Qt Quick – Qt C++ programming basics

Timo Strömmer, Jan 7, 2011

Contents

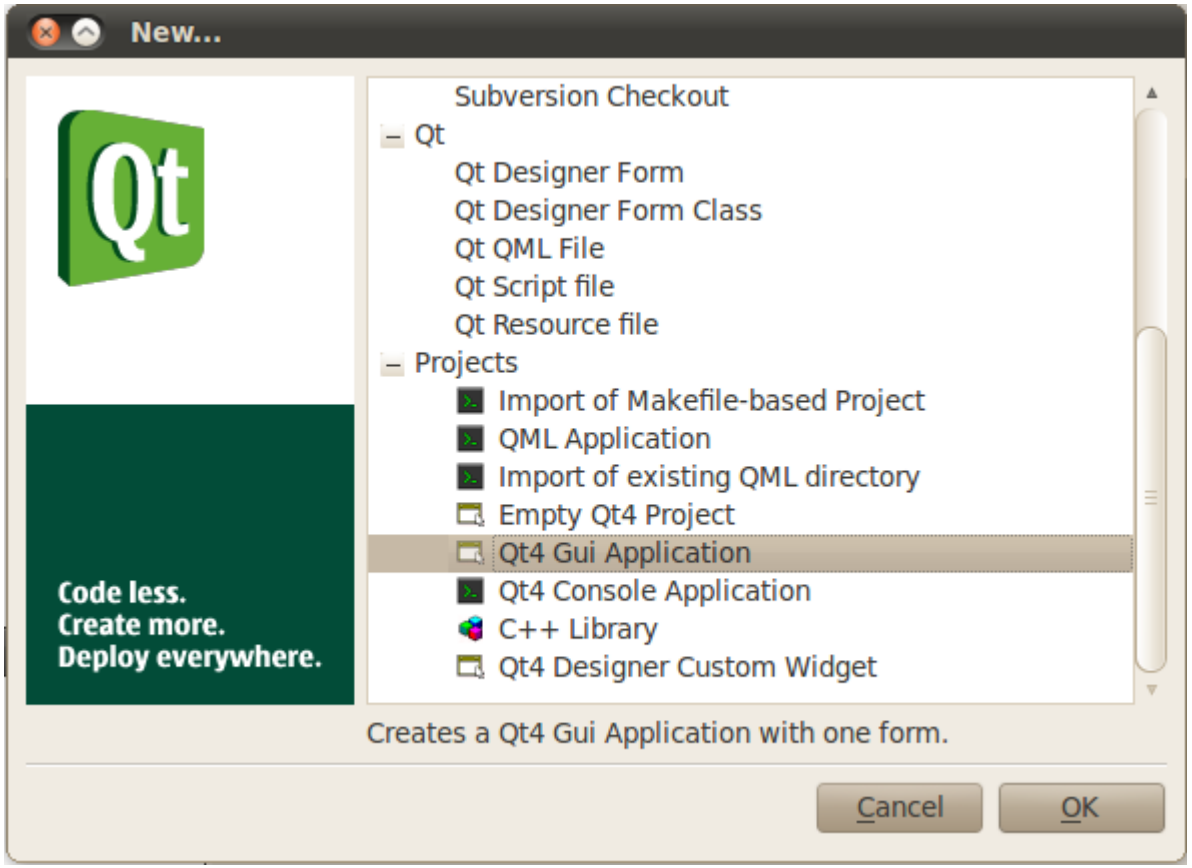
Contents

<symbio>

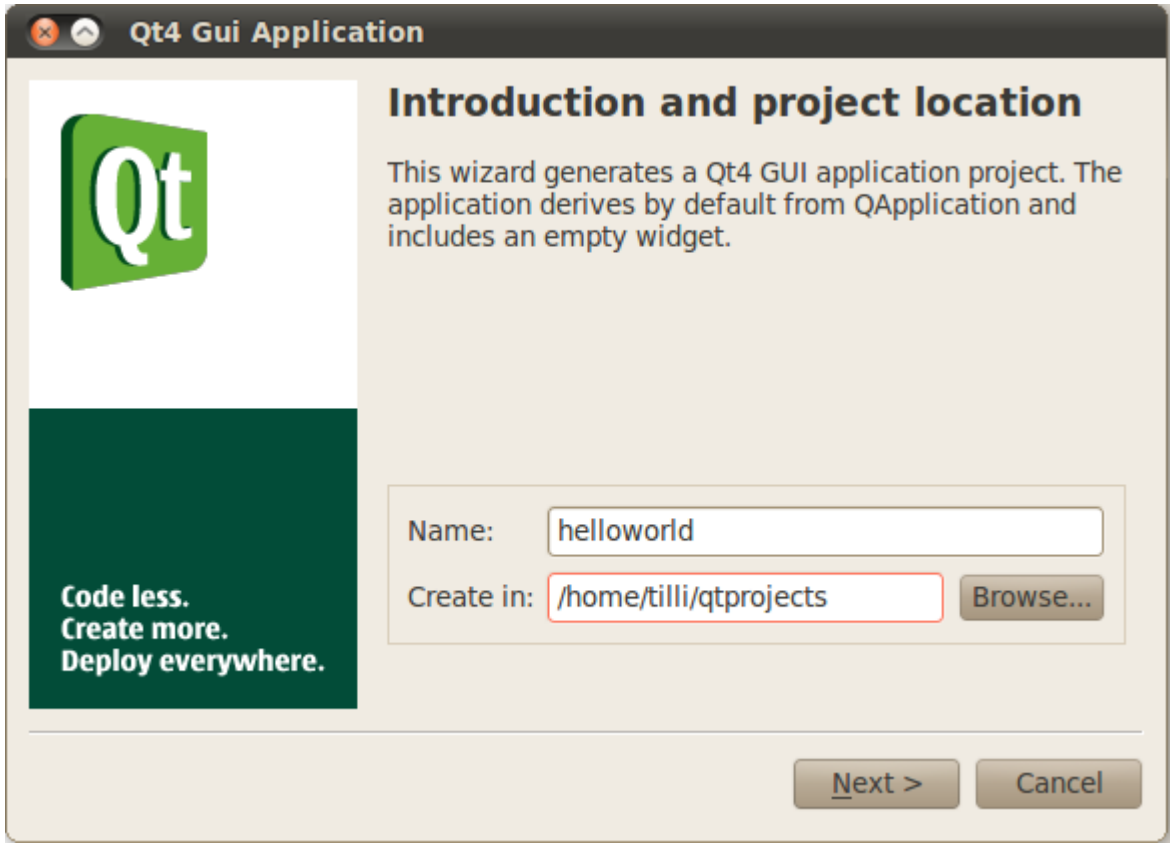
Creating a C++ project

QT C++ QUICK START

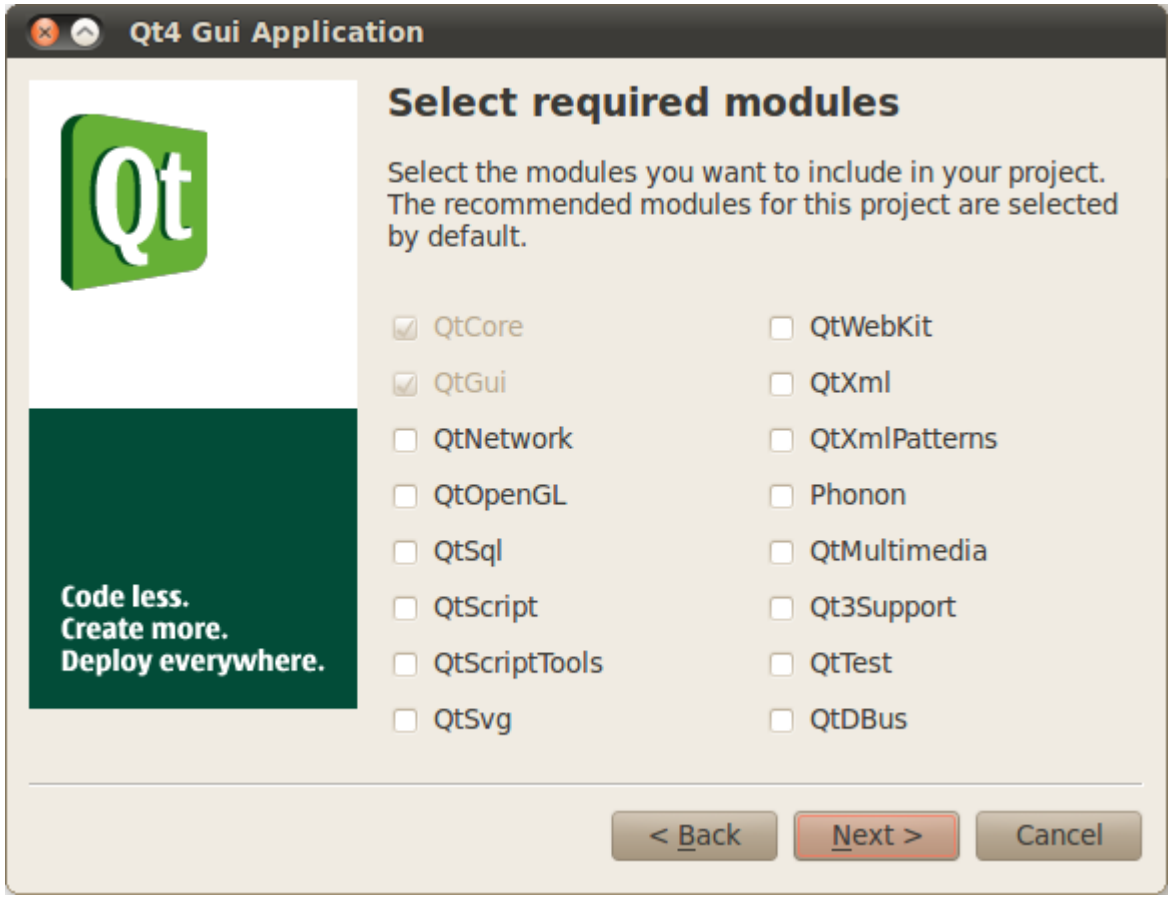
Quick start


New...
Subversion Checkout
Qt
Qt Designer Form
Qt Designer Form Class
Qt QML File
Qt Script file
Qt Resource file
Projects
Import of Makefile-based Project
QML Application
Import of existing QML directory
Empty Qt4 Project
Qt4 Gui Application
