.

4.3.3.1 Creating the text box with clear button

My custom text box component is called `LineEdit` and is based on (Ramakrishnan, 2007). Nonetheless, my version is more robust, clear button is switchable and read-only/disabled mode is supported. Take a look at Qonverter source code, class `LineEdit` for more information.

4.3.3.2 Creating the text box with status reporting

This type of text box builds on `LineEdit`. It is contained in the class called `MarkedLineEdit`. This component displays custom status icon which can be switched programmatically. See source code of the class for more information. You can see this component in action in unit converter function of Qonverter.

4.3.4 Tray icon and desktop integration

Desktop applications can be divided into two groups:

1. Applications which are executed manually if needed. They are closed after usage.

101

2. Applications which run constantly. They can be hidden (typically) into notification area of a desktop environment.

We probably use some applications very rarely, so they don't need to run all the time. We simply open them, do needed job and then close them. It's sometimes better to keep certain applications in main memory and hidden so that they don't disturb us. Tray icon is amazing tool to achieve that. This is typical for applications which are used regularly.

Qonverter supports both modes. If we want to use it irregularly and not too often, then Qonverter can run in single-window mode and quit if window gets closed. But if we are advanced users who use calculator often, then Qonverter can be hidden to a tray area and only a tray icon remains visible. Interaction with a tray icon is critical because its the only user interface element visible if application windows are hidden.

4.3.4.1 Single-window mode

This is the default mode for Qonverter. It is also used in desktop environments which don't support tray icon mode. This mode offers standard windows and dialogs. Qonverter exits when last window is closed by a user.

4.3.4.2 Tray icon mode

Tray icon mode (Figure 4.2) offers greater functionality. We can use windows and dialogs again. But Qonverter doesn't have to (but can) quit if last window is closed. It can minimize itself into notification area, resulting in last visible element – the tray icon. User can switch between modes freely via configuration dialog (Figure 4.3).
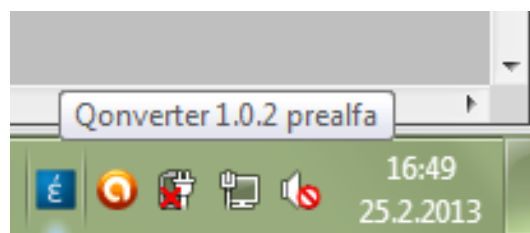
Figure 4.2: Qonverter tray icon
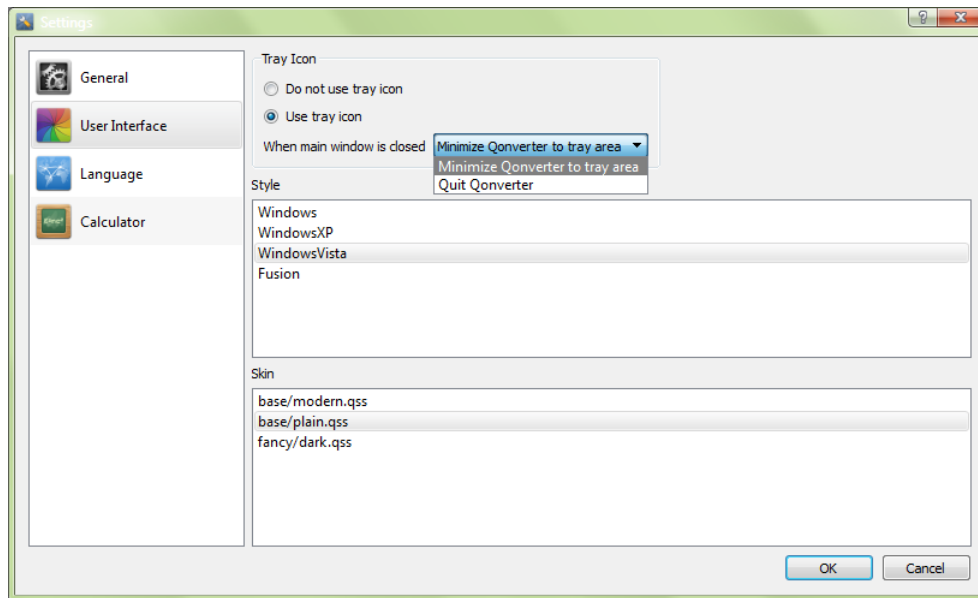
Figure 4.3: Qonverter mode selection

4.3.5 Displaying available functions, constants and variables

`Calculator` class offers information about constats, variables and functions via `ConstantsModel` class. It is used by functions/variables/constants overview dialog (see Figure 4.4). Model provides table-like data. Dialog uses `ConstantsView` class to display those data.

