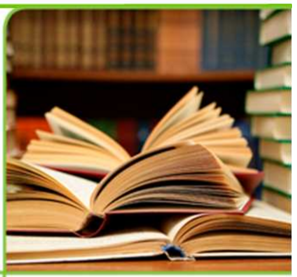
Qt in Education

Datatypes Collections and Files



Managing text

- Simple C strings are handy, but limited to your local character encoding

```
char *text = "Hello world!";
```

- The QString class attempts to be the modern string class
  - Unicode and codecs
  - Implicit sharing for performance

QString

- Stores Unicode strings capable of representing almost all writing systems in use today

- Supports conversion from and to different local encodings

```
QString::toLatin1 – QString::toLocal8Bit
```

- Provides a convenient API for string inspection and modification

Building Strings

- There are three main methods for building strings
- The `operator+` method

```
QString res = "Hello “ + name +
    “, the value is " + QString::number(42);
```

- The `QStringBuilder` method

```
QString res = "Hello “ % name %
    “, the value is " % QString::number(42);
```

- The `arg` method

```
QString res = QString("Hello %1, the value is %2")
    .arg(name)
    .arg(42);
```

QStringBuilder

- Using the + operator to join strings results in numerous memory allocations and checks for string lengths
- A better way to do it is to include `QStringBuilder` and use the `%` operator
- The string builder collects all lengths before joining all strings in one go, resulting in one memory allocation

```
QString res = "Hello “ % name %
    “, the value is %" % QString::number(42);
```

```
QString temp = "Hello “;
temp = temp % name;
temp = temp % “, the value is %"
temp = temp % QString::number(42);
```

Joining strings in small steps will cost you in performance

digia

QString::arg

- The arg method replaces `%1-99` with values

```
"%1 + %2 = %3, the sum is %3"
```

All instances of %n are replaced

- Can handle strings, chars, integers and floats

```
...).arg(qulonglong a)          ...).arg(QString, ... QString)
...).arg(short a)               ...).arg(int a)
...).arg(ushort a)              ...).arg(uint a)
...).arg(QChar a)               ...).arg(long a)
...).arg(char a)                ...).arg(ulong a)
...).arg(double a)              ...).arg(qlonglong
```

Up to nine strings in one go

- Can convert between number bases

```
...).arg(value, width, base, fillChar);
...).arg(42, 3, 16, QChar('0')); // Results in 02a
```

Substrings

- Access substrings using `left`, `right` and `mid`

```
QString s = "Hello world!";
r = s.left(5); // "Hello"
r = s.right(1); // "!"
r = s.mid(6,5); // "world"
```

- By not specifying a length to `mid`, the rest of the string is returned

```
r = s.mid(6); // "world!"
```

- Use `replace` to search and replace in strings

```
r = s.replace("world", "universe"); // "Hello universe!"
```

Printing to the console

- Qt is a toolkit primarily for visual applications, i.e. not focused on command line interfaces
- To print, use the `qDebug` function
  - It is always available, but can be silenced when building for release
  - Works like the `printf` function (but appends “\n”)
  - Using the `qPrintable` macro, it is easy to print QString texts

```
qDebug("Integer value: %d", 42);
qDebug("String value: %s", qPrintable(myQString));
```

  - Can be used with streaming operators when QtDebug is included

```
#include <QtDebug>

qDebug() << "Integer value:" << 42;
qDebug() << "String value:" << myQString;
qDebug() << "Complex value:" << myQColor;
```

From and to numbers

- Converting from numbers to strings

```
QString::number(int value, int base=10);
QString twelve = QString::number(12); // "12"
QString oneTwo = QString::number(0x12, 16); // "12"

QString::number(double value, char format='g', int precision=6);
QString piAuto = QString::number(M_PI); // "3.14159"
QString piScientific = QString::number(M_PI,'e'); // "3.141593e+00"
QString piFixedDecimal = QString::number(M_PI,'f',2); // "3.14"
```

- Converting from string to value

Cannot handle thousand group separators

```
bool ok;
QString i = "12";
int value = i.toInt(&ok);
if(ok) {
    // Converted ok
}
```

```
bool ok;
QString d = "12.36e-2";
double value = d.toDouble(&ok);
if(ok) {
    // Converted ok
}
```

Working with std::(w)string

- Converting from and to STL's strings come handy when interfacing third party libraries and other code
- Converting from STL strings

```
std::string ss = "Hello world!";
std::wstring sws = "Hello world!";

QString qss = QString::fromStdString(ss);
QString qsws = QString::fromStdWString(sws);
```

Assumes ASCII

- Converting to STL strings

```
QString qs = "Hello world!";
std::string ss = qs.toStdString();
std::wstring sws = qs.toStdWString();
```

Empty and null strings

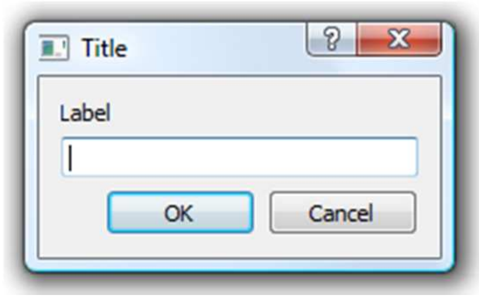
- A `QString` can be null, i.e. contain nothing

```
QString n = QString();

n.isNull();  // true
n.isEmpty(); // true
```

This is useful for passing no string, compared to an empty string.

