Brief Introduction to Qt Programming

August 8, 2011

Overview

PROGRAMMING WITH QT

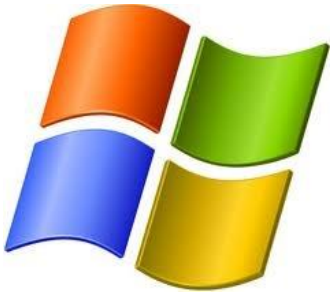
- **Introduction**
- Container Classes and Strings
- Widgets and GUIs
- Resources

What is Qt?

- Cross-platform C++ application development **framework**
- Effectively adds a few new C++ keywords (*signal*, *slot*, *emit*, etc)
- Provides facilities for:
  - GUIs
  - Internationalization
  - XML serialization
  - Media
  - More

Why use Qt?

- “Write once, compile anywhere”

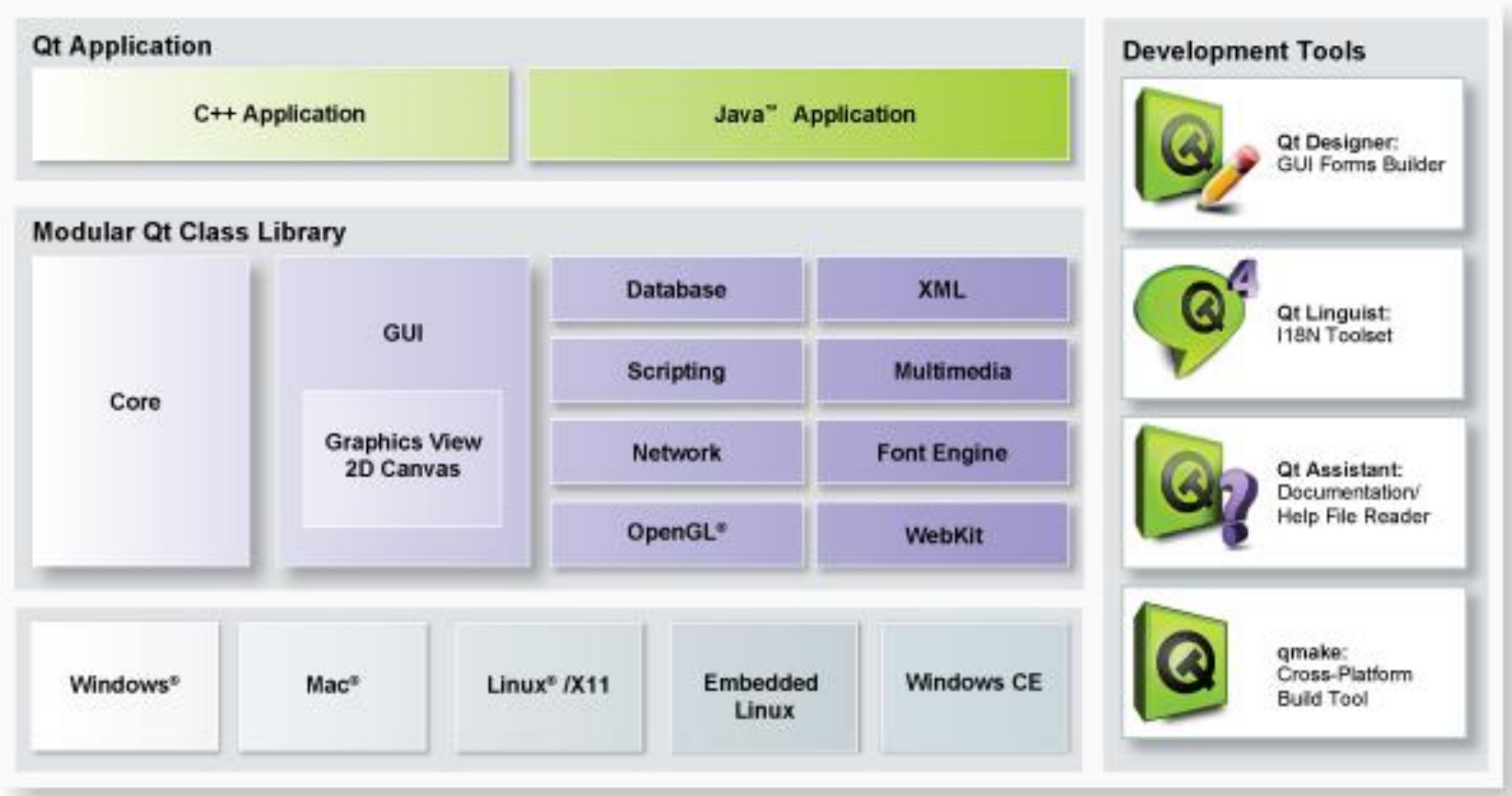



- GUIs look native (or pretty close to it) with just a recompile
- Commercial, GPL and LGPL licences

Who uses Qt?