4.3.6 Auto-completion

Auto-completion (see Figure 4.5) is known from advanced text editors or integrated development environments. It is usually displayed as a "floating" panel. User writes some text and auto-completion panel offers him available completions. This mechanism has use cases in calculator applications too. User doesn't have to remember names of built-in functions. Moreover, auto-completer can offer names of variables and constants.

4.3.7 Unit and currency converter

Unit converter doesn't contain special elements, except one thing. It doesn't contain single button for triggering a conversion. All conversions are done on-the-fly.
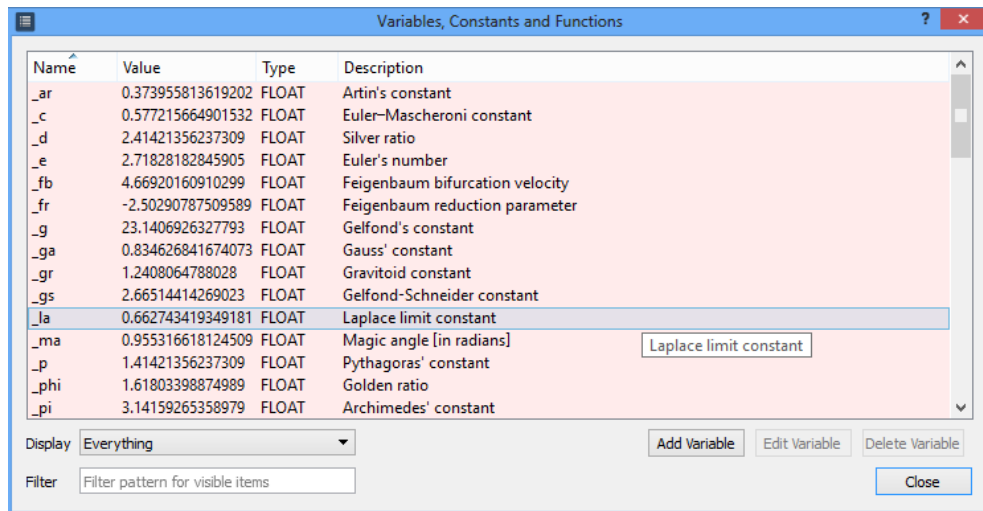
Figure 4.4: Dialog with overview of variables, constants and functions

User gets hints about what to do via placeholder texts of input controls (Figure 4.6).
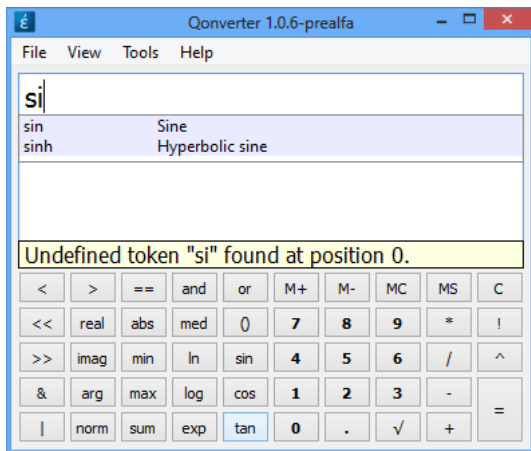
(a) Two functions in completion list
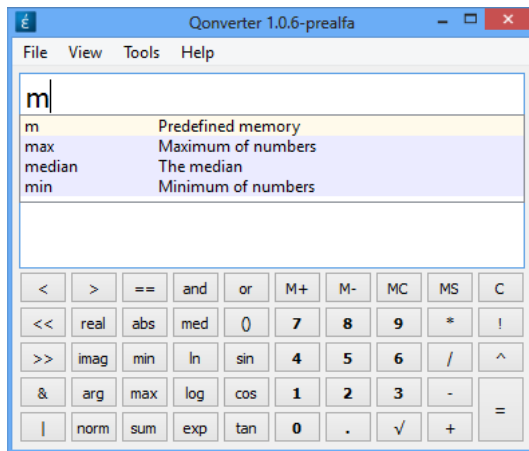
(b) Two functions and one variable in completion list