Qt4 Console Application
C++ Library
Qt4 Designer Custom Widget
Code less.
Create more.
Deploy everywhere.
Creates a Qt4 Gui Application with one form.
Cancel
OK

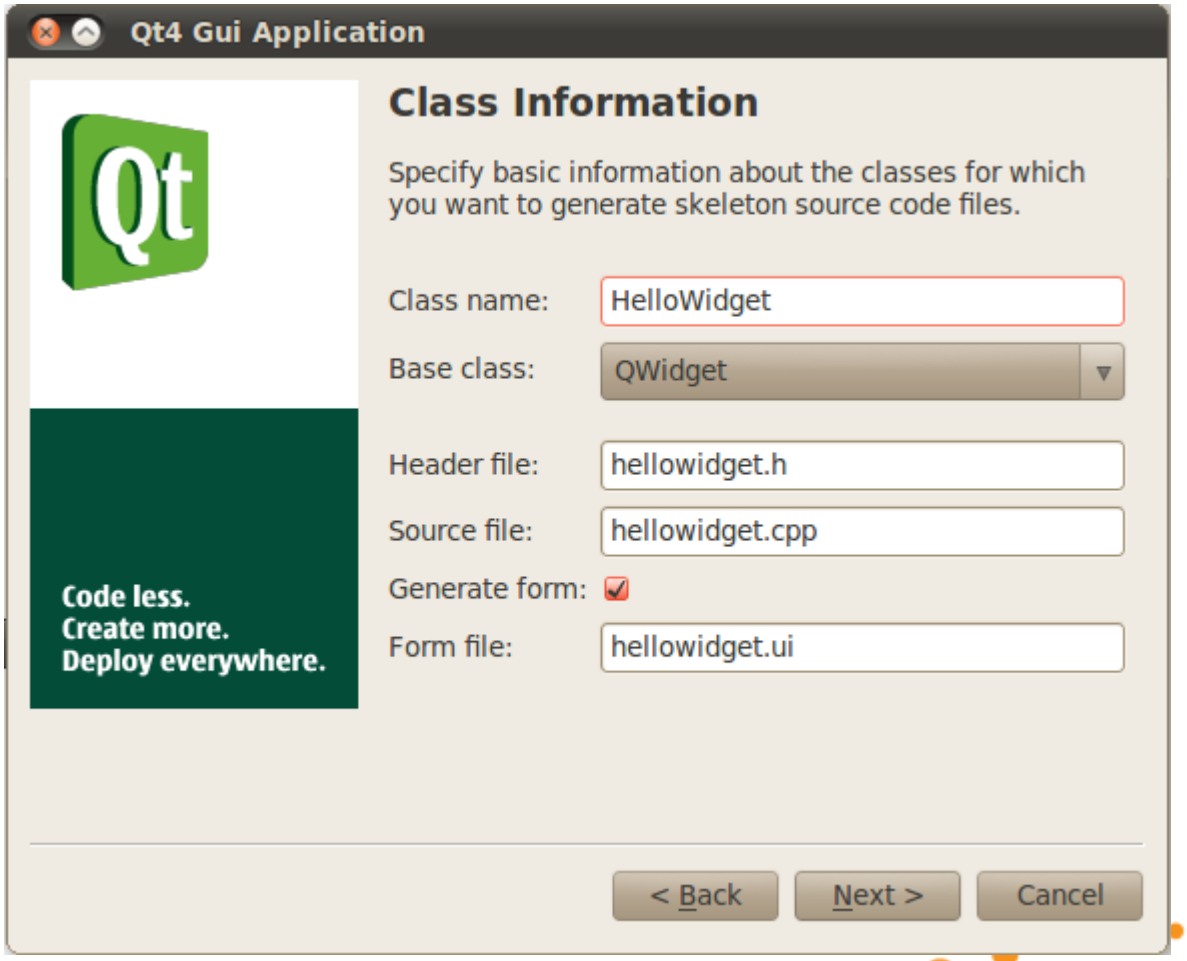
Quick start


Qt4 Gui Application
Introduction and project location
This wizard generates a Qt4 GUI application project. The application derives by default from QApplication and includes an empty widget.
Code less.
Create more.
Deploy everywhere.
Name:
helloworld
Create in:
/home/tilli/qtprojects
Browse...
Next >
Cancel