See `QInputDialog::getText` – returns a null string on Cancel.



- It can also be empty, i.e. contain an empty string

```
QString e =  "";

e.isNull();  // false
e.isEmpty(); // true
```

Splitting and joining

- A `QString` can be split into sub-strings

```
QString whole = "Stockholm - Copenhagen - Oslo - Helsinki";
QStringList parts = whole.split(" - ");
```

- The resulting object is a `QStringList`, which can be joined together to form a `QString`

```
QString wholeAgain = parts.join(", ");
    // Results in "Stockholm, Copenhagen, Oslo, Helsinki"
```

QStringList

- The `QStringList` is a specialized list type
- Designed for holding strings

  Provides a convenient API for working with the strings in the list

- The class uses implicit sharing
  - Copies on modification
  - Cheap to pass as value

Building and modifying string lists

- Use the << operator to add strings to string lists

```
QStringList verbs;
verbs = "running" << "walking" << "compiling" << "linking";
```

- The `replaceInStrings` method lets you search and replace within all strings of a `QStringList`.

```
qDebug() << verbs; // ("running", "walking", "compiling", "linking")
verbs.replaceInStrings("ing", "er");
qDebug() << verbs; // ("runner", "walker", "compiler", "linker")
```

Building and modifying string lists

- Use the << operator to add strings to string lists

```
QStringList verbs;
verbs = "running" << "walking" << "compiling" << "linking";
```

- The `replaceInStrings` method lets you search and replace within all strings of a `QStringList`.

```
qDebug() << verbs; // ("running", "walking", "compiling", "linking")
verbs.replaceInStrings("ing", "er");
qDebug() << verbs; // ("runner", "walker", "compiler", "linker")
```

Sorting and Filtering

- A `QStringList` can be sorted...

```
qDebug() << capitals; // ("Stockholm", "Oslo", "Helsinki", "Copenhagen")
capitals.sort();
qDebug() << capitals; // ("Copenhagen", "Helsinki", "Oslo", "Stockholm")
```

Case sensitive by default

- ...filtered...

```
QStringList capitalsWithO = capitals.filter("o");
qDebug() << capitalsWithO; // ("Copenhagen", "Oslo", "Stockholm")
```

- ...and cleaned for duplicate entries

```
capitals << capitalsWithO;
qDebug() << capitals; // ("Copenhagen", "Helsinki", "Oslo", "Stockholm",
                      //  "Copenhagen", "Oslo", "Stockholm")
capitals.removeDuplicates();
qDebug() << capitals; // ("Copenhagen", "Helsinki", "Oslo", "Stockholm")
```

Iterating over the strings

- Using the `operator[]` and `length` function, you can iterate over the contents of a `QStringList`

```
QStringList capitals;
for(int i=0; i<capitals.length(); ++i)
    qDebug() << capitals[i];
```

- Another option is to use the `at()` function which provides read-only access to the list items
- The `at()` function also avoids copying the data of the string, which `operator[]` does
- You can also use the `foreach` macro

```
QStringList capitals;
foreach(const QString &city, capitals)
    qDebug() << city;
```

Qt's Collections

- The interface of `QStringList` is not unique to the string list. `QStringList` is derived from `QList<QString>`
- `QList` is one of many of Qt's container template classes
  - `QLinkedList` – quick insert in the middle, access through iterators
  - `QVector` – uses continous memory, slow inserts
  - `QStack` – LIFO, last in – first out
  - `QQueue` – FIFO, first in – first out
  - `QSet` – unique values
  - `QMap` – associative array, sorted by key
  - `QHash` – associative array, faster than `QMap`, but requires hash
  - `QMultiMap` – associative array with multiple values per key
  - `QMultiHash` – associative array with multiple values per key

Slow or fast relative to `QList` which will be our reference and benchmark

Populating

- You can populate a `QList` using the `<<` operator

```
QList<int> fibonacci;
fibonacci << 0 << 1 << 1 << 2 << 3 << 5 << 8;
```

- The methods `prepend`, `insert` and `append` can also be used

```
QList<int> list;
list.append(2);
```

index 0: **2**

```
list.append(4);
```

index 0: **2**
index 1: **4**

```
list.insert(1,3);
```

index 0: **2**
index 1: **3**
index 2: **4**

```
list.prepend(1);
```

index 0: **1**
index 1: **2**
index 2: **3**
index 3: **4**

Removing

- Remove item from a `QList` using `removeFirst`, `removeAt`, `removeLast`

```
while(!list.isEmpty())
    list.removeFirst();
```

