Qt – Core features

Timo Strömmer, May 26, 2010

Contents

- C++ refresher
- Core features
  - Object model
  - Signals & slots
  - Event loop
- Shared data
  - Strings
  - Containers
- Private implementation pattern

Quick C++ refresher

Object-oriented programming with C++

Quick C++ OOP refresher

<symbio>

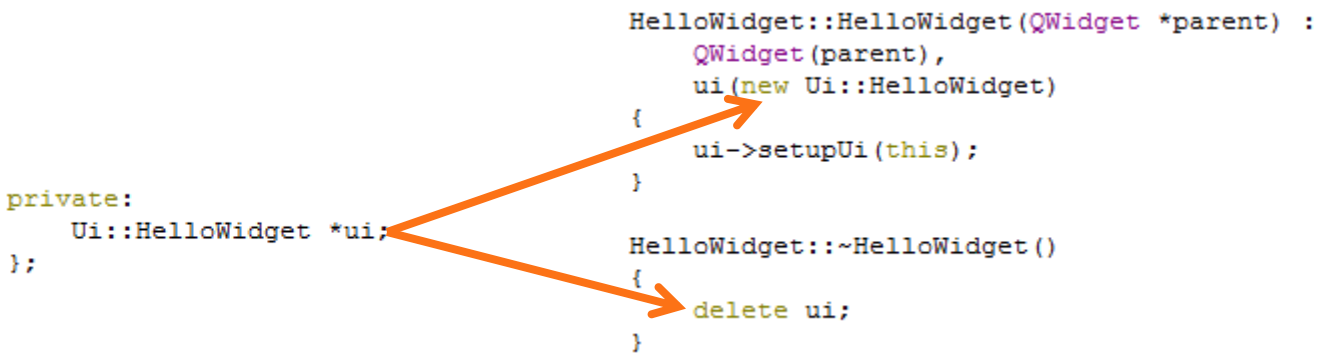
- A *class* defines the structure of an *object*
  - Objects are created with *new* operator and freed with *delete* operator
  - When object is created, its *constructor* is called and delete calls *destructor*

```
class HelloWidget : public QWidget
{
    Q_OBJECT

public:
    HelloWidget(QWidget *parent = 0);
    ~HelloWidget();

protected:
    void changeEvent(QEvent *e);

private:
    Ui::HelloWidget *ui;
};
```

```
HelloWidget::HelloWidget(QWidget *parent) :
    QWidget(parent),
    ui(new Ui::HelloWidget)
{
    ui->setupUi(this);
}

HelloWidget::~HelloWidget()
{
    delete ui;
}
```

Quick C++ OOP refresher

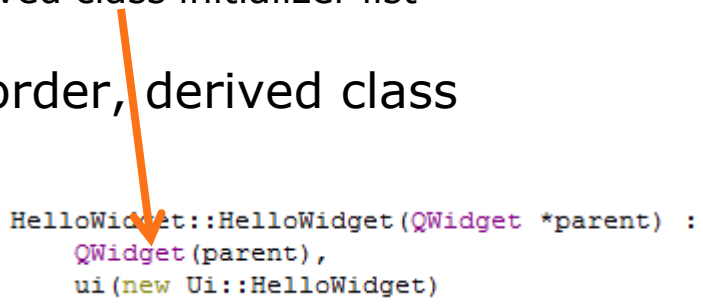
- Object data must be initialized in constructor and freed in destructor
  - C++ has constructor *initializer list*



Quick C++ OOP refresher

<symbio>

- A class may inherit other classes
  - *Derived* class gets all the properties of the *base* class
  - When object is created, base class constructor is called first
    - If base class constructor needs parameters, it needs to be explicitly called from derived class initializer list
  - Delete happens in reverse order, derived class destructor is called first



Memory management

<symbio>

- Calling *new* reserves an area of memory for the object
  - The area will stay reserved until *delete* is called or the program terminates
- If objects are not deleted, the program has *memory leaks*
  - Severity of the leaks depend on allocation size, number of allocations and life-time of program
  - Debugging can be costly

Memory management

<symbio>

- Objects allocated with *new* are reserved from program *heap*
- Other option is to allocate objects from program *stack*
  - Stack variables are automatically deleted when leaving the current program scope
- In general, anything based on QObject is allocated with *new*
  - Some exceptions of course…

Memory management

- Stack vs. heap

```
{
    QObject *obj = new QObject();
    QObject obj2();
    int value = 0;
}
```