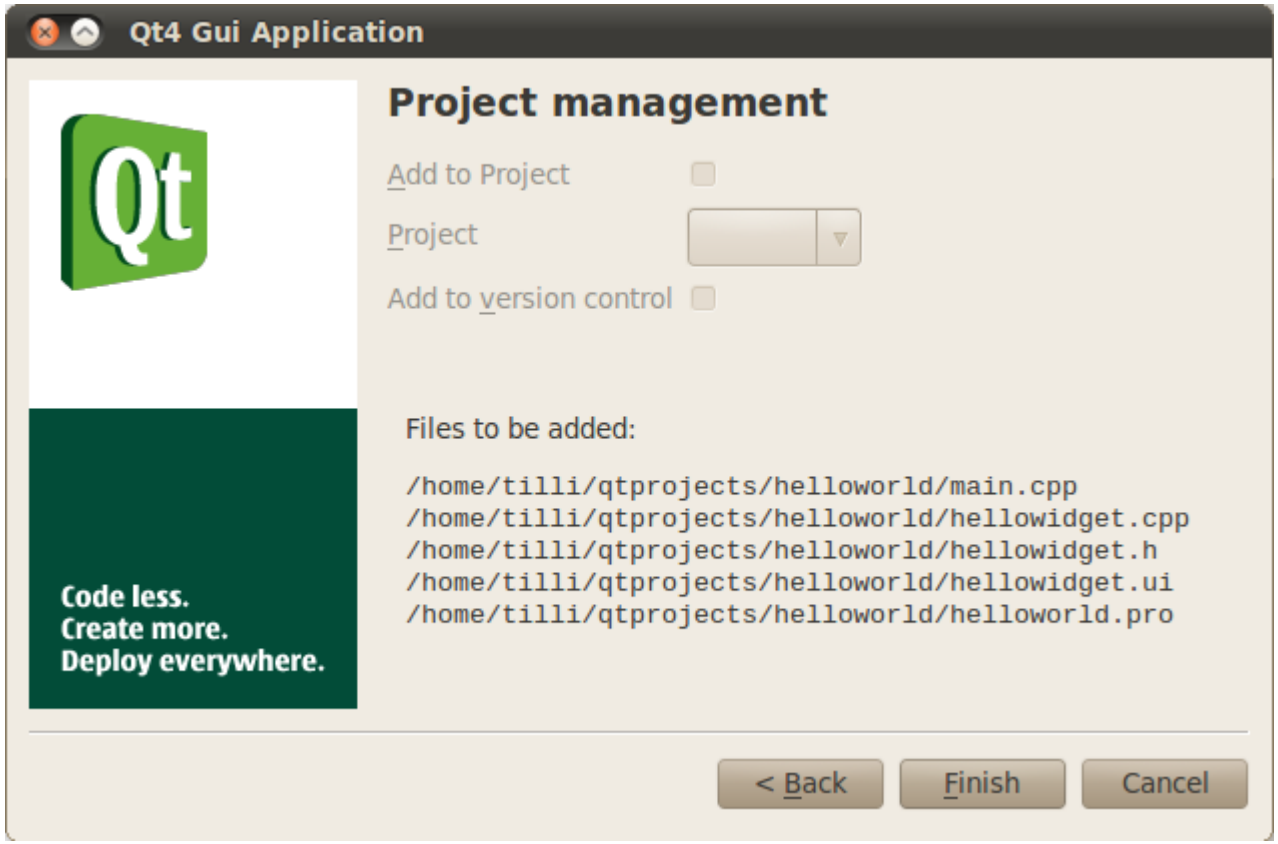
Quick start


Qt4 Gui Application
Select required modules
Select the modules you want to include in your project.
The recommended modules for this project are selected by default.
QtCore
QtGui
QtNetwork
QtOpenGL
QtSql
QtScript
QtScriptTools
QtSvg
QtWebKit
QtXml
QtXmlPatterns
Phonon
QtMultimedia
Qt3Support
QtTest
QtDBus
Code less.
Create more.
Deploy everywhere.
< Back
Next >
Cancel

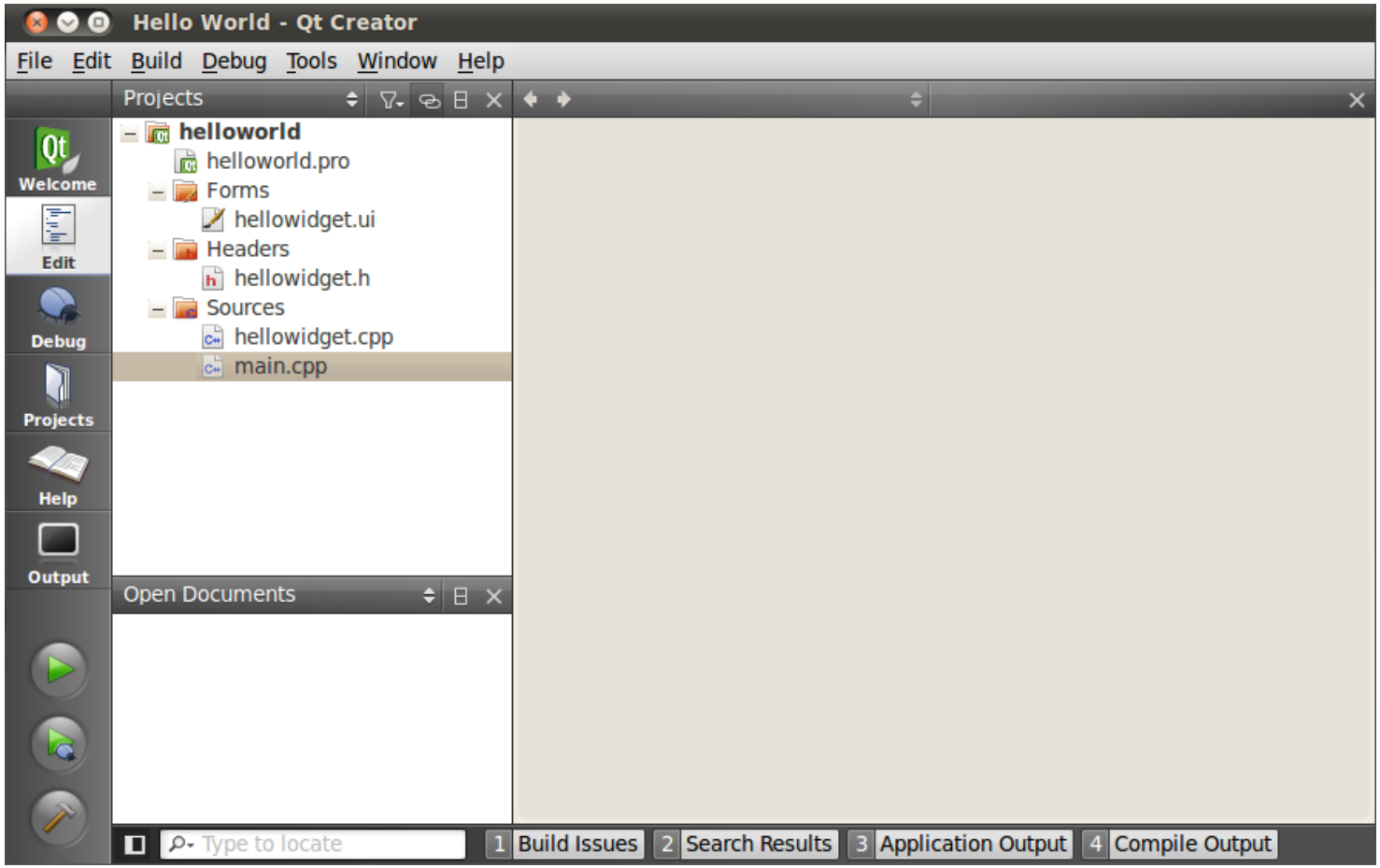
Quick start


Qt4 Gui Application
Class Information
Specify basic information about the classes for which you want to generate skeleton source code files.
Class name:
HelloWidget
Base class:
QWidget
Header file:
hellowidget.h
Source file:
hellowidget.cpp
Generate form:
Form file:
hellowidget.ui
Code less.
Create more.
Deploy everywhere.
< Back
Next >
Cancel