Remove simply removes – good for value types

- To take an item, use `takeFirst`, `takeAt`, `takeLast`

```
QList<QWidget*> widgets;
widgets << new QWidget << new QWidget;
while(!widgets.isEmpty())
    delete widgets.takeFirst();
```

Take removes and returns – good for pointers and when you need the value

- For removing items with a specific value, use `removeAll` or `removeOne`

```
QList<int> list;
list << 1 << 2 << 3 << 1 << 2 << 3;
list.removeAll(2); // Leaves 1, 3, 1, 3
```

Accessing

- The indexes of items in a `QList` are in the range 0 - `length-1`
- Individual list items can be accessed using `at` or the `[]` operator. `value` can be used if you can accept out of bound indexes.

```
for(int i=0; i<list.length(); ++i)
    qDebug("At: %d, []: %d", list.at(i), list[i]);

for(int i=0; i<100; ++i)
    qDebug(“Value: %d”, list.value(i));
```

Returns the *default-constructed value* when the index is out of range

- The `[]` operator returns a modifiable reference

```
for(int i=0; i<list.length(); ++i)
    list[i]++;
```

Iterating – Java style

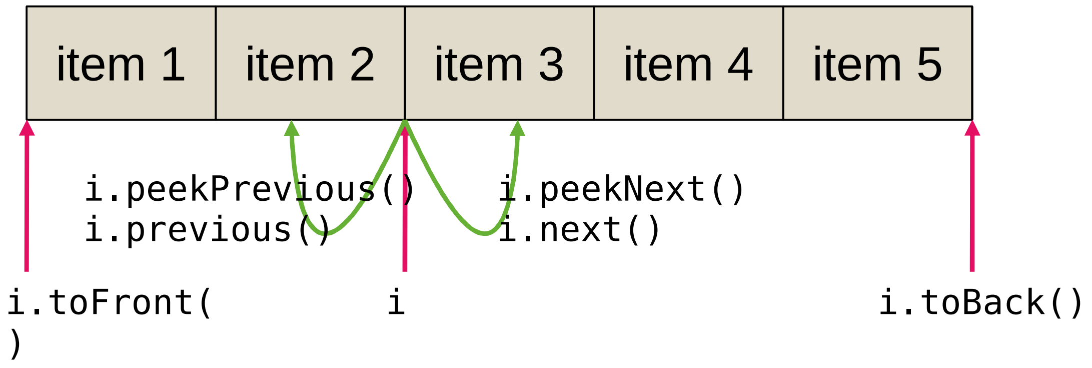
- Qt supports Java style iterators

```
QListIterator<int> iter(list);
        while(iter.hasNext())
            qDebug("Item: %d", iter.next());
```

Both return a value and step to the next position of the list

Use a `QMutableListIterator` if you need to modify the items

- Java style iterators point between entries
  - `toFront` places the iterator in front of the first item
  - `toBack` places the iterator after the last item
  - Refer to items using `peekNext` and `peekPrevious`
  - Move to items using `next` or `previous`



Iterating – STL style

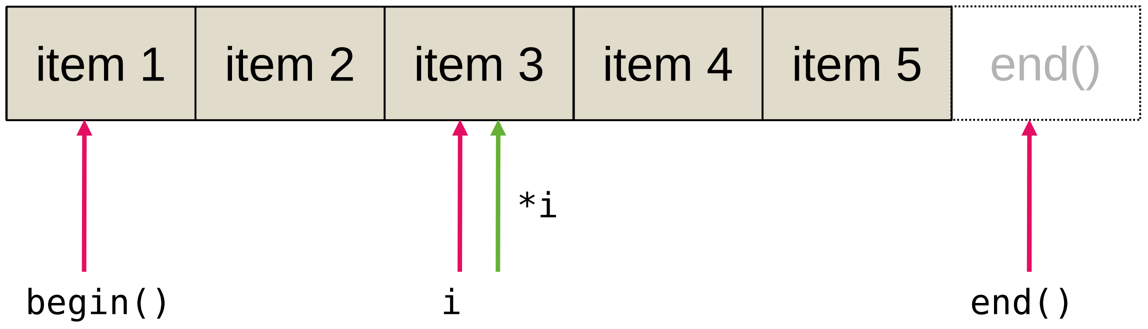
- Qt supports STL style iterators

```
for(QList<int>::const_iterator iter=list.begin();
    iter!=list.end(); ++iter)
    qDebug("Item: %d", *iter);
```

Use an `Iterator` if you need to modify the items

Both Qt and STL naming can be used. `Iterator|iterator` and `ConstIterator|const_iterator`

- STL iterators point at each item, and use invalid items as end markers
  - The first item is returned by `begin`
  - The end marker is returned by `end`
  - The `*` operator refers to the item value
  - When iterating backwards you must move the operator *before* accessing



Iterating for the lazy

- To iterate over a whole collection, use `foreach`

```
QStringList texts;
foreach(QString text, texts)
    doSomething(text);
```