Core features

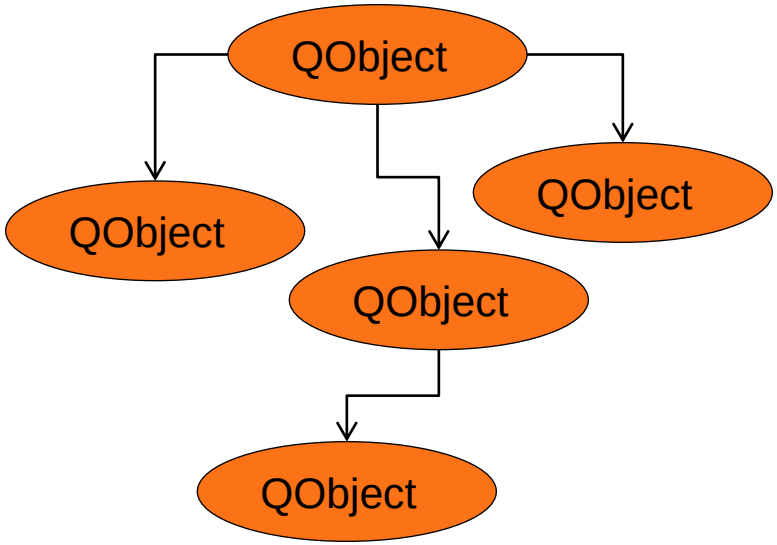
Object model, signals & slots and event loop

Object model

- Usual Qt program is based around a tree-based hierarchy of objects
  - Designed to help with C++ memory management
  - Based on QObject class
  - Do not confuse with class inheritance

Object model

- A QObject may have a *parent* object and number of *child* objects
- Object without parent is called a *root* object
- When an object is deleted, it will also delete all it's children



Object model and GUI

- All GUI components inherit from QWidget, which in turn inherits from QObject
  - Thus, GUI widgets are also arranged into tree hierarchy
  - The root widget is a *window*

  - Enabling / disabling or showing / hiding a widget will also affect its children

Signals and slots

- Qt way of making callback functions simple
  - Example cases
    - What happens when user presses a GUI button
    - What happens when data arrives from network
  - Similar semantics as with Java listeners
- A *signal* is *emitted*, which results in a function call to all *slots* that have been connected to the signal

Signals and slots

- Code to support signal-slot connections is generated by the *moc* tool when project is compiled
- Special keywords are used, which are interpreted by *moc*
  - Q_OBJECT, signals, slots, emit

Special keywords

- Q_OBJECT keyword must be added to every class that inherits from QObject base class
  - Tells *moc* to parse the class contents
  - QtCreator complains if missing

```
class Emitter : public QObject {
public:
    void doSomething() { emit changed(); }
signals:
    void changed();
};
```

you forgot the Q_OBJECT macro

Special keywords

- *signals* keyword is used to start a block of signal definitions
  - Signal functions are not implemented. Instead, the code for them is generated by *moc*
  - Signals can have parameters as any normal function
    - A slot that is connected to signal must have matching parameter count and types

```
signals:
    void helloSignal();
    void signalWithParams(const QString &data, qint32 value);

public slots:
    void hello();
```

Special keywords

- *slots* keyword starts a block of slot definitions
  - Each slot is a normal C++ function
    - Can be called directly from code
  - Normal visibility rules apply when called directly from code
    - However, signal-slot connections will ignore visibility and thus it's possible to connect to private slot from anywhere

```
public slots:
    void publicSlot();

protected slots:
    void protectedSlot();

private slots:
    void internalSlot();
```

Special keywords

<symbio>

- *emit* keyword is used to send a notification to all slots that have been connected to the signal
  - Object framework code loops over the slots that have been connected to the signal and makes a regular function call to each

Connecting signals to slots

<symbio>

- Connections are made with *QObject::connect* static functions
  - No access control, anyone can connect anything
  - Class headers are not needed if signal and slot function signatures are known
- Component-based approach
  - Components provide services
  - Controller makes the connections between components

Connecting signals to slots


```
class Emitter : public QObject {
    Q_OBJECT
public:
    void doSomething() { emit changed(); }
signals:
    void changed();
};

class Observer : public QObject {
    Q_OBJECT
public slots:
    void notifyChange() {
    }
};

class Manager : public QObject {
    Q_OBJECT
public:
    Manager() : emitter(new Emitter(this)),
                observer(new Observer(this)) {
    }
    void connectObjects() {
        QObject::connect(emitter, SIGNAL(changed()),
                         observer, SLOT(notifyChange()));
        emitter->doSomething();
    }
private:
    Emitter *emitter;
    Observer *observer;
};
```