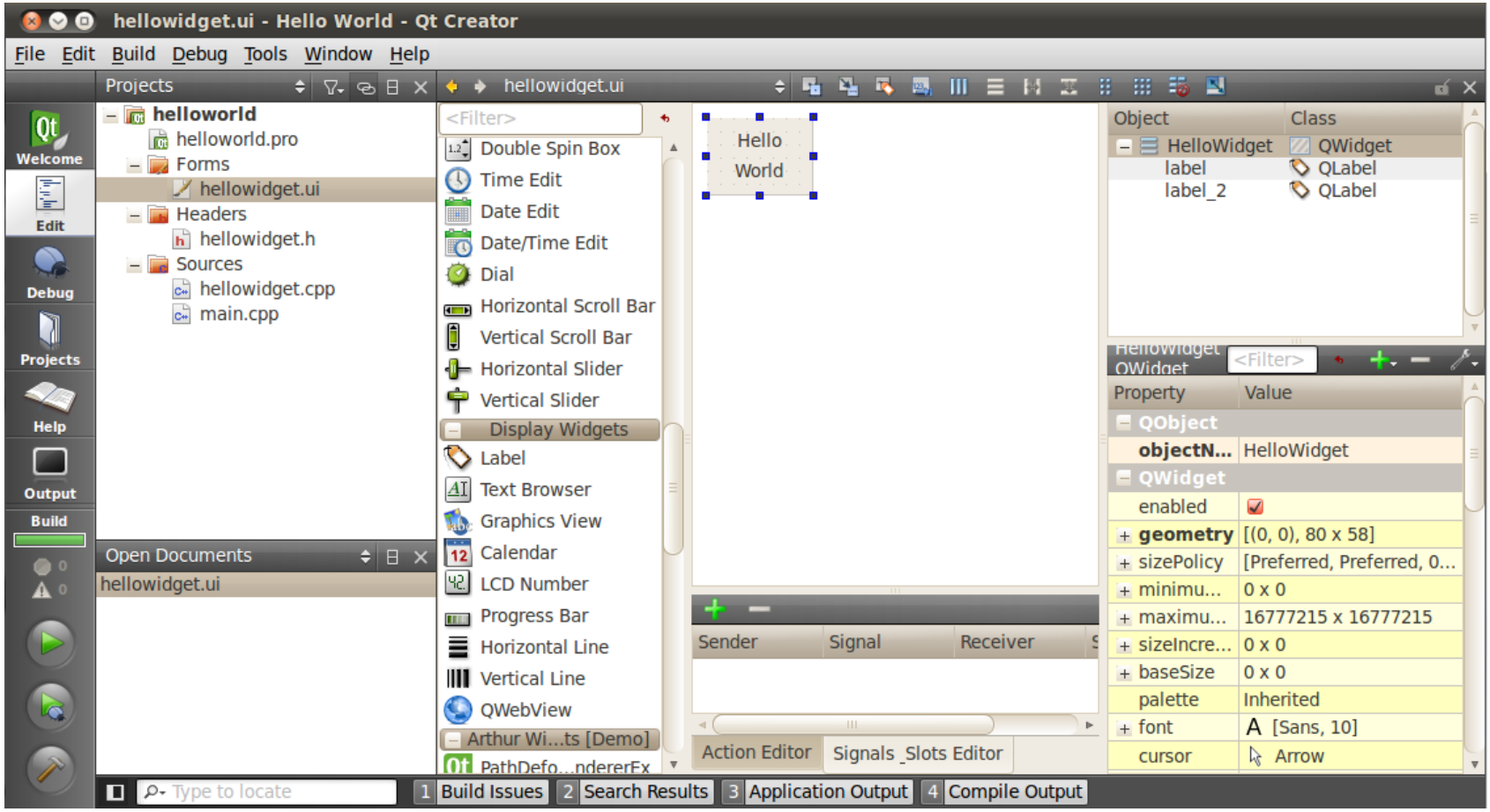
Quick start


Qt4 Gui Application
Project management
Add to Project
Project
Add to version control
Files to be added:
/home/tilli/qtprojects/helloworld/main.cpp
/home/tilli/qtprojects/helloworld/hellowidget.cpp
/home/tilli/qtprojects/helloworld/hellowidget.h
/home/tilli/qtprojects/helloworld/hellowidget.ui
/home/tilli/qtprojects/helloworld/helloworld.pro
Code less.
Create more.
Deploy everywhere.
< Back
Finish
Cancel

Quick start


Hello World - Qt Creator
File Edit Build Debug Tools Window Help
Projects
helloworld
helloworld.pro
Forms
hellowidget.ui
Headers
hellowidget.h
Sources
hellowidget.cpp
main.cpp
Welcome
Edit
Debug
Projects
Help
Output
Open Documents
Type to locate
1 Build Issues
2 Search Results
3 Application Output
4 Compile Output

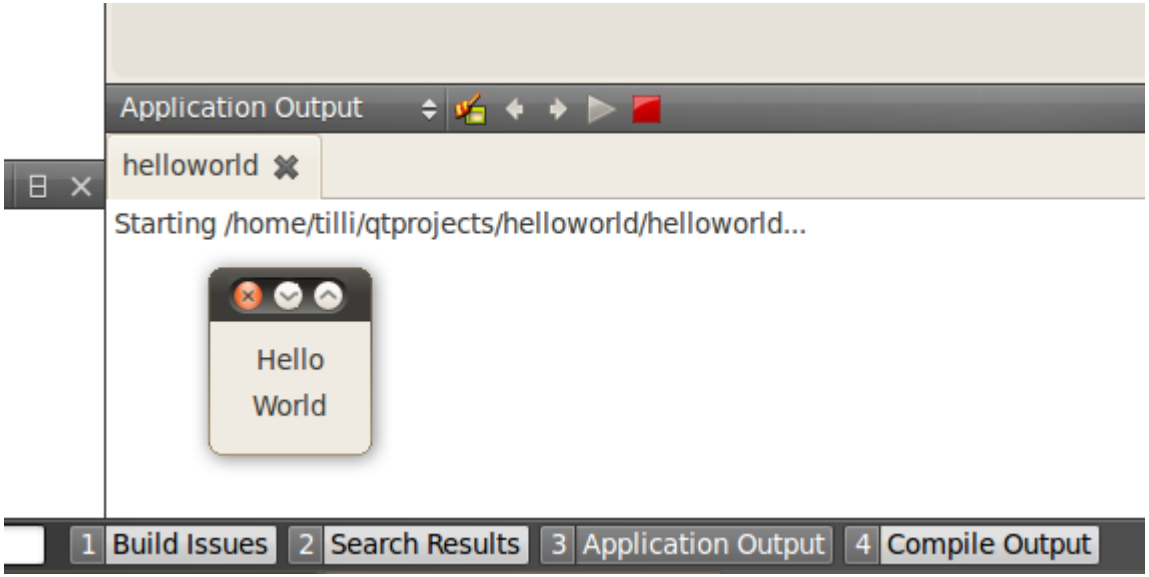
Quick start


hellowidget.ui - Hello World - Qt Creator
File Edit Build Debug Tools Window Help
Projects
helloworld
helloworld.pro
Forms
hellowidget.ui
Headers
hellowidget.h
Sources
hellowidget.cpp
main.cpp
Open Documents
hellowidget.ui
Welcome
Edit
Debug
Projects
Help
Output
Build
hellowidget.ui
Hello World
Double Spin Box
Time Edit
Date Edit
Date/Time Edit
Dial
Horizontal Scroll Bar
Vertical Scroll Bar
Horizontal Slider
Vertical Slider
Display Widgets
Label
Text Browser
Graphics View
Calendar
LCD Number
Progress Bar
Horizontal Line
Vertical Line
QWebView
Object
Class
HelloWidget
QWidget
label
QLabel
label_2
QLabel
Property
Value
QObject
objectN...
HelloWidget
QWidget
enabled
geometry
[(0, 0), 80 x 58]
sizePolicy
[Preferred, Preferred, 0...
minimu...
0 x 0
maximu...
16777215 x 16777215
sizeIncre...
0 x 0
baseSize
0 x 0
palette
Inherited
font
A [Sans, 10]
cursor
Arrow
Sender
Signal
Receiver
Action Editor
Signals _Slots Editor
Type to locate
1 Build Issues
2 Search Results
3 Application Output
4 Compile Output

Quick start

<symbio>

- Build with Ctrl+B, run with Ctrl+R



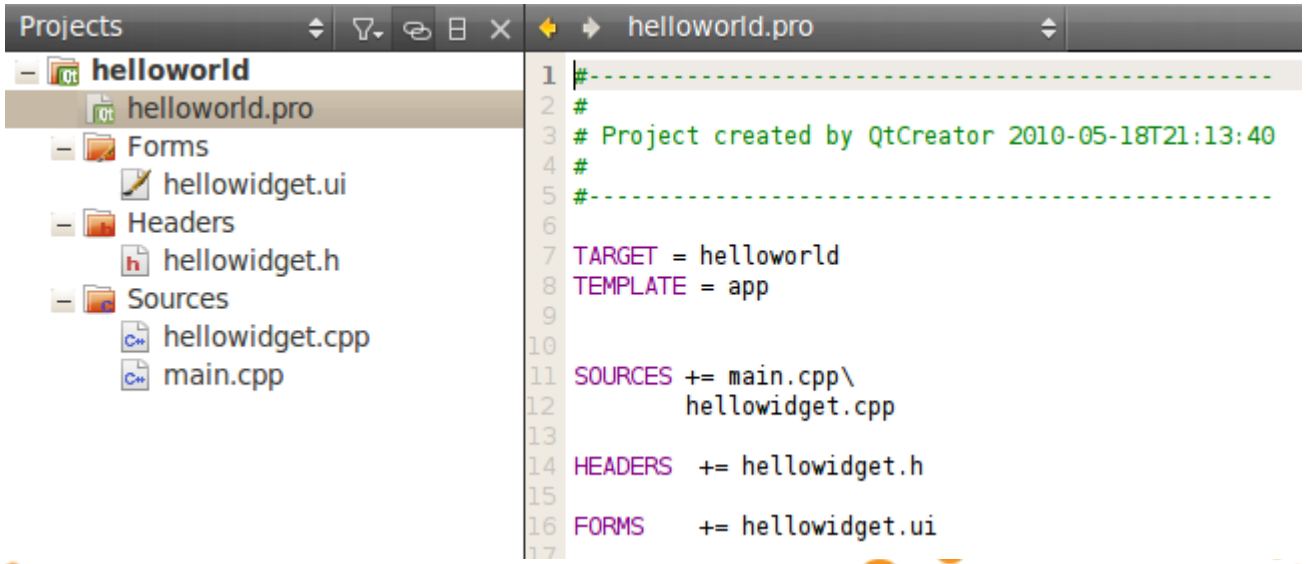
<symbio>

Project files overview

QT C++ PROJECTS

Qt project file

- A *.pro* file with same name as the directory it sits in
- Processed by *qmake* to generate platform-specific build files