Figure 4.5: Auto-completion in Qonverter
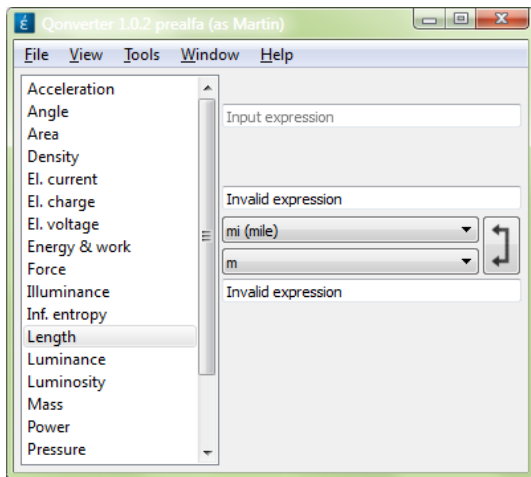
(a) Initial state
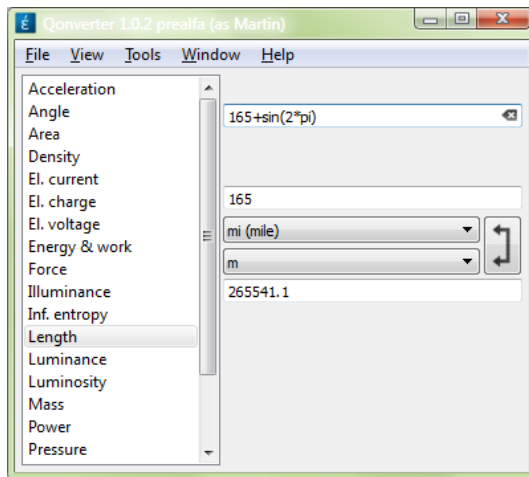
(b) With calculated and converted value

Figure 4.6: Unit converter overview

Chapter 5

Maintaining Qonverter

Many ambitious software projects offer applications of great quality but something is wrong. Even if rock-solid software is written, it's not used or well-known. Actually writing a piece of software is just starting phase of application life cycle. True merit comes from good application maintenance and deployment.

5.1   Storing source code

Reachable and efficient source code storage is critically important for collaborative application development. Event if we are lonely developers, we need to use safe storage to backup and protect our source code. Answer for this is version control system. There are some favorite systems such as Git (Torvalds, 2013) or Subversion (The Apache Software Foundation, 2012). One of these is a good choice. I recommend using Git because it's more robust, provides more functions and is supported by both Github (www.github.com) and Google Code (www.code.google.com) services.

Google Code and Github are free hosting services to hold and protect your source code. Moreover, each and every user is provided with ability to present his (or her) software product and share publicly.

Let's suppose we have written our software and we need to store its source code via Git:

1. Obtain Git client. Good choice for Windows is Cygwin (www.cygwin.com). Cygwin (Figure 5.1) is port of Bash to Windows operating system. It allows you to install Git during its installation.

2. Create new project on Google Code.

3. Navigate to root directory of your project and hit "git init" in Cygwin.

4. Navigate to ".git" subdirectory and edit "config" with text editor.

5. Change file to look similar to one in Listing 41. Section "remote" is important because it sets correct user name and password for your Google Code on-line Git repository.

6. Your local repository is ready for usage.

Listing 41: Git repository config file

```
[core]
    repositoryformatversion = 0
    filemode = false
    bare = false
    logallrefupdates = true
    ignorecase = true
[remote "origin"]
    fetch = +refs/heads/*:refs/remotes/origin/*
    url = https://GOOGLE-USER-NAME:GOOGLE-ACCOUNT-PASSWORD@code.
        google.com/p/PROJECT-NAME/
[branch "master"]
    remote = origin
    merge = refs/heads/master
```

One Step Further
Note that Google Code supports just open-source projects. Github supports both open and closed source projects.

5.1.1 Working with Git

We completed the setup of our imaginary project. Basically we need just three Git commands to start saving our code to on-line repository:

git add LIST-OF-FILES
> Adds selected files to new commit.

git commit -m 'Message'
> Creates new local revision from marked files.

git push origin master
> Uploads local revision to on-line repository.

