Qt Essentials - Basic Types Module

Qt Essentials - Training Course

Produced by Nokia, Qt Development Frameworks

Material based on Qt 4.7, created on December 15, 2010

http://qt.nokia.com

Module: Core Classes

- String Handling
- Container Classes
- File Handling
- Variants

Module Objectives

Qt provides a set of basic data types:

- **String handling classes:**
  - Unicode-aware string and character classes
  - Used throughout the Qt API
  - Regular expression engine for pattern matching
- **Container classes:**
  - Common containers: lists, sets, maps, arrays, ...
  - Including STL and Java-style iterators
- **File handling classes:**
  - Reading and Writing Files
  - Files and Text Streams
- **Variants:**
  - Variant basics
  - Application notes

Module: Core Classes

- String Handling
- Container Classes
- File Handling
- Variants

Text Processing with QString

Strings can be created in a number of ways:

- Conversion constructor and assignment operators:

```
QString str("abc");
str = "def";
```

- From a number using a static function:

```
QString n = QString::number(1234);
```

- From a char pointer using the static functions:

```
QString text = QString::fromLatin1("Hello Qt");
QString text = QString::fromUtf8(inputText);
QString text = QString::fromLocal8Bit(cmdLineInput);
```

Text Processing with QString

- Created from other strings
  - Using `operator+` and `operator+=`

```
QString str = str1 + str2;
fileName += ".txt";
```

- `simplified()` // removes duplicate whitespace
- `left()`, `mid()`, `right()` // part of a string
- `leftJustified()`, `rightJustified()` // padded version

```
QString s = "apple";
QString t = s.leftJustified(8, '.'); // t == "apple..."
```

Text Processing with QString

Data can be extracted from strings.

- Numbers:

```
int value = QString::toInt();
float value = QString::toFloat();
```

- Strings:

```
QString text = ...;
QByteArray bytes = text.toLatin1();
QByteArray bytes = text.toUtf8();
QByteArray bytes = text.toLocal8Bit();
```

Text Processing with QString

Obtaining raw character data from a `QByteArray`:

```
char *str = bytes.data();
const char *str = bytes.constData();
```

Care must be taken:

- Character data is only valid for the lifetime of the byte array.
- Either copy the character data or keep a copy of the byte array.

Text Processing with `QString`

Strings can be tested with:

- `length()`
  - returns the length of the string.
- `endsWith()` and `startsWith()`
  - test whether string starts or ends with an other string
- `contains()`
  - returns whether the string matches a given expression
  - `count()` tells you how many times.
- `indexOf()` and `lastIndexOf()`
  - search for next matching expressions, and return its index

Expression can be characters, strings, or regular expressions

Text Processing with `QStringList`

- `QString::split()` and `QStringList::join()`
  - `split` one string or `join` many strings using a substring
- `QStringList::replaceInStrings()`
  - search/replace on all strings in a list
- `QStringList::filter()`
  - return list of items matching given pattern or substring

URLs

- `QUrl` handles URLs

```
QUrl url("http://qt.nokia.com");
url.scheme()  // http
url.host();   // qt.nokia.com
```

- Convenience methods
  - Constructing from components (user, password, protocol, etc.)
  - Splitting URL into components
- `QUrlInfo` provides info about the content the URL points to
  - similar to what `QFileInfo` does for local files.
- `QDesktopServices`
  - can *open* URL, and configure how URLs are opened.

```
QDesktopServices::openUrl(QUrl("http://qt.nokia.com"));
```

Working with Regular Expressions

- `QRegExp` supports
  - Regular expression matching
  - Wildcard matching
- Most regular expression features are supported.
- `QRegExp` objects can also be used for searching strings.
- `QRegExp` offers text capturing regular expressions, where a subexpression can be extracted using `QString cap(int)` and `QStringList capturedTexts()`.

```
QRegExp rx("^\\d\\d?$");    // match integers 0 to 99
rx.indexIn("123");          // returns -1 (no match)
rx.indexIn("-6");           // returns -1 (no match)
rx.indexIn("6");            // returns 0 (matched as position 0)
```

See QRegExp Documentation

Module: Core Classes

- String Handling
- Container Classes
- File Handling
- Variants

Container Classes

General purpose template-based container classes

- **`QList<QString>`** - *Sequence Container*
  - Other: QLinkedList, QStack , QQueue ...
- **`QMap<int, QString>`** - *Associative Container*
  - Other: QHash, QSet, QMultiMap, QMultiHash