Qt project basics

- Project name and type
  - TARGET, TEMPLATE
- Project files
  - SOURCES, HEADERS, FORMS
- Project configuration
  - CONFIG, QT

Project templates

<symbio>

- Basic TEMPLATE types: app, lib, subdirs
  - Executable files (console or GUI) are created with the *app* type
    - GUI is default, console needs *CONFIG += console*
  - Libraries (static and shared) are created with *lib* type
    - Shared default, static needs *CONFIG += staticlib*
  - Sub-directory template is used to structure large projects into hierarchies

Project name

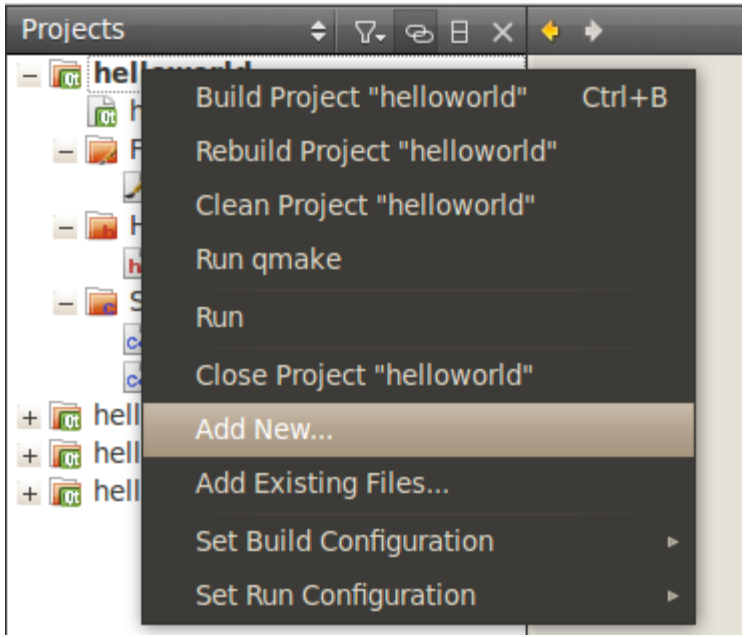
- Project TARGET specifies the output file name
  - *TARGET = helloworld*
- Affected by template and platform
  - Executable name (*name*, *name*.exe etc.)
  - Library name (*libname.so*, *name.dll* etc.)

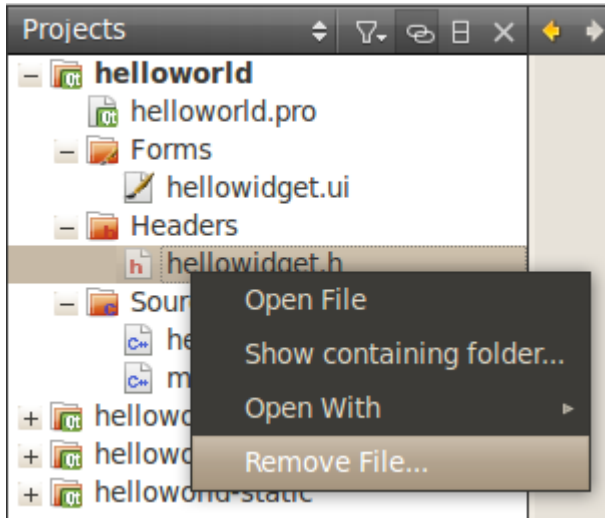
Project files

- SOURCES are obviously needed
- HEADERS also, as they are processed by *Qt meta-object compiler*
- UI form data (.ui files) are included with FORMS directive

Sources and headers

- QtCreator updates the directives in *.pro* file in most cases
  - Add and remove but no rename



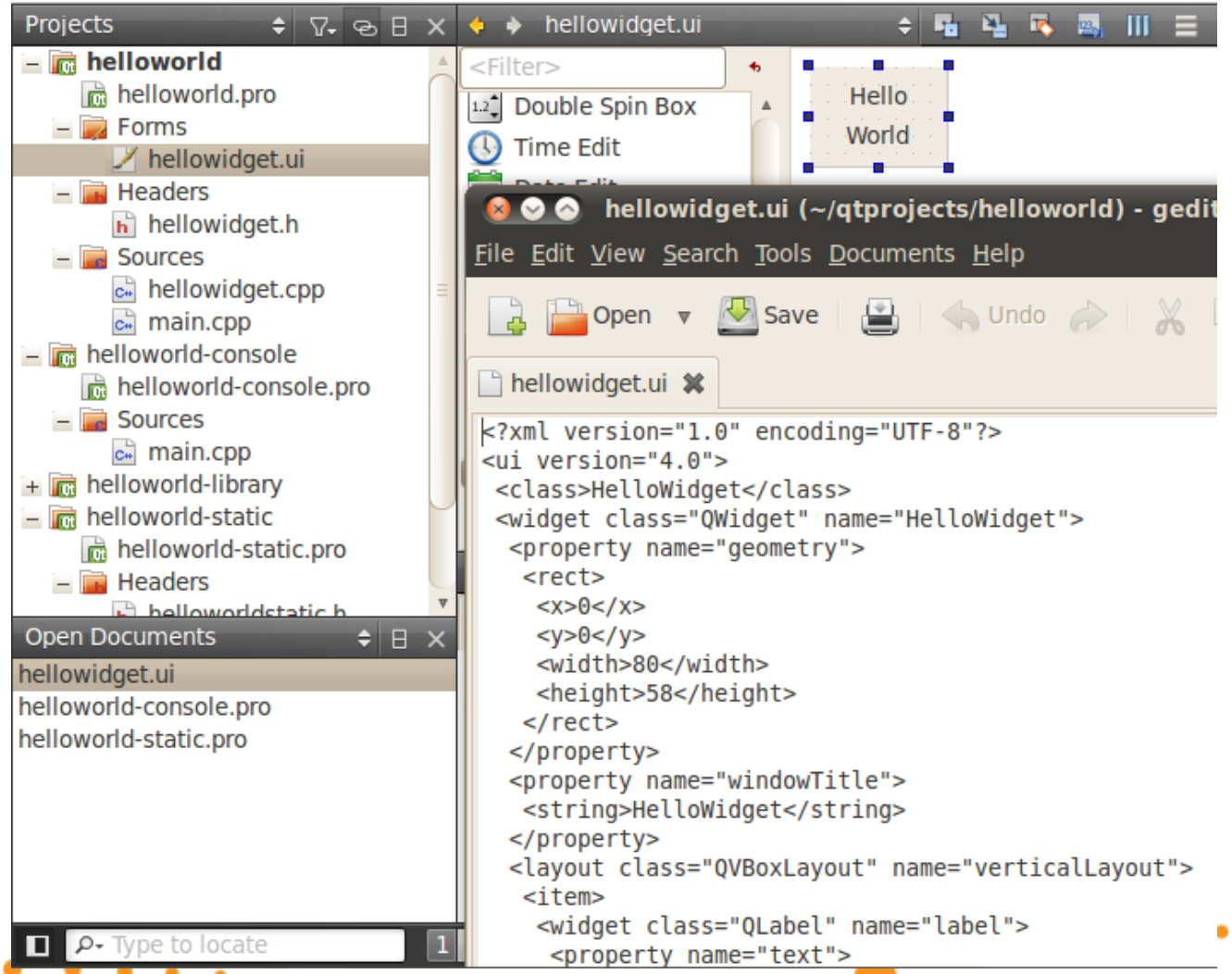


UI resources

<symbio>

- UI resource files are XML documents, which are processed by *uic* compiler during build
  - Generates C++ code from the resource and integrates it into project
- No need to edit manually, use QtCreator form editor instead

