QT: INTERNALS AND PRINCIPLES

Get to grips with Qt

Martin Rotter

February 10, 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

Big thanks belongs my bachelor thesis leader Mgr. Tomáš Kühr. Special praise goes to my closest friends, especially to my family and to people who let me know that love can be wonderful.

Preface

Whom 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 hugely experienced software engineer or self-taught enthusiast. Information included in this book can be useful either way.

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 form very complex science. 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 very 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 follow. Meta-object system is discussed too. Understanding meta-object system is the key for next progress and is the main precondition 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 (as of January, 2013) Qt 5.

What is not covered by this book?

As said earlier, it's not possible to cover all nooks of Qt libraries in one book. We will omit some hugely 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 but you will clap on them from time to time as they are needed for advanced Graphical User Interface (GUI) tweaking.

Some other parts of Qt are ignored too. 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 and its features. 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 prerequisites?

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, Stefanus, 2012) or in section 1.3.

Text formatting

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

Listing 1: Sample code fragment

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

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.

Licensing

Contents

Part I
Laboratory Qt

1 Foreword

Qt framework is one of the greatest libraries ever made. You probably use it and you don't even know about it. If you use Skype (for online communication) or K Desktop Environment (KDE), 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 flow 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 by, 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 very roots in Norway. Original creators are Haavard Nord and Eirik Chambe-Eng. Basically Qt framework consists of:

- set of libraries written in C++

- meta-object compiler
- QtScript interpreter
- tools for internationalization and 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 section 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 exciting years, preffering desktop development.

But as we know, things have changed and smartphones became massively popular lately. 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. But Nokia somehow was not able to utilize potential of Qt and sold it to another company called Digia.

1.2.1 Licensing

Qt uses two separate licenses:

1. **Commercial license**, which provides you (as indie developer) with possibility 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. It may cost around several thousands US dollars and this price may get even higher if you buy license for more platforms or if you have bigger development team. This license is usually bought by developers who want to sell their software for money and/or stay closed-source, otherwise open-source license is much better choice.

[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 runs on Windows, Linux or Mac OS X. It's Portable Operating System Interface (POSIX)-compatible.

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.

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 (see (Stallman, Richard M. 2007)) for your projects.

Licenses have always been quite a problem for Qt framework. Commercial license was fine. But 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 got 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 famous 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 very tiny example of basic techniques in Listing 2.

Listing 2: Basic OOP techniques in C++

```cpp
/* Base class declaration */
class BaseClass {
    public:
        BaseClass();

        void whoAmI() const;
};

/*
 * Class declaration
 * This class inherits BaseClass.
 */
class InheritingClass : public BaseClass {
    public:
        InheritingClass();

        void whoAmI() const;
};

/* Example usage of BaseClass and InheritingClass classes. */
int main() {
    BaseClass class_1;
    InheritingClass class_2;
    class_1.whoAmI();
    class_2.whoAmI();

    BaseClass *class_3 = &class_2;
    class_3->whoAmI();

    ((InheritingClass*) class_3)->whoAmI();

    return 0;
}
```

Listing 3: Output of application from Listing 2

```
BaseClass instance constructed.
BaseClass instance constructed.
InheritingClass instance constructed.
I am BaseClass.
I am InheritingClass.
I am BaseClass.
I am InheritingClass.
```

C++ has many characteristics – some are bad while other ones may be great. Let's compare usefulness of its abilities.

SYNTAXbad

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 bizarre usage of this keyword. You can read about this topic in (Prata, Stephen, 2011, p. 90-92, p. 537).

POINTERS vs. REFERENCESbad

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 MANAGEMENTbad, good

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

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 (or perhaps she) 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.

In the other, manual management of allocated objects gives programmer bigger power to control application memory consumption 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.

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.

THREADING$^{\text{bad}}$

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

FAST CODE EXECUTION$^{\text{great}}$

C++ code execution is amazingly fast compared to other modern programming languages. Direct compilation (see more in section 4) into machine

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

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 descendingly-valued 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 display in Figure 1.

Listing 4: Quicksort implementation in C++

```
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];
    int i = p - 1;
    int j;
    for (j = p; j < r; j++) {
        if (array[j] <= x) {
            i += 1;
            swap(&array[i], &array[j]);
        }
    }
    swap(&array[i + 1], &array[r]);
    return i + 1;
}

void QuickSort::swap(int *lhs, int *rhs) {
    int temp = *lhs;
    *lhs = *rhs;
    *rhs = temp;
}
```

Listing 5: Quicksort implementation in C#

```
static void 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);
    }
}

static int partition(int[] array, int p, int r) {
    int x = array[r];
    int i = p - 1;
    int j;
    for (j = p; j < r; j++) {
        if (array[j] <= x) {
            i += 1;
            swap(ref array[i], ref array[j]);
        }
    }
    swap(ref array[i + 1], ref array[r]);
    return i + 1;
}

static void swap(ref int lhs, ref int rhs) {
    int temp = lhs;
    lhs = rhs;
    rhs = temp;
}
```

Listing 6: Quicksort implementation in “unsafe” C#

```
static unsafe void 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);
    }
}

static unsafe int partition(int* array, int p, int r) {
    int x = array[r];
    int i = p - 1;
    int j;
    for (j = p; j < r; j++) {
        if (array[j] <= x) {
            i += 1;
            swap(&array[i], &array[j]);
        }
    }
    swap(&array[i + 1], &array[r]);
    return i + 1;
}

static unsafe void swap(int* lhs, int* rhs) {
    int* temp = lhs;
    lhs = rhs;
    rhs = temp;
}
```

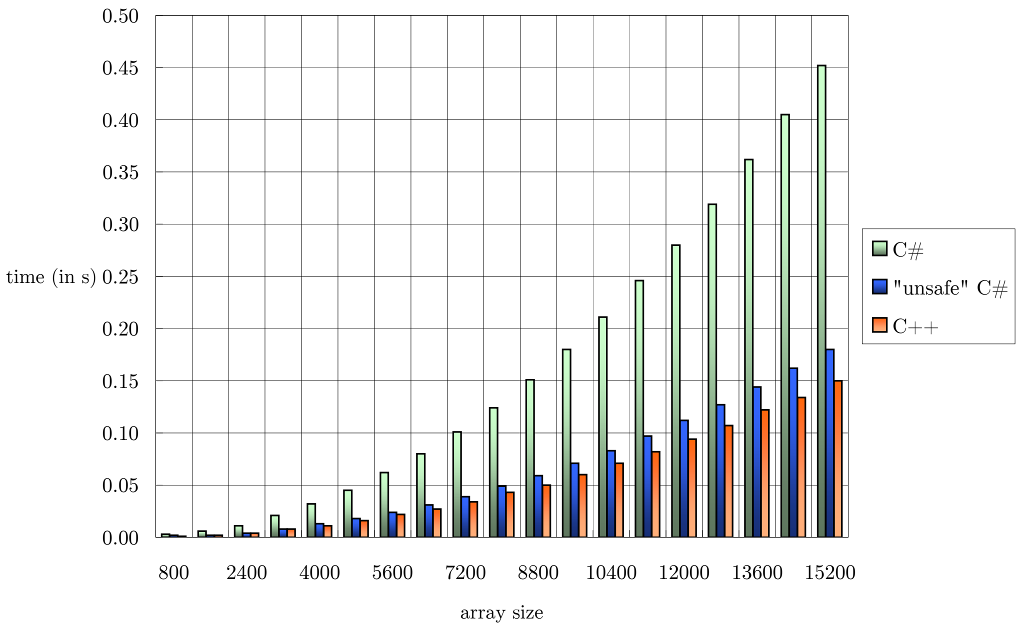


Figure 1: Results of C++ vs. C# comparison

We see that C++ outperformed classic C# implementation, while being around 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 (perhaps some 3D graphical computation) is needed to be done.