See typical and very basic work with Git in Figure 5.2.
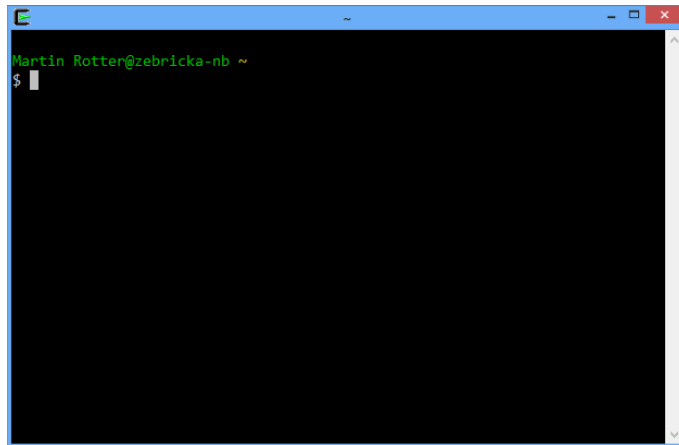
Figure 5.1: Cygwin user interface

5.2 Deploying Qt applications

Open-source projects have specific deployment phase. Compiled binaries aren't distributed regularly and user compiles his own executables directly from source code. Someone may think that special knowledge is needed to do that but it is not true.
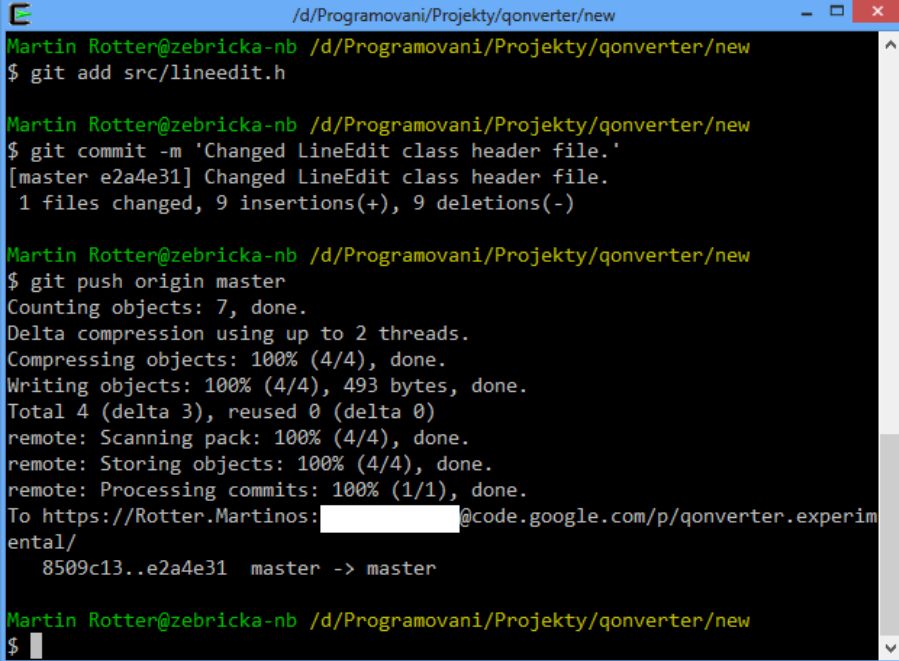
Nowadays, many semi-automatic build systems are available and it's good to use one. Qt applications require build system, which is cross-platform and supports all Qt versions. There is one build system that meets these criteria perfectly. It's called CMake (Kitware, 2013).

CMake takes care of application compilation and installation. It's easy to use. Programmer needs to write one script per software project and user runs just two commands in command line (for example in Cygwin) to install the software. CMake can be installed via Cygwin installation program.

5.2.1 Writing CMake application script

CMake script contains all information needed to setup and compile input source code. It can provide extra functionality for installing and packaging compiled files too.

CMake script is usually contained in single text file (called `CMakeLists.txt`) which is written by application programmer or distributor. Qonverter is not different. It offers CMake script too. It can be found on (Rotter, 2013).

Figure 5.2: Using Git from Cygwin

5.2.1.1 Basic data contained in CMake Qonverter script

Let's make a journey into the center of the `CMakeLists.txt` file of Qonverter. It contains collection of macros and command which guide potential user from compilation to installation of Qonverter. Even Qonverter translators can take advantage of this script, as we will see later. We split `CMakeLists.txt` into several fragments and investigate these.