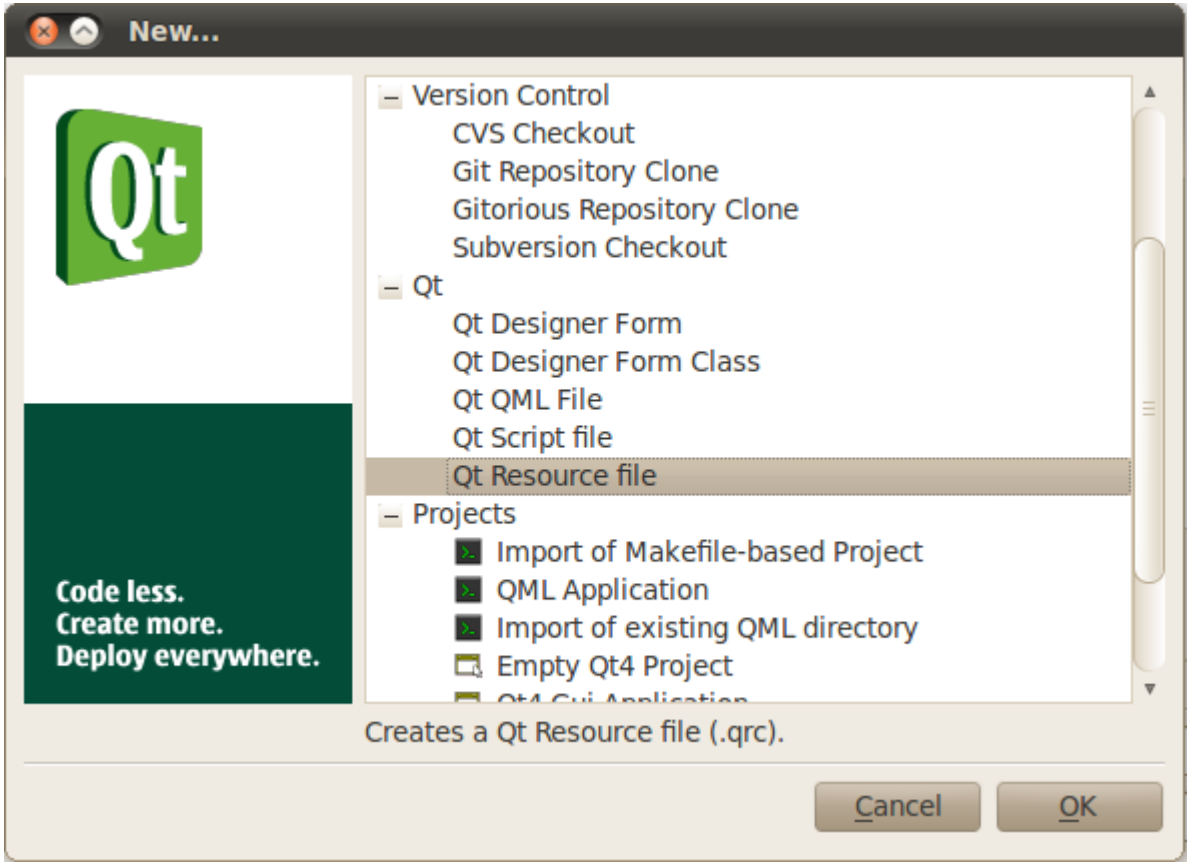
UI resources



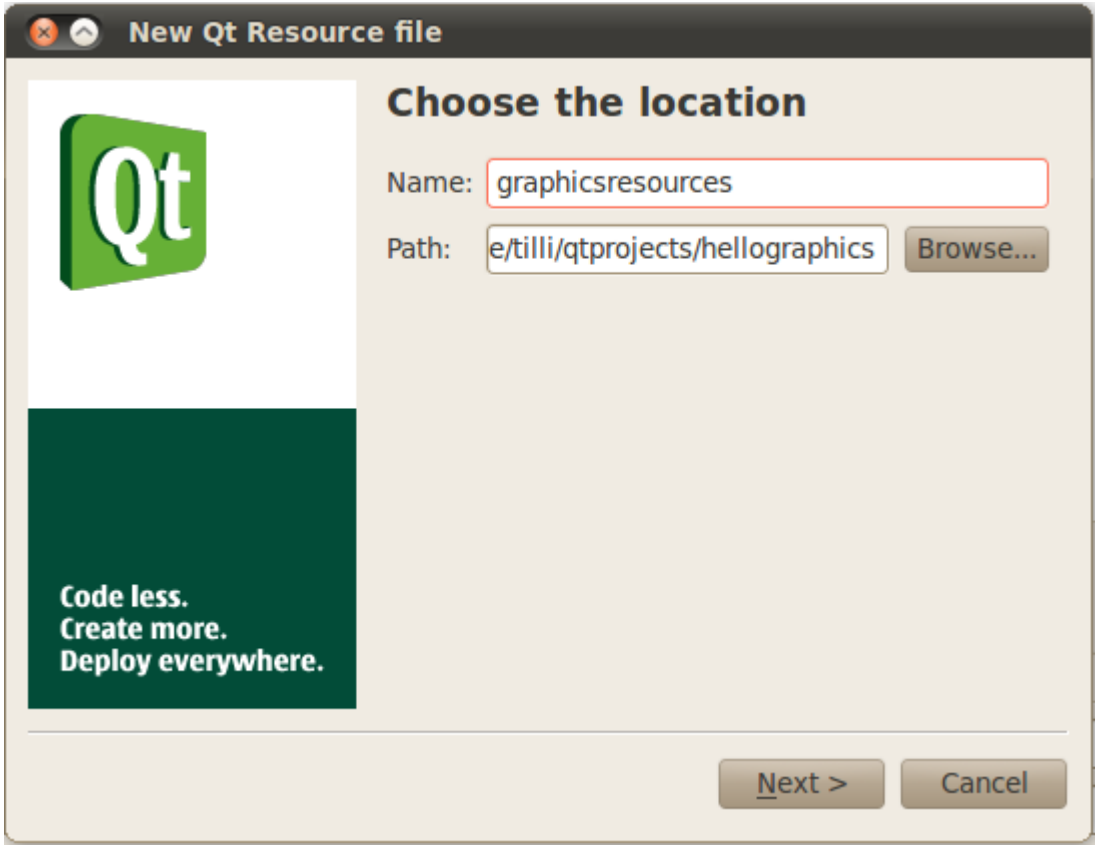
Other resources

- Resource file specifies a collection of data that should be bundled into the binary file
  - For example pictures and QML files
- QtCreator can help add resources to project
  - Qt project file has RESOURCES statement, which contains a list of *.qrc* files
  - *qrc* file is a text file, which is parsed by *resource compiler* during project build

Resource files

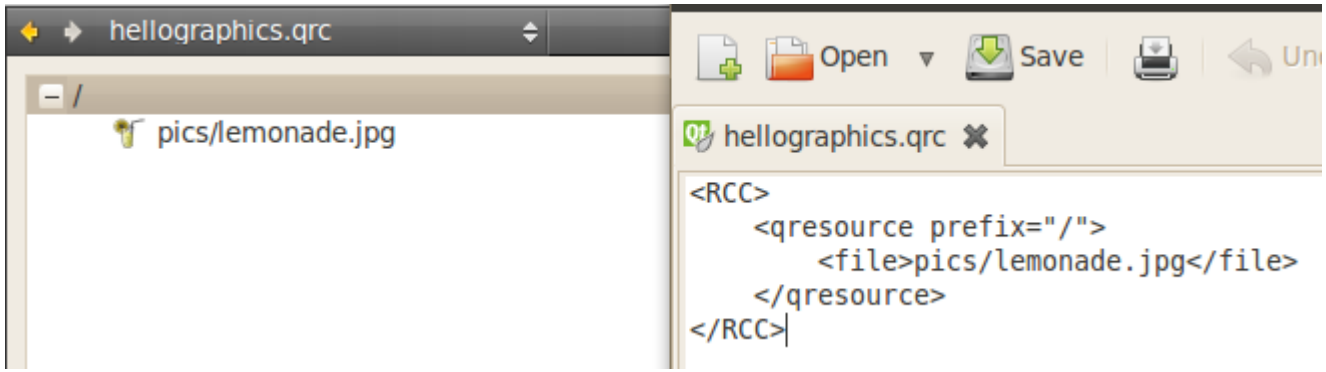


Resource files



Resource files

- After resource file has been created, add a *prefix* into it and a file under the *prefix*