HUGE COMMUNITY$^{\text{great}}$

Plenty of world-renowned software is written using C++ , including many 3D games, almost each program from Adobe and Chromium web browser. Many C++ books are available, making it easier to learn.

MEMORY CONSUMPTION$^{\text{good}}$

C++ applications, as stated, need no virtual machine for their execution. They load just base C++ library and extra libraries if needed. Such approach makes significant opposite to robust and greedy (as for memory) runtime environments of certain high-level languages. We can mention primarily .NET Framework and Java Runtime Environment (JRE).

CODE PORTABILITY$^{\text{bad}}$

When it comes to code portability (same as multi-platformity), C++ leaves its audience uncertain. Users 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, however, which can result in pain rooting from hypothetical need to port your application to another platform.

MISSING CONSTRUCTS$^{\text{bad}}$

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

1.3.1 Version 11 and its enhancements

C++ programming language was, for the first time, 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 depending on time and 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 a massively extensive (Du Toit, Stefanus, 2012) or goe through its finest new properties right now. It is recommended to know something about C++ 11 as its support is turned on in Qt 5 by default on supported 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 evil, 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 some thing like:

```
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();
```

Initializer lists

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

```
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. But what if you know all values in compile time? In older C++ you would have to insert all values one by one. (Note that there is no `CustomList(const std::list &list)` constructor available.) But C++ 11 allows you to use initializer list:

Listing 7: Initializer list usage

```
 1 class CustomList {
 2     private:
 3         std::list<int> m_list;
 4 
 5     public:
 6         CustomList();
 7         CustomList(std::initializer_list<int> values) {
 8             for (int &value : values) {
 9                 m_list.push_back(value);
10             }
11         }
12 
13         void insert(int i) {
14             m_list.push_back(i);
15         }
16 };
17 
18 // Creating CustomList instance and filling it with values.
19 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 too.

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

Type deduction

C++ is statically typed language. So you, as programmer, 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

Well, lambda expressions were the most expected feature. Known from functional languages (e. g. Common Lisp, Scheme), they rapidly penetrated even 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 (or she) wants to allocate lambda expression on the stack or on the heap:

```
auto twice_function_stack = [] (double input_1) -> double {
    return input_1 * double;
};

auto twice_function_heap = new auto ([] (double input_1) -> double {
    return input_1 * double;
});
```

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

Lambda expression can be used as function parameter too. Very simple (and kind of naive) implementation of in-place map function can look like the on in Listing 8.

Listing 8: Lamda expression as function parameter

```cpp
#include <iostream>
#include <functional>
#include <list>


// In-place map function.
// Executes func for each member of input_list
void map(const std::function<void(double&)> &func, std::list<double>
    &input_list) {

    // Range-based for-loop.
    for (double &value : input_list) {
        func(value);
    }
}

int main(int argc, char *argv[]) {
    // Create simple list using list initializer.
    std::list<double> my_list = {1.7, 2.8, 4.9, 5.9, 0.0};

    // Instantiate lambda expression (anonymous function).
    auto func_twice = [&] (double &input) {
        input *= 2;
    };

    // Use lambda expression as function parameter.
    map(func_twice, my_list);

    for (double value : my_list) {
        std::cout << value << "␣";
    }
    return 0;
}
```

Lambda expression make huge impact on Qt 5.

Null pointers

It is quite common to type something like `int *variable = NULL` in C++ 03. 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 has always been quite tiny. It included just necessary classes, nothing special. But time goes forward, so that standard library must too. Number of fine classes were added.

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)

Good bonus for every C++ programmer. No need to use 3rd party tuples implementations. More classes are new in the standard library, look at (Du Toit, Stefanus, 2012) for more.

1.4 Qt components

Qt consists of libraries, tools and other supplemental software. You have already seen very brief list of Qt components in subsection 1.1. Libraries themselves are divided into so-called *modules*.[5] You can learn more about modules in section 2. Let's look into Qt library collection more thoroughly. Qt library collection contains these main components:

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 or can server as the main force for creating graphics-related software, e. g. painting applications, video editors or perhaps chart designer programs.

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

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 support of various operating systems and platforms is one if its key features if now the biggest one. Supported operating systems are:

- Windows (+ Windows Embedded Compact)
- Linux
- Mac OS X
- OS/2 (eComStation)
- 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. You can blame Digia for bad approach but it's better not to compromise sometimes. Qt 3 support was dropped too.
- Brand new Qt component named Qt Platform Abstraction (QPA), allowing you to easier port Qt 5 for 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.[6] Meta-object system was tweaked too and you will be informed about those changes later.
- New multi-platform user interface style called Fusion is available (Figure 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.
2. You can download open-source Qt framework directly from www.qt-project.org.
3. You use a Linux distribution which can equip you with Qt framework via some kind of native packaging system.

[6]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.