Using const references helps improve performance. Not using it, still does not let you change the content of the list

```
QStringList texts;
foreach(const QString &text, texts)
    doSomething(text);
```

- Caveat! Make sure to copy the list when it is returned *by value*

```
const QList<int> sizes = splitter->sizes();
QList<int>::const_iterator i;
for(i=sizes.begin(); i!=sizes.end(); ++i)
     processSize(*i);
```

Copying is cheap thanks to implicit sharing

Interacting with STL

- A `QList` can be converted to and from the corresponding `std::list`

```
QList<int> list;
list << 0 << 1 << 1 << 2 << 3 << 5 << 8 << 13;

std::list<int> stlList = list.toStdList();

QList<int> otherList = QList<int>::fromStdList(stlList);
```

From Qt list to STL list

From STL list to Qt list

- Converting from and to STL means doing a deep copy of the list's contents – no implicit sharing takes place

Other collections

- What are the alternatives to `QList` and how do they compare to `QList`
- `QLinkedList`
  - Slow when using indexed access
  - Fast when using iterators
  - Fast (constant time) insertion into the middle of the list
- `QVector`
  - Uses continuous memory
  - Slow inserts and prepending

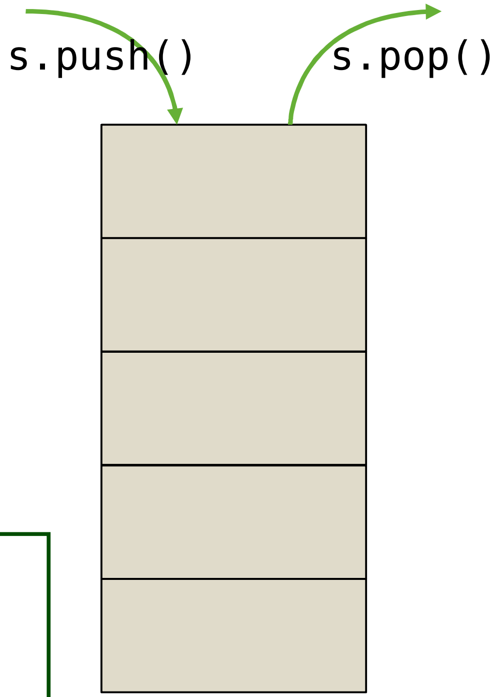
Other collections

Collection	Index access	Insert	Prepend	Append
`QList`	O(1)	O(n)	Amort. O(1)	Amort. O(1)
`QLinkedList`	O(n)	O(1)	O(1)	O(1)
`QVector`	O(1)	O(n)	O(n)	Amort. O(1)

- Notice that amortized behavior means unpredictable times in real-time setting
- Other collections
  - `QStringList` – based on `QList`
  - `QStack` – based on `QVector`
  - `QQueue` – based on `QList`
  - `QSet` – based on `QHash`

Special cases - QStack

- A stack if a LIFO container
  last in, first out



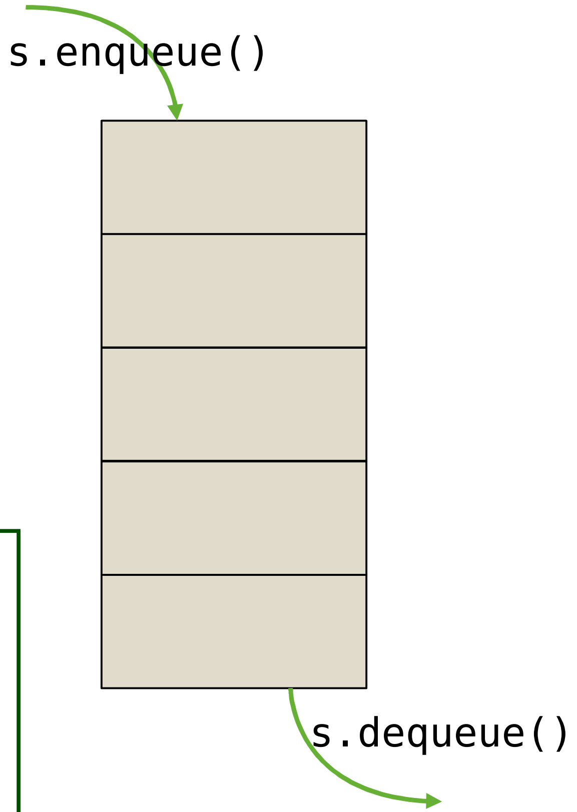
- Items are pushed onto the stack
- Items are popped off the stack
- The top item can be seen using `top()`

```
QStack<int> stack;
stack.push(1);
stack.push(2);
stack.push(3);
qDebug("Top: %d", stack.top()); // 3
qDebug("Pop: %d", stack.pop()); // 3
qDebug("Pop: %d", stack.pop()); // 2
qDebug("Pop: %d", stack.pop()); // 1
qDebug("isEmpty? %s", stack.isEmpty()?"yes":"no");
```