Listing 42: Basic Qonverter information in CMakeLists.txt script

```
1  cmake_minimum_required(VERSION 2.8.9)
2
3  # Setup basic variables.
4  project(qonverter)
5  set(APP_NAME "Qonverter")
6  set(APP_LOW_NAME "qonverter")
7  set(APP_VERSION "1.0.2-prealfa")
8  set(APP_AUTHOR "Martin Rotter")
9  set(APP_URL "http://code.google.com/p/qonverter")
10
11 message(STATUS "[qonverter] Welcome to Qonverter compilation process
       .")
12 message(STATUS "[qonverter] Compilation process begins right now.")
```

```
13
14  .
15  .
16  .
17
18  # Find includes in corresponding build directories.
19  set(CMAKE_INCLUDE_CURRENT_DIR ON)
20
21  # Instruct CMake to run moc automatically when needed.
22  set(CMAKE_AUTOMOC ON)
```

We see (Listing 42) that `CMakeLists.txt` contains basic information assigned to variables. Moreover, it produces messages which are printed to the standard output if script is executed by CMake. Last line of Listing 42 is Qt-specific. It turns the meta-object compiler on. Meta-object compiler then goes through all Qonverter headers (which are specified in script too) and creates corresponding meta-object features if needed.

Listing 43: Compiler settings in CMakeLists.txt script
```
1   # Unicode settings.
2   set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -DUNICODE")
3   add_definitions(-DUNICODE -D_UNICODE)
4   if(WIN32)
5       # UNICODE support with Visual C++ and MinGW.
6       set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -D_UNICODE")
7       set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -D_UNICODE")
8   endif()
9
10  # Enable compiler warnings.
11  if(CMAKE_COMPILER_IS_GNUCXX)
12    add_definitions(-Wall)
13  endif()
```

Proper compiler settings (Listing 43) are important here. Qonverter is totally Unicode-based application because it allows special characters, such as square root character ($\sqrt{\ }$), to be used in calculator expressions.

Listing 44: C++ 11 check in CMakeLists.txt script
```
1   # Check for C++ 11 features availability.
2   if("${CMAKE_CXX_COMPILER_ID}" MATCHES "GNU")
3       execute_process(
4       COMMAND ${CMAKE_CXX_COMPILER} -dumpversion OUTPUT_VARIABLE
          GCC_VERSION
5       )
```

```
 6     if(NOT (GCC_VERSION VERSION_GREATER 4.7 OR GCC_VERSION
           VERSION_EQUAL 4.7))
 7         message(FATAL_ERROR "Your C++ compiler does not support C
               ++11.")
 8     else()
 9     add_definitions(-std=c++11)
10     endif()
11 elseif("${CMAKE_CXX_COMPILER_ID}" MATCHES "Clang")
12     set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -stdlib=libc++")
13 elseif(${MSVC_VERSION} VERSION_LESS 1600)
14     message(FATAL_ERROR "Your C++ compiler does not support C++11.")
15 endif()
```

Qonverter is written using C++ 11 and CMake script needs to ensure that selected compiler is valid. Unfortunately, there is no macro provided by CMake to do C++ 11 checks. Best way is to check versions of major compilers (MSVC, GCC, Clang) manually. These three compilers are the only ones tested by Qonverter. See Listing 44, it is initially written by Matthias Vallentin, all credits go to him.

One last elementary thing needs to be done. We need to load source files, headers and other needed files. It's simple, paths to files are added to variable, as seen in Listing 45. Headers and other files (skins, translations, . . .) are prepares in the same way.

Listing 45: Typical source files variable in CMakeLists.txt script
```
 1 # Add form files.
 2 set(APP_FORMS
 3     ui/formmain.ui
 4     ui/formabout.ui
 5     ui/formsettings.ui
 6     ui/calculator/formcalculator.ui
 7     ui/unitconverter/formunitconverter.ui
 8     ui/currencyconverter/formcurrencyconverter.ui
 9     ui/formvariables.ui
10 )
```

5.2.1.2   Advanced macros and packaging

We have set some basic stuff, compiler is detected and Unicode is enabled. We need to load Qt 5. CMake 2.8.9 (and later) supports Qt 5.

Listing 46: Detect and load Qt 5 in CMakeLists.txt script
```
 1 # Find all needed Qt modules.
 2 find_package(Qt5Sql)
```

112

```
3  find_package(Qt5Widgets)
4  find_package(Qt5Xml)
5  find_package(Qt5Network)
6  find_package(Qt5LinguistTools)
7
8  # Wrap files, create moc files.
9  qt5_wrap_cpp(APP_MOC ${APP_HEADERS})
10 qt5_wrap_ui(APP_UI ${APP_FORMS})
11 qt5_add_resources(APP_RCC ${APP_RESOURCES})
12
13 if(Qt5LinguistTools_FOUND)
14     message(STATUS "[qonverter] Qt Linguist Tools found. Translations
            will get refreshed.")
15     qt5_add_translation(APP_QM ${APP_TRANSLATIONS})
16 else()
17     message(STATUS "[qonverter] Qt Linguist Tools NOT found. No
            refreshing for translations.")
18 endif()
```

Macro `find_package` loads necessary Qt modules. If module is not found, then error message is triggered and script fails to complete. Additionaly, special Qt 5 related macros are run. These create additional files needed for successful compilation. If translation tools are found, translation files are added.