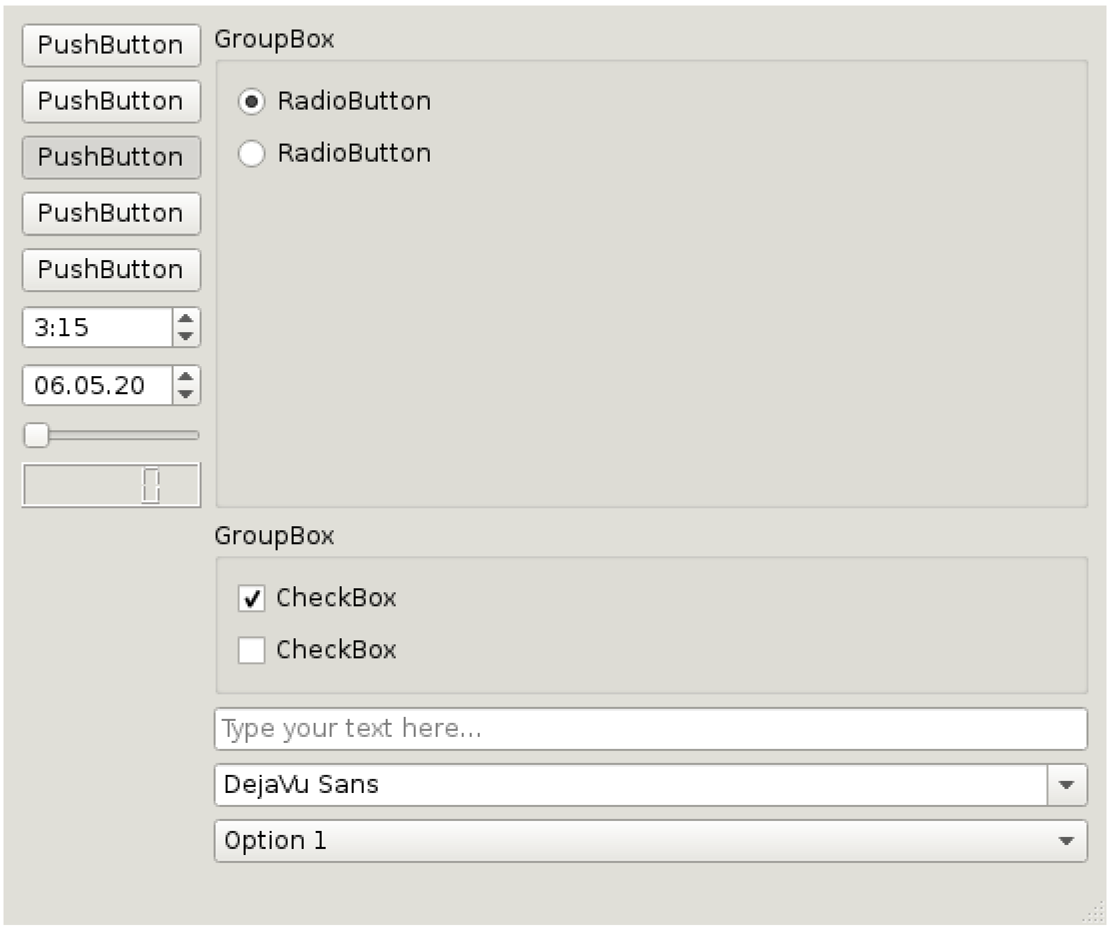


Figure 2: Qt Fusion style example

Qt can be downloaded as executable installer file which containes binaries precompiled for you as well as documentation and other needed tools. Sometimes manual compilation is needed.[7]

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

1.5.1 Installing Qt on Windows

Qt installation on Windows is fairly straightforward if binaries are available. All you need to do is obtain the setup executable file and follow instructions. Some

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

troubles might occur, though.

Let's assume that Qt was installed in `c:\Qt\Qt5.0.0\`.

You need to tweak 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:

   ```
   c:\Qt\Qt5.0.0\5.0.0\msvc2010\bin\
   c:\Qt\Qt5.0.0\Tools\QtCreator\bin\ (If Qt Creator is installed
       too.)
   ```

   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

   ```
   %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 for many major distributions, e. g. Ubuntu, Archlinux, Fedora, Debian or Mint.
2. Classical installation via executable file:
   (a) Obtain installation file from www.qt-pro-ject.org.
   (b) Open terminal and navigate to folder containing obtained installation file.

(c) Change permissions on the file:

```
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

```
QTDIR=/opt/qt5/5.0.0/gcc
PATH=$PATH:$QTDIR/bin
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 (and static too) libraries sizes.

Qt sources are always contained within compressed file. All you need to do is to have correctly installed C++ compiler (GCC or MSVC are recommended). 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).

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

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 section 4.

Let's dig into Qt libraries now. Qt offers very rich and diverse functionality (see subsection 1.4), ranging from network communication to painting vector pictures.

2.1 Modules

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

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.

2.1.1 Linking

Each module usually depends on QtCore module, including QtWidgets module. Moreover, QtWidgets module depends on QtGui module. So each Qt-based application with user interface has to be linked against 3 or more 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 executable file to run successfully. 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

```
[root@arch-linux 04-guiapp]# ldd -d -r -u 04-guiapp
Unused direct dependencies:
/opt/qt5/5.0.0/gcc/lib/libQt5Gui.so.5
/usr/lib/libGL.so.1
/usr/lib/libpthread.so.0
/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 called explicitly in our sample application. You will learn more about linking in section 4.

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

2.2 Tree-like class structure

Cleverly developed library has smart class structure which makes that library easily maintainable, expandable and functional. *Class inheritance* is used very extensively if library design is something we need to deal with. Read (Prata, Stephen, 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 practice 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 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 3) 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 very 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, Christian, 2010, p. 84). Java follows very similar class hierarchy ideas.

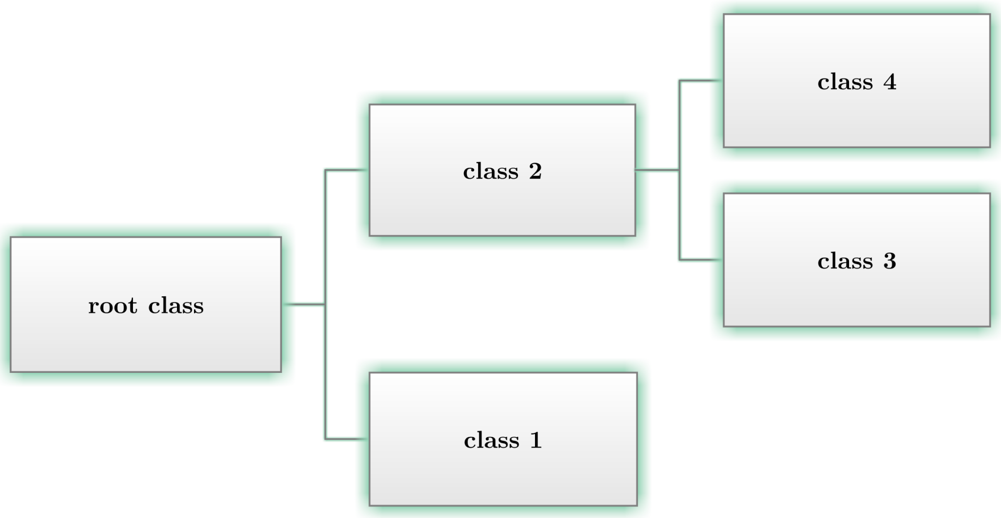


Figure 3: Typical library tree-like class structure

3 Using Qt framework

Basic Qt library structure is known to us but we need to know something about Qt-related development evnironments and tools. You can develop Qt applications in any text editor or in major 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 4) 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.

