

Технология ООП

Санкт-Петербургский государственный политехнический университет

8 ноября 2011

- отладчик
- `QObject::dumpObjectInfo()`
- `Q_ASSERT()` — выводит предупреждение, если не равно `true`
- `Q_CHECK_PTR()` — проверяет указатель на `NULL`
- `qDebug()`, `qWarning()`, `qFatal()`

В процессе отладки рекомендуется присваивать объектам имена при помощи функции `QObject::objectName()`

```
qDebug() << "Test" << QString("String") << QChar('x'  
    ') << QRect(0, 10, 50, 40);
```

```
qDebug("%d", i);
```

Содержатся в заголовочном файле QtGlobal

- qMax, qMin
- qAbs, qRound

```
int max = qMax<int>(3, 5);  
int abs = qAbs(-5);  
int rnd = qRound(-5.2);
```

Типы Qt

- qint8, qunit8
- ...
- qint64, quint64
- qlonglong
- qulonglong

Последовательные и ассоциативные.

Последовательные:

- `QVector<T>`
- `QList<T>`
- `QLinkedList<T>`
- `QStack`
- `QQueue`

Ассоциативные:

- `QSet<T>`
- `QMap<K,T>`
- `QMultiMap<K,T>`
- `QHash<K,T>`
- `QMultiHash<K,T>`

Во всех контейнерных классах переопределены операции:

- `==` `!=`
- `=`
- `[]` , кроме `QSet`
- `begin()` `end()`
- `clear()`
- `insert()`
- `remove()`
- `size()` `count()`
- `value()` , кроме `QSet`
- `empty` `isEmpty()`

Java Style:

```
QList<QString> list;  
QListIterator<QString> it(list);  
while(it.hasNext()) { qDebug() << it.next(); }
```

Если необходимо изменять значения списка то для этого надо воспользоваться QMutableListIterator'ом.

Java Style:

```
QList<QString> list;  
QMutableListIterator<QString> it(list);  
while(it.hasNext()) { if(it.next() == "test") { it.  
    setValue("new"); } }
```

STL Style:

```
QList<QString> list;  
QList<QString>::iterator it = list.begin();  
for(; it != vec.end(); it++)  
{ qDebug() << "Element_" << *it }
```

Если не планируется изменять значения элементов, то эффективнее использовать `const_iterator`;

```
QList<QString> list;  
QList<QString>::const_iterator it = list.constBegin  
    ();  
for (; it != vec.constEnd(); it++)  
{ qDebug() << "Element_is" << *it }
```

foreach — макрос который позволяет перебрать элементы

```
QList<QString> list;  
foreach(QString str, list) { qDebug() << str; }
```

Методы последовательных контейнеров

- +
- += «
- at()
- back() last()
- front() first()
- contains()
- indexOf()
- lastIndexOf()
- push_back() append()
- push_front() prepend()

```
QVector<QString> vec;  
vec.append("string");
```

Представляет собой динамический массив. Вставка в начало и в середину происходят за время $O(n)$, в конец массива может выполняться за время $O(1)$.

- `data()`
- `resize()`
- `reserve()`

```
QVector<QString> vec;  
vec.push_back("string");  
vec.push_back("string2");  
vec.push_back("string3");
```

Представляет собой упорядоченный набор связанных друг с другом элементов. Поиск выполняется за время $O(1)$, вставка в середину за $O(n)$, вставка в начало или конец списка как правило за $O(1)$, эти времена достигаются за счет того что внутри QList реализован на базе массива.

- `swap()`
- `move()`
- `takeAt()`

```
QList<QString> lst;  
lst << "str1" << "str2";  
QList<QString>::iterator it;  
for( it = lst.begin(); it != lst.end(); ++it )  
{ qDebug() << *it; }
```

Двусвязный список. Вставка в любое место списка выполняется за время $O(1)$, поиск за $O(n)$.

```
QLinkedList<QString> lst;  
lst << "str1" << "str2";  
QLinkedList<QString>::iterator it;  
for( it = lst.begin(); it != lst.end(); ++it )  
{ qDebug() << *it; }
```

Стек — реализует структуру данных, работающую по принципу LIFO. Наследуется от QVector'a.

- push()
- pop()
- top()

```
QStack<int> stack;  
stack.push(1);  
stack.push(2);  
stack.push(3);  
while (!stack.isEmpty())  
    cout << stack.pop() << endl;
```

Стек — реализует структуру данных, работающую по принципу FIFO. Наследуется от QList'a.

- enqueue()
- dequeue()
- head()

```
QQueue<int> queue;  
queue.enqueue(1);  
queue.enqueue(2);  
queue.enqueue(3);  
while (!queue.isEmpty())  
    cout << queue.dequeue() << endl;
```

Для всех контейнеров этого типа доступны методы:

- `contains()`
- `erase()`
- `find()`
- `insert()` отсутствует в `QSet`
- `insertMulti()` отсутствует в `QSet`
- `key()` отсутствует в `QSet`
- `value()` отсутствует в `QSet`
- `keys()` отсутствует в `QSet`
- `values()`

QMap<K,T> QMultiMap<K,T>

Словари хранят элементы одного и того же типа, индексируемые ключевыми значениями, в QMap — ключи должны быть уникальны в отличие от QMultiMap (элементы словаря отсортированы по ключу). Вставка и поиск элементов осуществляются за время $O(\log n)$.