Special Cases - QQueue

- A queue is a FIFO container
  first in, first out
- Items are enqueued into the queue
- Items are dequeued from the queue
- The first item can be seen using `head()`

```
QQueue<int> queue;
queue.enqueue(1);
queue.enqueue(2);
queue.enqueue(3);
qDebug("Head: %d", queue.head());   // 1
qDebug("Pop: %d", queue.dequeue()); // 1
qDebug("Pop: %d", queue.dequeue()); // 2
qDebug("Pop: %d", queue.dequeue()); // 3
qDebug("isEmpty? %s", queue.isEmpty()?"yes":"no");
```



Special cases - QSet

- A set contains values, but only one instance of each value.
- It is possible to determine if a value is a part of the set or not

```
QSet<int> primes;
primes << 2 << 3 << 5 << 7 << 11 << 13;
for(int i=1; i<=10; ++i)
    qDebug("%d is %sprime", i, primes.contains(i)?"":"not ");
```

- You can also iterate over a set, to see all values

```
foreach(int prime, primes)
    qDebug("Prime: %d", prime);
```

- It is possible to convert a `QList` to a `QSet`

```
QList<int> list;
list << 1 << 1 << 2 << 2 << 2 << 3 << 3 << 5;
QSet<int> set = list.toSet();
qDebug() << list; // (1, 1, 2, 2, 2, 3, 3, 5)
qDebug() << set;  // (1, 2, 3, 5)
```

Key-value collections

- The QMap and QHash classes let you create associative arrays

```
QMap<QString, int> map;

map["Helsinki"] = 1310755;
map["Oslo"] = 1403268;
map["Copenhagen"] = 1892233;
map["Stockholm"] = 2011047;

foreach(const QString &key, map.keys())
    qDebug("%s", qPrintable(key));

if(map.contains("Oslo"))
{
    qDebug("Oslo: %d",
              map.value("Oslo"));
}
```

```
QHash<QString, int> hash;

hash["Helsinki"] = 1310755;
hash["Oslo"] = 1403268;
hash["Copenhagen"] = 1892233;
hash["Stockholm"] = 2011047;

foreach(const QString &key, hash.keys())
    qDebug("%s", qPrintable(key));
```

Using QMap

- The `QMap` class requires an `operator<` to be defined for the key type

  This operator is used to keep the keys in order

- Populating is done using `operator[]` or `insert`

```
map["Stockholm"] = 2011047;

map.insert("London", 13945000);
```

- For reading, use `value` combined with `contains`

```
if(map.contains("Oslo"))
    qDebug("Oslo: %d",
              map.value("Oslo"));

qDebug("Berlin: %d", map.value("Berlin",42));
```

Use value instead of [] to avoid adding items by mistake

Optional default value. If not specified a *default constructed* value is returned.

Hashing

- `QMap` uses keys of the type that are given in the template
- `QHash` uses `uint` values
- The key type is *hashed* to a `uint` value

- Working with `uint` values potentially improves performance
- Hashed values mean that keys are not sorted
- The hash function must be designed to avoid collisions to achieve good performance

Using QHash

- The key type must provide a `qHash` function and `operator==` to `QHash`

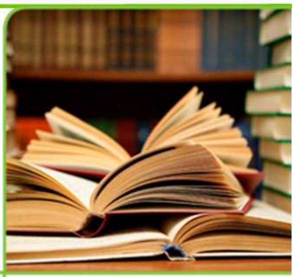
```
uint qHash(const Person &p)
{
    return p.age() ^ qHash(p.name());
}

bool operator==(const Person &first, const Person &second)
{
    return ((first.name() == second.name()) &&
            (first.age() == second.age()));
}
```

The hash function should be optimized according to known heuristics

- Populating and reading is identical to `QMap`

Multiple values per key



- QMultiMap and QMultiHash provide associative arrays supporting multiple values per key

```
QMultiMap<QString,int> multiMap;

multiMap.insert("primes", 2);
multiMap.insert("primes", 3);
multiMap.insert("primes", 5);
    ...
multiMap.insert("fibonacci", 8);
multiMap.insert("fibonacci", 13);

foreach(const QString &key, multiMap.uniqueKeys())
{
    QList<int> values = multiMap.values(key);
    QStringList temp;
    foreach(int value, values)
        temp << QString::number(value);

    qDebug("%s: %s", qPrintable(key), qPrintable(temp.join(",")));
}
```

There is no [], use insert

QMap *and* QHash *support this too using* insertMulti

keys repeat each key for each value, use uniqueKeys to get each key once

value returns the last insert for each key, values return a list of all values for the key

Break

Qt type definitions

- C++ does not define the size of types strictly across platforms

sizeof(void*) = ?

ARM = 4 bytes
x86 = 4 bytes
IA64 = 8 bytes
...