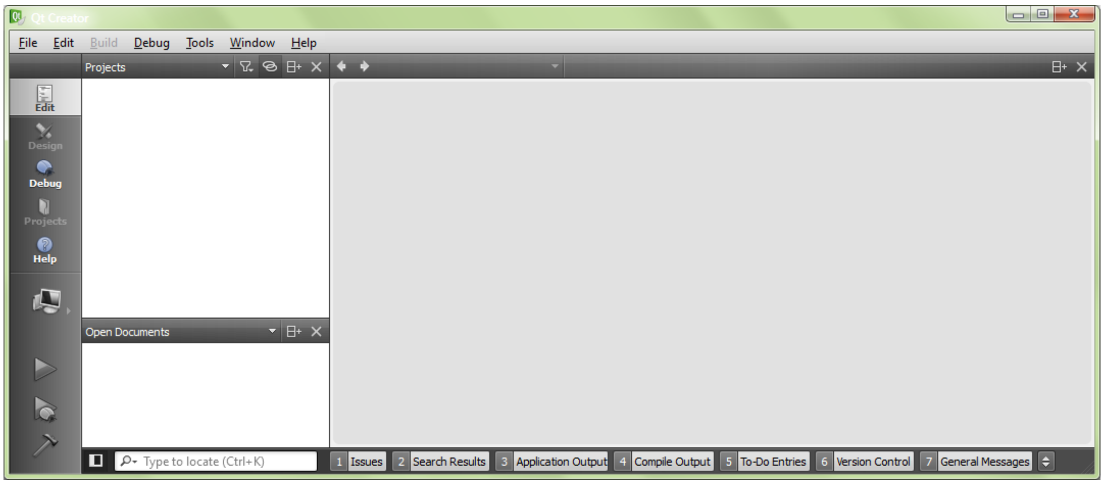
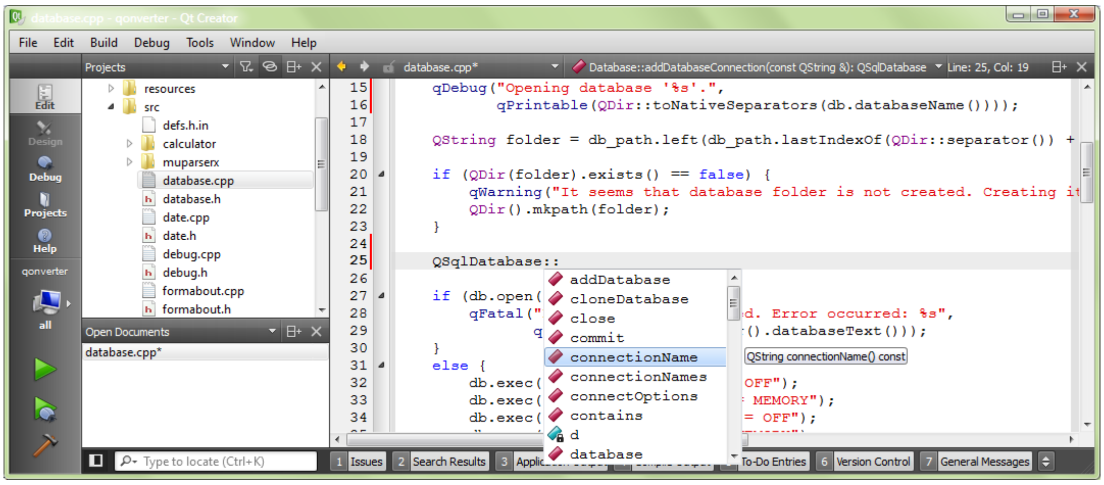


Figure 4: Qt Creator empty environment

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 5)
- 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 7)
- context-aware help (see Figure 6)
- support for simultaneously installed Qt frameworks
- integrated Qt Designer for GUI design
- sharing source code via online services
- many more. . .

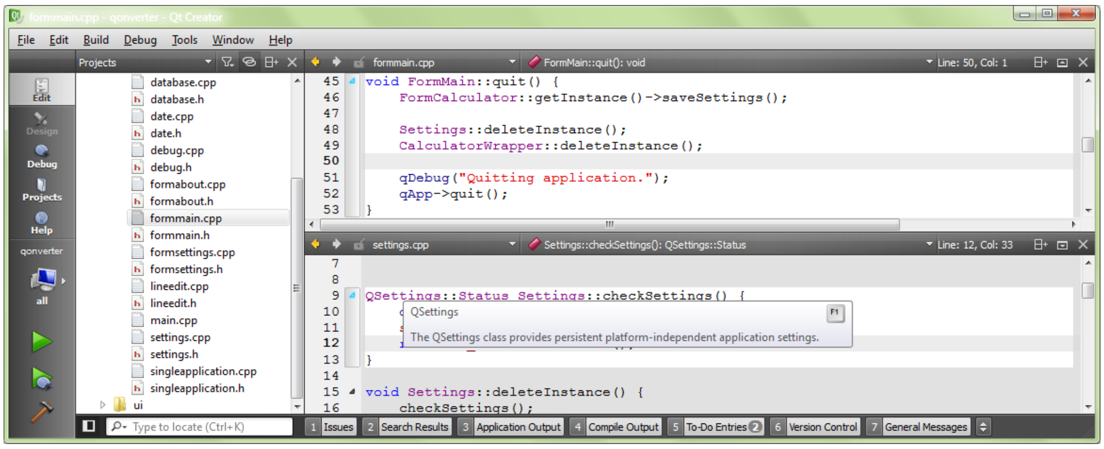


```
    qDebug("Opening database '%s'.",
           qPrintable(QDir::toNativeSeparators(db.databaseName())));

    QString folder = db_path.left(db_path.lastIndexOf(QDir::separator()) +

    if (QDir(folder).exists() == false) {
        qWarning("It seems that database folder is not created. Creating it
        QDir().mkpath(folder);
    }

    QSqlDatabase::

    if (db.open(
        qFatal("                           d. Error occurred: %s",
               q                           ().databaseText()));
    }
    else {
        db.exec(                           OFF");
        db.exec(                           MEMORY");
        db.exec(                           = OFF");
```

Figure 5: Qt Creator auto-completion

3.1.1 Speeding-up Qt Creator

You can make Qt Creator less memory-hungry by disabling some unneeded plug-ins. You can disable plugins in `About -> About Plugins`... menu. Minimal setup for desktop Qt development using CMake may look like the one in Table 1.