Qt's Container Classes compared to STL

- Lighter, safer, and easier to use than STL containers
- If you prefer STL, feel free to continue using it.
- Methods exist that convert between Qt and STL
  - e.g. you need to pass `std::list` to a Qt method

Using Containers

- Using `QList`

```
QList<QString> list;
list << "one" << "two" << "three";
QString item1 = list[1]; // "two"
for(int i=0; i<list.count(); i++) {
  const QString &item2 = list.at(i);
}
int index = list.indexOf("two"); // returns 1
```

- Using `QMap`

```
QMap<QString, int> map;
map["Norway"] = 5; map["Italy"] = 48;
int value = map["France"]; // inserts key if not exists
if(map.contains("Norway")) {
  int value2 = map.value("Norway"); // recommended lookup
}
```

Algorithm Complexity

Concern

How fast is a function when number of items grow

- Sequential Container

	Lookup	**Insert**	**Append**	**Prepend**
QList	O(1)	O(n)	O(1)	O(1)
QVector	O(1)	O(n)	O(1)	O(n)
QLinkedList	O(n)	O(1)	O(1)	O(1)

- Associative Container

	Lookup	**Insert**
QMap	O(log(n))	O(log(n))
QHash	O(1)	O(1)

all complexities are amortized

Storing Classes in Qt Container

- Class must be an *assignable data type*
- Class is *assignable*, if:

```
class Contact {
public:
  Contact() {} // default constructor
  Contact(const Contact &other); // copy constructor
  // assignment operator
  Contact &operator=(const Contact &other);
};
```

- *If copy constructor or assignment operator is not provided*
  - C++ will provide one (uses member copying)
- *If no constructors provided*
  - Empty default constructor provided by C++

Requirements on Container Keys

- Type `K` as key for `QMap`:
  - `bool K::operator<( const K& )` or
    `bool operator<( const K&, const K& )`

```
bool Contact::operator<(const Contact& c);
bool operator<(const Contact& c1, const Contact& c2);
```

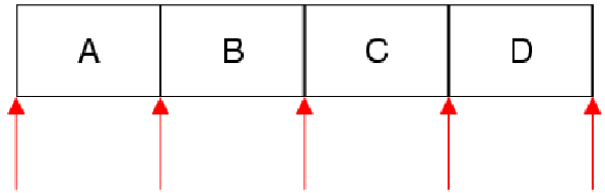
  - See QMap Documentation
- Type `K` as key for `QHash` or `QSet`:
  - `bool K::operator==( const K& )` or
    `bool operator==( const K&, const K& )`
  - `uint qHash( const K& )`
  - See QHash Documentation

Iterators

- Allow reading a container's content sequentially
- **Java-style iterators**: simple and easy to use
  - `QListIterator<...>` for read
  - `QMutableListIterator<...>` for read-write
- **STL-style iterators** slightly more efficient
  - `QList::const_iterator` for read
  - `QList::iterator()` for read-write
- Same works for QSet, QMap, QHash, ...

Iterators Java style

- Iterator points between items
- Example QList iterator




```
QList<QString> list;
list << "A" << "B" << "C" << "D";
QListIterator<QString> it(list);
```

- Forward iteration

```
while(it.hasNext()) {
  qDebug() << it.next();    // A B C D
}
```

- Backward iteration

```
it.toBack(); // position after the last item
while(it.hasPrevious()) {
  qDebug() << it.previous(); // D C B A
}
```

See QListIterator Documentation

Modifying During Iteration

- Use *mutable* versions of the iterators
  - e.g. `QMutableListIterator`.

```
QList<int> list;
list << 1 << 2 << 3 << 4;
QMutableListIterator<int> i(list);
while (i.hasNext()) {
  if (i.next() % 2 != 0)
    i.remove();
}
// list now 2, 4
```

- `remove()` and `setValue()`
  - Operate on items just jumped over using next()/previous()
- `insert()`
  - Inserts item at current position in sequence
    - *Remember iterator points between item*
  - `previous()` reveals just inserted item

Iterating Over QMap and QHash

- `next()` and `previous()`
  - Return Item class with `key()` and `value()`
- Alternatively use `key()` and `value()` from iterator