Depends on CPU architecture, operating system, compiler, etc

- For cross platform code it is important to define all types in a strict manner

Cross platform types

Type	Size	Minimum value	Maximum value
uint8	1 byte	0	255
uint16	2 bytes	0	65 535
uint32	4 bytes	0	4 294 967 295
uint64	8 bytes	0	18 446 744 073 709 551 615
int8	1 byte	-128	127
int16	2 bytes	-32 768	32 767
int32	4 bytes	-2 147 483 648	2 147 483 647
int64	8 bytes	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
quintptr	“pointer sized”	n/a	n/a
qptrdiff	“pointer sized”	n/a	n/a
qreal	fast real values	n/a	n/a

All types are defined in the `<QtGlobal>` header

Qt complex types

- Qt comes with a range of complex classes and types

QFont

QList

QColor

QString QRect QPen

QBrush

QSize

QPoint QImage

QPixmap

QByteArray

QVariant

- Sometimes, you want to be able to return any type through a generic interface

```
const QVariant &data(int index);
void setData(const QVariant &data, int index);
```

Data can be a string,
a picture, a color, a brush
an integer value, etc

- The `QVariant` class can be treated as a union
  - It would be impossible to create a union of Qt types as unions require default constructors
  - The variant class can contain custom complex types, e.g. QColor belongs to QtGui, QVariant to QtCore – unions cannot be extended with more types once they have been declared

Using QVariant

- The basic types are handled using the constructor and `to`*`Type`* methods

```
QVariant v;
int i = 42;
qDebug() << "Before:" << i; // Before: 42
v = i;
i = v.toInt();
qDebug() << "After:" << i;  // After: 42
```

- Non-QtCore types, such as custom types, are handled using the `setValue` method and templated `value<`*`type`*`>` method

```
QVariant v;
QColor c(Qt::red);
qDebug() << "Before:" << c; // Before: QColor(ARGB 1, 1, 0, 0)
v.setValue(c);
c = v.value<QColor>();      // After: QColor(ARGB 1, 1, 0, 0)
qDebug() << "After:" << c;
```

A custom complex type

- We implement a trivial class holding a person's name and age

```
class Person
{
public:
    Person();
    Person(const Person &);
    Person(const QString &, int);

    const QString &name() const;
    int age() const;

    void setName(const QString &);
    void setAge(int);

    bool isValid() const;

private:
    QString m_name;
    int m_age;
};
```

Does not have to be a QObject.

```
Person::Person() : m_age(-1) {}

...

void Person::setAge(int a)
{
    m_age = a;
}

bool Person::isValid() const
{
    return (m_age >= 0);
}
```

QVariant with Person

- Attempting to pass a `Person` object through a `QVariant` object fails

```
qmetatype.h:200: error: 'qt_metatype_id' is not a member of 'QMetaTypeId<Person>'
```

- Declaring the type in the meta-type system solves this

```
class Person
{
   ...
};

Q_DECLARE_METATYPE(Person)

#endif // PERSON_H
```

QVariant with Person

- When the type is registered as a meta type, Qt can store it in a `QVariant`

```
QVariant var;
var.setValue(Person("Ole", 42));
Person p = var.value<Person>();
qDebug("%s, %d", qPrintable(p.name()), p.age());
```

- Requirements for declared types
  - Public default constructor
  - Public copy constructor
  - Public destructor

And then it breaks...

- When working with signals and slots, most connections are direct
  - With direct connections, the type works
  - There are queued connections, i.e. non-blocking, asynchronous where the type does not work (e.g. across thread boundaries)

```
connect(src, SIGNAL(), dest, SLOT(), Qt::QueuedConnection);

...

QObject::connect: Cannot queue arguments of type 'Person'
(Make sure 'Person' is registered using qRegisterMetaType().)
```

Error message is printed at run-time

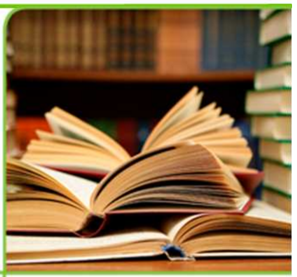
Registering the type

- The error message tells us what it needed
- The `qRegisterMetaType` function must be called before the connection is made (usually from `main`)

```
int main(int argc, char **argv)
{
    qRegisterMetaType<Person>();
    ...
```

Files and file systems

- Referring to files and directories in a cross platform manner poses a number of problems
  - Does the system have drives, or just a root?
  - Are paths separated by “/” or “\”?
  - Where does the system store temporary files?
  - Where does the user store documents?
  - Where is the application stored?



Paths

- Use the `QDir` class to handle paths

```
QDir d = QDir("C:/");
```

- Learn to use the static methods to initialize

```
QDir d = QDir::root(); // C:/ on windows
```

```
QDir::current() // Current directory
QDir::home()    // Home directory
QDir::temp()    // Temporary directory

// Executable directory path
QDir(QApplication::applicationDirPath())
```

Finding directory contents

- The `entryInfoList` returns a list of information for the directory contents

```
QFileInfoList infos = QDir::root().entryInfoList();
foreach(const QFileInfo &info, infos)
    qDebug("%s", qPrintable(info.fileName()));
```

Lists files and directories in arbitrary order.