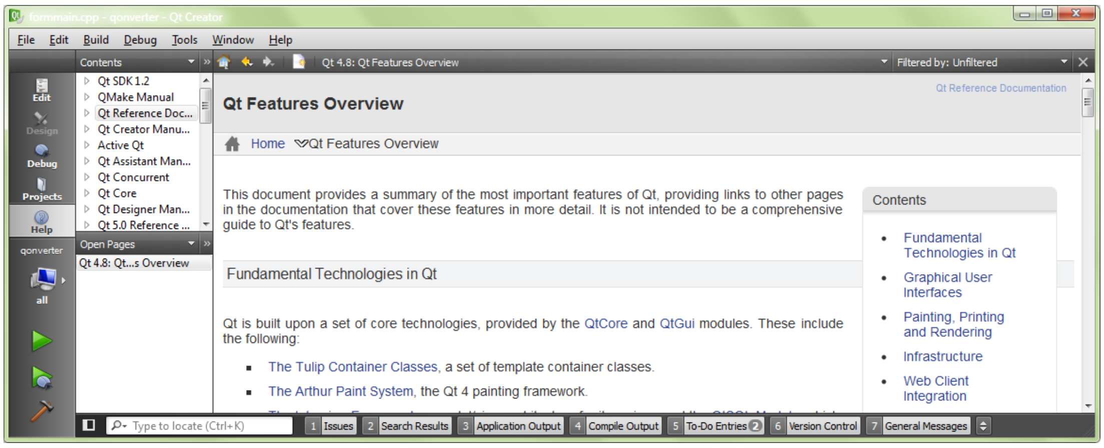
```
void FormMain::quit() {
    FormCalculator::getInstance()->saveSettings();

    Settings::deleteInstance();
    CalculatorWrapper::deleteInstance();

    qDebug("Quitting application.");
    qApp->quit();
}
```

```
QSettings::Status Settings::checkSettings() {
}

void Settings::deleteInstance() {
    checkSettings();
```

QSettings

The QSettings class provides persistent platform-independent application settings.

Figure 6: Qt Creator context-aware help



Qt Features Overview

This document provides a summary of the most important features of Qt, providing links to other pages in the documentation that cover these features in more detail. It is not intended to be a comprehensive guide to Qt's features.

Fundamental Technologies in Qt

Qt is built upon a set of core technologies, provided by the QtCore and QtGui modules. These include the following:

- The Tulip Container Classes, a set of template container classes.
- The Arthur Paint System, the Qt 4 painting framework.

Contents

- Fundamental Technologies in Qt
- Graphical User Interfaces
- Painting, Printing and Rendering
- Infrastructure
- Web Client Integration

Figure 7: Qt Creator full reference documentation

3.2 Qt Designer

Qt Designer (see Figure 8) is a tool for user interface design. It is standalone application which is integrated into Qt Creator too.

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 9) is virtual container for one compiler and one specific Qt library (for example Qt 5 library). Moreover, kit specifies primary debugger and and other stuff needed to compile your projects. Kit (or

Table 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

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.

3.4 Code conventions

Writing working application isn't, by far, 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".[8]

3.4.1 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 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's actually joy to browse it.

[8]You should assure yourself you have some base to build on before you proceed. If certainty is not solid, then take a look at (McConnell, Steve C. 2004, p. 40-77).

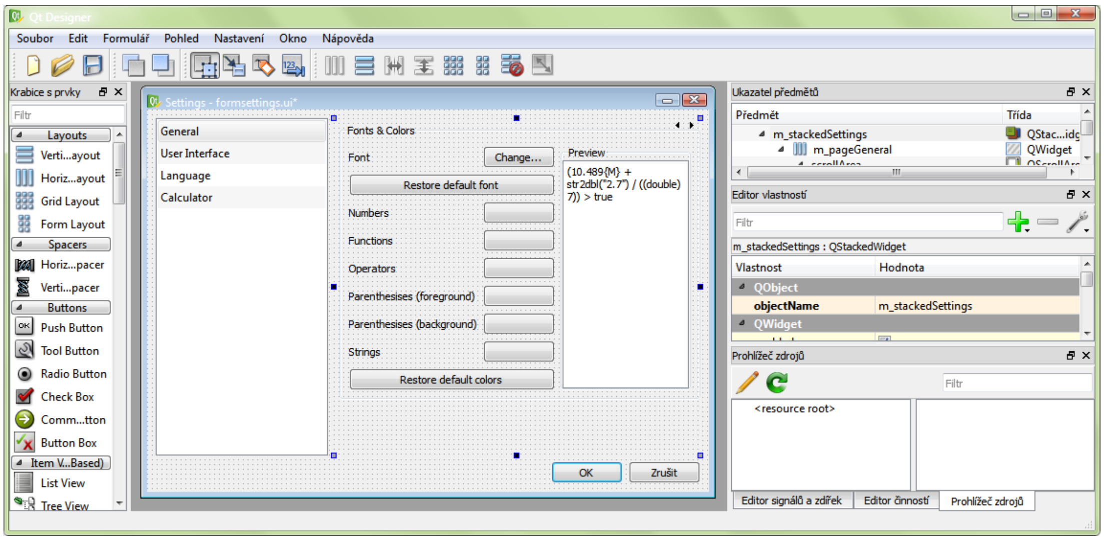
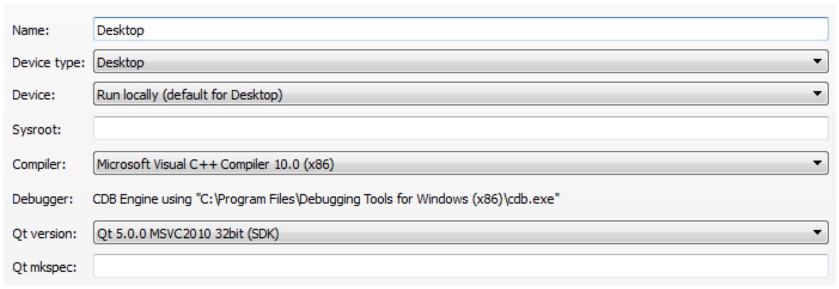


Figure 8: Qt Designer environment



Name:	Desktop
Device type:	Desktop
Device:	Run locally (default for Desktop)
Sysroot:	
Compiler:	Microsoft Visual C++ Compiler 10.0 (x86)
Debugger:	CDB Engine using "C:\Program Files\Debugging Tools for Windows (x86)\cdb.exe"
Qt version:	Qt 5.0.0 MSVC2010 32bit (SDK)
Qt mkspec:	

Figure 9: Qt Creator kit setup

Let's compare source code to formatted text (e. g. text of a book). Good books do have their content 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 lines 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 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 a *conventions*. Everything (almost) can have its style and conventions. Style makes things usable, predictable. Style is good.

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.

3.4.2 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

```
#ifndef CAR_H
#define CAR_H

#include <QtCore/QDebug>


class Car {
    private:
        unsigned int NumberOfWheels;
    public:
        Car(int w);
        void showMeThisParticularCar();
    private:
        bool owner;
};
#endif // CAR_H
```

Listing 14: Good code style

```
#ifndef CAR_H
#define CAR_H


class Car {
    public:
        // Creates new car.
        Car(int number_of_wheels, bool has_owner = true);

        // Displays information about this car.
        void showCar();

    private:
        int m_numberOfWheels; // Stores count of wheels of this car.
        bool m_hasOwner; // True if this cas is owned by someone.
};

#endif // CAR_H
```

There are considerable differences between these two code fragments. Conventions importance is not probably obvious because of code length but it will grow rapidly with regard to code complexity and length. We should examine lines of Listing 14 now.

