

**Методические указания к лабораторным работам по курсу  
алгоритмические языки и программирование.  
Дополнительные материалы.**

Самарев Роман Станиславович,  
канд. техн. наук,  
доцент каф. «Компьютерные системы и сети»  
samarev [ ] acm.org

Москва,  
МГТУ им. Н.Э.Баумана,  
2011

## Оглавление

1. Измерение времени выполнения программы.....	3
1.1. ANSI C.....	3
1.2. Windows API.....	3
1.3. Циклы ожидания.....	4
1.4. Пример измерения времени выполнения.....	5
2. Использование стандартных классов контейнеров.....	6
2.1. Классы STL.....	6
2.2. Основные принципы работы с контейнерами.....	7
2.2.1. Добавление элементов в последовательности.....	7
2.2.2. Обращение к элементам .....	8
2.2.3. Использование неупорядоченных типов.....	8
2.2.4. Стандартные алгоритмы.....	9
2.3. Классы Qt.....	9
2.4. Примеры использования QSet и QMap.....	11
3. Графика с использованием библиотеки Qt.....	16
3.1. Система координат.....	16
3.2. Преобразование системы координат.....	17
3.3. Основные функции для формирования графических примитивов.....	17
3.4. Пример программы формирования графического изображения.....	22
3.5. Пример рисования изображения и формирования тела вращения.....	25
4. Документирование исходных кодов программы.....	34
4.1. Комментарии doxygen.....	34
4.2. Специальная разметка .....	35
4.3. Настройка генератора документации.....	36
4.4. Пример оформления исходного текста.....	42

# 1. Измерение времени выполнения программы

При отладке программ актуальна задача оценки времени выполнения конкретного фрагмента, что позволяет понять, насколько оптимально реализован алгоритм. Следует сразу отметить, что при измерении необходимо помнить о том, что большинство функций измерения времени имеют погрешность измерения 1 мс, а их вызов вносит определенные задержки (порядка единиц-десятков мкс). Таким образом, для повышения точности необходимо измерять интервалы, существенно больше 1 мс, например имеющие порядок секунд. Если необходимо измерить время выполнения фрагмента программы существенно меньшей величины следует организовать счетный цикл, реализующий многократное повторение одного и того же фрагмента программы.

Кроме того, следует помнить, что время однократного выполнения одного и того же фрагмента программ в разных состояниях вычислительной системы (оно может менять несколько раз в секунду) может существенно различаться.

Результат измерения при многократном повторении в цикле, приведенный к единичному выполнению, будет показывать усредненное значение времени выполнения, причем в этом случае можно добиться точности существенно выше 1 мс.

Основной принцип измерения заключается в следующем. Перед измеряемым фрагментом необходимо вставить функцию измерения текущего времени (в любых единицах), значение которого помещаем в переменную. После измеряемого фрагмента вставляем аналогичную функцию. Разность последнего и первого измерений времени есть время выполнения фрагмента программы. Пример использования функций приводится ниже.

## 1.1. ANSI C

В соответствии со стандартом ANSI C компилятор C в библиотеке функций должен существовать файл `time.h`, в котором определены функции для работы со временем. Таким образом, функции, определенные в ANSI C не зависят от операционной системы и аппаратной платформы.

```
#include <time.h>
clock_t clock(void);
```

Функция `clock` возвращает число тактов таймера с момента запуска программы. Для преобразования результата в секунды его нужно разделить на макроопределение `CLOCKS_PER_SEC`. Выдается общее количество процессорного времени, прошедшего с момента начала выполнения программы в единицах, определенных машинно-зависимым макро `CLOCKS_PER_SEC`. Если такое измерение провести нельзя, то выдается -1. В большинстве реализаций библиотек C функция `clock()` имеет разрешение в 1 миллисекунду. Если тип `clock_t` объявлен как: `typedef long clock_t;`, то через 24.3 дня возвращаемые значения будут отрицательными.

```
time_t time ( time_t * timer )
```

Функция `time()` предназначена для определения календарного времени в секундах с 1-го января 1970 года. Если значение `timer` не `NULL`, оно будет возвращено в качестве результата. Функция возвращает -1 в случае, если время определить невозможно.

## 1.2. Windows API

Функции Windows целесообразно использовать в тех случаях, когда код программы не планируется переносить на другие платформы.

```
#include <windows.h>
DWORD GetTickCount();
```

Функция `GetTickCount()` возвращает время с момента загрузки Windows в миллисекундах. Максимальное значение - 49.7 дней, после чего счет начинается сначала. В большинстве случаев предпочтительно использование именно этой функции Windows.

```
BOOL QueryPerformanceCounter( LARGE_INTEGER *lpPerformanceCount );
```

Функция `QueryPerformanceCounter` возвращает значение таймера высокой точности. Позиционируется как таймер высокой точности, однако реальная точность возвращаемого значения зависит от версии Windows. Особенность использования данной функции заключается в том, что счетчик связан с конкретным процессором, что требует, чтобы текущий поток выполнялся на одном и том же процессоре (или его ядре). Для этого следует устанавливать маску допустимых процессоров функцией `SetThreadAffinityMask`.

```
BOOL QueryPerformanceFrequency( LARGE_INTEGER *lpFrequency );
```

Функция `QueryPerformanceFrequency()` возвращает частоту таймера высокой точности. Если возвращаемое значение 0 – таймер не существует или получить значение частоты не удалось. В документации MSDN не упоминается, что возвращаемое значение является частотой процессора, однако приближенно можно считать возвращаемую величину именно частотой процессора, а значение, возвращаемое `QueryPerformanceCounter` значением счетчика тактов ядра процессора (у каждого ядра свой счетчик тактов).

### 1.3. Циклы ожидания

Внимание! **Недопустимо использовать в программе** реализацию цикла ожидания, подобную `while (clock() < endwait) {}`, так как для выполнения будет использовано всё доступное процессорное время текущего процессора, расходуемое на определение текущего времени (`clock()`) с максимально возможной скоростью. Другие процессы, выполняемые на текущем компьютере, будут вытеснены планировщиком задач операционной системы на время работы данного цикла (в разных операционных системах поведение будет различаться).

Более правильной реализацией будет `while (clock() < endwait) { Sleep(0);}`, однако точность ожидания будет находиться в пределах одного кванта процессора, т.е. порядка 5-30 миллисекунд и будет зависеть от загрузки текущего процессора.

```
void wait ( int seconds )
{
    clock_t endwait;
    endwait = clock () + seconds * CLOCKS_PER_SEC ;
    while (clock() < endwait) { Sleep(0);}
}
```

В случаях, когда требуется более высокая точность, необходимо пользоваться ждущими таймерами, событиями и другими средствами, которые зависят от используемой операционной системы.

Примечание: реализации указанной выше функции существуют во всех операционных системах и называются `Sleep(int milliseconds)` в MS Windows, `usleep(int microseconds)` в UNIX-системах.

## 1.4. Пример измерения времени выполнения

```
#include <windows.h>

#include <time.h>
#include <stdlib.h>
#include <stdio.h>

int main()
{
    time_t    t0, t1;
    clock_t   c0, c1;
    DWORD     tt0, tt1;

    // ***** средства PerformanceCounter *****
    LARGE_INTEGER pct0, pct1, freq;
    QueryPerformanceFrequency(&freq); // определяем частоту таймера
    HANDLE hThread = GetCurrentThread();
    // устанавливаем выполнение текущего потока на первом ядре первого процессора
    DWORD_PTR prevMask = SetThreadAffinityMask( hThread, 1 );
    // завершаем квант процессорного времени для гарантированного переключения
    // на заданный процессор
    Sleep(0);
    // *****

    // измеряем время начала отсчета 4-мя способами
    // В реальной программе следует выбрать какой-нибудь один!
    t0 = time(NULL);           // способ 1
    c0 = clock();              // способ 2
    tt0 = GetTickCount();      // способ 3 (только MS Windows)
    QueryPerformanceCounter(&pct0); // способ 4 (только MS Windows)

    // ждем 1 секунду
    Sleep(1000);

    // измеряем время окончания 4-мя способами
    t1 = time(NULL);
    c1 = clock();
    tt1 = GetTickCount();
    QueryPerformanceCounter(&pct1);

    // разрешаем выполнение потока на всех доступных процессорах
    SetThreadAffinityMask( hThread, prevMask );

    // выводим результат измерения
    printf ("\tTime in milliseconds\n");
    printf ("\telapsed wall clock time: %ld\n", (long) (t1 - t0)*1000 );
    printf ("\telapsed CPU time:          %f\n",
            (float) (c1 - c0)/CLOCKS_PER_SEC*1000.0 );
    printf ("\telapsed GetTickCount time: %ld\n", (tt1 - tt0) );
    printf ("\telapsed PerformanceCounter time: %d\n",
            (unsigned int) ( ((double)pct1.QuadPart / (double) freq.QuadPart -
            (double)pct0.QuadPart / ( double) freq.QuadPart) * 1000) );

    return 0;
}
```

## 2. Использование стандартных классов контейнеров

Стандартные классы контейнеры широко используются для создания сложных динамических структур данных. Наиболее распространенной является библиотека STL (Standard Template Library) - библиотека шаблонов, входящая в состав стандарта C++ ISO Standard - ISO/IEC 14882. Библиотека STL в настоящее время реализована во всех версиях C++ для всевозможных платформ, в противном случае реализация C++ не будет удовлетворять ISO/IEC 14882. В то же время в последние годы интенсивно развивается библиотека Qt, в которой предпринимается попытка по унификации средств во всех возможных направлениях, в том числе в направлении стандартных контейнеров.

### 2.1. Классы STL

Контейнер	Описание
<b>Последовательности (Массивы/Связанные списки) – упорядоченные коллекции</b>	
vector	Динамический массив, обеспечивающий произвольный доступ подобно массиву языка C с возможностью изменения размера в случае вставки или удаления объекта. Физически выделяется линейная область памяти для всех элементов и обеспечивается перевыделение памяти в случае, если выделенной области недостаточно. В стандартной реализации каждое последующее выделение памяти удваивает текущий объем. Вставка и удаления объекта в конец вектора занимает фиксированное время. Вставка и удаления любого не последнего элемента занимает линейное время. Для массива элементов типа bool существует специальный тип bitset. <code>#include &lt;vector&gt;</code>
list	Двусвязный список, элементы которого в памяти размещаются произвольным образом. В противоположность типу vector, обеспечивается медленный доступ к конкретному элементу (линейное время), зато за постоянное время обеспечивается вставка и удаление любого элемента. <code>#include &lt;list&gt;</code>
deque	Дек (от англ. deq - double ended queue, т.е очередь с двумя концами). Реализован как вектор, в котором вставка или удаление элементов в конец или начало производится за постоянное время, однако корректность итераторов при таких операциях не гарантируется. <code>#include &lt;deque&gt;</code>
<b>Ассоциативные контейнеры – неупорядоченные коллекции</b>	
set	Математическое множество, реализующее операции объединения, пересечение, разность, симметричную разность, а также проверку на включение. Тип данных, из которого формируется множество должен реализовывать оператор <code>operator&lt;()</code> либо должны быть указана конкретная функция сравнения. Тип реализован с использованием балансированного дерева. Вставка или удаление элемента в множестве не нарушает итератор текущего положения во множестве. <code>#include &lt;set&gt;</code>
multiset	Тип аналогичен типу set за исключением того, что возможны дубликаты значений. <code>#include &lt;set&gt;</code>

Контейнер	Описание
<a href="#">map</a>	Ассоциативный массив, обеспечивающий отображение одного типа данных – ключ (key) к другому - значению (value). Тип key должен реализовывать операцию сравнения operator<() либо должна быть указана конкретная функция сравнения. Тип реализован с использованием сбалансированного дерева. <code>#include &lt;map&gt;</code>
multimap	Тип аналогичен типу multiset за исключением того, что возможны дубликаты ключей. <code>#include &lt;map&gt;</code>
<b>Адаптеры контейнеров</b>	
queue priority_queue stack	Контейнеры-интерфейсы к типам-последовательностям, обеспечивающие доступ с использованием соответствующих функций по принципу FIFO, упорядоченная очередь и LIFO, соответственно.

## 2.2. Основные принципы работы с контейнерами

Подключение библиотеки стандартных классов производится включением соответствующего заголовочного файла. Следует отметить, что все классы STL помещены в пространство имен std, следовательно в программе сразу после включения необходимых заголовков целесообразно вписать строку:

```
using namespace std;
```

либо использовать типы вместе со спецификатором std, например std::vector<int>.

Весьма полезным типом библиотеки STL является тип строка – std::string, декларация которого находится в файле <string>, а также классы-потoki ввода/вывода, задекларированные в заголовке <iostream> и имеющие имена cout, cin, cerr для потока вывода, ввода и ошибок, соответственно.

Внимание! Количество элементов контейнеров требует переменную типа **size\_t**, а не int. То же относится к индексу типа vector. Причина заключается в том, что **size\_t** зависит от разрядности ОС и в 64-х битной ОС будет иметь размер 64 бита, в то время как в 32-х битной ОС его размер будет соответствовать типу int, т.е. 32 бита.