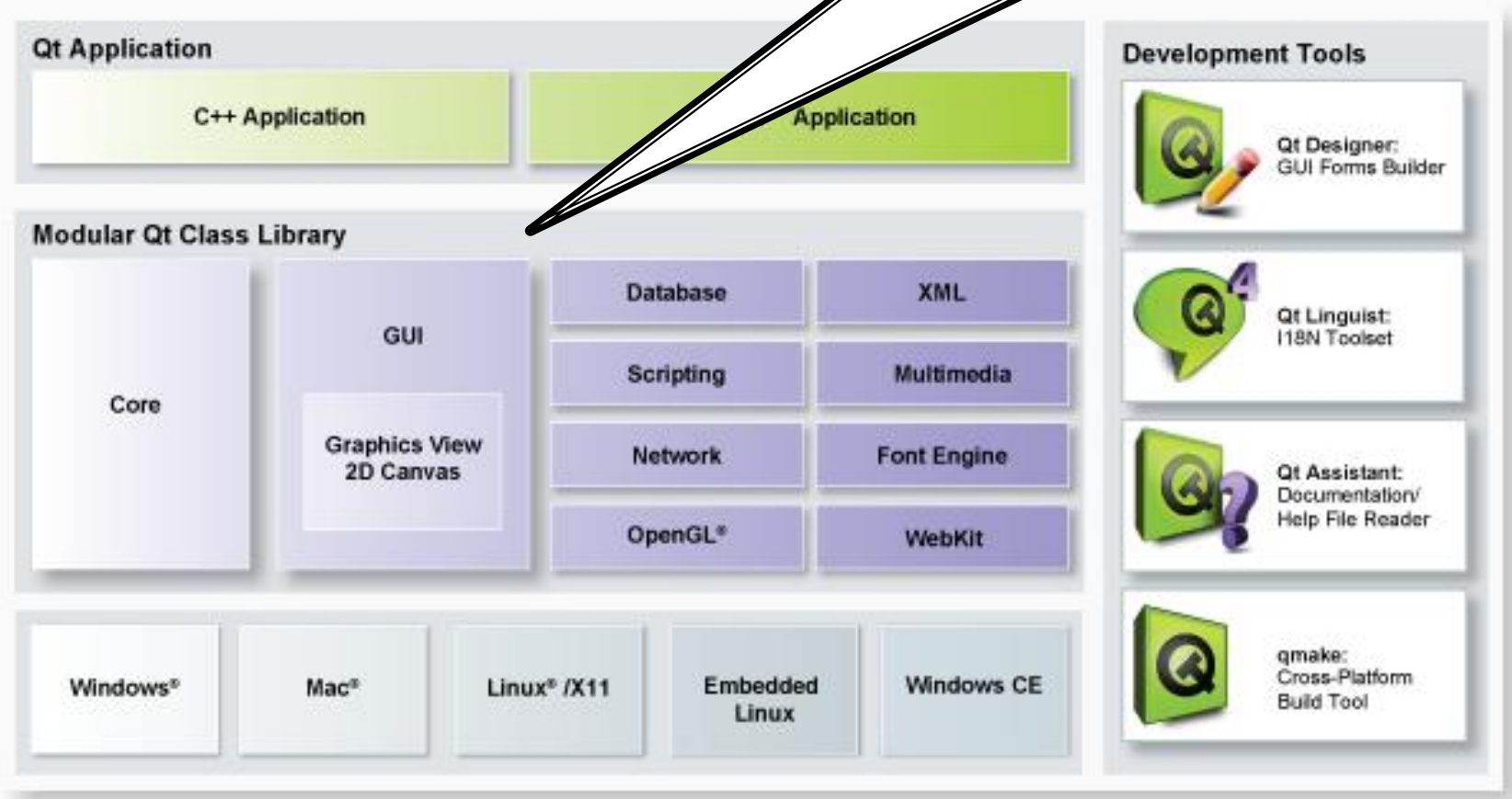
- European Space Agency
- Google Earth
- K Desktop Environment (KDE)
- Adobe Photoshop Album
- VLC Player
- Autodesk Maya
- …