3.4.3 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, it's a matter of taste.

```
#ifndef CAR_H
#define CAR_H


class Car {
```

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

```
class Car {
    public:
```

Comment your code. Comments are essential part of source code. Always comment parts of code which are not straightforward. Uncommented code buys you ticket to hell.

```
    // 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.

```
    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 methods. In Camel notation, all words in a compound (except first word) begin with upper-case letter and are not separated by spaces or any other character.

```
    void showCar();
```

Use Hungarian notation for data members of each and every class. In Hungarian notation, all variable names have prefix which signals data type or purpose of the variable. It is recommended to delimite prefixes 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.

```
    int m_numberOfWheels; // Signs count of wheels of this car.
    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 rest of source code.

```
#include <QDebug>

#include "car.h"


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.

```
    m_numberOfWheels = number_of_wheels;
    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 some newly forged one in the future.[9] This code 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, you could include parts of a module you would never use in your application.

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

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

4 Compilation process

Compinilers 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 compiler (of object-oriented language) is sometimes called *object code*. Transformaton from source code into object code is not straigtworward 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 librarydynamic-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 perform such a process is called *linker*. Linking can be divided into static and dynamic, see page 32 for more information.

4.2 Executable files 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.[10]

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 10). 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 to actual code. 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 11 differs from classic C++ code compilation process because Meta-object compiler (moc) comes into 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 becaus it is essential part of Qt Meta-object system (mos).

[10]Information in this paragraph is not entirely true but it's "correct enough" for our purposes.

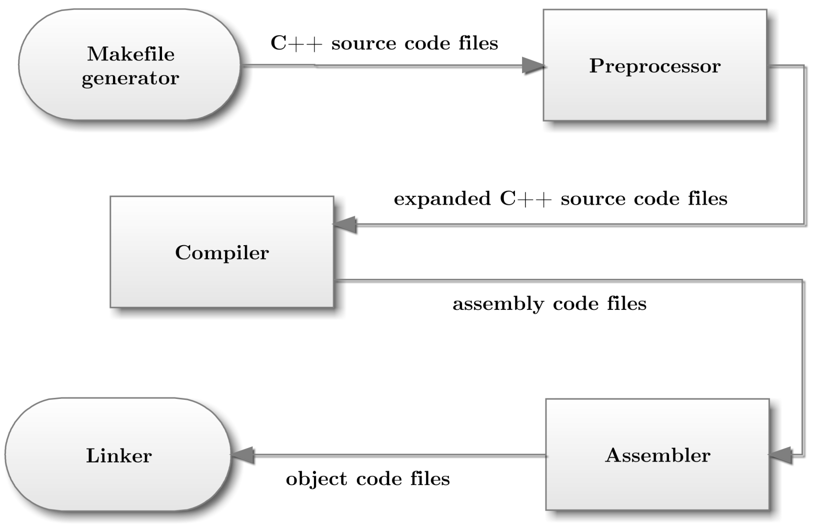


Figure 10: Classic C++ code compilation process

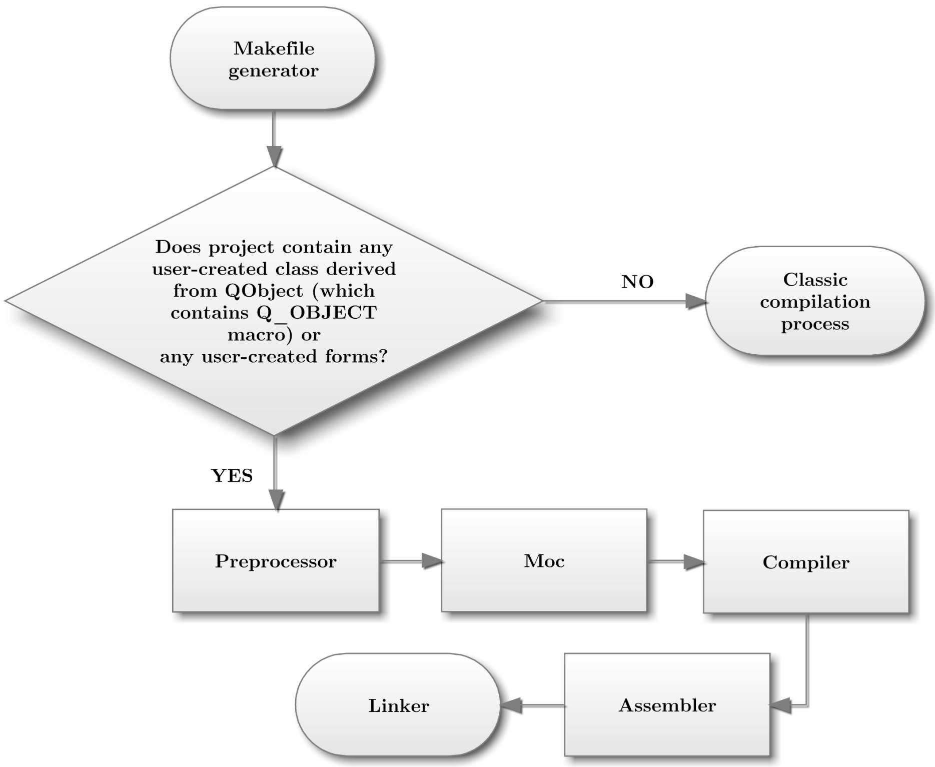

Makefile generator
Does project contain any user-created class derived from QObject (which contains Q_OBJECT macro) or any user-created forms?
NO
Classic compilation process
YES
Preprocessor
Moc
Compiler
Linker
Assembler

Figure 11: Qt-way C++ code compilation process

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`. QtGlobal contains following:

- type clones (Table 2) for every standard C++ type
- functions
- macros

Table 2: Qt Creator minimal plugins setup on x86 architecture (Linux)

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`.

5.2 Producing console outputs with QDebug class

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).

Listing 15: Basic QDebug usage

```
qDebug() << "Print this to standard output.";
qDebug("Print number %d.\n", 10);
```

First `qDebug` usage requires explicit `<QDebug>` inclusion because `<<` operator is used. Second usage acts as wrapper for the `printf` function from standard C library. You can use also `qWarning`, `qCritical` or `qFatal` functions in accordance 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:

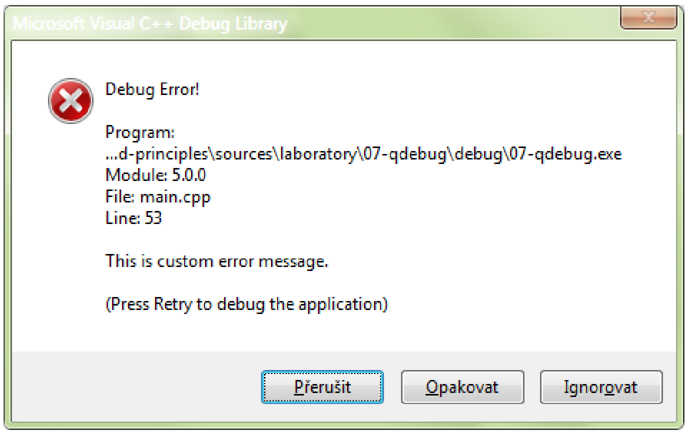
1. You need to implement global (or static) function with signature `void (*function)(QtMsgType, const QMessageLogContext &, const QString &)`. Typical implementation may look like Listing 16.

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

Listing 16: Typical printing handler for QDebug

```
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 12).