### 2.2.1. Добавление элементов в последовательности

Вектор строк:

```
vector<string> str;
```

Добавление в конец вектора:

```
str.push_back( "5" );  
str.push_back( "8" );
```

Добавление после первого элемента при условии, что он существует:

```
str.insert(v.begin() + 1, "Текст" );
```

Список строк:

```
list<string> str;
```

Добавление в конец списка:

```
str.push_back( "5" );
```

Добавление в начало списка:

```
str.push_front( "8" );
```

Добавление элемента перед итератором iter:

```
str.insert(iter, "Текст" );
```

## 2.2.2. Обращение к элементам

Доступ к элементам типа `vector` возможен двумя способами – по индексу либо с использованием итераторов. Обращение к типу `list` возможно только через итератор.

```
vector<int> nums;  
size_t i;
```

Добавляем элементы с использованием:

```
for (i=0; i<10; i++) nums.push_back(i);
```

Выводим значения с использованием прямого индекса:

```
for (i=0; i< nums.size(); i++)  
    cout << " " << nums[i] << endl;
```

Выводим значения с использованием индекса через итератор первого элемента:

```
for (i=0; i< nums.size(); i++)  
    cout << " " << *( nums.begin() + i) << endl;
```

Выводим значения с использованием итераторов:

```
vector<int>::iterator iter;  
for (iter = nums.begin(); iter!=nums.end(); iter++)  
    cout << " " << *(iter) << endl;
```

Если используется тип `list`, то доступ к элементам возможен только через итераторы.

Выводим значения списка в обратном порядке:

```
list<int> nums;  
...  
list<int>::iterator iter;  
for (iter = nums.end(); iter!=nums.begin(); iter--)  
    cout << " " << *(iter) << endl;
```

## 2.2.3. Использование неупорядоченных типов

Пример. Использование множества.

Декларируем множество строк и итератор для дальнейшей работы:

```
set<string> strs;  
set<string>::iterator iter;
```

Добавляем элементы:

```
strs.insert( "5" );  
strs.insert( "8" );  
strs.insert("Текст" );
```

Ищем элемент во множестве:

```
iter=strs.find("123");
```

Если нашли – удаляем:

```
if( iter != strs.end() )  
    strs.erase( iter );
```

Выводим всё, что осталось:

```
for( iter=strs.begin(); iter!=strs.end(); iter++)  
    cout << " " << *(iter) << endl;
```

Пример. Использование ассоциативного массива.

Определяем функцию сравнения строк в порядке убывания:

```
class StrLess  
{  
public:
```



```

bool operator() (const string & s1, const string & s2) const
{
    return s1 > s2;
};
};

```

Определяем тип для сокращения дальнейших записей. Ключ – тип string, значение – тип int, функция сравнения - StrLess.

```
typedef map<string, int, StrLess> StrMap;
```

Декларируем массив.

```
StrMap strs;
StrMap::iterator iter;
```

Добавляем элементы парами ключ-значение. Используем для этого конструкцию ::value\_type для ранее определенного типа:

```
strs.insert(StrMap::value_type( "text1", 1354123) );
strs.insert(StrMap::value_type( "text2", 2634234) );
```

Ищем итератор в массиве по заданному ключу:

```
iter = strs.find( "text3" );
if( iter!=strs.end() )
    strs.erase( iter );
```

Выводим все оставшиеся элементы. Особенность – итератор имеет два поля, поэтому обращение через first и second для ключа и значения, соответственно.

```
for( iter=strs.begin(); iter!=strs.end(); iter++)
    cout << "key=" << (*iter).first <<
         "val=" << (*iter).second << endl;
```

## 2.2.4. Стандартные алгоритмы

Библиотека STL включает набор функций, облегчающих обработку данных, помещенных в классы-контейнеры. Среди таких алгоритмов можно выделить поиск, сортировка, копирование, обмен указанных элементов, замена, сортировка в обратном порядке и пр.

Пример. Сортировка элементов типа vector.

```
vector<int> nums;
```

...

Выполняем сортировку первых пяти элементов (при условии, что они существуют) в естественном для int порядке, т.е. по возрастанию.

```
sort ( nums.begin(), nums.begin()+5 );
```

Выполняем сортировку элементов с пятого и до конца с использованием своей функции сравнения, обеспечивающей сортировку по убыванию.

```
bool cmpfunc (int i,int j) { return (j<i); }
sort (nums.begin()+5, nums.end(), cmpfunc);
```

## 2.3. Классы Qt

Классы контейнеры Qt по составу и программному интерфейсу в значительной степени повторяют классы STL, позволяют выполнять преобразования контейнеров Qt в STL и обратно для обеспечения совместимости с уже написанными приложениями и библиотеками. Основное отличие Qt от STL заключается в том, что Qt не является жестко стандартизированной библиотекой, следовательно может активно изменяться от версии к версии как в части интерфейса так и реализации. Основной аргумент в использовании контейнеров Qt заключается в том, что производитель позиционирует их как более эффективные, чем STL. Для работы с контейнерами предусмотрено два стиля – стиль Java и

стиль STL. Все основные принципы использования STL также применимы к Qt. Классы контейнеры относятся к группе классов Generic Containers.

1. **QList<T>** - Класс хранит список значений данного типа (T), доступ к которым возможен по индексу. Внутренне QList реализован как массив, что обеспечивает быстрый доступ к элементу по индексу. Элементы могут добавляться либо в конец списка, используя QList::append() и QList::prepend(), или могут быть вставлены в середину с использованием QList::insert(). В отличие от других классов контейнеров, QList оптимизирован по производительности и реализован наиболее компактно. Класс QStringList наследует QList<QString>.
2. **QLinkedList<T>** - Класс подобен QList, за исключением того, что для обращения к элементу используются итераторы, а не индекс. Обеспечивает лучшую производительность, чем QList при вставке в середину большого списка и более строгую работу с итераторами. (Итератор-указатель на элемент списка QLinkedList остается актуальным до тех пор, пока элемент существует, в то время как итератор в списке QList может стать ошибочным в случае добавления или удаления элементов.)
3. **QVector<T>** - Класс сохраняет массив значений заданного типа, выравнивая размещение в памяти. Вставка в начало или в середину вектора может быть достаточно медленной, т.к. это приводит к перемещению большого числа элементов в памяти.
4. **QStack<T>** - Класс стека реализован на основе вектора QVector и обеспечивает принцип "last in, first out" (LIFO).
5. **QQueue<T>** - Класс очереди реализован на основе списка QList и обеспечивает принцип "first in, first out" (FIFO) semantics.
6. **QSet<T>** - Обеспечивает формирование математического множества без повторений элементов и их быстрый поиск. Класс основан на QHash, что принципиально отличает его от класса std::set. Значения QSet, извлекаемые с помощью итератора не являются отсортированными. Также следует помнить о расходе памяти при использовании методов, основанных на хэш-таблицах.
7. **QMap<Key, T>** - Обеспечивает возможность формирования словаря (ассоциативный массив), который отображает значения ключей типа Key на значения типа T. Обычно каждый ключ ассоциирован с одним значением. QMap сохраняет данные в порядке следования ключей Key. Если необходим другой порядок, следует использовать класс QHash.
8. **QMultiMap<Key, T>** - Потомок QMap, обеспечивающий интерфейс формирования ассоциативного массива, в котором один ключ может быть ассоциирован с несколькими значениями.
9. **QHash<Key, T>** - Класс почти аналогичен QMap, однако реализует быстрый поиск ключей (по хэш-функции). QHash хранит данные в произвольном порядке.
10. **QMultiHash<Key, T>** - потомок QHash, обеспечивающий интерфейс для хранения нескольких значений для одного ключа.

Пример использования списка:

```
QList<QString> list;
list << "A" << "B" << "C" << "D";

QList<QString>::iterator i;
for (i = list.begin(); i != list.end(); ++i)
    *i = (*i).toLower();
```

Дополнительные алгоритмы обработки данных в контейнерах реализованы в функциях, декларированных в файле заголовков <QtAlgorithms>.

## 2.4. Примеры использования QSet и QMap

Текст программы, приведенной ниже, состоит из четырех отдельных примеров. Код сгруппирован таким образом, чтобы сделать их максимально автономными.

```
#include <QtCore/QCoreApplication>
#include <QSet>
#include <QTextStream>
#include <QTime>

const int MAX_WORDS = 20;
const int MAX_CHARS = 2;
//*****
// Пример использования QSet с типом на основе стандартного

/// класс, нечувствительный к регистру
class InsensitiveString : public QString
{
public:
    /// Функция сравнения. Спецификаторы const являются обязательными!
    /// Используется только в том случае, если хэш-значения совпали,
    /// поэтому сравнивается равенство значений.
    bool operator==( const InsensitiveString & other ) const
    {
        return this->toLower() == other.toLower();
    };
};

/// Определяем хэш-функцию для типа InsensitiveString
uint qHash( const InsensitiveString & var )
{
    /// приводим к нижнему регистру и к стандартной qHash(const QString&)
    return qHash( var.toLower() );
};

void test_qset()
{
    /// используем стандартный поток вывода для вывода текста в консоль
    QTextStream out(stdout);
    out<<"*****"<<__FUNCTION__<<"*****"<<endl;
    /// объявляем множество
    QSet <InsensitiveString> st;
    InsensitiveString str;
    /// заносим случайно сгенерированные значения
    for( int i=0; i<MAX_WORDS; i++ ){
        str.clear();
        for( int j=0; j<MAX_CHARS; j++ )
            str.append( QChar( (qrand()%('z'-'A')) + 'A' ) );
        st.insert( str );
    };
    /// формируем новые строки и проверяем, входят ли
    for( int i=0; i<MAX_WORDS; i++ ){
        str.clear();
        for( int j=0; j<MAX_CHARS; j++ )
            str.append( QChar( (qrand()%('z'-'A')) + 'A' ) );

        /// проверяем, есть ли уже такие значения во множестве
        /// и выводим результат
        if(st.find( str ) != st.end() ){
            out<<"Val " << str << " exist" << endl;
        } else
            out<<"Val " << str << " not exist" << endl;
    };
    /// выводим все элементы множества. Обратите внимание, что элементы
    /// не сортированы !!!
    QSetIterator<InsensitiveString> i(st);
    while (i.hasNext())
        out << i.next() << endl;
};
```

```

//*****
// Пример использования QSet с новым типом
#include <qHash>
// класс, определяющий новый тип
class RawData
{
public:
    /// конструктор по умолчанию инициализирует случайным образом значение
    RawData(){ m_val = qrand()%10 - 5; };

    /// функция получения значения. Спецификатор const здесь необходим!
    int val() const { return m_val;};

    /// Функция сравнения. Спецификаторы const являются обязательными!
    /// Используется только в том случае, если хэш-значения совпали,
    /// поэтому сравнивается равенство значений.
    bool operator== ( const RawData & other ) const
    {
        return this->m_val == other.m_val;
    };
protected:
    int m_val;
};

/// Определяем новую хэш-функцию, позволяющую работать с новым типом
uint qHash( const RawData & var )
{
    // поскольку значение простое, воспользуемся стандартной хэш-функцией
    // для целого типа
    return qHash( var.val() );
};

void test_qset2()
{
    // используем стандартный поток вывода для вывода текста в консоль
    QTextStream out(stdout);
    out<<"*****"<<__FUNCTION__<<"*****"<<endl;

    // определяем множество и заполняем его
    QSet <RawData> st;
    for( int i=0; i<MAX_WORDS; i++ ){
        RawData data;
        st.insert( data );
    };

    // создаем новые элементы и проверяем, есть ли такие же
    for( int i=0; i<MAX_WORDS; i++ ){
        RawData data;
        if(st.find( data ) != st.end() ){
            out<<"Val " << data.val() << " exist" << endl;
        } else
            out<<"Val " << data.val() << " not exist" << endl;
    };

    // выводим все элементы множества.
    // Значения не сортированы!!!
    QSetIterator<RawData> i(st);
    while (i.hasNext())
        out << i.next().val() << endl;
};

```

```

//*****
// Пример использования QMap с типом на основе стандартного
#include <QMap>
// класс, нечувствительный к регистру
class InsensitiveString2 : public QString
{
public:
    // переопределяем функцию сравнения. Поскольку будем использовать QMap,
    // которая использует дерево, необходимо переопределить operator<
    bool operator< ( const InsensitiveString2 & other ) const
    {
        return this->toLower() < other.toLower();
    };
};

void test_qmap()
{
    // используем стандартный поток вывода для вывода текста в консоль
    QTextStream out(stdout);
    out<<"*****"<<__FUNCTION__<<"*****"<<endl;

    // объявляем map, имеющий ключ типа InsensitiveString2 и фиктивное
значение
    QMap <InsensitiveString2, int> st;
    InsensitiveString2 str;
    // заносим случайно сгенерированные ключи с одинаковыми значениями
    for( int i=0; i<MAX_WORDS; i++ ){
        str.clear();
        for( int j=0; j<MAX_CHARS; j++ )
            str.append( QChar( (qrand()%('z'-'A')) + 'A' ) );
        st.insert( str, 0 );
    };
    // формируем новые строки и проверяем, входят ли
    for( int i=0; i<MAX_WORDS; i++ ){
        str.clear();
        for( int j=0; j<MAX_CHARS; j++ )
            str.append( QChar( (qrand()%('z'-'A')) + 'A' ) );

        // проверяем, есть ли уже такие значения во множестве
        // и выводим результат
        if(st.find( str ) != st.end() ){
            out<<"Val " << str << " exist" << endl;
        } else
            out<<"Val " << str << " not exist" << endl;
    };

    // выводим все элементы map.
    // Здесь все элементы выводятся в порядке сортировки
    // (по возрастанию, игнорируя регистр)
    QMapIterator <InsensitiveString2, int> iter(st);
    while (iter.hasNext()){
        iter.next();
        out << iter.key() << endl;
    };
};