Architecture



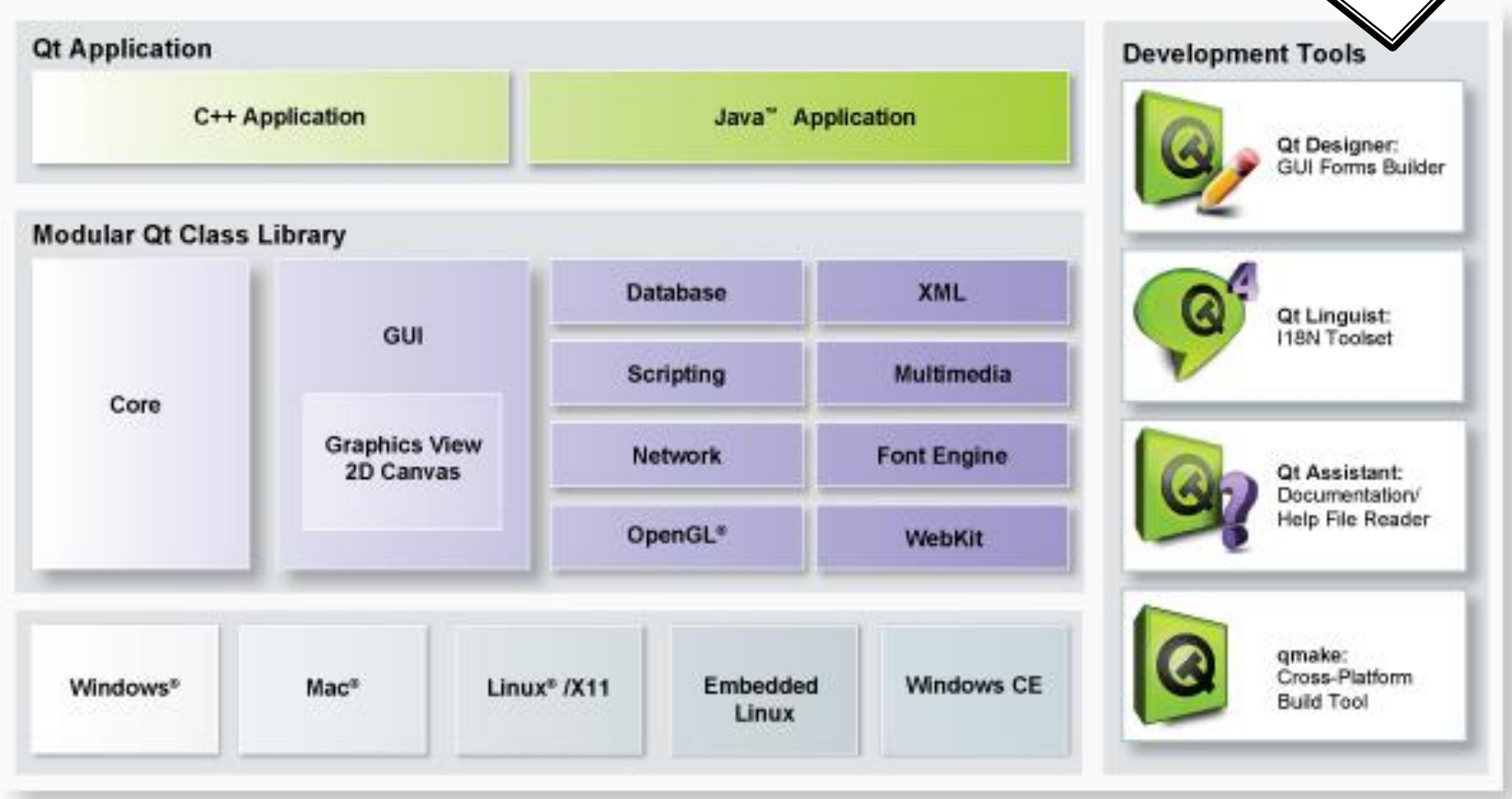
Architecture

C++ class library for writing GUIs, database access, XML parsing, etc

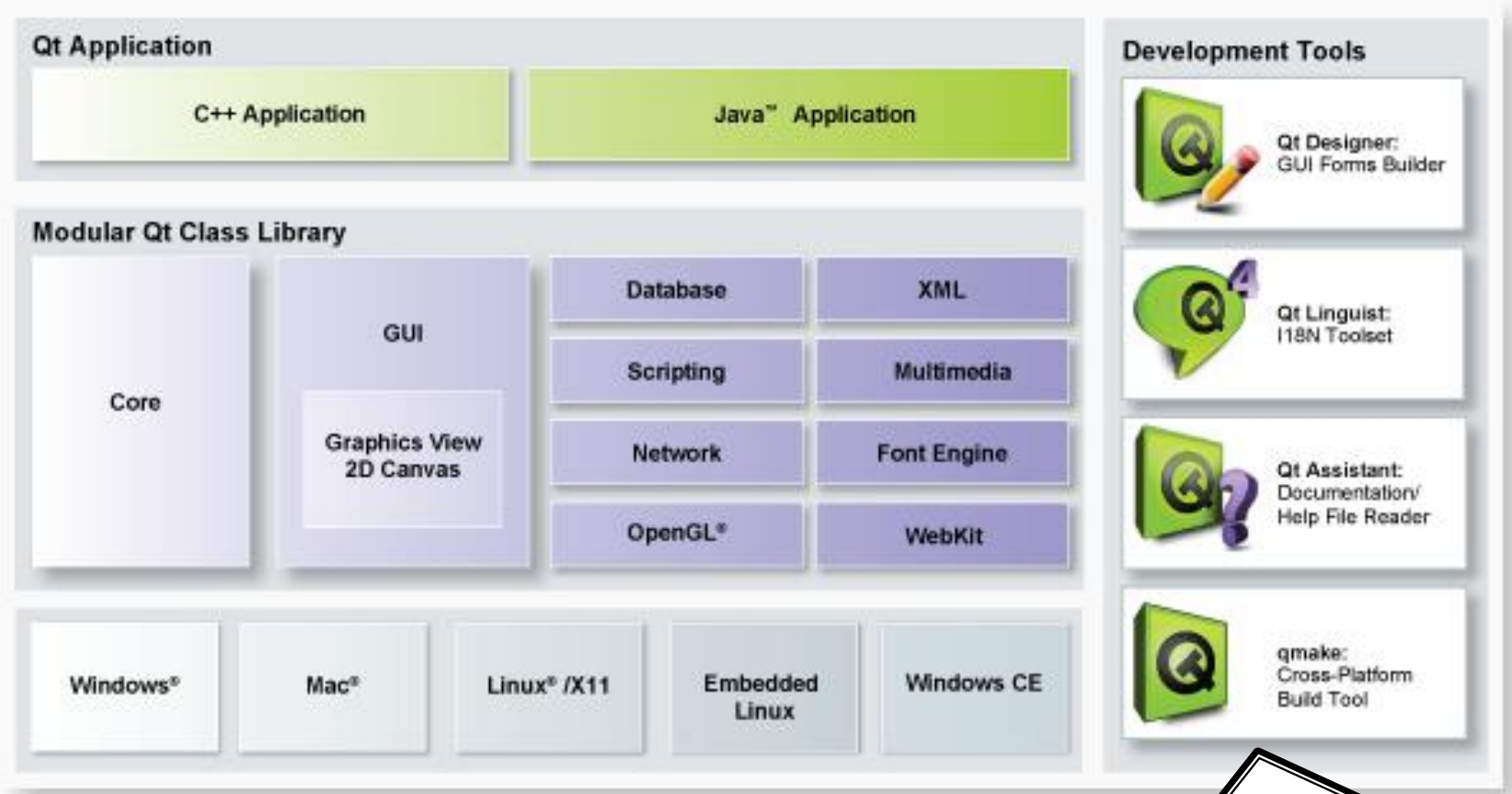


Architecture

Development tools (you generally don't have to use them if you prefer your own)



Architecture



qmake enables you to write one project file (with the extension ".pro") and generate platform-specific project files from that as needed

Hello World


```
#include <QApplication>
#include <QLabel>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    QLabel label("Hello world!");
    label.show();
    return a.exec();
}
```

Hello World


```
#include <QApplication>
#include <QLabel>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    QLabel label("Hello world!");
    label.show();
    return a.exec();
}
```

Headers named after the class they define

Hello World


```
#include <QApplication>
#include <QLabel>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    QLabel label("Hello world!");
    label.show();
    return a.exec();
}
```

All Qt apps have a QApplication or QCoreApplication instance

Hello World


```
#include <QApplication>
#include <QLabel>

int main(int argc, char *argv[])
{
    QApplication a(argc, a

    QLabel label("Hello world!");
    label.show();
    return a.exec();
}
```

The label widget becomes our window because it has no parent

Hello World


```
#include <QApplication>
#include <QLabel>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    QLabel label("Hello
    label.show();
    return a.exec();
}
```