Signals and slots

- Comparing Qt and Java

```
class Emitter : public QObject {
    Q_OBJECT
public:
    void doSomething() { emit changed(); }
signals:
    void changed();
};

class Observer : public QObject {
    Q_OBJECT
public slots:
    void notifyChange() {
    }
};

class Manager : public QObject {
    Q_OBJECT
public:
    Manager() : emitter(new Emitter(this)),
                observer(new Observer(this)) {
    }
    void connectObjects() {
        QObject::connect(emitter, SIGNAL(changed()),
                         observer, SLOT(notifyChange()));
        emitter->doSomething();
    }
private:
    Emitter *emitter;
    Observer *observer;
};
```

```
import java.util.ArrayList;

interface ChangeEventListener {
    void notifyChange();
}

class Emitter {
    private ArrayList<ChangeEventListener> listeners =
        new ArrayList<ChangeEventListener>();
    void addChangeListener(ChangeEventListener listener) {
        listeners.add(listener);
    }
    void doSomething() { changed(); }
    private void changed() {
        for (int i = 0; i < listeners.size(); i++) {
            listeners.get(i).notifyChange();
        }
    }
}

class Observer implements ChangeEventListener {
    public void notifyChange() {
    }
}

public class Manager {
    private Emitter emitter = new Emitter();
    private Observer observer = new Observer();
    void connectObjects() {
        emitter.addChangeListener(observer);
        emitter.doSomething();
    }
}
```

Event loop

<symbio>

- Purpose of event loop is to keep program running and responsive to whatever happens
  - User interaction
  - Interaction with environment
- Basic idea

```
while (isRunning()) {
    if (hasEvent()) {
        processEvent();
    } else {
        waitAMoment();
    }
}
```

Event processing

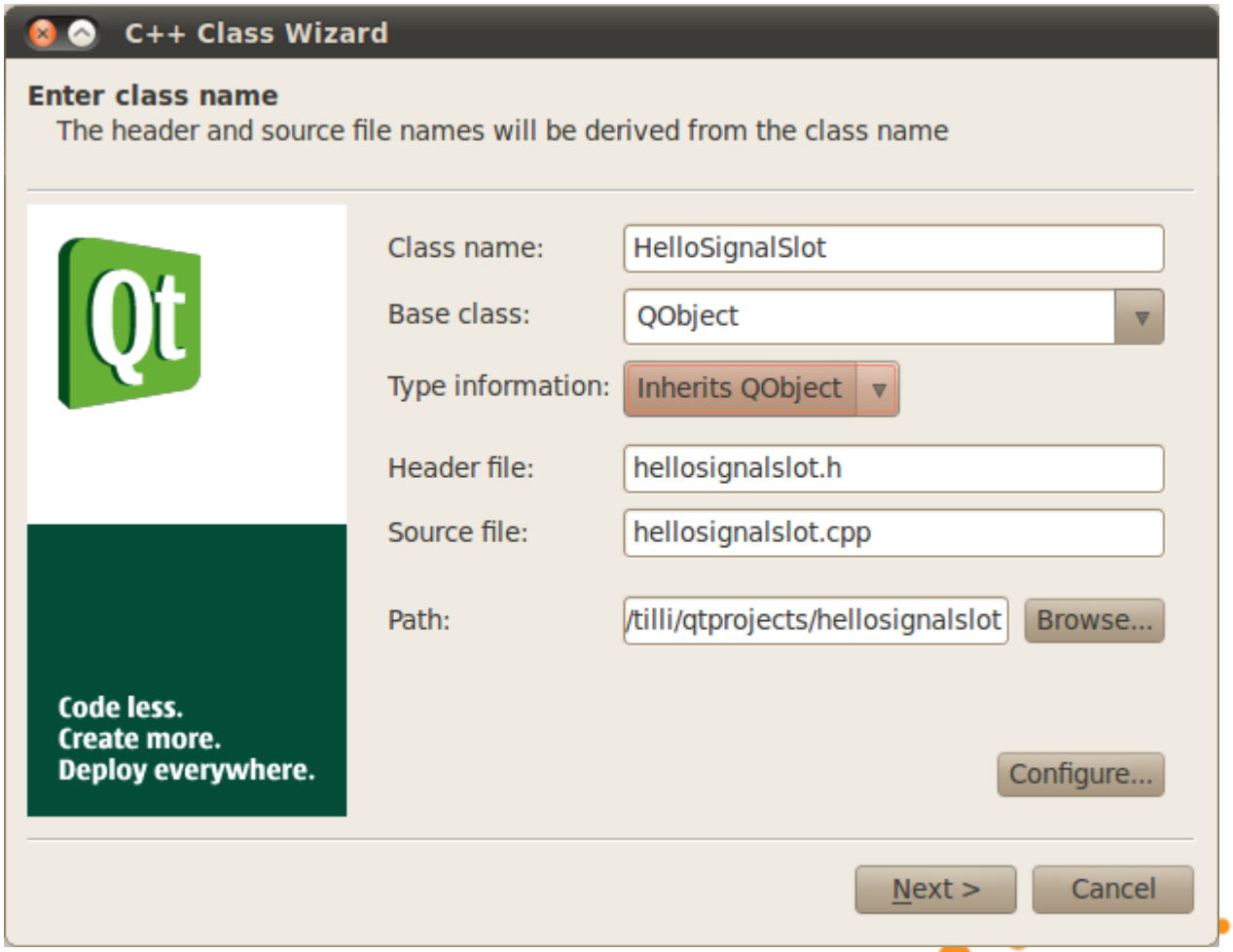
- Any Qt object may be a target for events
  - void QCoreApplication::postEvent (QObject * receiver, QEvent * event)
    - Adds an event to a queue
  - bool QObject::event (QEvent * e)
    - Processes an event
  - Usually an event is propagated out as signal emission
    - For example, mouse click on button generates an *activated* signal