```

```

//*****
// Пример использования QMap с новым типом

// класс, определяющий новый тип
class RawData2
{
public:
    /// конструктор по умолчанию инициализирует случайным образом значение
    RawData2(){ m_val = qrand()%10 - 5; };
    /// функция получения значения. Спецификатор const здесь необходим!
    int val() const { return m_val;};

    /// Функция сравнения. Спецификаторы const являются обязательными!
    /// Поскольку используем QMap, то operator<
    bool operator< ( const RawData2 & other ) const
    {
        return this->m_val < other.m_val;
    };
protected:
    int m_val;
};

void test_qmap2()
{
    // используем стандартный поток вывода для вывода текста в консоль
    QTextStream out(stdout);
    out<<"*****"<<__FUNCTION__<<"*****"<<endl;

    // объявляем map, имеющий ключ типа InsensitiveString2, int
    // и заполняем парами ключ-фиктивное значение (не используемое)
    QMap <RawData2, int> st;
    for( int i=0; i<MAX_WORDS; i++ ){
        RawData2 data;
        st.insert( data,0 );
    };

    // создаем новые элементы и проверяем, есть ли такие же
    for( int i=0; i<MAX_WORDS; i++ ){
        RawData2 data;

        // проверяем, есть ли уже такие значения во множестве
        // и выводим результат
        if(st.find( data ) != st.end() ){
            out<<"Val " << data.val() << " exist" << endl;
        } else
            out<<"Val " << data.val() << " not exist" << endl;
    };

    // выводим все элементы map.
    // Здесь все элементы выводятся в порядке сортировки
    QMapIterator <RawData2, int> iter(st);
    while (iter.hasNext()){
        iter.next();
        out << iter.key().val() << endl;
    };
};

```

```

//*****

int main(int argc, char *argv[])
{
    // инициализируем генератор псевдослучайных чисел текущим временем
    QTime time;
    qsrand(time.secsTo(QTime::currentTime()));

    test_qset();
    test_qset2();
    test_qmap();
    test_qmap2();

    // ждем, пока пользователь нажмет Enter
    QTextStream(stdin).readLine();
    return 0;
}

```

Файл проекта для создания консольного Qt-приложения выглядит следующим образом:

```

TEMPLATE = app
TARGET =
DEPENDPATH += .
INCLUDEPATH += .
CONFIG += console

SOURCES += qt_containers.cpp

```

При использовании Qt остается возможность выбора классов контейнеров из библиотеки STL или из комплекта Qt. Если создается библиотека, где использование классов контейнеров необходимо в интерфейсных функциях, лучше выбрать контейнеры STL, реализация которых обязательна для всех версий C++. В остальных случаях выбор за разработчиком.

При использовании той или иной структуры данных следует помнить о необходимости балансирования между временем выполнения и размером занимаемой памяти, как например, в случае выбора между `std::set`, реализованного с помощью дерева и `QSet`, реализованного с помощью хэш-таблицы. Также отметим, что несмотря на то, что с помощью `QMap` можно получить аналог `std::set`, код программы будет содержать не используемые значения, что увеличивает расход памяти и затрудняет понимание кода программы.

Дополнительно см.

- Плаугер П., Степанов А., Ли М., Массер Д. STL - стандартная библиотека шаблонов C++. BHV-Санкт-Петербург, 2004
- Дэвид Р. Мюссер, Жилмер Дж. Дердж, Атул Сейни. C++ и STL. Справочное руководство (2-е издание). Вильямс, 2010.
- Макс Шлее - Qt 4.5. Профессиональное программирование на C++. Издательство: БХВ-Петербург, 2010 г.—т, 896 стр.,— ISBN 978-5-9775-0398-3

### 3. Графика с использованием библиотеки Qt

Qt предоставляет несколько вариантов рисования двумерных объектов. Если предполагается лишь вывод изображений на экран необходимо использовать классы группы рисования (The Paint System). В случае, когда необходимо взаимодействие с нарисованными объектами, следует использовать классы группы графического отображения (The Graphics View Framework). Для ускорения двумерной графики для специальных устройств возможно использование модуля OpenGL. Рисование 3-х мерных сцен требует использования модуля QtOpenGL. Более подробно см. документацию по библиотеке Qt (QtAssistant, <http://doc.trolltech.com/> , частичный перевод: <http://doc.crossplatform.ru/qt/> ).

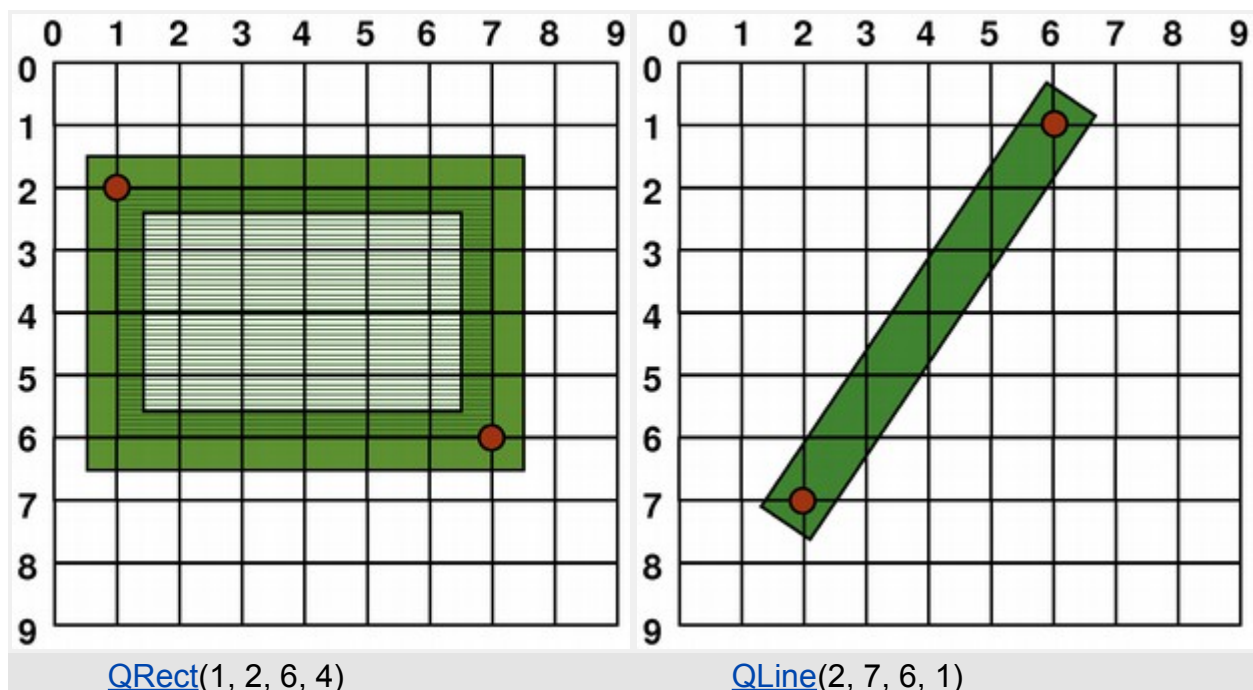
Основным классом группы рисования является класс QPainter предоставляет методы для рисования большинства графических примитивов: [drawPoint\(\)](#), [drawPoints\(\)](#), [drawLine\(\)](#), [drawRect\(\)](#), [drawRoundedRect\(\)](#), [drawEllipse\(\)](#), [drawArc\(\)](#), [drawPie\(\)](#), [drawChord\(\)](#), [drawPolyline\(\)](#), [drawPolygon\(\)](#), [drawConvexPolygon\(\)](#) [drawCubicBezier\(\)](#). Предусмотрены два полезных метода [drawRects\(\)](#) и [drawLines\(\)](#), рисующие заданное количество прямоугольников или линий из массива [QRects](#) или [QLines](#) с использованием текущих пера и кисти. Класс QPainter также предоставляет метод [fillRect\(\)](#), который заливает заданный прямоугольник [QRect](#) с использованием заданной кисти [QBrush](#), а также метод [eraseRect\(\)](#) стирающий область внутри прямоугольника. Все указанные методы имеют реализации как целочисленные, так и с плавающей точкой.

В случае, когда требуется вывод сложных изображений, особенно когда требуется вывод их отдельных частей, целесообразно использование класса [QPainterPath](#) вместо QPainter с тем же набором методов, и метод QPainter::[drawPath\(\)](#) для отображения результата.

Класс [QPainterPath](#) представляет собой контейнер операций рисования, предоставляя возможность сформировать шаблон, который может быть многократно использован.

#### 3.1. Система координат

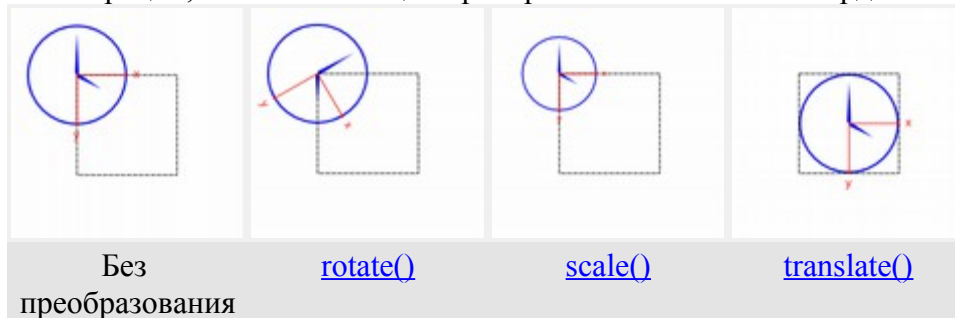
Для формирования изображения используется логическая система координат (см. главу документации The Coordinate System). Начало отсчета – левый верхний угол. Значение x увеличивается по горизонтали направо, значение y – по вертикали вниз. Здесь и далее будут использованы иллюстрации из документации Qt.





### 3.2. Преобразование системы координат

Следует учитывать, что система координат, в которой осуществляется формирование изображения отличается от системы координат устройства вывода. В первую очередь это касается размеров логического пространства, в котором осуществляется формирование изображения и разрешения устройства, на котором будет производиться вывод. Класс QPainter имеет операции, обеспечивающие преобразование системы координат вывода.



```
void QPainter::rotate ( qreal angle )
```

Обеспечивает поворот системы на заданный угол по часовой стрелке.

```
void QPainter::scale ( qreal sx, qreal sy )
```

Масштабирует систему координат. Например, если размер области при построении был 200 пикселей, а устройство отображения имеет разрешение, хранимое в переменной side, то преобразование следует сделать как painter.scale( side / 200.0, side / 200.0);

```
void QPainter::translate ( const QPointF & offset )
```

Смещает начало координат на заданную величину.

### 3.3. Основные функции для формирования графических примитивов

Qt предоставляет несколько классов с близким набором методов. Рассмотрим некоторые одноименные методы классов [QPainterPath](#) и QPainter. Для всех графических примитивов предоставляется возможность выбора способа задания координат и параметров в виде отдельных точек, в виде массива, в виде объектов специализированного класса. Координаты могут задаваться целочисленно или при помощи типа с плавающей точкой.

```
void drawArc ( const QRectF & rectangle, int startAngle, int spanAngle )
```

```
void drawArc ( const QRect & rectangle, int startAngle, int spanAngle )
```

```
void drawArc ( int x, int y, int width, int height, int startAngle, int spanAngle )
```

Вывод дуги в рамках заданного прямоугольника. startAngle, spanAngle указывают угол начала и окончания дуги и имеют значение в 1/16 долях градуса, т.е. полный круг имеет значение 5760 (16 \* 360). Начало отсчета соответствует положению трех часов. Положительные значения указывают направление против часовой стрелки, отрицательные – по часовой.

```
QRectF rectangle(10.0, 20.0, 80.0, 60.0);
int startAngle = 30 * 16;
int spanAngle = 120 * 16;

QPainter painter(this);
painter.drawArc(rectangle, startAngle, spanAngle);
```

```
void drawChord ( const QRectF & rectangle, int startAngle, int spanAngle )
```

```
void drawChord ( const QRect & rectangle, int startAngle, int spanAngle )
```

```
void drawChord ( int x, int y, int width, int height, int startAngle, int spanAngle )
```

Вывод хорды в рамках заданного прямоугольника. Заливка образованной фигуры производится с помощью установленной кисти по умолчанию [brush\(\)](#). Параметры `startAngle`, `spanAngle` указывают угол начала и окончания дуги и имеют значение в 1/16 долях градуса, т.е. полный круг имеет значение 5760 (16 \* 360). Начало отсчета соответствует положению трех часов. Положительные значения указывают направление против часовой стрелки, отрицательные – по часовой.



```
QRectF rectangle(10.0, 20.0, 80.0, 60.0);
int startAngle = 30 * 16;
int spanAngle = 120 * 16;

QPainter painter(this);
painter.drawChord(rect, startAngle, spanAngle);
```

```
void drawConvexPolygon ( const QPointF * points, int pointCount )
void drawConvexPolygon ( const QPoint * points, int pointCount )
void drawConvexPolygon ( const QPolygonF & polygon )
void drawConvexPolygon ( const QPolygon & polygon )
```