- You can add filters to skip files or directories

```
QDir::Dirs
QDir::Files
QDir::NoSymLinks
```

Dirs, files or symbolic links?

```
QDir::Readable
QDir::Writable
QDir::Executable
```

Which files?

```
QDir::Hidden
QDir::System
```

Hidden files? System files?

Finding directory contents

- You can also specify sort order

```
QDir::Name
QDir::Time
QDir::Size
QDir::Type
```

Sort by ...

```
QDir::DirsFirst
QDir::DirsLast
```

Dirs before or after files

```
QDir::Reversed
```

Reverse order

- Listing all directories from the home directory ordered by name

```
QFileInfoList infos =
    QDir::root().entryInfoList(QDir::Dirs, QDir::Name);
foreach(const QFileInfo &info, infos)
    qDebug("%s", qPrintable(info.fileName()));
```

Filter

Order

Finding directory content

- Finally, you can add name filters

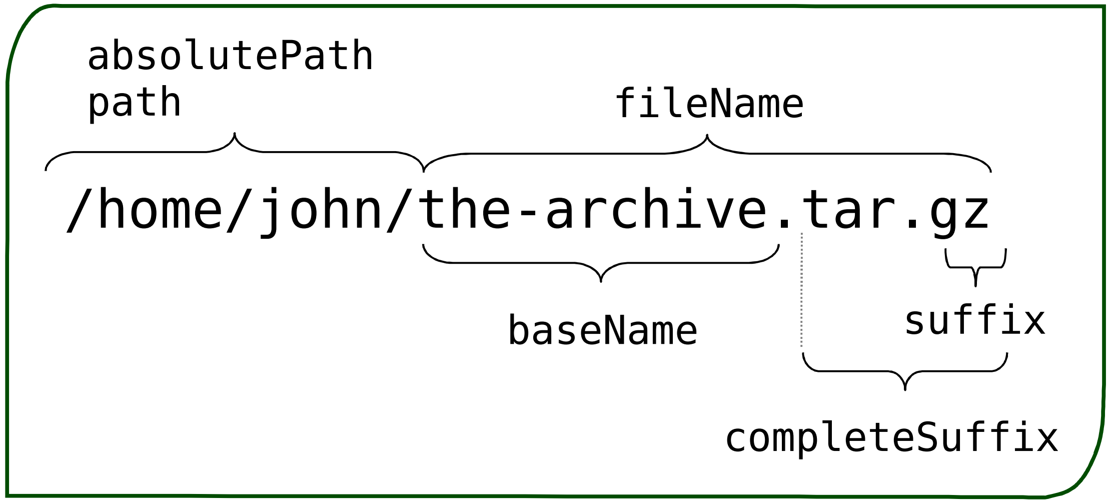
```
QFileInfoList infos =
    dir.entryInfoList(QStringList() << "*.cpp" << "*.h",
                      QDir::Files, QDir::Name);
foreach(const QFileInfo &info, infos)
    qDebug("%s", qPrintable(info.fileName()));
```

All **cpp** and **h** files

QFileInfo

- Each `QFileInfo` object has a number of methods
  - `absoluteFilePath` – full path to item
  - `isDir` / `isFile` / `isRoot` – type of item
  - `isWriteable` / `isReadable` / `isExecutable` – permission for file

Great for creating new `QDir` objects when traversing.



Opening and reading files

- The `QFile` is used to access files

```
QFile f("/home/john/input.txt");

if (!f.open(QIODevice::ReadOnly))
    qFatal("Could not open file");
```

```
QByteArray data = f.readAll();
processData(data);
```

Reads all data in one go.

```
while(!f.atEnd())
{
    QByteArray data = f.read(160);
    processData(data);
}
```

Reads up to 160 bytes each time.

```
f.close();
```

Writing to files

- When writing files, open it in `WriteOnly` mode and use the write method to add data to the file

```
QFile f("/home/john/input.txt");

if (!f.open(QIODevice::WriteOnly))
    qFatal("Could not open file");

QByteArray data = createData();
f.write(data);

f.close();
```

- Files can also be opened in `ReadWrite` mode
- The flags `Append` or `Truncate` can be used in combination with write-enabled modes to either append data to the file or to truncate it (i.e. clear the file from its previous contents)

```
if (!f.open(QIODevice::WriteOnly|QIODevice::Append))
```

The QIODevice

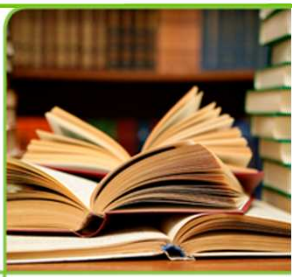
- `QFile` is derived from `QIODevice`
- The constructors `QTextStream` and `QDataStream` take a `QIODevice` pointer as an argument, not a `QFile` pointer
- There are `QIODevice` implementations
  - `QBuffer` – for reading and writing to memory buffers
  - `QextSerialPort` – for serial (RS232) communication (3rd party)
  - `QAbstractSocket` – the base of TCP, SSL and UDP socket classes
  - `QProcess` – for reading and writing to processes' standard input and output