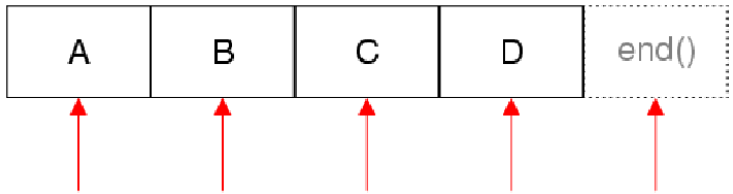
```
QMap<QString, QString> map;
map["Paris"] = "France";
map["Guatemala City"] = "Guatemala";
map["Mexico City"] = "Mexico";
map["Moscow"] = "Russia";

QMutableMapIterator<QString, QString> i(map);
while (i.hasNext()) {
  if (i.next().key().endsWith("City"))
    i.remove();
}
// map now "Paris", "Moscow"
```

Demo core-types/ex-container

STL-style Iterators

- Iterator points at item
- Example QList iterator




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

- Forward mutable iteration

```
for (i = list.begin(); i != list.end(); ++i) {
    *i = (*i).toLower();
}
```

- Backward mutable iteration

```
i = list.end();
while (i != list.begin()) {
    --i;
    *i = (*i).toLower();
}
```

- `QList<QString>::const_iterator` for read-only

The `foreach` Keyword

- It is a macro, feels like a keyword

`foreach` (*variable*, *container*) *statement*

```
foreach (const QString& str, list) {
  if (str.isEmpty())
    break;
  qDebug() << str;
}
```

- `break` and `continue` as normal
- Modifying the container while iterating
  - results in container being copied
  - iteration continues in unmodified version
- Not possible to modify item
  - iterator variable is a const reference.

Algorithms

- STL-style iterators are compatible with the STL algorithms
  - Defined in the STL <algorithm> header
- Qt has own algorithms
  - Defined in <QtAlgorithms> header
- *If STL is available on all your supported platforms you can choose to use the STL algorithms*
  - The collection is much larger than the one in Qt.

Algorithms

- **qSort**(begin, end) sort items in range
- **qFind**(begin, end, value) find value
- **qEqual**(begin1, end1, begin2) checks two ranges
- **qCopy**(begin1, end1, begin2) from one range to another
- **qCount**(begin, end, value, n) occurrences of value in range
- and more ...

Counting 1's in list

```
QList<int> list;
list << 1 << 2 << 3 << 1;
int count = 0;
qCount(list, 1, count); // count the 1's
qDebug() << count;      // 2  (means 2 times 1)
```

- For parallel (ie. multi-threaded) algorithms
  - See `QtConcurrent` Documentation

See QtAlgorithms Documentation

Algorithms Examples

- Copy list to vector example

```
QList<QString> list;
list << "one" << "two" << "three";
QVector<QString> vector(3);
qCopy(list.begin(), list.end(), vector.begin());
// vector: [ "one", "two", "three" ]
```

- Case insensitive sort example

```
bool lessThan(const QString& s1, const QString& s2) {
    return s1.toLower() < s2.toLower();
}
// ...
QList<QString> list;
list << "AlPha" << "beTA" << "gamma" << "DELTA";
qSort(list.begin(), list.end(), lessThan);
// list: [ "AlPha", "beTA", "DELTA", "gamma" ]
```

Implicitly Sharing and Containers

Implicit Sharing

If an object is copied, then its data is copied *only when* the data of one of the objects is changed

- Shared class has a pointer to shared data block
  - Shared data block = reference counter and actual data
- Assignment is a shallow copy
- Changing results into deep copy (detach)

```
QList<int> l1, l2; l1 << 1 << 2;
l2 = l1; // shallow-copy: l2 shares date with l1
l2 << 3; // deep-copy: change triggers detach from l1
```

Important to remember when inserting items into a container, or when returning a container.

Module: Core Classes

- String Handling
- Container Classes
- File Handling
- Variants

Working With Files

Rule

For portable file access do not use the native functions like `open()` or `CreateFile()`, but Qt classes instead.