Debug Error!

Program:
...d-principles\sources\laboratory\07-qdebug\debug\07-qdebug.exe
Module: 5.0.0
File: main.cpp
Line: 53

This is custom error message.

(Press Retry to debug the application)

Figure 12: Application crash dialog in Windows

One Step Further
Debugging outputs should always be written in English.

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.

The is another (simpler) way of forcing Qt to format console outputs if you want to tweak just format. You may tweak `QT_MESSAGE_PATTERN` build environment variable to display application name or other useful information. Head to the documentation (Qt-Project, 2012b) for more information.

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 entity that extends another object, providing us certain kind of information *beyond* that particular object or set of objects. Meta-objects lie beyond actual objects, forming (kind of "higher") abstraction layer of any Qt application. We can name this layer *meta-echelon*.

Each class instance exposes its private data through *methods* to its users – other classes. Publicly available class members (methods or *properties*[11]) 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 says completely nothing about object inner structure and representation,e. g. it doesn't expose type of on object (in 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

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

certain object is possible at runtime but with one important addition. Compiler supporting 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) 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

```
#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-object features 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 application. 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()const` method. Moreover, dynamic translation system is enabled by this macro too. You will learn about Qt applications translation later.

6.5 QObject class – the cradle of meta-objects

`QObject` class is the very base class for each meta-object-system-enabled class and 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 13). Should copy of `QObject` instance point to the parent of the original `QObject` instance?

   Consider situation in the Figure 14. "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 14) 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.

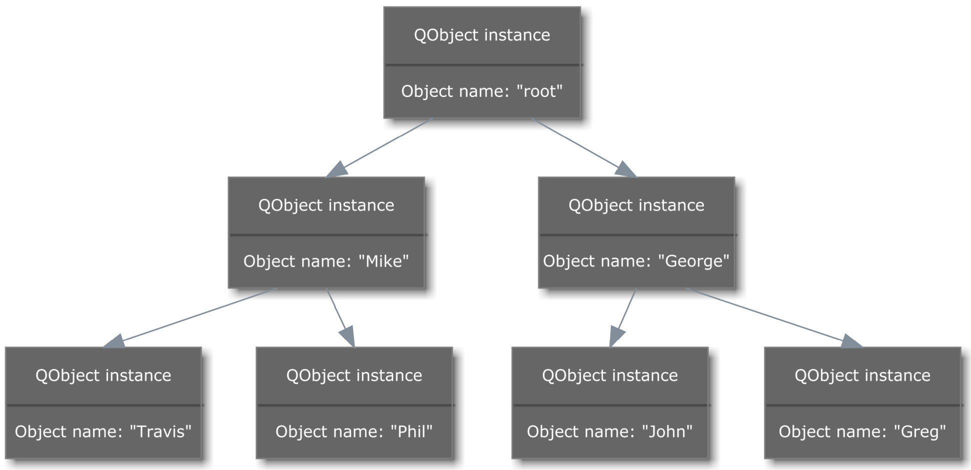


Figure 13: QObject instances tree hierarchy

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 windows is cleared from memory, then all its children are cleared from memory too which is desired behavior. This behavior makes memory management more automatic.

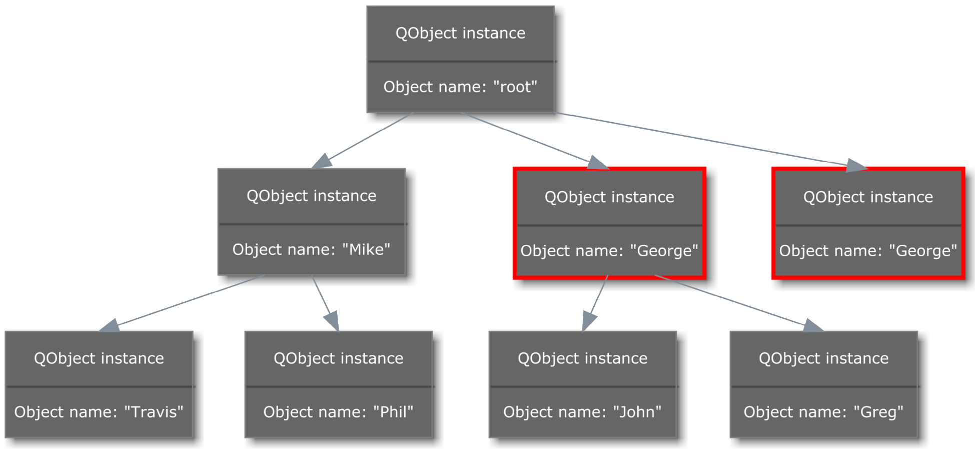


Figure 14: Broken QObject instances tree hierarchy

`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 section 8.

Existence of tree hierarchies impacts positively on several topics:

- more sophisticated memory management[12]
- 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
```

[12]Tree hierarchies form just part of Qt memory management as you will see in section 7.

```

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

/* source file (myqobject.cpp) */
#include "myqobject.h"


MyQObject::MyQObject(QObject *parent) : QObject(parent) {
}
```

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 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, 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 the signal occurrence.

Figure 15: Typical textbox example

Imagine typical *textbox* control (Figure 15). This textbox could offers 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.

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. 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 signals emitted by another entity, then we say that there is *signal–slot connection* between these 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.

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 58 correspond with terms and known from .NET-based languages. (Nigel, Christian, 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 the application.

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. Slots declaration is preceded by `slots` keyword (line 15) with any modifier. You can have public slot as well as private slot. It's just a matter of 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 63 in chapter Signals and slots with regard to meta-object compiler.