QApplication::exec() runs the message loop and exits when the user closes the window

Overview

PROGRAMMING WITH QT

- Introduction
- **Container Classes and Strings**
- Widgets and GUIs
- Resources

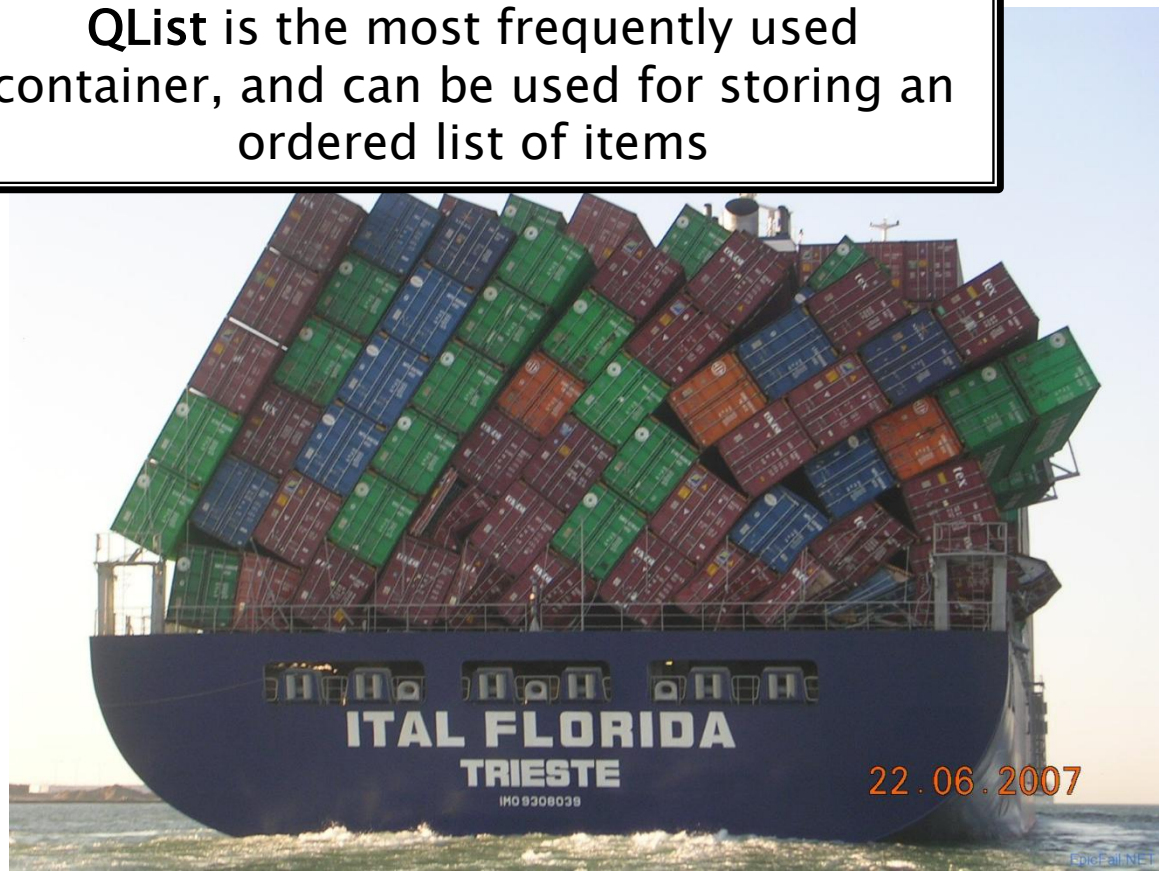
Qt Container Classes

- **QList**
- QMap
- QStack
- QQueue
- QSet
- QVector
- Others…

Qt Container Classes

- **QList**
- QMap
- QStack
- QQueue
- QSet
- QVector
- Others…

QList is the most frequently used container, and can be used for storing an ordered list of items



Qt Container Classes

- **QList**
- QMap
- QStack
- QQueue
- QSet
- QVector
- Others…

QMap is frequently used when one needs to store key-value pairs and perform fast lookups on the key value

Qt Container Classes

- Grow as needed

```
QList<QString> list;
list << "Item1" << "Item2";
list.append("Item3");
list.push_back("Item4");
```

- Iteration using various methods

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

Qt Container Classes

- Grow as needed

Multiple methods for appending (<<, **append, push_back**) are all equivalent.

```
QList<QString> list;
list << "Item1" << "Item2";
list.append("Item3");
list.push_back("Item4");
```