Trying it out

<symbio>

- Create a new console application with QtCreator
- Add a new QObject-based class into it
  - See next slide

Trying it out



Trying it out

hellosignalslot.h — HelloSignalSlot

```
1  #ifndef HELLOSIGNALSLOT_H
2  #define HELLOSIGNALSLOT_H
3
4  #include <QObject>
5
6  class HelloSignalSlot : public QObject
7  {
8  Q_OBJECT
9  public:
10     explicit HelloSignalSlot(QObject *parent = 0);
11
12 signals:
13     void helloSignal();
14
15 public slots:
16     void hello();
17
18 };
19
20 #endif // HELLOSIGNALSLOT_H
```

hellosignalslot.cpp — HelloSignalSlot::h

```
1  #include "hellosignalslot.h"
2
3  HelloSignalSlot::HelloSignalSlot(QObject *parent) :
4      QObject(parent)
5  {
6  }
7
8  void HelloSignalSlot::hello()
9  {
10     qDebug("Hello signal!!!");
11     emit helloSignal();
12 }
```

Trying it out

- Copy the main function from *hellosignalslot* example into your main function
- Build & run

```
int main(int argc, char *argv[])
{
    qDebug("Start!!!");

    // Main loop of console application
    QCoreApplication a(argc, argv);

    // Creates a timer and hello object with the QCoreApplication as parent
    QTimer *timer = new QTimer(&a);
    HelloSignalSlot *hello = new HelloSignalSlot(&a);

    // Connects signals to slots
    QObject::connect(timer, SIGNAL(timeout()), hello, SLOT(hello()));
    QObject::connect(hello, SIGNAL(helloSignal()), &a, SLOT(quit()));

    // Starts timer and runs main loop
    timer->start(10000);
    int result = a.exec();

    qDebug("Quit!!!");
    return result;
}
```

What was done?

- The program event loop was created

```
// Main loop of console application
QCoreApplication a(argc, argv);
```

- Note that *new* operator was not used
  - Object was allocated on *stack*
  - Stack variables will be automatically deleted at the end of the scope they belong to
    - In this case the scope is the *main* function
    - Thus, *delete* is not needed

What was done?

- Two objects were created
  - QCoreApplication object was assigned as the parent object
  - Thus, parent will delete them when it is deleted

```
// Creates a timer and hello object with the QCoreApplication as parent
QTimer *timer = new QTimer(&a);
HelloSignalSlot *hello = new HelloSignalSlot(&a);
```

- Note: timer and hello could also be allocated from stack
  - But parent must not be used in that case (why?)

What was done?

- Objects were connected together

```
// Connects signals to slots
QObject::connect(timer, SIGNAL(timeout()), hello, SLOT(hello()));
QObject::connect(hello, SIGNAL(helloSignal()), &a, SLOT(quit()));
```

- Note that timer and hello object don’t know anything about each other

What was done?

- Timer and event loop were started

```
// Starts timer and runs main loop
timer->start(10000);
int result = a.exec();
```

```
void HelloSignalSlot::hello()
{
    qDebug("Hello signal!!!");
    emit helloSignal();
}
```

  - When event loop is active
    - The timer gets an event from the system and emits *timeout* signal after 10 seconds
    - *timeout* signal is connected to the *hello* slot
    - Hello slot prints something and emits *helloSignal*
    - *helloSignal* is connected to event loop *quit* slot
    - *Quit* slot stops the event loop and thus *exec* function returns and program quits

Short exercise

- Open the *hellosignalslot* example that presented in previous slides
- Change it so that it prints ”Hello” and ”World” with 5-second interval and quits after the second print