QMap<K,T> QMapMultiMap<K,T>

```
QMap<QString, int> map
map["one"] = 1;
map["three"] = 3;
map["seven"] = 7
map.insert("twelve", 12);
int num1 = map["thirteen"];
int num2 = map.value("thirteen");
QMap<QString, int>::iterator it = map.begin();
for( ; it != map.end(); it++)
{
qDebug << it.key() << it.value();
}
if(map.contains("one")) { qDebug << "one"; }
```

Если необходимо связать с одним ключем несколько значений, например, в адресной книге, то необходимо использовать структуру QMultiMap.

```
QMultiMap<QString, int> map
map.insert("twelve", 12);
map.insert("twelve", 13);
QMap<QString, int>::iterator it = map.find("twelve");
for( ; it != map.end() && it.key() == "twelve"; it++)
{
qDebug << it.key() << it.value();
}
if(map.contains("one")) { qDebug << "one"; }
```

QHash<K,T>

В отличие от словарей не используют сортировку по ключу, а используют хэш-таблицу, что позволяет работать с этой коллекцией намного быстрее. Время доступа к элементам $O(1)$, в худшем случае $O(n)$, время вставки элемента $O(1)$, в худшем случае $O(n)$.

При использовании оператора [], как для QHash так и для QMap следует учитывать, что если при сравнении элемент не обнаружен он будет создан.

```
QHash<QString, int> hash;  
hash.insert("twelve", 12);  
hash.insert("twelve", 13);  
QHash<QString, int>::iterator it = hash.find("  
    twelve");  
for( ; it != hash.end() && it.key() == "twelve";  
    it++)  
{  
qDebug << it.key() << it.value();  
}
```

Для создания коллекции из собственных классов, необходимо переопределить оператор `==` и специализированную функцию `qHash`, которая должна возвращать уникальное число для каждого находящегося в хеше элемента.

Является частным случаем таблицы QHash, хранит внутри себя только ключи без значений. Позволяет выполнять операции над множествами, такие как объединение, пересечение, разность.

- intersect()
- subtract()
- toList()
- unite()

```
QSet<QString> set1;  
QSet<QString> set2;  
set1 << "str1";  
set2 << "str2";  
set1.unite(set2);  
set1.intersect(set2);
```

Входят в заголовочный файл `QtAlgorithms` и предоставляют операции применяемые к контейнерам.

- `qBinaryFind()`
- `qCopy()`
- `qCopyBackward()`
- `qCount()`
- `qDeleteAll()`
- `qEqual()`
- `qFill()`
- `qFind()`
- `qLowerBound()`
- `qUpperBound()`
- `qSwap()`

Для функции qSort необходимо чтобы были переопределены операторы сравнения строк.

```
QList<int> list;  
list << 33 << 12 << 68 << 6 << 12;  
qSort(list.begin(), list.end());  
// list: [ 6, 12, 12, 33, 68
```

Отвечает за поиск элементов в коллекции, возвращает итератор установленный на первый найденный элемент.

```
QStringList list;  
list << "one" << "two" << "three";
```

```
QStringList::iterator i1 = qFind(list.begin(),  
    list.end(), "two");  
// i1 == list.begin() + 1
```

```
QStringList::iterator i2 = qFind(list.begin(),  
    list.end(), "seventy");  
// i2 == list.end()
```

Позволяет сравнить две коллекции различных типов, например QList и QVector. В качестве первого и второго параметра передаются итераторы указывающие на начало и конец первой последовательности, а в качестве третьего итератор на начало второй последовательности.

```
QStringList list;  
list << "one" << "two" << "three";  
  
QVector<QString> vect(3);  
vect[0] = "one";  
vect[1] = "two";  
vect[2] = "three";  
  
bool ret1 = qEqual(list.begin(), list.end(), vect.  
    begin());  
// ret1 == true  
  
vect[2] = "seven";  
bool ret2 = qEqual(list.begin(), list.end(), vect.  
    begin());  
// ret2 == false
```

Заполнить коллекцию какими-либо заданными значениями.

```
QStringList list;  
list << "one" << "two" << "three";  
  
qFill(list.begin(), list.end(), "eleven");  
// list: [ "eleven", "eleven", "eleven" ]  
  
qFill(list.begin() + 1, list.end(), "six");  
// list: [ "eleven", "six", "six" ]
```

Регулярные выражения

Описываются классом QRegExpr, представляют из себя шаблон который предназначен для поиска текста в строке.

```
. //a.b  
^ и $ //^Abc$  
[] //[abc]  
- //[0-9A-Za-z]  
^ //[^def]  
* //A*b  
+ //A+b  
? //A?b  
{n} //A{3}b  
{n,} //a{3,}b  
{,n} //a{,3}b  
{n,m} //a{2,3}b  
| //ac|bc  
\b //a\b  
\B //a\Bd
```

```
( ) //(ab|ac)ad
\d //q\d //q2
\D //a\D //ad
\s //
\S //a
\w //c
\W //2
```

```
QRegExp reg("
    [0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}
");
QString str("my_ip_address_is_192.168.0.1_");
qDebug() << str.contains(reg) : 1 ? 0;
QRegExp rx("^\\d\\d?$");
rx.indexIn("123");
rx.indexIn("-6");
rx.indexIn("6");
```

```
QRegExp rx ("^\\S+$");  
rx.indexIn ("Hello_world");  
rx.indexIn ("This_is-OK");
```

```
QRegExp rx ("\\b(mail|letter|correspondence)\\b");  
rx.indexIn ("I_sent_you_an_email");      // returns  
    -1 (no match)  
rx.indexIn ("Please_write_the_letter"); // returns  
    17
```

```
QRegExp rx ("*.html");  
rx.setPatternSyntax (QRegExp::Wildcard);  
rx.exactMatch ("index.html");  
rx.exactMatch ("default.htm");  
rx.exactMatch ("readme.txt");
```

Позволяет хранить объекты произвольных типов.

```
QVariant v(123);  
int x = v.toInt();  
v = QVariant("hello");  
int y = v.toInt();  
QString s = v.toString();
```