Streaming to files

- The read and write methods feel awkward in many situations – handling complex types, etc
- A modern approach is to use stream operators
- Qt offers two types of stream operators
  - For handling text files
  - For handling binary file formats

QTextStream

- The `QTextStream` class handles reading and writing from text based files
- The class is
  - codec aware (uses locale by default, but can be set explicitly as well)
  - aware of lines and words
  - aware of numbers



Writing to text streams

- Use the << operator and modifiers, much as using STL's streams

```
QFile f(...);
if(!f.open(QIODevice::WriteOnly))
    qFatal("Could not open file");

QTextStream out(&f);
out << "Primes: " << qSetFieldWidth(3) << 2 << 3 << 5 << 7 << endl;
```

Numbers can be put directly into the stream

Modifier, sets the minimum field width

Adds a line break to the stream

Results in:

```
Primes:   2  3  5  7
```

Reading using text streams

- It is possible to read the file line by line

```
QTextStream in(&f);
while(!f.atEnd())
    qDebug("line: '%s'", qPrintable(in.readLine()));
```

- You can also extract words and numbers

```
QTextStream in(&f);
QString s;
int i;
in >> s >> i;
```

- Use `atEnd` to determine if you've reached the end of the file

Handling binary files

- The `QDataStream` class is used for streaming bytes
  - Guarantees byte-ordering (default big endian)
  - Supports basic types
  - Support Qt complex types
  - Supports adding custom complex types

```
if (!f.open(QIODevice::WriteOnly))
    qFatal("Could not open file");

QDataStream ds(&f);
ds << QString("Unicode string data");
```

Simply pass a pointer to a `QFile` object to the stream constructor to setup a stream for a specific file.

Data streams as a file format

- When basing a file format on `QDataStream` there are some details to keep in mind
  - Versioning – as Qt's structures evolve, so do their binary serialization formats. Using `QDataStream::setVersion`, you can explicitly force a specific serialization format to be used.
  - Type information – Qt does not add type information, so you need to keep track of which you store in what order.
  - Byte ordering – Qt's data streams are big endian by default, but when using the class for handling legacy file formats you can set the byte ordering using `QDataStream::setByteOrder`.

Data streams as a file format

```
QFile f("file.fmt");
if (!f.open(QIODevice::WriteOnly))
    qFatal("Could not open file");

QDataStream out(&f);
out.setVersion(QDataStream::Qt_4_6);

quint32 value = ...;
QString text = ...;
QColor color = ...;

out << value;
out << text;
out << color;
```

Versions down to Qt 1.0 are supported here.

Ensure to match types and order when writing and reading streams.

```
QFile f("file.fmt");
if (!f.open(QIODevice::ReadOnly))
    qFatal("Could not open file");

QDataStream in(&f);
in.setVersion(QDataStream::Qt_4_6);

quint32 value = ...;
QString text = ...;
QColor color = ...;

in >> value;
in >> text;
in >> color;
```

If you want to serialize objects of mixed types without specifying the order of the types you can serialize `QVariant` objects.

Streaming custom types

- By implementing the stream operators << and >> custom types can be streamed from and to data streams

```
QDataStream &operator<<(QDataStream &out, const Person &person)
{
    out << person.name();
    out << person.age();
    return out;
}

QDataStream &operator>>(QDataStream &in, Person &person)
{
    QString name;
    int age;
    in >> name;
    in >> age;
    person = Person(name, age);
    return in;
}
```

A friend function could have accessed `m_age` and `m_name` directly.

Streaming custom types

- To be able to stream a custom type contained in a `QVariant` object the stream operators must be registered

```
qRegisterMetaTypeStreamOperators<Person>("Person");
```

- When the variant is streamed, it adds the name of the data type to ensure that it can be restored from the stream later

```
00: 0x00 0x00 0x00 0x7f 0x00 0x00 0x00 0x00   ________
08: 0x07 0x50 0x65 0x72 0x73 0x6f 0x6e 0x00   _Person_
16: 0x00 0x00 0x00 0x06 0x00 0x4f 0x00 0x6c   _____O_l
24: 0x00 0x65 0x00 0x00 0x00 0x2a             e___*
```

Custom types check-list

- Implement
  - `Type::Type()` – Public default constructor
  - `Type::Type(const Type &other)` – Public copy constructor
  - `Type::~Type()` – Public destructor
  - `QDebug operator<<` – Convenient debugging
  - `QDataStream operator<<` and `>>` – Streaming
- Register
  - `Q_DECLARE_METATYPE` – In header
  - `qRegisterMetaType` – In main
  - `qRegisterMetaTypeStreamOperators` – In main