Вывод выпуклого многоугольника из массива точек, определяющих координаты вершин. Если заданный многоугольник не выпуклый, т.е. существует хотя бы один угол менее 180 градусов, результат будет непредсказуемым.



```
static const QPointF points[4] = {
    QPointF(10.0, 80.0),
    QPointF(20.0, 10.0),
    QPointF(80.0, 30.0),
    QPointF(90.0, 70.0)
};

QPainter painter(this);
painter.drawConvexPolygon(points, 4);
```

```
void drawEllipse ( const QRectF & rectangle )
void drawEllipse ( const QRect & rectangle )
void drawEllipse ( int x, int y, int width, int height )
void drawEllipse ( const QPointF & center, qreal rx, qreal ry )
void drawEllipse ( const QPoint & center, int rx, int ry )
```

Вывод эллипса, вписанного в указанный прямоугольник. Размер эллипса будет равен размеру прямоугольника плюс размер пера.



```
QRectF rectangle(10.0, 20.0, 80.0, 60.0);

QPainter painter(this);
painter.drawEllipse(rectangle);
```

```
void drawLine ( const QLineF & line )
void drawLine ( const QLine & line )
void drawLine ( const QPoint & p1, const QPoint & p2 )
void drawLine ( const QPointF & p1, const QPointF & p2 )
void drawLine ( int x1, int y1, int x2, int y2 )
```

Вывод линии с указанными точками.



```
QLineF line(10.0, 80.0, 90.0, 20.0);

QPainter painter(this);
painter.drawLine(line);
```

```

void drawLines ( const QLineF * lines, int lineCount )
void drawLines ( const QLine * lines, int lineCount )
void drawLines ( const QPointF * pointPairs, int lineCount )
void drawLines ( const QPoint * pointPairs, int lineCount )
void drawLines ( const QVector<QPointF> & pointPairs )
void drawLines ( const QVector<QPoint> & pointPairs )
void drawLines ( const QVector<QLineF> & lines )
void drawLines ( const QVector<QLine> & lines )

```

Вывод линий из массива или вектора координат.

```

void drawPie ( const QRectF & rectangle, int startAngle, int spanAngle )
void drawPie ( const QRect & rectangle, int startAngle, int spanAngle )
void drawPie ( int x, int y, int width, int height, int startAngle, int spanAngle )

```

Вывод сегмента круга в рамках заданного прямоугольника. Заливка образованной фигуры производится с помощью установленной кисти по умолчанию [brush\(\)](#). Параметры `startAngle`, `spanAngle` указывают угол начала и окончания дуги и имеют значение в 1/16 долях градуса, т.е. полный круг имеет значение 5760 (16 \* 360). Начало отсчета соответствует положению трех часов. Положительные значения указывают направление против часовой стрелки, отрицательные – по часовой.



```

QRectF rectangle(10.0, 20.0, 80.0, 60.0);
int startAngle = 30 * 16;
int spanAngle = 120 * 16;

QPainter painter(this);
painter.drawPie(rectangle, startAngle, spanAngle);

```

```

void drawPoint ( const QPointF & position )
void drawPoint ( const QPoint & position )
void drawPoint ( int x, int y )

```

Вывод точки с заданными координатами.

```

void drawPoints ( const QPointF * points, int pointCount )
void drawPoints ( const QPoint * points, int pointCount )
void drawPoints ( const QPolygonF & points )
void drawPoints ( const QPolygon & points )

```

Вывод точки из массива, либо заданных как вершины многоугольника класса `QPolygon`.

```

void drawPolygon ( const QPointF * points, int pointCount,
                  Qt::FillRule fillRule = Qt::OddEvenFill )
void drawPolygon ( const QPoint * points, int pointCount,
                  Qt::FillRule fillRule = Qt::OddEvenFill )
void drawPolygon ( const QPolygonF & points, Qt::FillRule fillRule = Qt::OddEvenFill )
void drawPolygon ( const QPolygon & points, Qt::FillRule fillRule = Qt::OddEvenFill )

```

Вывод многоугольника по заданному массиву вершин. Возможно указание алгоритма заливки через параметр `fillRule`.



```

static const QPointF points[4] = {
    QPointF(10.0, 80.0),
    QPointF(20.0, 10.0),
    QPointF(80.0, 30.0),
    QPointF(90.0, 70.0)
};

QPainter painter(this);
painter.drawPolygon(points, 4);

```

```

void drawPolyline ( const QPointF * points, int pointCount )
void drawPolyline ( const QPoint * points, int pointCount )
void drawPolyline ( const QPolygonF & points )
void drawPolyline ( const QPolygon & points )

```

Вывод последовательности прямых, соединяющих заданные точки. Отличается от вывода многоугольника тем, что последняя точка не соединяется с первой.

```

static const QPointF points[3] = {
    QPointF(10.0, 80.0),
    QPointF(20.0, 10.0),
    QPointF(80.0, 30.0),
};

QPainter painter(this);
painter.drawPolyline(points, 3);


```

```

void drawRect ( const QRectF & rectangle )
void drawRect ( const QRect & rectangle )
void drawRect ( int x, int y, int width, int height )

```

Вывод прямоугольника с заданными координатами левого верхнего угла и размеров. Размер прямоугольника будет определяться как заданный размер плюс размер пера.



```

QRectF rectangle(10.0, 20.0, 80.0, 60.0);

QPainter painter(this);
painter.drawRect(rectangle);

```

```

void drawRects ( const QRectF * rectangles, int rectCount )
void drawRects ( const QRect * rectangles, int rectCount )
void drawRects ( const QVector<QRectF> & rectangles )
void drawRects ( const QVector<QRect> & rectangles )

```

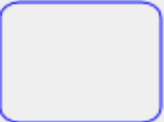
Вывод множества прямоугольников по координатам, заданных в виде массива или вектора координат прямоугольников.

```

void drawRoundedRect ( const QRectF & rect, qreal xRadius, qreal yRadius,
    Qt::SizeMode mode = Qt::AbsoluteSize )
void drawRoundedRect ( const QRect & rect, qreal xRadius, qreal yRadius,
    Qt::SizeMode mode = Qt::AbsoluteSize )
void drawRoundedRect ( int x, int y, int w, int h, qreal xRadius, qreal yRadius,
    Qt::SizeMode mode = Qt::AbsoluteSize )

```

Вывод прямоугольника со скругленными углами с заданными координатами левого верхнего угла и размерами. xRadius, yRadius определяют радиусы эллипса, вписываемого в углы. Если установлен режим скругления Qt::RelativeSize, xRadius, yRadius определяются в процентах от половины ширины/высоты прямоугольника.



```

QRectF rectangle(10.0, 20.0, 80.0, 60.0);

QPainter painter(this);
painter.drawRoundedRect(rectangle, 20.0, 15.0);

```

```

void drawText ( const QPointF & position, const QString & text )
void drawText ( const QPoint & position, const QString & text )
void drawText ( const QRectF & rectangle, int flags, const QString & text,
                QRectF * boundingRect = 0 )
void drawText ( const QRect & rectangle, int flags, const QString & text,
                QRect * boundingRect = 0 )
void drawText ( int x, int y, const QString & text )

```

Вывод текста text начиная с точки, заданной координатами.

```

void drawText ( int x, int y, int width, int height, int flags, const QString & text,
                QRect * boundingRect = 0 )
void drawText ( const QRectF & rectangle, const QString & text,
                const QTextOption & option = QTextOption() )

```

Вывод текста text, вписывая в прямоугольник, заданный координатами и размерами. Имеется возможность указать режим выравнивания текста внутри прямоугольника.

```

Qt by      QPainter painter(this);
Trolltech painter.drawText(rect, Qt::AlignCenter, tr("Qt by\nNokia"));

```

```

void eraseRect ( const QRectF & rectangle )
void eraseRect ( const QRect & rectangle )
void eraseRect ( int x, int y, int width, int height )

```

Удаляет область, заданную координатами прямоугольника.

```

void fillRect ( const QRectF & rectangle, const QBrush & brush )
void fillRect ( int x, int y, int width, int height, Qt::BrushStyle style )
void fillRect ( const QRect & rectangle, Qt::BrushStyle style )
void fillRect ( const QRectF & rectangle, Qt::BrushStyle style )
void fillRect ( const QRect & rectangle, const QBrush & brush )
void fillRect ( const QRect & rectangle, const QColor & color )
void fillRect ( const QRectF & rectangle, const QColor & color )
void fillRect ( int x, int y, int width, int height, const QBrush & brush )
void fillRect ( int x, int y, int width, int height, const QColor & color )
void fillRect ( int x, int y, int width, int height, Qt::GlobalColor color )
void fillRect ( const QRect & rectangle, Qt::GlobalColor color )
void fillRect ( const QRectF & rectangle, Qt::GlobalColor color )

```

Вывод залитого прямоугольника, режим заливки определяется указанной кистью brush или цветом color.

### 3.4. *Пример программы формирования графического изображения*

```
/**
 * @mainpage Формирование элементарных графических изображений
 * В примере показано создание приложения, которое формирует
 * элементарное графическое изображение с использованием некоторых
 * возможностей библиотеки Qt - классов QPainter и QPainterPath.
 * @file main.cpp
 * @brief Запуск приложения.
 * @details Создается окно из класса Painter.
 * Производится запуск цикла обработки сообщений.
 */
#include <QApplication>
#include "painter.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    Painter painter;
    painter.show();
    return app.exec();
}



---



/**
 * @file painter.h
 * @brief Декларация классов программы.
 */
#include <QWidget>

class RenderArea;
/**
 * @class Painter
 * @brief Главное окно приложения.
 * @details Обеспечивает формирование окна приложения, содержащего
 * область отображения рисунка и кнопку выхода
 */
class Painter : public QWidget
{
    Q_OBJECT

public:
    Painter();

protected:
    RenderArea * m_RenderArea; ///< Область отображения рисунка.
};

/**
 * @class RenderArea
 * @brief Область отображения рисунка.
 * @details Формирует, запоминает и отображает набор графических примитивов
 */
class RenderArea : public QWidget
{
    Q_OBJECT

public:
    RenderArea(QWidget *parent = 0);

    /// @brief Минимальный размер области отображения
    /// @details Изменение размеров окна приложения производится с учетом
```

```

    /// допустимых размеров всех элементов. Функция обеспечивает корректное
    /// отображение области графических примитивов
    /// @return Допустимые минимальные размеры
    QSize minimumSizeHint() const;

    /// @brief Размер области отображения по умолчанию
    /// @details При создании окна автоматически рассчитываются размеры
    /// по указанным рекомендованным размерам всех элементов.
    /// @return рекомендованные размеры
    QSize sizeHint() const;

protected:
    /// @brief Обработчик события перерисовки
    virtual void paintEvent(QPaintEvent * event);
private:
    QPainterPath m_Path; ///< Шаблон отображения графических примитивов.
};



---


/**
 * @file painter.cpp
 * @brief Реализация классов программы.
 */
#include <QPainter>
#include <QPushButton>
#include <QVBoxLayout>
#include "painter.h"

/// @brief Конструктор главного окна
/// @details Устанавливает режим выравнивания элементов и создает их.
Painter::Painter()
{
    QVBoxLayout *layout = new QVBoxLayout;

    m_RenderArea = new RenderArea( this );
    QPushButton * btn = new QPushButton( "Close", this );
    QObject::connect(btn, SIGNAL(clicked(bool)),
        this, SLOT(close()));

    layout->addWidget( m_RenderArea );
    layout->addWidget( btn );

    setLayout(layout);
};

/// @brief Конструктор области отображения
/// @param parent - Если значение NULL, то область отображения станет
/// самостоятельным окном. В противном случае - виджетом-потомком.
/// @details Именно в этом методе формируется набор графических примитивов,
/// которые будут отображены при перерисовке.
RenderArea::RenderArea(QWidget *parent)
: QWidget(parent)
{
    setBackgroundRole(QPalette::Base);
    // Формируем прямоугольник
    m_Path.moveTo(20.0, 30.0);
    m_Path.lineTo(80.0, 30.0);
    m_Path.lineTo(80.0, 70.0);
    m_Path.lineTo(20.0, 70.0);
    // Замыкаем контур - проводим линию от последней точки к первой
    m_Path.closeSubpath();

    // Формируем сектор круга (в англ. - pie - кусок пирога)
    m_Path.moveTo(50.0, 50.0);

```

```

    m_Path.arcTo(20.0, 30.0, 60.0, 40.0, 60.0, 240.0);
    // Замыкаем контур
    m_Path.closeSubpath();
}

QSize RenderArea::minimumSizeHint() const
{
    return QSize(50, 50);
}

QSize RenderArea::sizeHint() const
{
    return QSize(100, 100);
}

void RenderArea::paintEvent(QPaintEvent *)
{
    // Готовим объект отображения
    QPainter painter(this);

    // Определяем режим масштабирования со сглаживанием
    painter.setRenderHint(QPainter::Antialiasing);

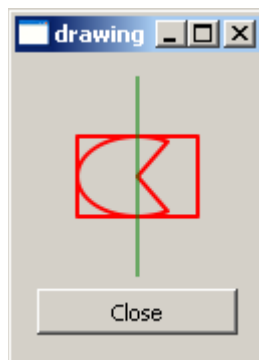
    // Определяем масштаб в зависимости от текущих геометрических размеров
    // области отображения
    painter.scale(width() / 100.0, height() / 100.0);

    // Создаем перо для отображения контуров. Используем красный цвет и толщину
    // линий равную 2.
    painter.setPen(
        QPen(QColor(255,0,0), 2, Qt::SolidLine, Qt::RoundCap, Qt::RoundJoin));

    // Отображаем контуры
    painter.drawPath( m_Path );

    // Нарисуем дополнительную вертикальную линию
    painter.setPen(
        QPen(QColor(0,127,0), 1, Qt::SolidLine, Qt::RoundCap, Qt::RoundJoin));
    painter.drawLine(50.0, 0.0, 50.0, 100.0);
}