- Resource is identified with a path, quite similarly as a file
  - :/<prefix>/<resource-name>

<symbio>

Building a Qt project

QT C++ PROJECTS

Build from command line

<symbio>

- Run *qmake* in the directory, which contains the *.pro* file
  - Generates project *Makefile*
- Run *make* to build project
  - Runs *uic*, *moc* and *rc* to generate *ui_<form>.h*, *moc_<class>.cpp* and *rc_<resource>.cpp* files
  - Compiles the sources to object *.o* files
  - Links the object files together and with Qt modules to produce the project target

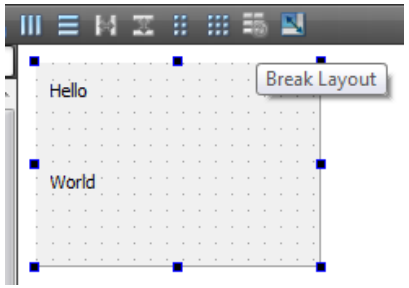
Shadow builds

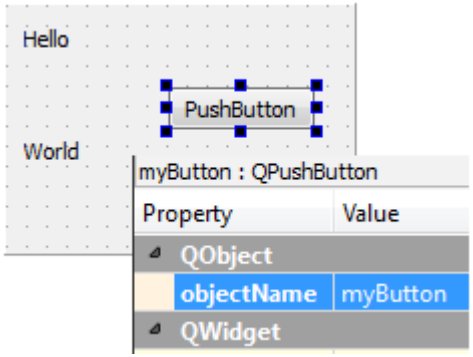
<symbio>

- Interactive part
  - Qt creator project properties
  - Shadow build output in file system

Excercise

- Open the *helloworld* example into GUI designer
  - Add a button widget
    - Break the form layout first
    - Change button object name to *myButton*
  - Right-click and select *Go to slot*
    - Selected *clicked* slot
    - Add *qDebug("click…");* to code
- Build and run





<symbio>

Shared libraries

QT C++ PROJECTS

Shared libraries

- A shared library contains code that is loaded once and shared by all executables that use it
- Saves resources compared to *a static library*, which is especially important in mobile devices

Exporting from project

<symbio>

- In order to be used, a *library* needs to provide an API
  - Public headers are included into client project
  - Client is linked against the library
- Project contents are exported with help of makefiles
  - Run *make install* in project directory
    - Files and paths need to be specified first

Public headers

<symbio>

- Project file variables
  - Project files support user-defined variables
    - For example FOO = 5
  - Variables can be referenced with $$<name>
    - For example $$FOO would be replaced with 5
- Public headers can be separated from private headers with help of a variable

```
PUBLIC_HEADERS += helloworldlibrary.h \
    helloworld-library_global.h

HEADERS += $$PUBLIC_HEADERS \
    hwlibprivate.h
```

Exporting from project

<symbio>

- INSTALLS directive is used to specify what and where to install
  - var.path specifies where to install
    - Path is relative to project directory
  - var.files specify what to install
    - *target.files* is pre-defined to contain project binaries

```
TARGET = helloworld-library

PUBLIC_HEADERS += helloworldlibrary.h \
    helloworld-library_global.h

public_headers.path = ../inc
public_headers.files = $$PUBLIC_HEADERS
target.path = ../bin
INSTALLS += target \
    public_headers
```

Using exported libraries

<symbio>

- To use the library, a project needs to find the exported data
  - INCLUDEPATH for headers
  - LIBS for libraries
    - -L<path>
    - -l<library-name>

```
INCLUDEPATH += ../inc
LIBS += -L../bin -lhelloworld-library
```

- Examples in *helloworld-console* and *helloworld-library* projects

Object-oriented programming with Qt/C++

C++ INTRODUCTION

C++ OOP basics

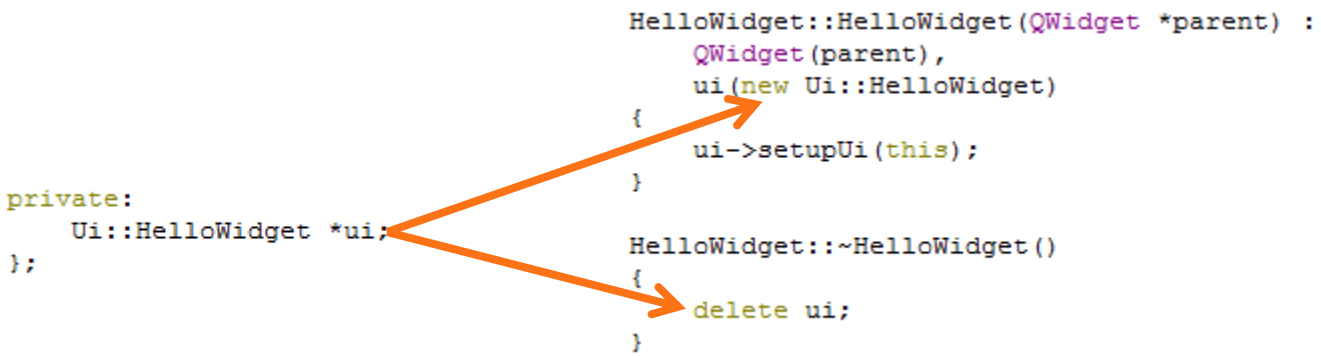
- A *class* defines the structure of an *object*
  - Objects are created with *new* operator and freed with *delete* operator
  - When object is created, its *constructor* is called and delete calls *destructor*

```
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;
}
```

C++ OOP basics

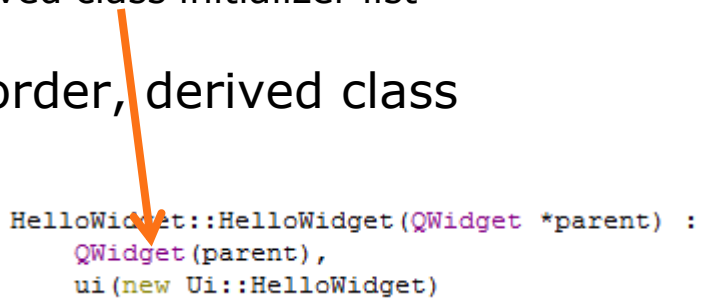
- Object data must be initialized in constructor and freed in destructor
  - C++ has constructor *initializer list*