- Iteration using various methods

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

Qt Container Classes

- Implicitly shared

```
QList<QString> list1;
list1 << “Item1” << “Item2”;

QList<QString> list2 = list1;
```

- Easy constructs for iteration: *foreach*!!!

```
QList<QString> list;

foreach (const QString &str, list)
    qDebug() << str;
```

Qt Container Classes

- Implicitly shared

```
QList<QString> list1;
list1 << "Item1" << "Item2";

QList<QString> list2 = list1;
```

Only a quick, constant-time, shallow copy of `list1` is made here. If `list1` or `list2` is modified later, a deep copy will be made at that time. This is called **Implicit Sharing** and is a core Qt concept.

- Easy constructs

```
QList<QString

foreach (const QString &str, list)
    qDebug() << str;
```

Qt Strings

- **QString!**
- Built-in Unicode support
- Useful methods (`trimmed()`, `split()`)
- Cheap copying (implicit sharing)

- Believe it or not, they can actually contain embedded '`\0`' characters…

Qt Strings

- Concatenation and formatting

```
QString first = “James”;
QString last = “Kirk”;

QString fullName = first + “ “ + last;

QString fullName2 =
  QString(“%1 %2”).arg(first).arg(last);
```

- Translation

```
QString translatedSend = tr(“Send”);
```

Qt Strings

Both `fullName` and `fullName2` end up the same: “James Kirk”

- Concatenation and formatting

```
QString first = “James”;
QString last = “Kirk”;

QString fullName = first + “ “ + last;

QString fullName2 =
  QString(“%1 %2”).arg(first).arg(last);
```

- Translation

```
QString translatedHello = tr(“Hello”);
```

Qt Strings

- Concatenation and formatting

```
QString first = "James";
QString last = "Kirk";

QString fullName =

QString fullName2 =
  QString("%1 %2").arg(first).arg(
```

If a translation for "Hello" is available at runtime, tr("Hello") will return that translated string instead of "Hello".

- Translation

```
QString translatedHello = tr("Hello");
```

Qt Strings

- Splitting/joining on a delimiter character

```
QString numbers = "1,2,54,23,7";
QStringList nums = numbers.split(',');
int sum = 0;
foreach (QString num, numbersList)
     sum += num.toInt();
```

- Removing leading or trailing whitespace

```
QString name = "  William Shatner\n";
QString trimmedName = name.trimmed();
```

Qt Strings

- Splitting/joining on a

```
QString numbers = "
QStringList nums = numbe          lit(',');
int sum = 0;
foreach (QString num, nu          bersList)
     sum += num.toInt();
```

Also available: `QString::toLong()`, `QString::toDouble()`, `QString::toLower()`, etc

- Removing leading or trailing whitespace

```
QString name = "  William Shatner\n";
QString trimmedName = name.trimmed();
```

Qt Strings

- Splitting/joining on a delimiter character

```
QString numbers = "1,2,54,23,7";
QStringList nums = numbers.split(',');
int sum = 0;
foreach (QString num, numbersList)
     sum += num.toI
```

Afterwards, `trimmedName` equals "William Shatner"

- Removing leading or trailing whitespace

```
QString name = "  William Shatner\n";
QString trimmedName = name.trimmed();
```

Overview

PROGRAMMING WITH QT

- Introduction
- Container Classes and Strings
- **Widgets and GUIs**
- Resources

The King of Classes: QObject

- Most objects inherit from **QObject:**
  - QWidget
  - QThread
  - Etc..
- Together with the *moc* precompiler, enables many Qt features…

What is QObject good for?

- Signals and slots
- Memory management (parents kill their children)
- Properties
- Introspection (`QObject::className()`, `QObject::inherits()`)

BUT:

- QObjects cannot be copied (they have “identity”)

Signals and Slots

- General-purpose way for Qt objects to notify one-another when something changes.
- Simple *Observer* pattern:

```
QPushButton *button = new
QPushButton("About", this);

QObject::connect(
    button, SIGNAL(clicked()),
    qApp, SLOT(aboutQt()));
```

Signals and Slots

- General-purpose way for Qt objects to notify one-another when something changes.
- Simple *Observer* pattern:

```
QPushButton *button =
QPushButton("About", 

QObject::connect(
    button, SIGNAL(clicked()),
    qApp, SLOT(aboutQt()));
```

When the button *emits* the "clicked()" *signal*, the global application's "aboutQt()" *slot* will be executed.