Qt 5 introduced new system for linking modules. As we know, each module is represented by dynamic-link (or static-link) library file. CMake can link our compiled executable against correct libraries which is easy to get done with CMake (Listing 47).

Listing 47: Link against Qt 5 in CMakeLists.txt script

```
1  # Use modules from Qt.
2  qt5_use_modules(${EXE_NAME}
3      Core
4      Widgets
5      Sql
6      Network
7      Xml
8  )
```

There is one little problem. Qt 5 CMake scripts don't automatically link against Qt 5 platform library. Qt 5 platform library is responsible for appropriate "categorization" of executable file. Typical categories are:

- console application,

- GUI-based application.

If CMake is not told that our application contains user interface, then console window is displayed (along with user interface) if our application launches. Thus, extra linking is needed (Listing 48).

Listing 48: Link against Qt 5 platform library in CMakeLists.txt script

```
1  if(WIN32)
2      add_executable(${EXE_NAME} WIN32
3          ${APP_SOURCES}
4          ${APP_FORMS}
5          ${APP_RCC}
6          ${APP_QM}
7      )
8      target_link_libraries(${EXE_NAME} Qt5::WinMain)
9  else()
10     add_executable(${EXE_NAME}
11         ${APP_SOURCES}
12         ${APP_FORMS}
13         ${APP_RCC}
14         ${APP_QM}
15     )
16 endif()
```

Qonverter CMake script supports semi-automatic instalation via "make install" command. `CMakeLists.txt` script contains code which is responsible for that.

Listing 49: Installation code in CMakeLists.txt

```
1  elseif(UNIX)
2      message(STATUS "[qonverter] You will probably install on Linux.")
3      install(TARGETS ${EXE_NAME} RUNTIME DESTINATION bin)
4      install(FILES ${CMAKE_CURRENT_BINARY_DIR}/resources/desktop/
          qonverter.desktop DESTINATION share/applications)
5      install(FILES resources/graphics/qonverter.png DESTINATION share/
          icons/hicolor/256x256/apps/)
6      install(FILES ${APP_QM} DESTINATION share/qonverter/l10n)
7      install(FILES ${APP_SKIN_PLAIN} DESTINATION share/qonverter/skins
          /base)
8      install(FILES ${APP_SKIN_MODERN} DESTINATION share/qonverter/
          skins/base)
9  endif()
```

Each supported platform has its own installation code because some files need to be installed in different paths on each platform. On Windows, for example, all files of an application are installed in single root directory. On Unix-like operating systems, on the other hand, files of one application are often spread over across the file system directory tree. Binaries may be placed in /usr/bin directory, icons

in `/usr/share/icons` directory and so on.

CMake even supports packaging of source code. Imagine that you are software developer and new version of your application comes out. You want to distribute its source code. Easiest way to do that is to pack all source code into single archive. That's what CMake does. Listing 50 show fragment of Qonverter script which does the job too via new command "make dist".

Listing 50: Packing support for Qonverter

```
1  # Custom target for packaging.
2  set(CPACK_PACKAGE_NAME ${APP_LOW_NAME})
3  set(CPACK_PACKAGE_VERSION ${APP_VERSION})
4  set(CPACK_SOURCE_GENERATOR "TGZ")
5  set(CPACK_SOURCE_PACKAGE_FILE_NAME "${CPACK_PACKAGE_NAME}-${
      CPACK_PACKAGE_VERSION}")
6  set(CPACK_IGNORE_FILES "/CVS/;/\\\\.svn/;/\\\\.git/;\\\\.swp$;/
      CMakeLists.txt.user;\\\\.#;/#;\\\\.tar.gz$;/CMakeFiles/;CMakeCache
      .txt;\\\\.qm$;/build/;\\\\.diff$;.DS_Store'")
7  set(CPACK_SOURCE_IGNORE_FILES ${CPACK_IGNORE_FILES})
8
9  # Load packaging facilities.
10 include(CPack)
11
12 # make dist implementation.
13 add_custom_target(dist COMMAND ${CMAKE_MAKE_PROGRAM} package_source)
```

5.2.2 Using CMake scripts

Previous chapter was about CMake scripts from the view of the programmer. Let's assume that we are interested in installing software which is available with CMake script. In that situation we only need working compiler, installed CMake and source code with valid script.