```



**Рисунок 1 – Внешний вид окна приложения формирования изображения**



### 3.5. *Пример рисования изображения и формирования тела вращения*

```
/**
 * @mainpage Формирование элементарных графических изображений
 * В примере показано создание приложения, которое формирует
 * элементарное графическое изображение с использованием некоторых
 * возможностей библиотеки Qt – классов QPainter и QPainterPath.
 * @file main.cpp
 * @brief Запуск приложения.
 * @details Создается окно из класса Painter.
 * Производится запуск цикла обработки сообщений.
 */
#include <QApplication>
#include <QTranslator>
#include "painter.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    // Устанавливаем язык для преобразования строк, использующих функции
    // tr() или translate().
    // В данном примере они содержатся в файле painter.cpp.
    // Внимание! Строки, которые будут преобразовываться через QTranslator
    // должны содержать только латинские символы!
    // Файл ru.ts сгенерирован утилитой lupdate ../painter.cpp -ts ru.ts и
    // отредактирован в Qt Linguist
    // Файл ru.qm сформирован утилитой lrelease ru.ts
    QTranslator translator;
    // путь, начинающийся с :/ означает, что требуемый файл находится
    // в файле-ресурсе приложения. Если оставить только translations/ru.qm,
    // то приложение будет искать файл в директории translations относительно
    // текущей директории
    if( translator.load(":/translations/ru.qm") )
        qApp->installTranslator(&translator);

    Painter painter;
    painter.show();
    return app.exec();
}



---


/**
 * @file painter.h
 * @brief Декларация классов программы.
 */
#include <QWidget>
#include <QVector>
#include <QFrame>

class RenderArea;
class InputImage;
class QLineEdit;
/**
 * @class Painter
 * @brief Главное окно приложения.
 * @details Обеспечивает формирование окна приложения, содержащего
 * область отображения рисунка и кнопку выхода
 */
class Painter : public QWidget
{
    Q_OBJECT
```

```

public:
    Painter();

protected:
    RenderArea * m_RenderArea; ///< Область отображения рисунка.
    InputImage * m_InputArea;  ///< Область рисования кривой для вращения.
    QLineEdit * m_Exy;         ///< Шаг поворота при формировании тела
    вращения.
    QLineEdit * m_Eyz;         ///< Угол поворота в градусах плоскости yz
    QLineEdit * m_Exz;         ///< Угол поворота в градусах плоскости xz

    public slots:
        void convert();
};

///< @class BaseRenderArea
///< @brief Базовый класс для панелей ввода и отображения.
class BaseRenderArea: public QFrame
{
    Q_OBJECT

public:
    ///<@brief Структура для хранения координат в массиве
    struct PointCoords {
        int x;
        int y;
        PointCoords() { x=y=0; };
        PointCoords(int x1, int y1) {
            x=x1; y=y1;
        };
    };
    BaseRenderArea(QWidget *parent = 0);

    ///< @brief Минимальный размер области отображения
    ///< @details Изменение размеров окна приложения производится с учетом
    ///< допустимых размеров всех элементов. Функция обеспечивает корректное
    ///< отображение области графических примитивов
    ///< @return Допустимые минимальные размеры
    QSize minimumSizeHint() const;

    ///< @brief Размер области отображения по умолчанию
    ///< @details При создании окна автоматически рассчитываются размеры
    ///< по указанным рекомендованным размерам всех элементов.
    ///< @return рекомендованные размеры
    QSize sizeHint() const;

    ///< @brief Формирует тело вращения по указанному массиву координат
    void setPoints( const QVector<PointCoords> &,
        double xy_step_angle, double yz_angle, double xz_angle );

protected:
    float m_PenWidth; ///< Ширина пера отображения по умолчанию.

    ///< @brief Обработчик события перерисовки
    virtual void paintEvent(QPaintEvent * event);

    QPainterPath m_Path; ///< Шаблон отображения графических примитивов.
};

/**
@class RenderArea
@brief Область отображения тела вращения.
@details Формирует и отображает рисунок
*/

```

```

class RenderArea : public BaseRenderArea
{
    Q_OBJECT

public:
    RenderArea(QWidget *parent = 0);

    /// @brief Метод для формирования тела вращения по массиву значений
    void setPoints( const QVector<PointCoords> &,
                   double xy_step_angle, double yz_angle, double xz_angle );
};

/**
@class InputImage
@brief Область формирования кривой.
@details Обеспечивает формирование, отрисовку кривой.
*/
class InputImage : public BaseRenderArea
{
    Q_OBJECT

public:
    InputImage(QWidget *parent = 0);

    /// @brief Метод для сканирования введенного изображения
    void getPoints( QVector<PointCoords> & );

    public slots:
        /// @brief Очистка изображения
        void clearImage();

protected:
    /// @brief Обработчик события нажатия клавиши мыши
    virtual void mousePressEvent ( QMouseEvent * event );
    /// @brief Обработчик события перемещения мыши
    virtual void mouseMoveEvent ( QMouseEvent * event );

    /// @brief Вспомогательный метод преобразования координат
    QPoint from(const QPoint&);
};



---


/**
@file painter.cpp
@brief Реализация классов программы.
*/
#include <QPainter>
#include <QPushButton>
#include <QVBoxLayout>
#include <QHBoxLayout>
#include <QMouseEvent>
#include <QLineEdit>
#include <QSpacerItem>
#include <QImage>
#include <QLabel>
#include "painter.h"

#define _USE_MATH_DEFINES
#include "math.h"

// Размер поля отображения
const float _x_aspect=400;
const float _y_aspect=400;

/// @brief Конструктор главного окна
/// @details Устанавливает режим выравнивания элементов и создает их.
Painter::Painter()

```

```

{
    QVBoxLayout *layout = new QVBoxLayout;
    QHBoxLayout *hlayout = new QHBoxLayout;

    // Создаём панели для ввода кривой и вывода тела вращения
    m_InputArea = new InputImage( this );
    m_RenderArea = new RenderArea( this );

    // формируем отдельную панель для органов управления
    QFrame * controls = new QFrame();
    QVBoxLayout *controllayout = new QVBoxLayout;
    controls->setLayout( controllayout );
    controls->setSizePolicy( QSizePolicy::Fixed, QSizePolicy::Expanding );

    m_Exy=new QLineEdit( "30", this );
    m_Exz=new QLineEdit( "0", this );
    m_Eyz=new QLineEdit( "50", this );
    QPushButton * btn_convert = new QPushButton( tr("Convert") );
    QPushButton * btn_clear = new QPushButton( tr("Clear") );

    QObject::connect(btn_clear, SIGNAL(clicked(bool)),
        m_InputArea, SLOT(clearImage()));

    QObject::connect(btn_convert, SIGNAL(clicked(bool)),
        this, SLOT(convert()));

    // наполняем панель с органами управления
    controllayout->addWidget( new QLabel(tr("xy step")) );
    controllayout->addWidget( m_Exy );
    controllayout->addWidget( new QLabel(tr("xz angle")) );
    controllayout->addWidget( m_Exz );
    controllayout->addWidget( new QLabel(tr("yz angle")) );
    controllayout->addWidget( m_Eyz );
    controllayout->addWidget( btn_convert );
    controllayout->addItem( new QSpacerItem(10,10, QSizePolicy::Minimum,
QSizePolicy::Expanding) );
    controllayout->addWidget( btn_clear );

    // устанавливаем порядок отображения панелей
    hlayout->addWidget( m_InputArea );
    hlayout->addWidget( controls );
    hlayout->addWidget( m_RenderArea );

    QPushButton * btn = new QPushButton( tr("Close"), this );
    QObject::connect(btn, SIGNAL(clicked(bool)),
        this, SLOT(close()));

    // формируем окончательный вид, добавляем кнопку Close
    layout->addLayout(hlayout);
    layout->addWidget( btn );

    setLayout(layout);
};

/// @details Сканирует панель ввода, преобразует введенные значения углов,
///          запускает формирование тела вращения
void Painter::convert()
{
    double xy_step_angle, yz_angle, xz_angle;
    QVector<BaseRenderArea::PointCoords> data;

    xy_step_angle = m_Exy->text().toDouble()*M_PI/180.0;
    yz_angle = m_Eyz->text().toDouble()*M_PI/180.0;
    xz_angle = m_Exz->text().toDouble()*M_PI/180.0;

```

```

        m_InputArea->getPoints(data);
        m_RenderArea->setPoints(data, xy_step_angle, yz_angle, xz_angle);
};

/// @details Конструктор базового класса панелей ввода и отображения.
///          Устанавливает формат рамки и инициализирует общие переменные.
/// @param parent - Если значение NULL, то область отображения станет
///               самостоятельным окном. В противном случае - виджетом-потомком.
BaseRenderArea::BaseRenderArea(QWidget *parent):QFrame(parent)
{
    // формируем рамку вокруг виджета
    setFrameStyle(QFrame::Panel | QFrame::Raised);
    setLineWidth(2);
    //Установим перо по умолчанию, равное 2
    m_PenWidth=2;
};

QSize BaseRenderArea::minimumSizeHint() const
{
    return QSize(100, 100);
}
QSize BaseRenderArea::sizeHint() const
{
    return QSize(_x_aspect, _y_aspect);
}

void BaseRenderArea::paintEvent(QPaintEvent *event)
{
    // Готовим объект отображения
    QPainter painter(this);

    // Определяем режим отображения со сглаживанием
    painter.setRenderHint(QPainter::Antialiasing);

    // Определяем масштаб в зависимости от текущих геометрических размеров
    // области отображения
    painter.scale( width()/_x_aspect, height()/_y_aspect );

    // Создаем перо для отображения контуров. Используем красный цвет и
толщину
    // линий равную m_PenWidth.
    painter.setPen(
        QPen(QColor(255,0,0), m_PenWidth, Qt::SolidLine, Qt::RoundCap,
Qt::RoundJoin));

    // Отображаем контуры
    painter.drawPath( m_Path );

    QFrame::paintEvent(event); // рисуем рамку вокруг панели вызовом предка
}

/// @brief Конструктор области отображения
RenderArea::RenderArea(QWidget *parent)
: BaseRenderArea(parent)
{
    setBackgroundRole(QPalette::Base);
    QFont font("Arial", 12);
    m_Path.addText( 10, _y_aspect/2, font, tr("Draw on the left panel"));
    m_PenWidth=0.5;
}

/// @param data - массив точек кривой
/// @param xy_step_angle - шаг поворота при формировании тела вращения в
градусах
/// @param yz_angle - угол поворота в градусах плоскости yz

```

```

/// @param xz_angle - угол поворота в градусах плоскости xz
void RenderArea::setPoints( const QVector<PointCoords> & data,
                           double xy_step_angle, double yz_angle,
double xz_angle )
{
    m_Path=QPainterPath(); // очищаем область вывода

    if(!data.size())return; // Если на входе ничего нет - выходим

    double phi;
    double x,y,z;
    double xw, yw;

    // для ускорения работы программы, синусы и косинусы
    // вычисляем заранее
    double cos_yz=cos(yz_angle), sin_yz=sin(yz_angle);
    double cos_xz=cos(xz_angle), sin_xz=sin(xz_angle);

    // цикл, формирующий тело вращения из линии
    for( phi=M_PI/24; phi<2*M_PI+M_PI/24; phi+=xy_step_angle){
        const PointCoords *point;
        double sin_r1=sin(phi), cos_r1=cos(phi);
        for(int i=0; i<data.size();i++){
            point=&data[i];

            // переназначаем координаты.
            z=point->x;
            x=point->y;
            // смотрим вдоль z. Поворачиваем плоскость x-y.
            // Поскольку y=0, вычисления относительно y отбрасываем
            xw=x*cos_r1; //-y*sin_fi;
            y=x*sin_r1; //+y*cos_fi;
            x=xw;

            // Поворачиваем в плоскости y-z
            yw=y*cos_yz-z*sin_yz;
            z=y*sin_yz+z*cos_yz;
            y=yw;

            // Поворачиваем в плоскости x-z
            xw=x*cos_xz-z*sin_xz;
            //z=x*sin_fi+z*cos_fi;// z - дальше не используется
            x=xw;

            //проецируем на экран, отбросив z
            //Смещаем в середину панели и масштабируем с коэффициентом 0.5
            xw= _x_aspect/2+x*0.5;
            yw=_y_aspect/2+y*0.5;

            if( !i ) m_Path.moveTo(xw,yw);
            else m_Path.lineTo(xw,yw);
        }
    }
    update();
};

InputImage::InputImage(QWidget *parent)
:BaseRenderArea(parent)
{
};

/// @details Преобразует координаты мыши в локальные координаты виджета
QPoint InputImage::from(const QPoint &point){
    QPoint pos = mapFrom( this, point );