Parent-child Memory Management

- Parents destroy their children… A good thing!?

```
void showAbout()
{
    QDialog aboutDialog(NULL);
    QLabel* label = new QLabel(
        "About this..", &aboutDialog);
    aboutDialog.exec();
}
```

- No memory leak! *aboutDialog*'s destructor deletes *label*.

Parent-child Memory Management

- Parents destroy their children… A good thing!?

```
void showAbout()
{
    QDialog aboutDialog(NULL);
    QLabel* label = new QLabel(
        "About this..", &aboutDialog);
    aboutDialog.exec();
}
```

`label` is freed when its parent, `aboutDialog`, goes out of scope.

- No memory leak! *aboutDialog*'s destructor deletes *label*.

Parent-child Memory Management

- So what happens if the child is on the stack?

```
void showAbout()
{
     QDialog* about = new QDialog(NULL);
     QLabel label(
          "About this..", aboutDialog);
     about->exec();
      delete about;
}
```

Parent-child Memory Management

- So what happens if the child is on the stack?
  - Crash!! Qt tries to *free* the stack-allocated child!

```
void showAbout()
{
    QDialog* about = new QDialog(NULL);
    QLabel label(
        "About this..", aboutDialog);
    about->exec();
    delete about;  // CRASH!
}
```

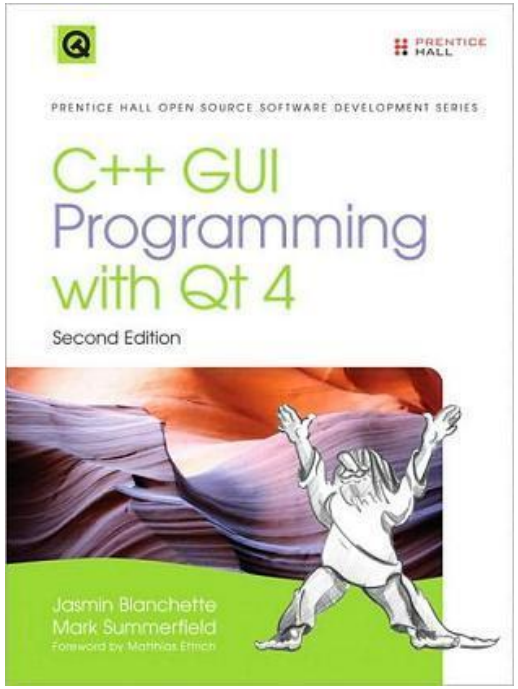
`about`'s destructor tries to free `label`, but label is on the stack, not the heap!

Overview

PROGRAMMING WITH QT

- Introduction
- Container Classes and Strings
- Widgets and GUIs
- **Resources**