More core features

Shared data objects

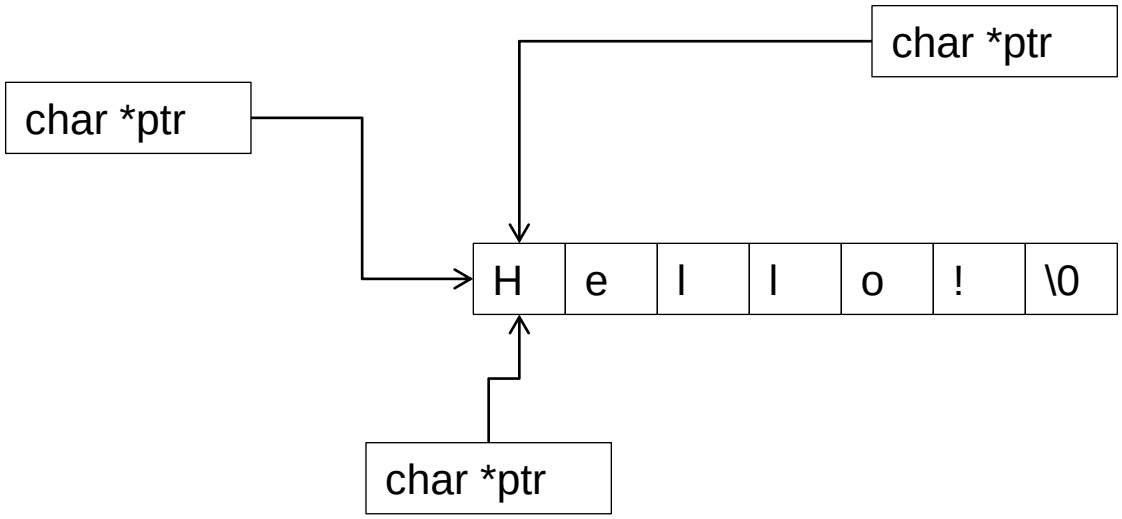
Shared data objects

- For some objects there's no use for features of the object model
  - Performance reasons
  - Strings, images, collections
- Object data is usually *implicitly shared*
  - Copy-on-write semantics
  - Easier to use that just pointers

Implicit sharing

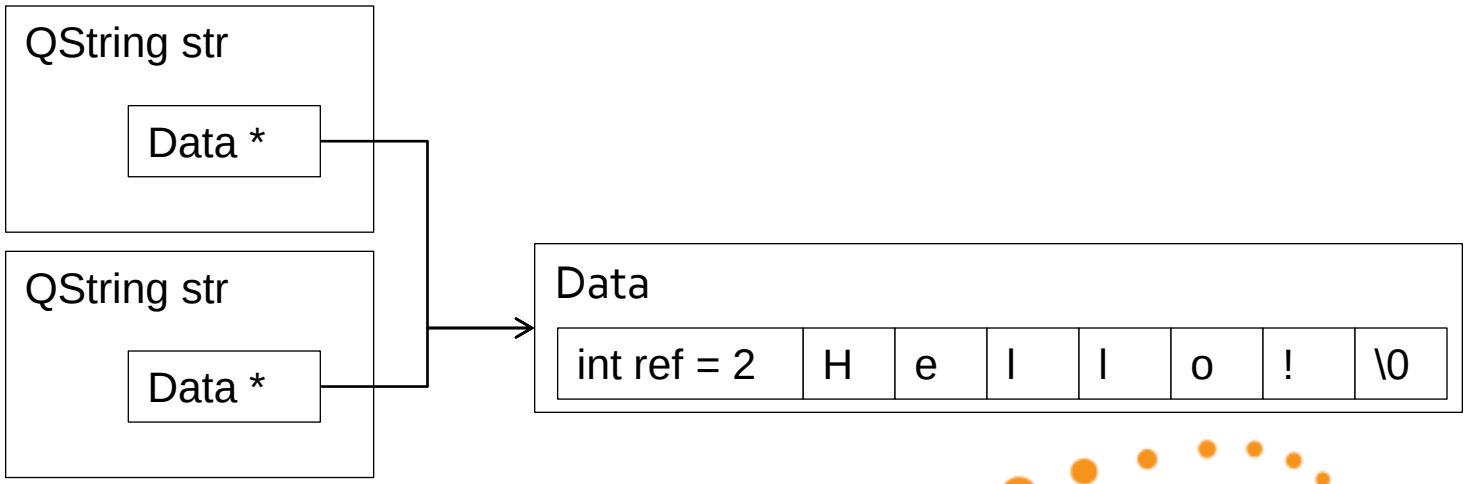
<symbio>

- In normal C++ an object is allocated and a pointer to it is passed around
  - Care must be taken that object is not deleted while it's still being pointed to