```

```

        return QPoint( pos.x()* _x_aspect/ width(), pos.y()*_y_aspect/ height() );
};

void InputImage::mousePressEvent ( QMouseEvent * event )
{
    //Если нажата левая кнопка мыши - рисуем
    if (event->buttons () & Qt::LeftButton){
        m_Path.moveTo( from(event->pos()) );
    }
}

void InputImage::mouseMoveEvent ( QMouseEvent * event )
{
    if (event->buttons () & Qt::LeftButton){
        m_Path.lineTo( from(event->pos()) );
    }
    // принудительно отрисовываем. Внимание! реализация упрощенная.
    // На каждое движение мыши будет отрисовываться всё, что было введено
    ранее
    update();
};

void InputImage::clearImage()
{
    m_Path=QPainterPath(); // Создаём пустой объект.
    update();// Очищаем изображение.
};

void InputImage::getPoints( QVector<PointCoords> & data )
{
    data.clear(); // очищаем массив на случай, если там что-то было
    QColor color(255,0,0); QRgb rgb_color=color.rgb();
    // формируем устройство отображения
    QImage image( _x_aspect, _y_aspect, QImage::Format_RGB32 );
    QPainter painter(&image);
    painter.setPen( QPen(color, 1, Qt::SolidLine, Qt::RoundCap,
Qt::RoundJoin));
    //выводим сохраненный рисунок
    painter.drawPath( m_Path );

    //сканируем столбцы изображения в матрице пикселей
    for(int x=0; x<_x_aspect; x++){
        //двигаемся снизу изображения
        for(int y=image.height(); y>=0; y--){
            QRgb rgb=image.pixel(x,y);
            //запоминаем первую точку столбца с соответствующим цветом
            if( rgb==rgb_color ){
                data.push_back( PointCoords(x,image.height()-y));
                break;
            }
        }
    }
};

```

---

Файл перевода translations/ru.ts, сформированный следующей утилитой:

**lupdate ../painter.cpp -ts ru.ts**

и отредактированный в программе Qt Linguist.

```

<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE TS>
<TS version="2.0" language="ru_RU">
<context>
    <name>Painter</name>
    <message>
        <location filename="../painter.cpp" line="42"/>
        <source>Convert</source>
    </message>

```

```

        <translation type="unfinished">Преобразовать</translation>
</message>
<message>
    <location filename="../../painter.cpp" line="43"/>
    <source>Clear</source>
    <translation type="unfinished">Очистить</translation>
</message>
<message>
    <location filename="../../painter.cpp" line="51"/>
    <source>xy step</source>
    <translation type="unfinished">шаг вращения</translation>
</message>
<message>
    <location filename="../../painter.cpp" line="53"/>
    <source>xz angle</source>
    <translation type="unfinished">угол xz</translation>
</message>
<message>
    <location filename="../../painter.cpp" line="55"/>
    <source>yz angle</source>
    <translation type="unfinished">угол yz</translation>
</message>
<message>
    <location filename="../../painter.cpp" line="66"/>
    <source>Close</source>
    <translation type="unfinished">Закрыть</translation>
</message>
</context>
<context>
    <name>RenderArea</name>
    <message>
        <location filename="../../painter.cpp" line="136"/>
        <source>Draw on the left panel</source>
        <translation type="unfinished">Рисовать слева</translation>
    </message>
</context>
</TS>

```

Внимание! В файле ресурсов подключается результат компиляции файла перевода translations/ru.qm, который необходимо получить вызовом **lrelease ru.ts**. Если перевод не требуется, не следует подключать в файле проекта использование файла-ресурсов приложения и этого перевода.

---

Файл ресурсов приложения g2.qrc

```

<RCC>
    <qresource prefix="/" >
        <file>translations/ru.qm</file>
    </qresource>
</RCC>

```

---

Файл проекта g2.pro (может быть сформирован автоматически):

```

TEMPLATE = app
TARGET =
DEPENDPATH += . translations
INCLUDEPATH += .

HEADERS += painter.h
SOURCES += main.cpp painter.cpp
RESOURCES += g2.qrc

```



Приложение позволяет нарисовать кривую (левой клавишей мыши на левой панели) и, по нажатию кнопки «Преобразовать», обеспечивает формирование тела вращения с указанными параметрами: шаг поворота кривой, а также угла наклона плоскостей xz и yz. Все значения указываются в градусах.

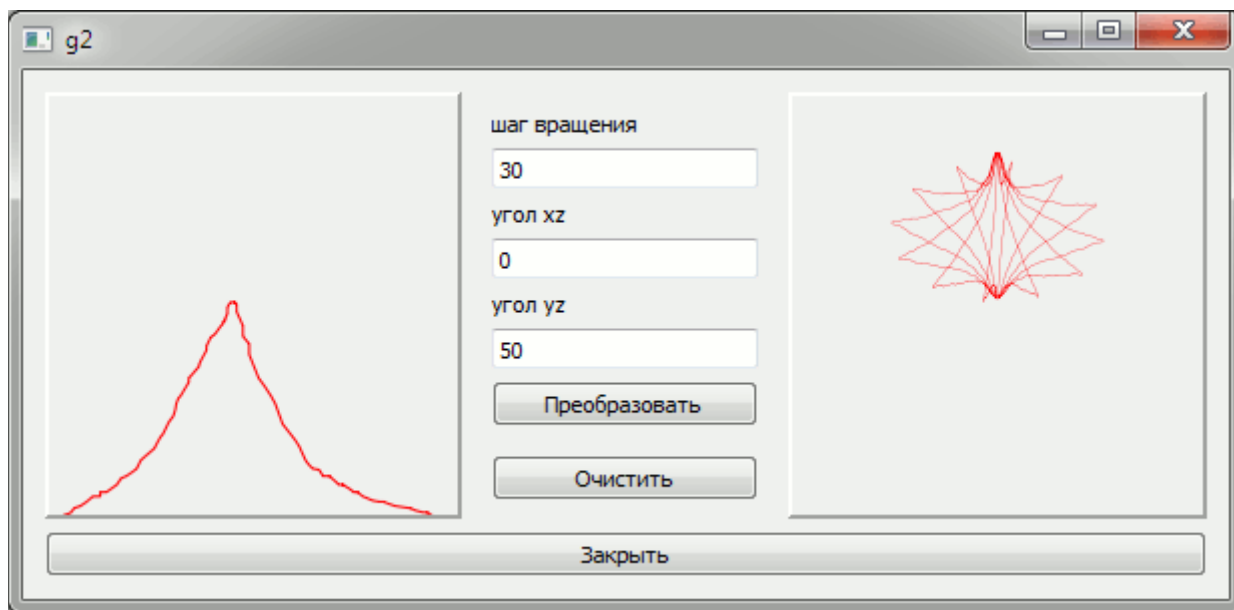


Рисунок 2 – Внешний вид приложения формирования тела вращения

Более подробно см.:

- примеры из комплекта Qt - qt\examples\painting\
- Макс Шлее - Qt 4.5. Профессиональное программирование на C++. Издательство: БХВ-Петербург, 2010 г.–т, 896 стр.,– ISBN 978-5-9775-0398-3
- А. В. Казанцев. Основы компьютерной графики для программистов. Учебное пособие. Казань, 2005.

## 4. Документирование исходных кодов программы

При ведении коллективно разработки одной из задач программиста является оформление исходных текстов программ. Для решения этой задачи разработано множество программных продуктов, которые позволяют совместить процесс формирования документации по исходным кодам с процессом их написания. Т.е. входными данными для них являются сами исходные тесты программ, а на выходе формируется документация в заданном формате. Среди бесплатных продуктов можно назвать следующие: AsmDoc, AutoDOC, Autoduck, CcDoc, CppDoc, Cxref, cxxwrap, Cxx2HTML, C2HTML, Doc++, DocClass, Doxygen, DoxyS (Doxygen fork/spinoff), Epydoc, gtk-doc, HappyDoc, HeaderDoc, HTMLgen, HyperSQL, Javadoc, KDoc, Natural Docs, phpDocumentor, PHPDoc, ReThree-C++, RoboDoc, ScanDoc, Synopsis, Tydoc, VBDOX.

Ключевой особенностью всех этих программ является способность прочитать исходные тексты программы, выделить все классы, методы, атрибуты, функции, построить зависимости между ними, а также выделить из исходных текстов комментарии, которые оставили программисты с тем, чтобы поместить их в формальный документ, формируемый на выходе. Большинство генераторов документации ориентировано на использование определенных языков программирования. Например Javadoc разработан компанией Sun Microsystems для формирования документации по библиотекам программирования Java, которые они же и разработали. Вследствие этого Javadoc является де-факто стандартом оформления исходных текстов программ, написанных на языке Java.

Применительно к языку программирования C++ не существует единого стандарта оформления исходных тестов, однако наиболее распространенным средством является doxygen (<http://www.stack.nl/~dimitri/doxygen/>). Это средство реализовано для большинства операционных систем, включая MS Windows, Linux, MacOS. Следует заметить, что doxygen способен работать с исходными текстами полностью без комментариев, однако в этом случае в результирующем отчете будут присутствовать лишь формальные имена без описания назначения.

### 4.1. Комментарии doxygen

Doxygen по умолчанию игнорирует все комментарии в программе, за исключением специально выделенных следующими способами:

Комментарии, относящиеся к строке, следующей за ними:

```
/// Функция что-то делает
void func1(int param1);

/**
 Функция делает что-то другое
*/
void func2(int param1);
```

Комментарий, относящийся к строке, в которой он находится:

```
void func1(
    int param1 ///<< Параметр предназначен для указания количества чего-то
);
```

Компилятор воспринимает все эти виды комментариев как обычные комментарии. Однако то, что doxygen реагирует лишь на специальную разметку позволяет совмещать комментарии, которые должны быть помещены в итоговый документ, например относящиеся к интерфейсу библиотеки, и те комментарии, которые предназначены лишь для программистов, которые будут в дальнейшем заниматься модификацией кода программы.

## 4.2. Специальная разметка

Применение комментариев специального вида является лишь первым шагом в разметке документа. Doxygen поддерживает набор служебных слов, позволяя разметить текст комментария таким образом, чтобы указать, что указанное имя является классом, параметром, возвращаемым значением, может принимать указанные значения, реализуется указанная формула и пр.

Команды должны быть написаны внутри комментария, воспринимаемого doxygen, причем команда начинается с префикса \ или @.

```
/**
 * @brief Функция делает что-то
 * @param pArr - массив элементов
 * @param param1 - количество элементов в массиве
 */
void func(const long * pArr, int count );
```

В примере иллюстрируются команды @brief и @param, позволяющие указать, что текст является кратким описанием самой функции и перечислить параметры. Это позволит doxygen стилистически правильно оформить описание функции и её параметров.

Полный список команд приводится в документации doxygen-manual в разделе Special Commands. Перечислим наиболее часто используемые команды.

Команда	Назначение
@brief	Краткое описание сущности, к которой относится данная команда (структура данных, функция, файл, страница текста). Краткие описания в итоговом отчете отображаются в таблицах с именами сущностей, к которым они относятся (по группам).
@details	Подробное описание сущности, к которой относится данная команда
@file	Имя файла, в котором находится описание. Позволяет формально отделить секцию, относящуюся к описанию файла, например от секции описания класса.
@addtogroup	При создании больших программных библиотек исходный код разбивается на модули. В то же время необходима единая документация на все исходные тексты. В этом случае целесообразно разбить описываемые разделы на группы, например по принципу отнесения исходных текстов к тому или иному модулю. Соответственно, команде addtogroup указывают имя группы, к которой необходимо отнести текущий файл.
@param	Описание параметра функции или метода. Для каждого параметра необходимо указывать свою команду param
@return	Описание возвращаемого значения функции.
@class	Указывает, что имя, после команды является классом.
@fn	Указывает, что имя, после команды является функцией.
@struct	Указывает, что имя, после команды является структурой.
@mainpage	Команда для добавления текста на главную страницу отчета.

### 4.3. Настройка генератора документации

Doxygen является консольным приложением, выполняющим формирование отчета в соответствии с заданными в параметрами, которые могут находиться например в файле doxyfile. Этот файл является текстовым, где перечислены параметры doxygen и их значения. Параметрами задаются директории с исходными текстами программ, опции обработки текстов, опции формирования результата. Для облегчения создания и управления этим файлом существуют графические оболочки, например doxywizard или doxygate.

Рассмотрим создание документации при помощи программы doxywizard. После запуска doxywizard (в меню Пуск/Программы/doxygen) будет отображено окно, подобно изображенному на рисунке 2.