Listing 19: Account class design

```
class Account : public QObject {
    Q_OBJECT

    public:
        // No copy constructor or assignment operator is declared,
        // just simple constructor.
        explicit Account(const QString &owner,
                         int deposit,
                         Bank *parent);

        // These are NOT slots.
        void status();
        QString name();

    public slots:
        // Used by customer who requests money from his account.
        // Customer can be either bank or account owner.
        void withdrawMoney(int sum);

        // Used by customer to save money to this account.
        // Customer can be either bank or account owner.
        void saveMoney(int sum);

    signals:
        // Emitted when money is withdrawn successfully from this
           account.
        void withdrawn(int sum);

        // Emitted when money is saved successfully into this account
           .
        void saved(int sum);

    private:
        QString m_owner;
        int m_deposit;
};
```

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

```
class Bank : public QObject {
    Q_OBJECT

    public:
        explicit Bank(QObject *parent = 0);
        void printAccounts();
        void transfer(const QString &from, const QString &to, int sum);

    signals:
        // Emitted both accounts are ready for money transfer and
        // money should be withdrawn from the first account.
        void withdrawalWanted(int sum);

    protected:
        void checkAccounts();
        Account *getAccountByName(const QString &name);

    protected slots:
        void serveAccount(int sum);

    private:
        QList<Account*> m_accounts;

        Account *m_sendingAccount;
        Account *m_waitingAccount;
};
```

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 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 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

```
1  void Bank::transfer(const QString &from, const QString &to, int sum) {
2      if (sum <= 0) {
3          qDebug("You␣cannot␣transfer␣sum␣%d␣USD.", sum);
4          return;
5      }
6
7      checkAccounts();
8
9      Account *acc_from;
10      Account *acc_to;
11      if ((acc_from = getAccountByName(from)) == NULL) {
12          qDebug("Source␣account␣is␣not␣registered␣in␣this␣bank.");
13          return;
14      }
15      if ((acc_to = getAccountByName(to)) == NULL) {
16          qDebug("Destination␣account␣is␣not␣registered␣in␣this␣bank.")
                ;
17          return;
18      }
19
20      m_sendingAccount = acc_from;
21      m_waitingAccount = acc_to;
22
23      connect(this, &Bank::withdrawalWanted, acc_from, &Account::
            withdrawMoney);
24      connect(acc_from, &Account::withdrawn, this, &Bank::serveAccount)
            ;
25
26      emit withdrawalWanted(sum);
27  }
```

Money transfer gets done by serving the target account, which is done by calling method `void serveAccount(int sum)`, but there still exists connection between the bank and the source account. This connection needs to be destroyed after the transfer completes so that another money transfer can be realized.

6.5.3.2 Explanations

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

```
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 which are passed to the slot when connected signal is emitted. Money sum was that argument in previous example.

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

```
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 to 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 job before the actual C++ code compilation (see Figure 11 on page 47). moc goes through every `QObject`-based class in project and seeks signals and slots. If it founds any, then:

1. New source file is created, this file is named `moc_original-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_*.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 $0, 1, 2, 3$.

Qt creates new meta-method which maps external method calls to signal/slot calls if signal is emitted or slot called. This mettod 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

```
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 checked in according to variable `_c`. It states if signal/slot really needs to be invoked or if some other situation occurred. In our case, let's suppose that signal/slot should be called. Execution processes to line 7. Id of wanted element is checked on line and if id is less than four, 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 compile time created entities but connections are not. There is no need to dig into technical aspects of connection. Generally, connection is collection of two pointers (source object and target object) and names of signals or slots. Signals and slots are invoked by name in run time. This 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 73).

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 55) 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 are 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 several object trees this way.

7 Memory management

Qt brings many new features into standard C++ memory management. We can stick with plain `new` and `delete` operator. Qt features are, however, highly addictive and useful. We have already heard about object trees (page 55) which are real base for Qt memory management because tree structure is quite natural. Therefore, object trees are used much and many memory leaks is 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.

8 Event system

Events do not Generally, events are reactions for some other actions. In software, actions can be divided into to 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 click or 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. There are plenty of various types of events. Typical event might be key-press-event or perhaps mouse-click-event or repaint-gui-event.

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 which 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 has one main 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 of Listing 23 results in entering to global event loop, so that events from application objects, e. g. from main application window, can be processed.

Listing 23: Global event loop

```
1 int main(int argc, char *argv[]) {
2     QApplication a(argc, argv);
3 
4     // Display main application window here.
5     .............
6 
7     // int QApplication::exec() triggers global event loop.
8     return a.exec();
9 }
```

Listing 24: Typical event loop structure

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

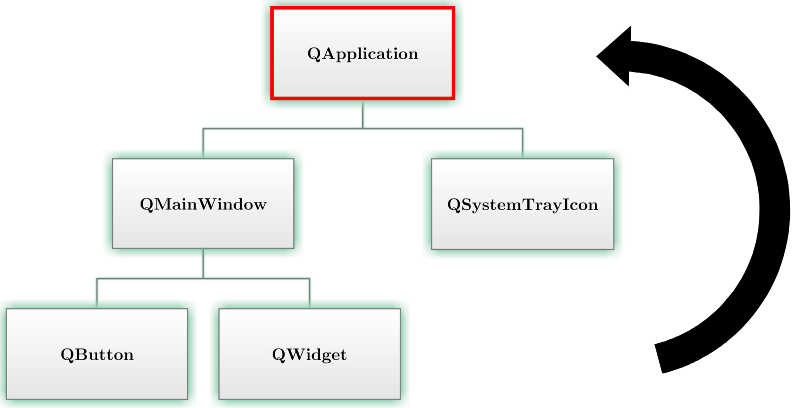


Figure 16: Typical event loop

Let's take a look at Figure 16 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 (object) is posted by `QSystemTrayIcon` and appended to event loop queue for further processing and `QSystemTrayIcon` possibly waits for backward event delivery from event loop.

`QApplication` 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 use `QButton` (ordinary button) instance by mouse, then paint-event is raised, because `QButton` needs to be repainted on the screen, and this event is propagated to parent object which is usually application window which 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` 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 or removed to particular `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
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()));
```

```
    }
    else if (event->removed()) {
        qDebug("Child␣%s␣(%s)␣was␣removed␣from␣%s.",
            event->child()->metaObject()->className(),
            qPrintable(event->child()->objectName()),
            qPrintable(objectName()));
    }

    QObject::childEvent(event);
}
```

8.1 Event filters

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

Listing 26: Using event filter

```
// myqobject.h
class MyQObject : public QObject {
    Q_OBJECT

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

    protected:
        bool eventFilter(QObject *object, QEvent *event);
};

// 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);
}
```

9 Threading

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

9.1 What is thread?

Operating system has only one duty – 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 between 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. They share joint data. This 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. Thread 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-build 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 is used as the backend for Qt threading support on Linux. Windows native threading facilities are used on Windows and, in fact, Qt uses native threading machinery on almost every supported platform.

Threading is very complicated and complex subject but at least basic usage (which is fine for most users) can be introduced. Basic class for working with threads is `QThread`. (Qt-Project, 2012b, QThread class) `QThread` should **never** be subclassed.

Part II

Real-world Qt

List of Figures

List of Tables

List of Listings

List of Abbreviations

ELF	Executable and Linkable Format
GCC	GNU Compiler Collection
GUI	Graphical User Interface
IL	Intermediate Language
JRE	Java Runtime Environment
KDE	K Desktop Environment
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
XML	Extensible Markup Language

References

Du Toit, Stefanus

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

McConnell, Steve C.

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

Nigel, Christian

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

Prata, Stephen

2011 *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).

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).

Index