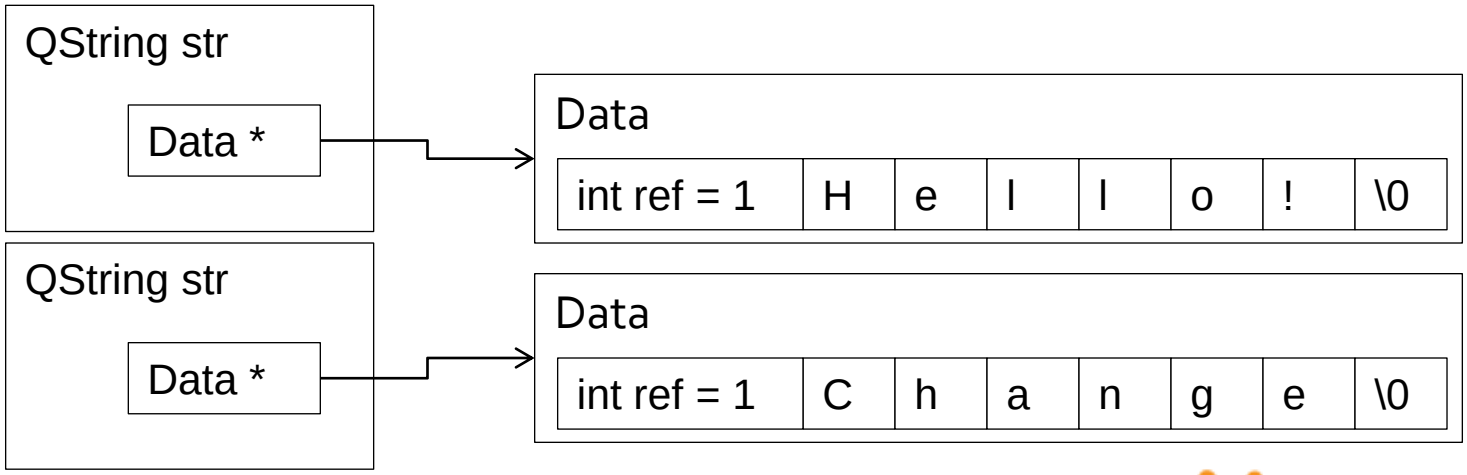
Implicit sharing

- In implicit sharing, a *reference counter* is associated with the data
  - Data pointer is wrapped into a *container* object, which takes care of deleting the data when reference count reaches 0



Implicit sharing

- Implicitly shared objects can be treated as simple values
  - Only the pointer is passed around



Terminology

- Copy-on-write
  - Make a shallow copy until something is changed
- Shallow copy
  - Copy just the pointer, not actual data
- Deep copy
  - Create a copy of the data

Copy on write


```
#include <QtCore/QCoreApplication>

class CopyOnWriteIllustration
{
public:
    void setInteger(qint32 i) { value = i; }
    qint32 integer() const { return value; }
    void setString(const QString &s) { str = s; }
    QString string() const { return str; }

private:
    qint32 value;
    QString str;
};

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

    CopyOnWriteIllustration cowi;
    cowi.setInteger(100);
    cowi.setString("Test");

    qint32 changedInt = cowi.integer();
    QString changedStr = cowi.string();
    changedInt += 5;
    changedStr.append(" Changed");

    qDebug("Object: %d,  Changed: %d", cowi.integer(), changedInt);
    qDebug("Object: %s,  Changed: %s", cowi.string().toLatin1().data(),
                                       changedStr.toLatin1().data());

    return a.exec();
}
```

Strings

- Two types of string
  - UNICODE strings (QString)
  - Byte arrays (QByteArray)
- In general, QString should be used
  - UNICODE, so can be localized to anything
  - Conversion between types is easy, but might have unexpected performance issues

Strings and implicit sharing

- Strings are implicitly shared, so in general, should be treated as a value
  - Returned from functions like value
  - Stored into objects as values
  - Function parameters should use constant reference, not value
    - *const QString &*

```
QString HelloWorld::hello() const
{
    QString str = "Hello World";
    return str;
}
```

```
void HelloWorld::setDescription(const QString &desc)
{
    description = desc;
}
```

String operations

- In Qt, a string can be changed
  - Thus, differs from java immutable strings
  - Modifying a string in-place is more efficient (especially with *reserve()* function)
    - However, some care must be taken to avoid changes in unexpected places

```
void HelloWorld::changeString(QString &str)
{
    str += "_changed";
}

QString HelloWorld::createNewString(const QString &str)
{
    return str + "_changed";
}
```

String operations

- QString supports various operators
  - ’+’, ’+=’, ’>’, ’<’, ’<=’, ’>=’, ’==’, ’!=’
  - Also work with literals
  - Character access with []

```
QString str1 = "hello";
QString str2 = "world";

if (str1 == "hello" && str2 != "wolrd") {
    qDebug("True");
}
```

Console output

- Qt has *qPrintable* function, which should be used when printing strings with *qDebug*

```
QString str = "Hello World";
qDebug("%s", qPrintable(str));
```

```
QByteArray str = "Hello World";
qDebug("%s", str.constData());
```

```
const char *str = "Hello World";
qDebug("%s", str);
```

Generic containers

- List containers
  - QList, QLinkedList, QVector, QStack, QQueue
  - Usually QList is best for ordinary tasks
  - QStringList for strings
- Associative containers
  - QSet, QMap, QHash, QMultiMap, QMultiHash
  - QMap for sorted, QHash for unsorted items