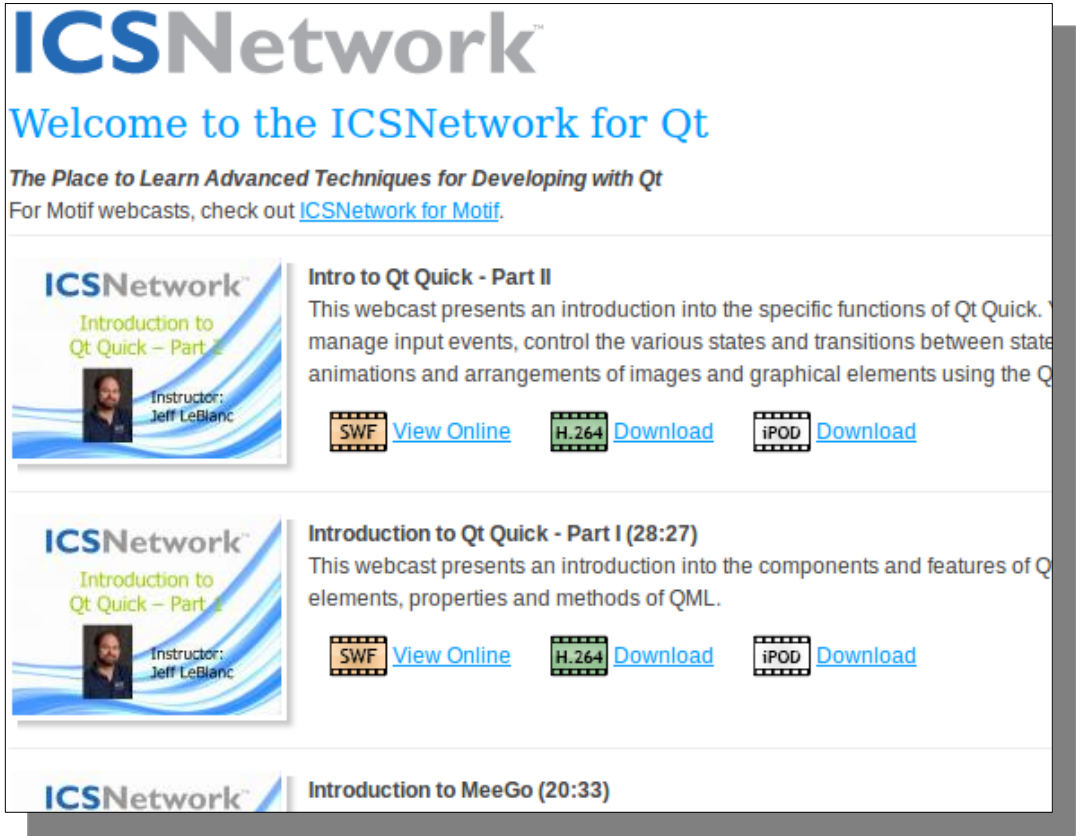
Where to get more information



Available as a free PDF from author at:

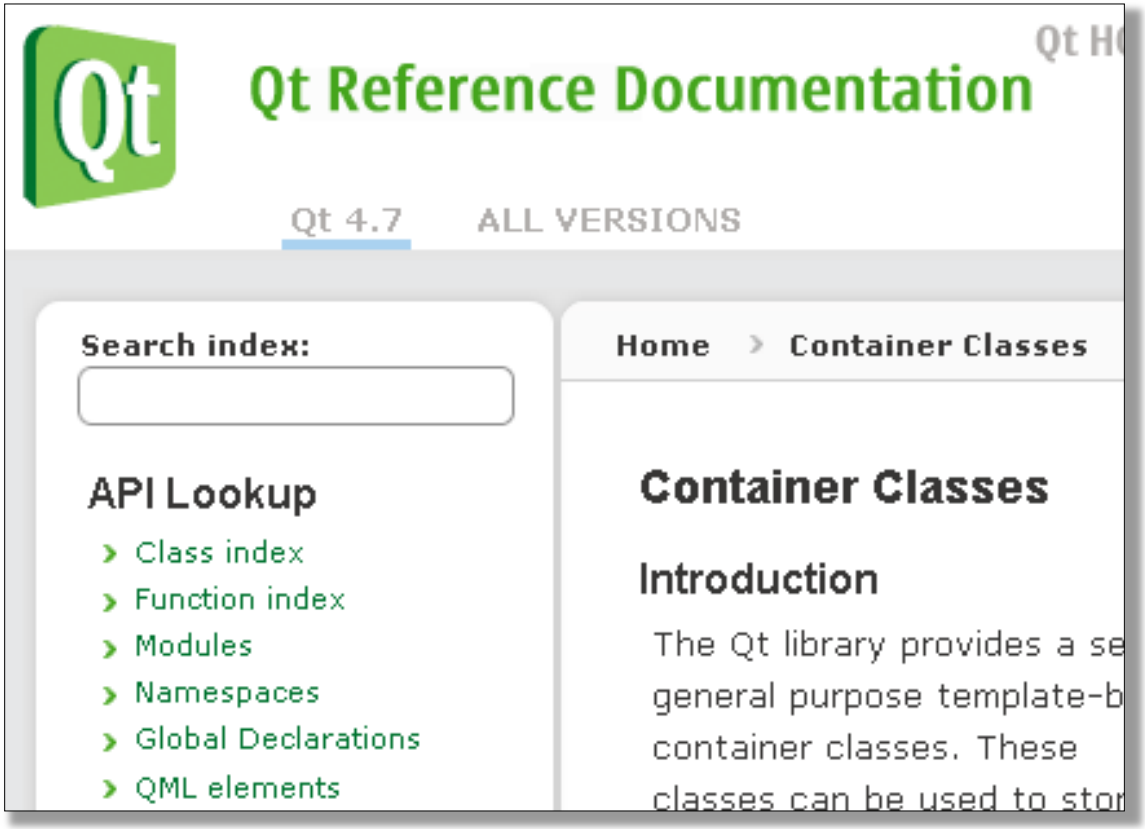
http://www.qtrac.eu/marksummerfield.html

Where to get more information



http://www.ics.com/learning/icsnetwork/

Where to get more information



http://doc.qt.nokia.com

THANK YOU