File Handling

- `QFile`
  - Interface for reading from and writing to files
  - Inherits QIODevice *(base interface class of all I/O devices)*
- `QTextStream`
  - Interface for reading and writing text
- `QDataStream`
  - Serialization of binary data

Additional

- `QFileInfo` - System-independent file information
- `QDir` - Access to directory structures and their contents

Reading/Writing a File

- Writing with text stream

```
QFile file("myfile.txt");
if (file.open(QIODevice::WriteOnly)) {
  QTextStream stream(&file);
  stream << "HelloWorld " << 4711;
  file.close();
}
```

- Reading with text stream

```
{
  QString text; int value;
  QFile file("myfile.txt");
  if (file.open(QIODevice::ReadOnly)) {
    QTextStream stream(&file);
    stream >> text >> value;
  }
} // file closed automatically
```

- Demo coretypes/ex-qtextstream

Text and Encodings

- Using `QTextStream`
  - Be aware of encoding
  - Need to call `QTextStream::setCodec(codec)`
- Alternative ways of reading a file
  - `QFile::readAll()` *returns* `QByteArray`
  - `QFile::readLine(maxlen)` *returns* `QByteArray`
  - `QTextStream::readAll()` *returns* `QString`
  - `QTextStream::readLine(maxlen)` *returns* `QString`

Reading/Writing Data - QDataStream

- Class QDataStream
  - Alternative to QTextStream
  - Adds extra information about the data
  - Portable between hardware architectures and operating systems
  - Not human-readable.

```
QFile file("file.dat");
if (file.open(QIODevice::WriteOnly)) {
  QDataStream out(&file);
  out << "Blue"; // string
  out << QColor(Qt::blue); // as QColor
}
```

- QDataStream can serialize many Qt classes
  - Not the case with QTextStream
- Common for QTextStream & QDataStream
  - Both can write onto memory or sockets (i.e. any QIODevice)

File Convenient Methods

- Media methods: `load(fileName)`, `save(fileName)`
  - for `QPixmap`, `QImage`, `QPicture`, `QIcon`
- `QFileDialog`
  - `QFileDialog::getExistingDirectory()`
  - `QFileDialog::getOpenFileName()`
  - `QFileDialog::getSaveFileName()`
- `QDesktopServices::storageLocation(type)`
  - returns default system directory where files of type belong
- File operations
  - `QFile::exists(fileName)`
  - `QFile::rename(oldName, newName)`
  - `QFile::copy(oldName, newName)`
  - `QFile::remove(fileName)`
- Directory Information
  - `QDir::tempPath()`
  - `QDir::home()`
  - `QDir::drives()`

Module: Core Classes

- String Handling
- Container Classes
- File Handling
- Variants

QVariant

- `QVariant`
  - Union for common Qt "value types" (copyable, assignable)
  - Supports implicit sharing (fast copying)
  - Supports user types
- For `QtCore` types

```
QVariant variant(42);
int value = variant.toInt(); // read back

qDebug() << variant.typeName(); // int
```

- For non-core and custom types:

```
variant.setValue(QColor(Qt::red));
QColor color = variant.value<QColor>(); // read back
qDebug() << variant.typeName(); // "QColor"
```

- See QVariant Documentation

Q_DECLARE_METATYPE

```
// contact.h:
class Contact {
public:
  void setName(const QString name);
  QString name() const;
  ...
};
// make Contact known to meta-type system
Q_DECLARE_METATYPE(Contact);
```

- Adds custom type to QVariant system.
- Type must support default construction, copy and assignment.
- Should appear after class definition in header file.

See Q_DECLARE_METATYPE Documentation

Custom Types and QVariant

- Custom Type stored in **QVariant**:

```
#include <QtGui>
#include "contact.h"   // must have Q_DECLARE_METATYPE there

int main(int argc, argv) {
    QApplication app(argc, argv);
    // ...
    Contact c; c.setName("Peter");
    QVariant v = QVariant::fromValue(c);
    Contact c2 = v.value<Contact>();
    qDebug() << c2.name(); // "Peter"
    qDebug() << v.typeName();  // prints "Contact"
```

qRegisterMetaType

- A **registered** type has a known string typename
- Can be dynamically constructed with `construct()`
- qRegisterMetaType() belongs in code, not header file.

```
#include "contact.h"   // must have Q_DECLARE_METATYPE there
int main(int argc, argv) {
    QApplication app(argc, argv);

    // Register string typename:
    qRegisterMetaType<Contact>("Contact");

    Contact c;  c.setName("Peter");
    QVariant v = QVariant::variantValue(c);
    qDebug() << v.typeName();  // prints "Contact"
```

See qRegisterMetaType Documentation See construct Documentation