C++ templates

- Containers are based on C++ templates
  - Type safety -> helps prevent errors
  - Type of object is specified within angle brackes
  - Only objects of specific type can be used
- Some examples:
  - QList<QPicture>
    - List of pictures
  - QHash<QString,QObject>
    - Name-to-object dictionary

List containers

<symbio>

- Lists are index-based, starting from 0
  - Fast access if index is known, slow to search
- Adding and removing items
  - append, insert, ’+=’, ’<<’
- Accessing items
  - at, ’[]’

```
QList<QString> strings;
strings.append("1");
strings << "2" << "3" << "4";
strings.insert(2, "2");
strings.removeOne("2");

qDebug("%s", qPrintable(strings[2])); // 3
```

Foreach statement

- Can be used to iterate over lists
- Takes a shallow copy of the container
  - If original container is modified while in loop, the one used in the loop remains unchanged

```
QString hello = "Hello World !!!";
QStringList strList = hello.split(" ");
foreach (QString str, strList) {
    qDebug("Part: %s", qPrintable(str));
}
```

Associative containers

<symbio>

- Associative containers are used to map keys to values
  - In QSet, key and value are the same
    - QSet<String>
  - Other containers have separate keys and values
    - QHash<QString,QString>
  - Normal versions have one-to-one mapping, *multi*-versions accept multiple values for single key
    - QMultiMap<QString, QObject *>

Creating new objects

<symbio>

- Two classes for reference counting
  - *QSharedData* is inherited by data container
  - *QSharedDataPointer* is used from public API

```
class Shared
{
public:
    Shared(const QString &name);
    ~Shared();

    QString name() const { return d->name; }
    void setName(const QString &name) { d->name = name; }

private:
    QSharedDataPointer<SharedPrivate> d;
};
```

```
class SharedPrivate : public QSharedData
{
public:
    SharedPrivate();
    SharedPrivate(const SharedPrivate &other);
    ~SharedPrivate();

    QString name;
};
```

Copy constructor

- Private part of shared data object requires a copy constructor, so *deep copy* is possible in case data is changed
  - Compiler can usually create one for you

```
SharedPrivate::SharedPrivate()
{
    qDebug("Private - Alloc");
}

SharedPrivate::~SharedPrivate()
{
    qDebug("Private - Delete");
}

SharedPrivate::SharedPrivate(const SharedPrivate &other)
    : QSharedData(other), name(other.name)
{
    qDebug("Private - Copy");
}
```

Shared data reminders

<symbio>

- Keep things simple
  - Do not use *new* to allocate shared data objects
  - Instead, think of them as values (like *int*)

Programming exercise

<symbio>

- Create a new console project *shareddata*
  - Add a shared data class *Shared* and corresponding private *SharedPrivate*
    - Add *QString name* member to *SharedPrivate*
    - Add *name* and *setName* functions to *Shared*
  - Add some *qDebug* statements to constructors and destructors of *Shared* and *SharedPrivate* to see what gets allocated and deleted

Programming exercise

- In main.cpp
  - Add function, which creates 5 *Shared* objects, adds them to a list and returns the list
  - Call it from *main()*
  - Loop over the contents of the list with *foreach*
    - Print object name from the loop
  - Replace *return a.exec()* with *return 0;*

Exercise notes

- Should allocate 5 *SharedPrivate* objects
- Number of allocated *Shared* objects is something different