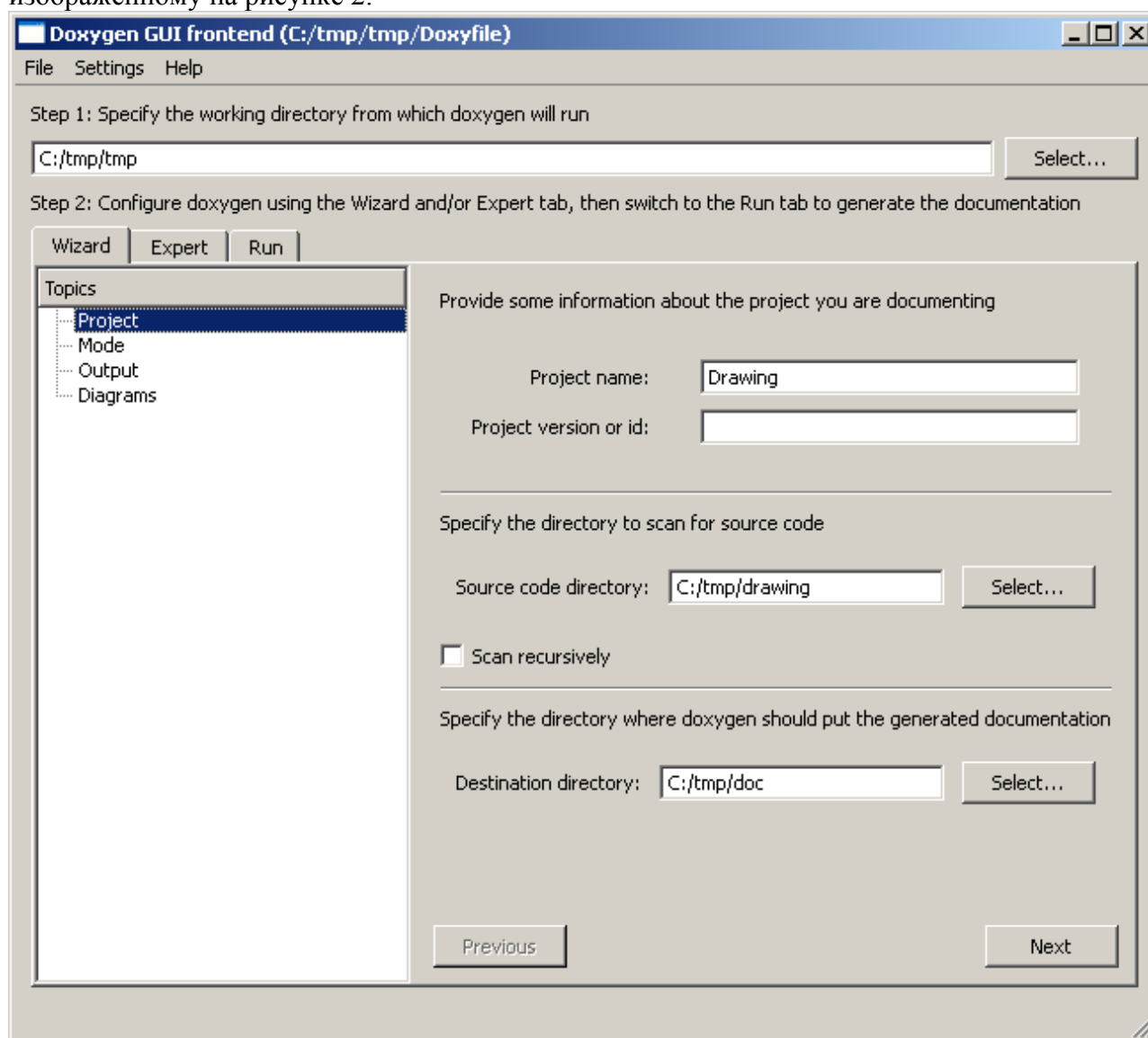
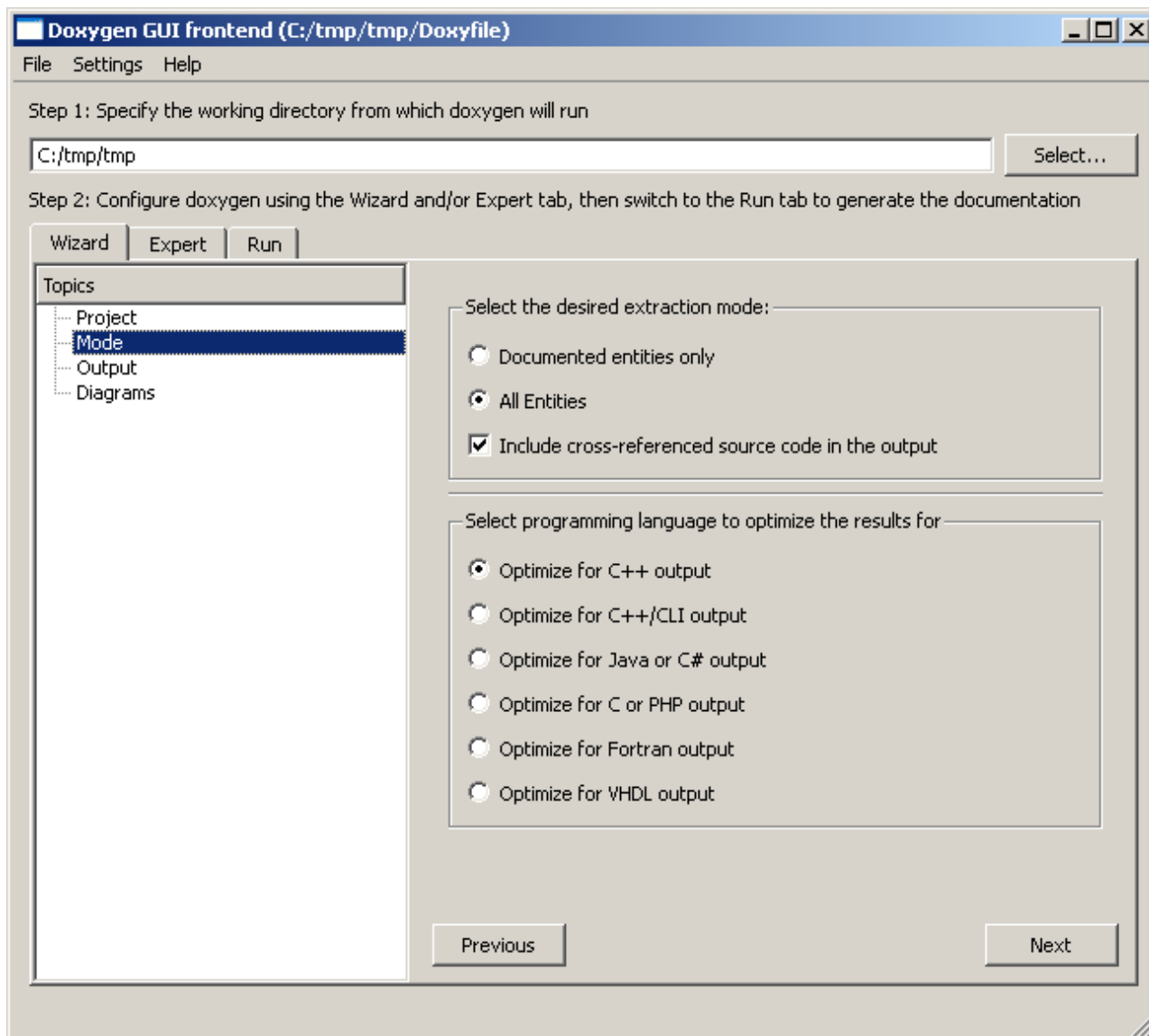


Рисунок 3 – Окно doxywizard. Ввод основных параметров.

Необходимо указать рабочую директорию (working directory), т.е. директорию, в которой будет размещаться временные файлы при создании документации, указать имя проекта (Project name), директорию с исходными кодами (Source code directory), а также директорию, в которую будет помещена документация (Destination directory). Если в проекте исходные тексты размещены в нескольких поддиректориях необходимо отметить “Scan recursively”.

После настройки основных параметров следует переключиться к закладке Mode (см. рисунок 3).



**Рисунок 4 – Окно doxywizard. Ввод параметров обработки текстов.**

Выбор режима обработки исходных текстов позволяет указать что конкретно требуется анализировать и для какого языка программирования. Например Extraction mode All Entries означает, что в документацию будут вынесены абсолютно все сущности, найденные в исходных текстах, а не только те, которые имеют специальную разметку. Флаг Include cross-referenced source code in the output означает, что в документацию будут помещены ссылки, позволяющие позиционироваться с их помощью на нужную строку нужного файла исходный кода.

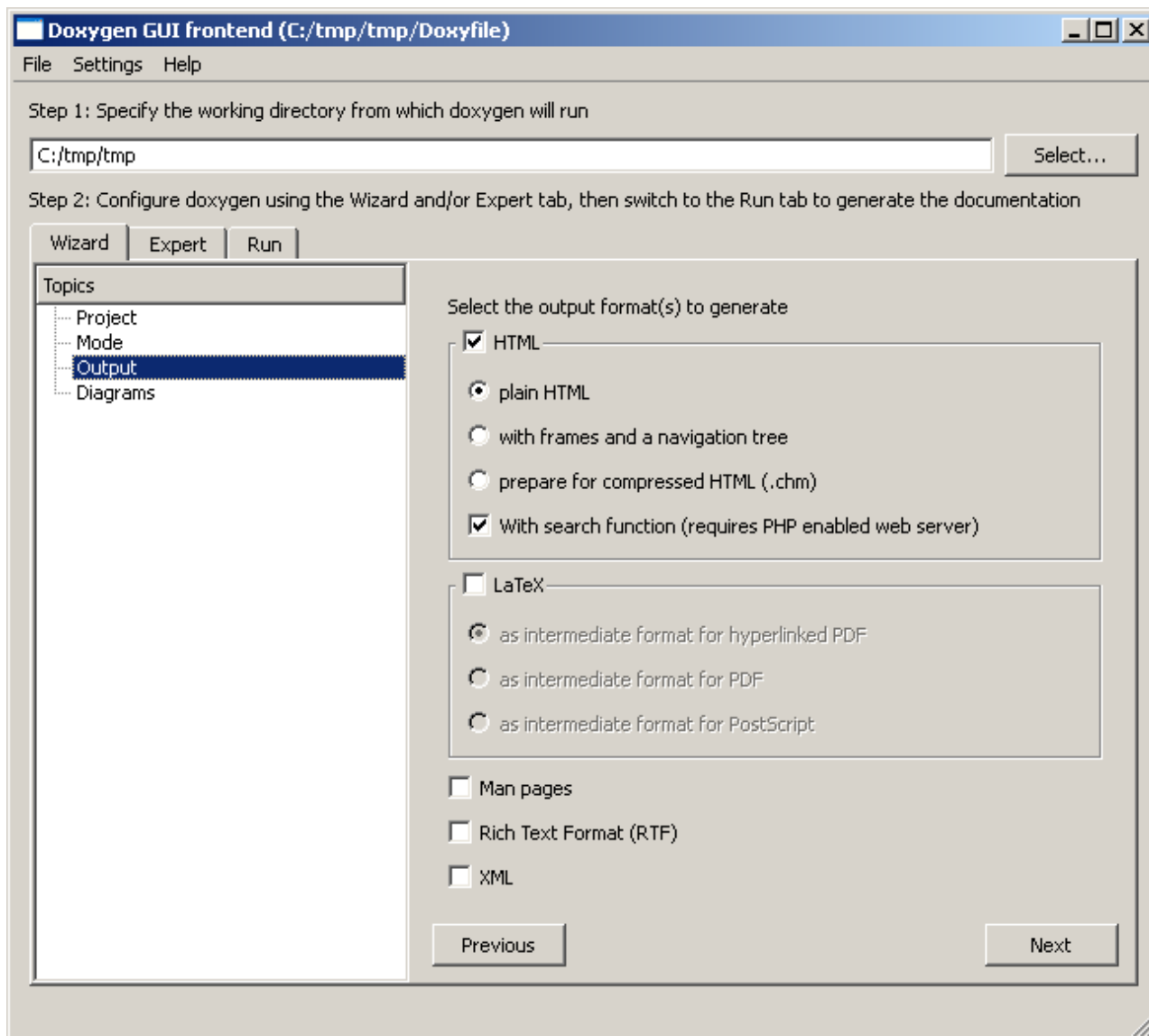


Рисунок 5 – Окно doxywizard. Ввод режима формирования результата.

Параметры вывода (рисунок 3) подразумевают выбор формата, в котором, будет сформирована документация. Наиболее часто используется формат HTML. Режим plain text предполагает создание страниц описания с гиперссылками. Формат With frames and a navigation tree позволит создать html-документацию, где помимо страниц описания будет присутствовать отдельное окно с деревом имен сущностей проекта. Флаг With search function целесообразно использовать лишь в тех случаях, когда документацию планируется размещать на WEB-сервере.