Usual installation contains three typical steps:

1. Open command line, navigate to unpacked source code root directory and check if `CMakeLists.txt` file exists.

2. Create new subdirectory "build", this directory will contain compiled files when compilation ends. Run this code in command line:

```
cd build && cmake ../ -DCMAKE_INSTALL_PREFIX=<path-to-installed-
    application> -DCMAKE_BUILD_TYPE=release
```

3. Run "make install" in command line.

Qonverter uses the same approach. User has to do just one thing, provide correeect target installation path. So on Windows, `cmake` call could look like this:

```
cd build && cmake ../ -DCMAKE_INSTALL_PREFIX="C:/Program Files/
    Qonverter" -DCMAKE_BUILD_TYPE=release
```

5.2.3 Publishing Qonverter for GNU/Linux

Linux-based operating systems differ from Windows operating systems in many things. One of them is the way of managing installed applications and libraries. Each Linux distribution contains sophisticated package manager – the software which handles installation and management of all software.

5.2.3.1 Publishing Qonverter for Archlinux

Archlinux is Linux-based operating system. It uses package manager called Pacman. Software packages are available online via so called "repositories". Each repository offers specific software. For example "core" repository offers critical software needed by every Archlinux instance. There exists special community-managed repository called Arch User Repository (AUR). Each and every user of Archlinux can store software packages in this repository and offer it to the community.

Each package for AUR needs to provide special script. This script specifies steps needed for successful compilation of the software (Archlinux Project, 2013).

116

Script should be contained within text file called "PKGBUILD". Listing 51 displays simple packaging script made for Git version of Qonverter.

Listing 51: AUR Script for Qonverter

```
1  # Maintainer: Martin Rotter <rotter.martinos@gmail.com>
2
3  pkgname=qonverter-git
4  pkgver=20130402
5  pkgrel=2
6  pkgdesc='Very simple and easy-to-use desktop calculator with unusual
       functions.'
7  arch=('i686' 'x86_64')
8  url="http://code.google.com/p/qonverter"
9  license=('GPL3')
10  depends=(qt5-base)
11  makedepends=(gcc git cmake)
12
13  _gitname=qonverter
14  _gitroot=https://Rotter.Martinos@code.google.com/p/qonverter/
15
16  build() {
17    cd ${srcdir}
18    msg "Cloning " ${_gitname} " repository..."
19
20    if [ -d ${_gitname} ] ; then
21      cd ${_gitname} && git pull origin master
22      msg "The local files are updated."
23    else
24      git clone ${_gitroot}
25    fi
26
27    msg "Git checkout done or server timeout."
28    cd ${srcdir}/${_gitname}
29
30    if [ ! -d "build" ]; then
31      mkdir build
32    fi
33
34    cd build
35
36    msg "Preparing files with cmake..."
37    cmake ../ -DCMAKE_INSTALL_PREFIX=/usr -DCMAKE_BUILD_TYPE=release
38  }
39
40  package() {
41    cd ${srcdir}/${_gitname}/build
42
43    msg "Compiling files..."
```

```
44    make DESTDIR=${pkgdir} install || return 1
45
46    msg "All files were successfully compiled."
47  }
```

Script contains two basic parts:

1. Lines 1 – 14 specify information about the software, including its name, version number and url. This section specifies build and run dependencies too. "Dependency" is another software which is needed to run software from this script, Qonverter in this case. Obviously, Qonverter needs Qt 5 for its run. GCC is needed for compilation. Variables "_gitname" and "_gitroot" are special. They specify name and url of used Git repository which contains the actual source code of our software.

2. Rest of script contains two functions which are called when software package is installed. These functions do everything needed for compilation and packaging of target software. It downloads source code from remote Git repository, compiles it with CMake and prints out informative messages.

Let's assume we have this script finished. We need to create the actual package. "PKGBUILD" file is the only file contained in this package. Package can be created by `makepkg --source` command, which is available on every Archlinux instance.