Opaque pointers

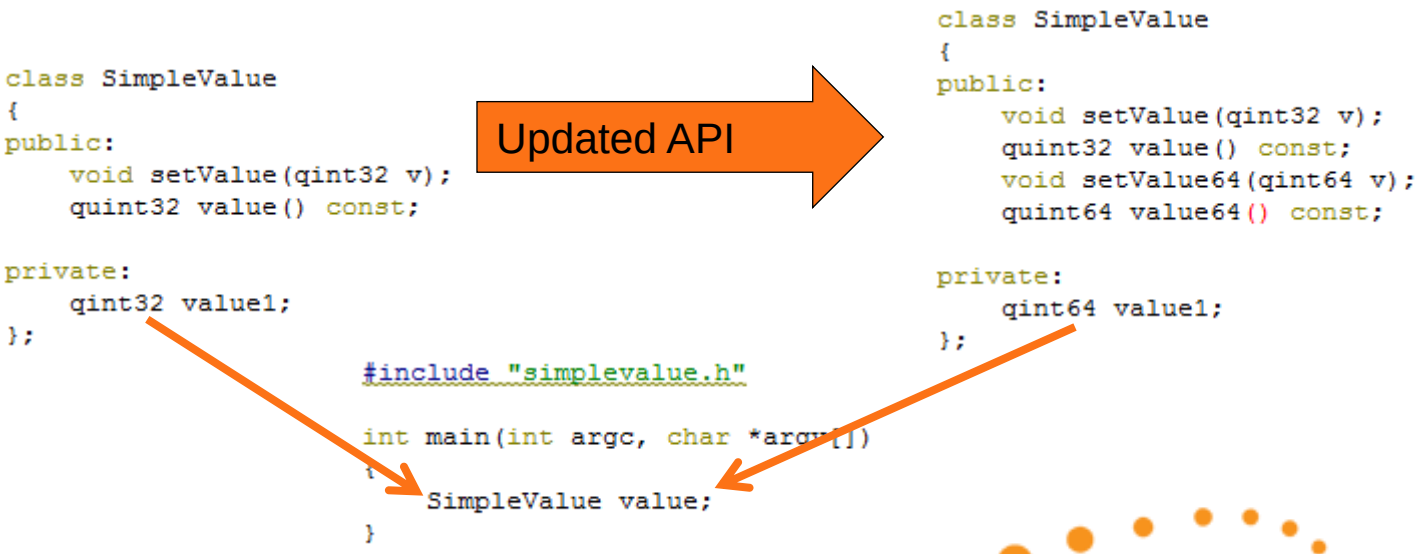
Planning for the future

Opaque pointers

- Private Implementation
  - Separates implementation from public API
    - Two classes that are linked together
  - Also called ”d” and ”q” pointers in Qt (or pimpl)
- Why?
  - Code maintenance
    - Hide dirty details from user of the API
  - Problem with C++ memory allocation
  - Different backend for different environments

Memory allocation

- The size of the allocated memory block is determined at *compile time*
  - If used from other library, size change results in *binary break*



Changing implementation

<symbio>

- Sometimes it’s necessary to implement features in platform-specific ways
  - Pre-processor macros within a source
  - Implementations for different platforms in different sources
    - Choice can be made in *.pro* file
- Whatever the case, the library public API should be the same in all platforms

Opaque pointers


```
class PIMPLSHARED_EXPORT Pimpl : public QObject
{
    Q_OBJECT

public:
    Pimpl(QObject *parent = 0);
    ~Pimpl();

    void setString(const QString &string);
    QString string() const;

signals:
    void stringChanged();

private:
    friend class PimplPrivate;
    PimplPrivate *privatePtr;
};
```

```
class PimplPrivate
{
    PimplPrivate();

    void setString(const QString &s);
    QString string() const;

private:
    friend class Pimpl;
    Pimpl *publicPtr;
    QString str;
};
```

- Only a pointer member in public API, so object size will not change when features are added

Object ownership

- Public object owns the private object
  - Allocated in constructor
  - Deleted in destructor

```
Pimpl::Pimpl(QObject *parent) : QObject(parent)
{
    privatePtr = new PimplPrivate();
    privatePtr->publicPtr = this;
}

Pimpl::~Pimpl()
{
    delete privatePtr;
}
```

Implementation options

- Totally separated
  - Public API delegates all function calls to the private counterpart
  - Private implementation emits public API signals
  - 2 sources, 2 headers – tedious to implement

```
void Pimpl::setString(const QString &string)
{
    privatePtr->setString(string);
}

QString Pimpl::string() const
{
    return privatePtr->string();
}
```

```
void PimplPrivate::setString(const QString &s)
{
    str = s;
    emit publicPtr->stringChanged();
}

QString PimplPrivate::string() const
{
    return str;
}
```

Implementation options

- Private class within public API source
  - Just data in private class
    - No need for two-way linking between private and public
  - Won't work if needed from other private sources
    - But you can switch to fully separated later if needed

```
class PimplPrivate
{
public:
    QString str;
};

Pimpl::Pimpl(QObject *parent) : QObject(parent)
{
    privatePtr = new PimplPrivate();
}

Pimpl::~Pimpl()
{
    delete privatePtr;
}

void Pimpl::setString(const QString &string)
{
    privatePtr->str = string;
    emit stringChanged();
}

QString Pimpl::string() const
{
    return privatePtr->str;
}
```

Programming exercise

Music library object model

Programming exercise

- Music library
  - Project was created on day 1
  - Contents of a music library:
    - MusicLibrary, Artist, Record, Song
  - Relations:
    - MusicLibrary has a list of artists
    - Artist has a list of records
    - Record has a list of songs

Programming exercise

- Object properties (set / get functions)
  - Artist:
    - Home page (QUrl)
  - Record:
    - Release date (QDateTime)
    - Cover image (QString, represents file name)
  - Song:
    - Number (int)
    - Song itself (QString, represents file name)

Programming exercise

<symbio>

- Common properties
  - All objects have a name (QString)
  - All objects must inherit QObject
  - Objects must emit a signal when a property changes
  - Optional
    - Add base class for common functionality

Programming exercise

<symbio>

- Optional
  - Separate public API and private implementation
  - Add functions to get:
    - All artists, records and songs of music library
    - All records of artist
    - All songs of record

SERIOUS ABOUT SOFTWARE