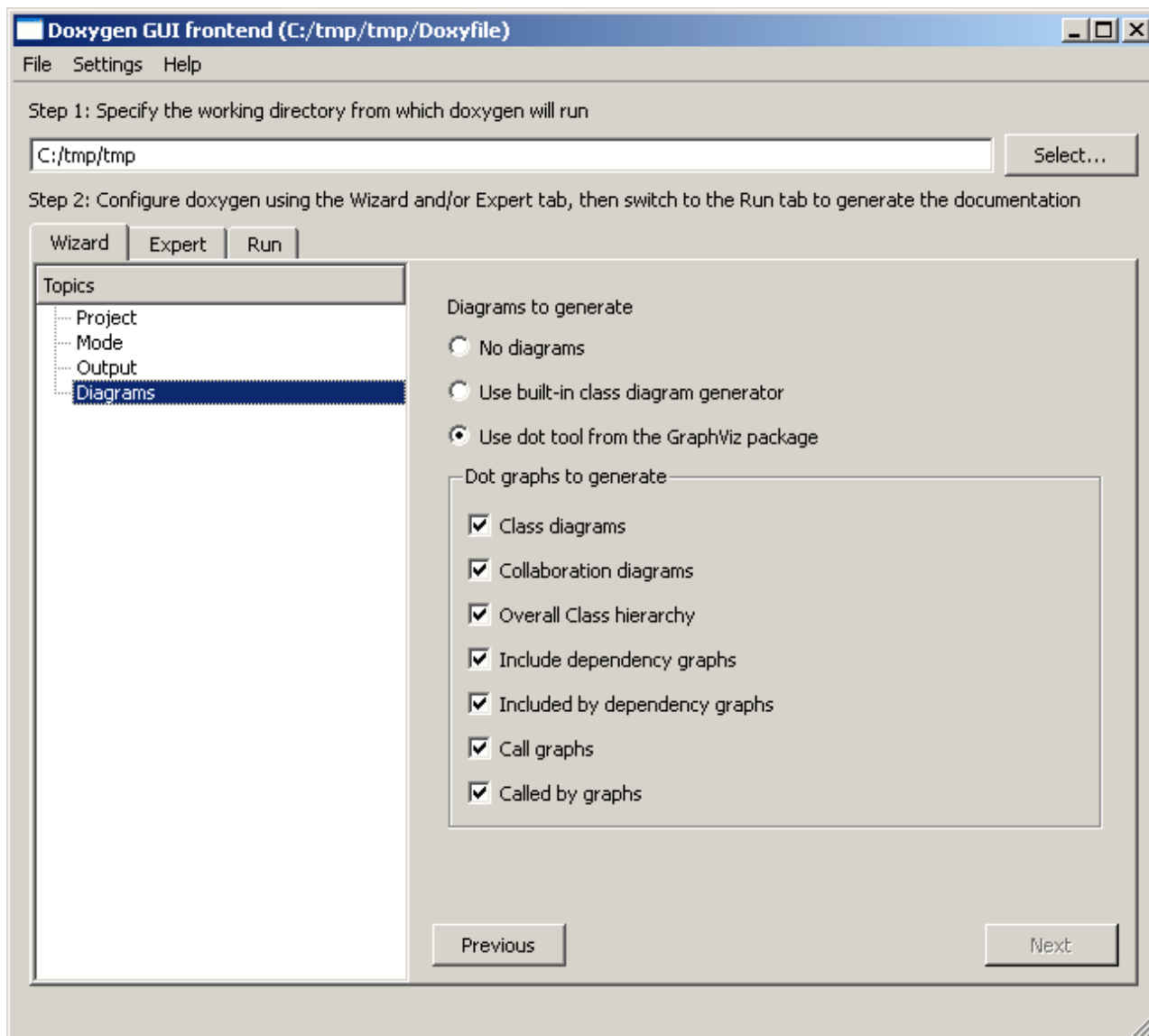


Рисунок 6 – Окно doxywizard. Выбор режима формирования диаграмм.

В состав документации могут быть помещены диаграммы (рисунок 4) среди которых диаграммы классов, диаграмма взаимодействия, диаграмма иерархии классов, графы зависимостей файлов исходных текстов, графы вызова функций. Для их построения требуется наличие специальной библиотеки GraphViz и средства dot.

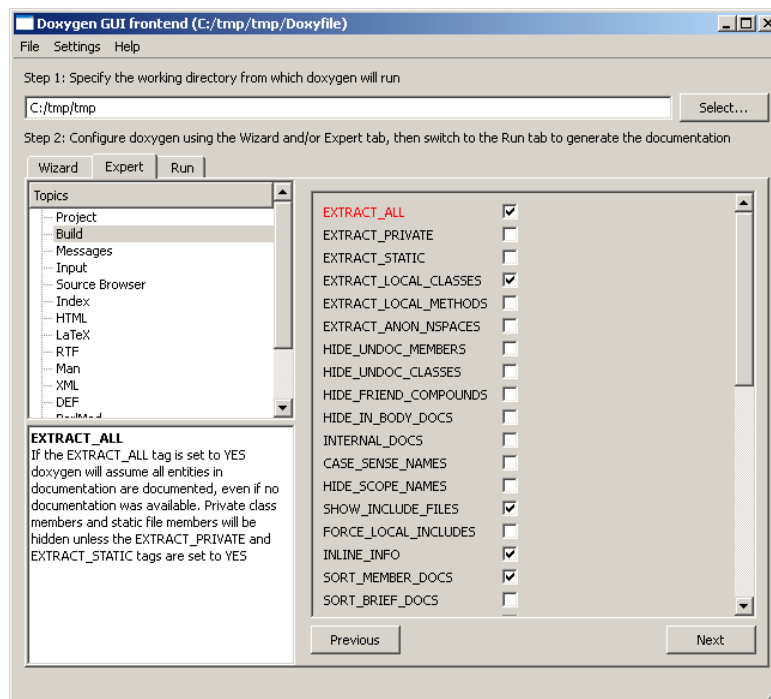


Рисунок 7 – Дополнительные настройки doxywizard. Режим анализа текстов.

После указания основных параметров можно уточнить расширенные параметры, относящиеся к каждой группе в отдельности (рисунок 5). При этом следует заметить, что параметры, выделенные красным обозначают что их текущее значение отличается от значения по умолчанию.

Язык документации (т.е. надписи обрамления) выбирается в группе Project – OUTPUT\_LANGUAGE.

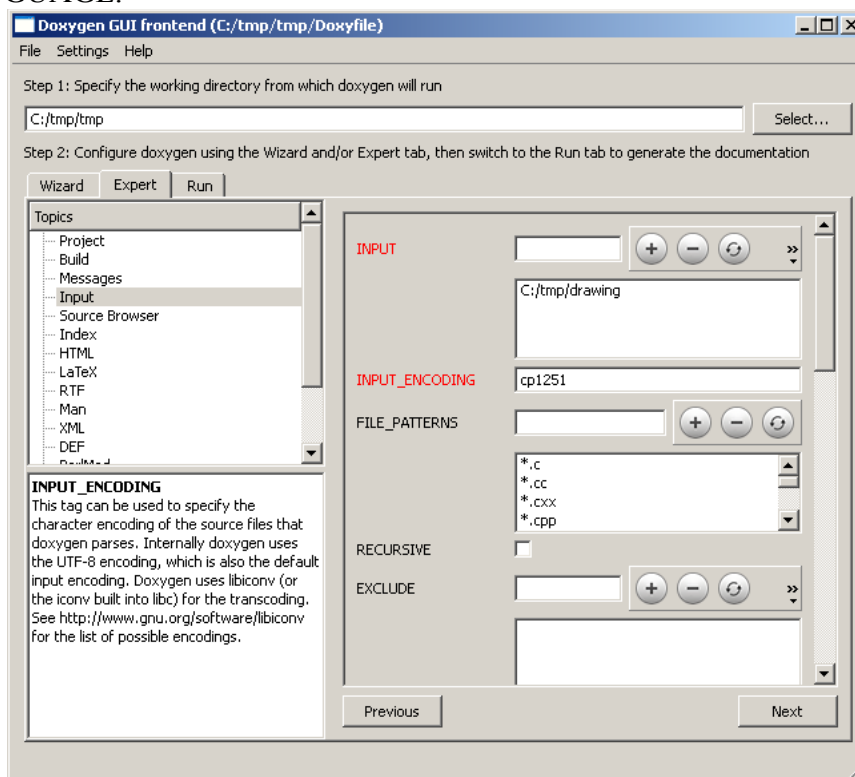


Рисунок 8 – Дополнительные настройки doxywizard. Формат исходных данных.

Среди параметров группы Input (рисунок 6) должен быть указан INPUT\_ENCODING в том случае, если в исходных текстах присутствуют комментарии, написанные буквами отличными от основного латинского алфавита. В частности cp1251 обозначает кодировку Windows.



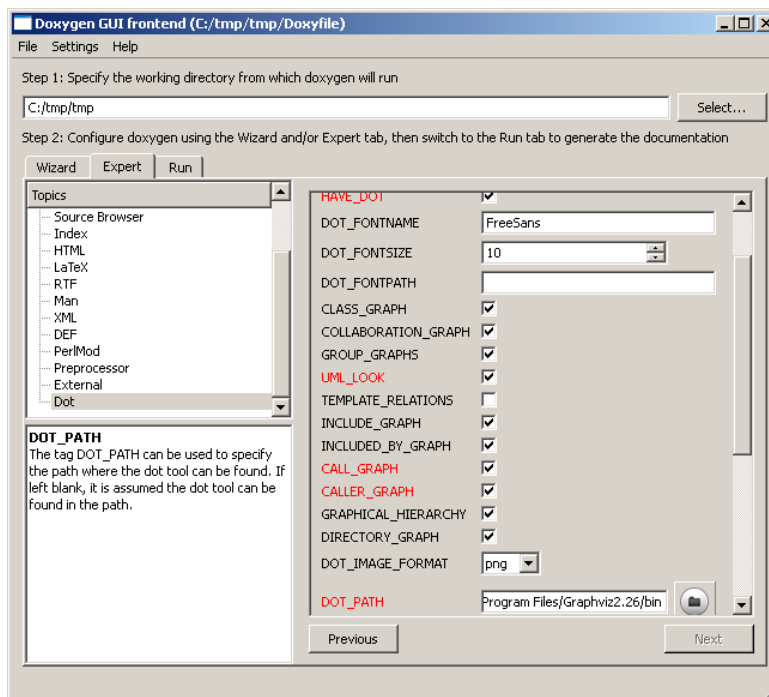


Рисунок 9 – Дополнительные настройки doxuwizard. Выбор типов диаграмм.

Если необходимо построить диаграммы, то в группе Dot (рисунок 7) следует указать, что используется программа dot (генератор изображений по текстовому описанию) – HAVE\_DOT, а также указать директорию, в которой расположена эта программа – DOT\_PATH. Программа dot входит в комплект GraphViz и в случае, когда директория DOT\_PATH указана в переменной окружения операционной системы PATH, указывать путь здесь уже не требуется.

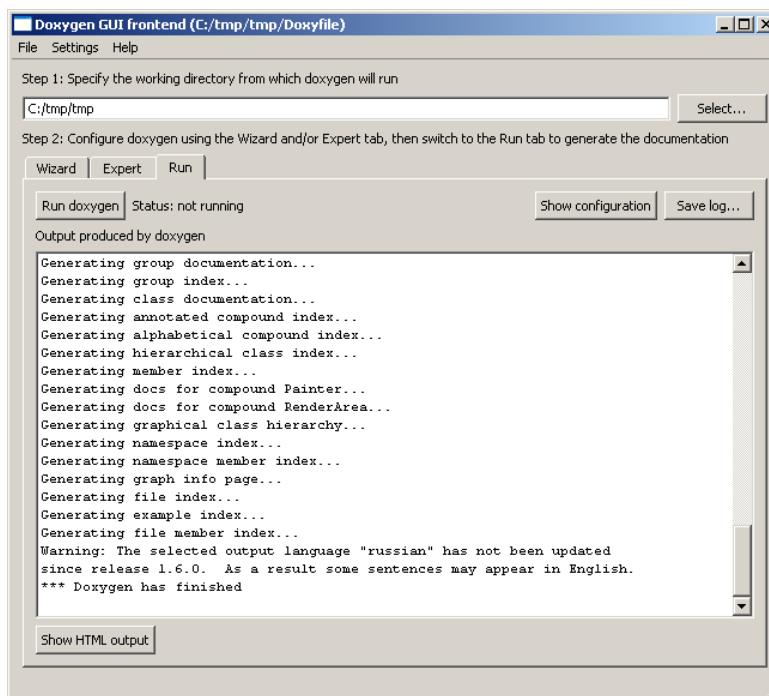


Рисунок 10 – Запуск doxygen.

После того, как заполнены значения всех необходимых параметров, можно сохранить файл параметров на диск(меню File/Save) и запустить генератор документации, нажав кнопку Run doxygen (рисунок 8).

## 4.4. Пример оформления исходного текста

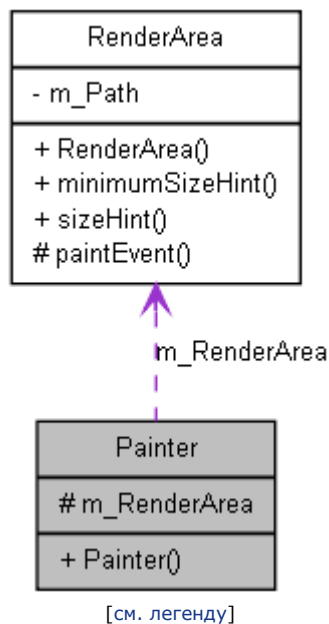
Пример разметки текста приведен в главе рисования с использованием Qt. Приведем фрагмент отчета в формате HTML, сформированный doxygen.

### Класс Painter

Главное окно приложения. [Подробнее...](#)

```
#include <painter.h>
```

Граф связей класса Painter:



[Полный список членов класса](#)

#### Открытые члены

**Painter** ()

Конструктор главного окна.

#### Защищенные данные

**RenderArea**

\*

**m\_RenderArea**

Область отображения рисунка.

#### Подробное описание

Главное окно приложения.

Обеспечивает формирование окна приложения, содержащего область отображения рисунка и кнопку выхода

## Конструктор(ы)

**Painter::Painter** ( )

Конструктор главного окна.

Устанавливает режим выравнивания элементов и создает их.

---

## Данные класса

**RenderArea\*** **Painter::m\_RenderArea** [protected]

Область отображения рисунка.

---

Объявления и описания членов классов находятся в файлах:

- [painter.h](#)
- [painter.cpp](#)