User needs to upload generated archive to AUR by going to https://aur. archlinux.org/, logging in and submitting it. Other users then download this packages manually from Archlinux website or use special tool (see https://wiki. archlinux.org/index.php/Yaourt) to obtain it. We can see that yaourt utility makes selection of packages very simple (Figure 5.3).

Figure 5.3: Package Selection in yaourt

Chapter 6

Conclusions

We have learned some basic facts about Qt throughout this book. Fundamental principles got discovered and we saw some source code fragments which complemented discussed matter. It is recommended to go through Qonverter source code to gain more useful source code (Rotter, 2013) snippets.

List of Figures

List of Tables

126

List of Listings

List of Abbreviations

API   Application Programming Interface
AUR   Arch User Repository

CSS   Cascading Style Sheets

ELF   Executable and Linkable Format

GCC   GNU Compiler Collection
GUI   Graphical User Interface

IL   Intermediate Language

JRE   Java Runtime Environment

KDE   K Desktop Environment

MinGW Minimalistic GNU for Windows
moc   Meta-object compiler
mos   Meta-object system
MSVC   Microsoft Visual C++
MVC   Model-view-controller

OOP   Object-oriented programming

PE   Portable Executable
POSIX   Portable Operating System Interface
PThreads POSIX Threads

QML   Qt Meta Language
QPA   Qt Platform Abstraction

RPN   reverse Polish notation

XML   Extensible Markup Language

Bibliography

Archlinux Project

    2013   *Arch Packaging Standards*, https://wiki.archlinux.org/index. php/Arch_Packaging_Standards (visited on 01/14/2013).

Berg, Ingo

    2013   *muParserX library*, http://code.google.com/p/muparserx (visited on 04/11/2013).

Du Toit, Stefanus

  16, 2012   *Working Draft N3337, Standard for Programming Language*, tech. rep., International Organization for Standardization/International Electrotechnical Commission.

KDE

    2013   *KDE Project Webp*, http://www.kde.org/ (visited on 03/26/2013).

Kitware

    2013   *CMake*, http://www.cmake.org/ (visited on 04/11/2013).

McConnell, Steve C.

    2004   *Code Complete: A Practical Handbook of Software Construction*, 2nd edition, Microsoft Press, ISBN: 0-7356-1967-0.

Microsoft

    2013   *Skype*, http://www.skype.com/ (visited on 03/27/2013).

Nigel, Christian

    2010   *Professional C# and .NET 4*, Wiley-Publishing, ISBN: 978-0-470-50225-9.

Prata, Stephen

    2011   *C++ Primer Plus*, 6th ed., Addison-Wesley, ISBN: 0-321-77640-2.

Qt-Project

    2012a   *Qt 5 Developer Changelog*, version d4a29a5, Qt-Project, http://qt.gitorious.org/qt/qtbase/blobs/HEAD/dist/changes-5.0.0 (visited on 01/21/2013).

    2012b   *Qt 5 Online Reference Documentation*, http://qt-project.org/doc/qt-5.0/ (visited on 01/14/2013).

    2012c   *Qt Online Wikipedia*, http://qt-project.org/wiki/ (visited on 01/14/2013).

    2013a   *Public Qt Git Repository*, http://qt.gitorious.org/ (visited on 05/10/2013).

    2013b   *Qt-Project Web*, http://qt-project.org/ (visited on 03/27/2013).

Ramakrishnan, Girish

    6, 2007   *Lineedit with a clear button*, June 6, 2007, http://blog.qt.digia.com/blog/2007/06/06/lineedit-with-a-clear-button/ (visited on 05/16/2013).

Rotter, Martin

    12, 2013   *Qonverter project Git repository*, Apr. 12, 2013, https://code.google.com/p/qonverter/source/ (visited on 04/12/2013).

Stallman, Richard M.

    2007   *GNU General Public License*, version 3, Free Software Foundation, http://www.gnu.org/copyleft/gpl.html (visited on 08/29/2012).

The Apache Software Foundation

    2012   *Subversion*, http://subversion.apache.org/ (visited on 04/11/2013).

Torvalds, Linus

    2013   *Git*, http://git-scm.com/ (visited on 04/11/2013).

Wikipedia

    2013a   *Clang*, http://en.wikipedia.org/wiki/Clang (visited on 04/11/2013).

    2013b   *eComStation operating system*, http://en.wikipedia.org/wiki/EComStation (visited on 04/11/2013).

    2013c   *Qt Framework History*, http://en.wikipedia.org/wiki/Qt_(framework) (visited on 03/26/2013).

Index

133