C++ OOP basics

<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

- 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

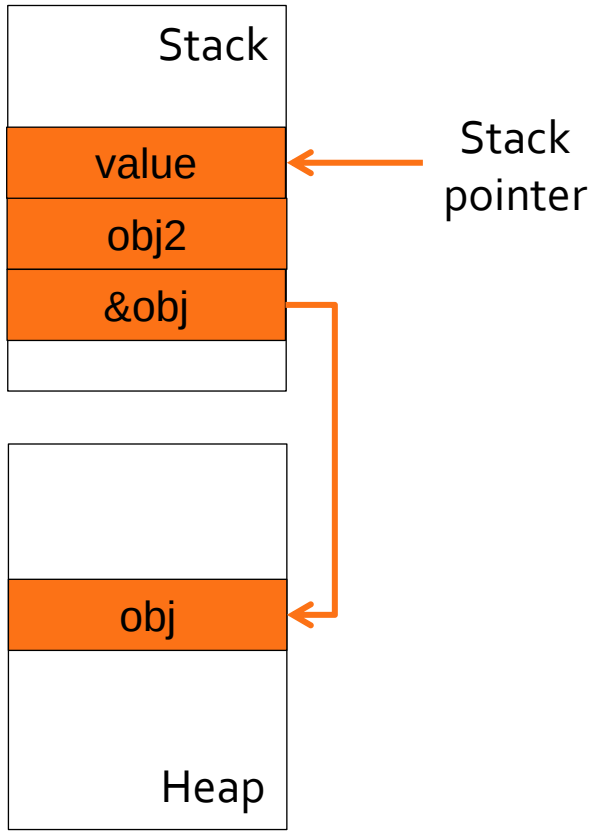
<symbio>

- 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;
}
```



Pointers and references

- An object allocated from heap is referenced by a *pointer* or a *reference*
  - *Pointer* is numeric value representing the memory address of the object
    - Usually 32 or 64 bits in size
    - *Dereference operator (*)* returns the value
  - *Reference variable* can be thought as being the object itself
    - Also note that the *reference operator* returns the pointer of an object

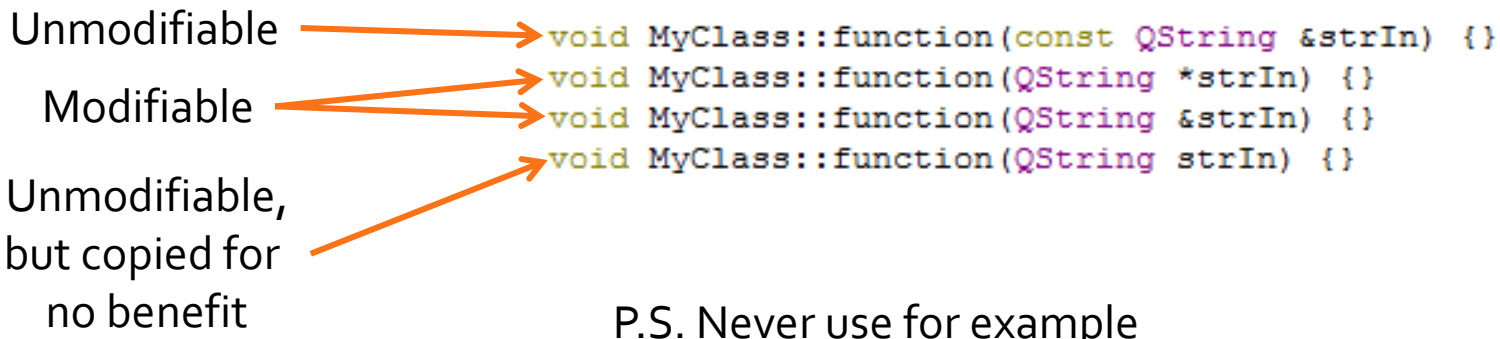
Pointers and references

- Following prints out *10* (why?)
  - Also note the memory leak

```
int stackAllocatedInteger = 10;
int &integerReference = stackAllocatedInteger;
int *heapAllocatedInteger = new int;
*heapAllocatedInteger = 20;
integerReference = 5;
heapAllocatedInteger = &integerReference;
qDebug() << stackAllocatedInteger + *heapAllocatedInteger;
```

Pointers and references

- References are not usually used as variables within functions
- Main use is *constant reference* for passing objects into functions as input parameters



P.S. Never use for example *const int* & parameter (why?)

Function return values

- Functions return value should always be thought as a copy
  - Thus, return heap-allocated objects by *pointer*
  - And stack-allocated objects by *value*
    - Never return a pointer to stack-allocated value

```
void anotherFunction()
{
    int someUselessVariable = 25;
}

int main(int argc, char *argv[])
{
    int *value = stupidFunction();
    anotherFunction();
    qDebug() << *value;
}
```

```
int *stupidFunction()
{
    int value = 5;
    return &value;
}
```

value is 25

Notes about const

- The *const* keyword indicates that something is not modifiable
  - *const* variables, *const* parameters, *const* member functions
  - A class member function which doesn't change the object should be marked *const*
    - Possibilities for better compiler optimizations
    - Class variables are *const* within a *const* function

```
QString MyClass::stringGetter() const
{
    return theString;
}
```

const string can be returned as it is copied

C++ hands-on

<symbio>

- Create a new Console Application project
  - Add a class, which doesn't inherit anything
  - Add a *QString* member
  - Add *get* and *set* functions for the member
    - Note: Qt naming conventions don't have *get*
      - *setFoo* is matched by *foo*
  - In main function, create instance from heap and set the member to some string value

Shared data objects

CORE FEATURES

Shared data objects

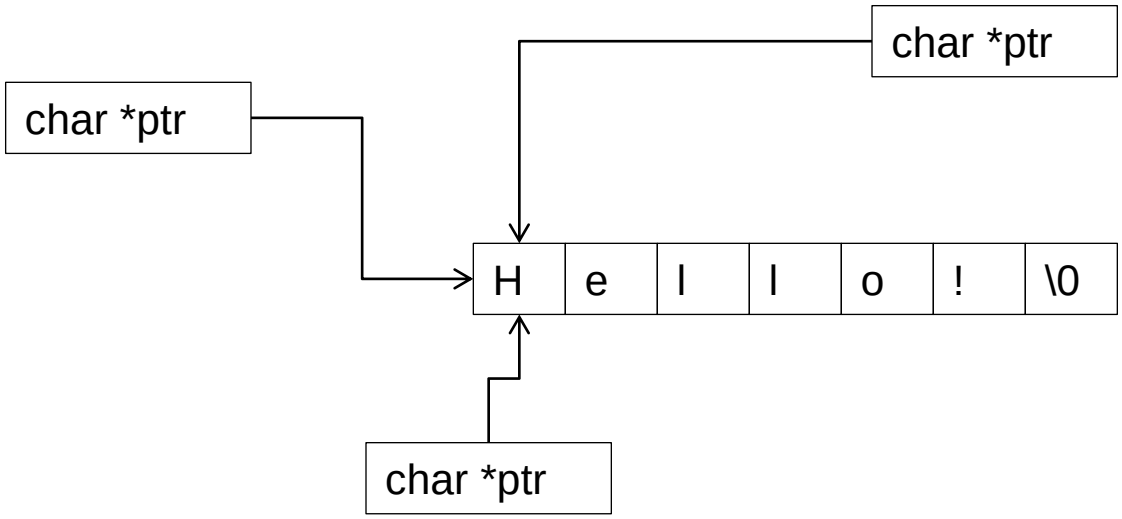
<symbio>

- A *shared data object* doesn't store the object data by itself
  - Instead, data is *implicitly shared*
    - With copy-on-write semantics
  - Easier to use that just pointers
    - The object can be thought as *simple value type*
- Examples:
  - Strings, images, collections

Implicit sharing

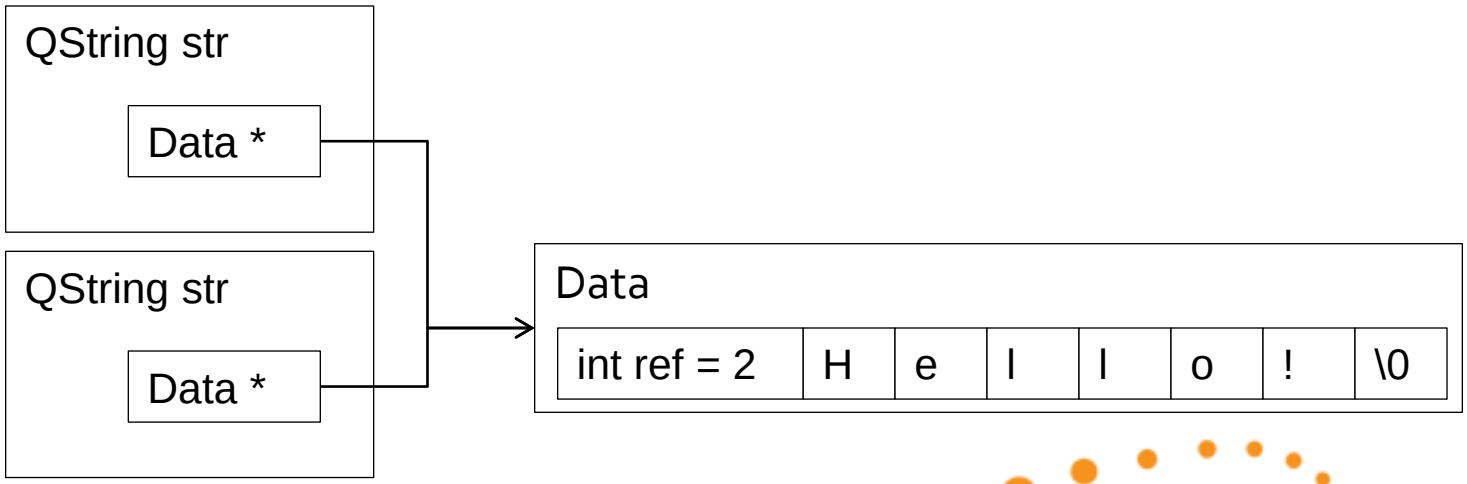
<symbio>

- 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



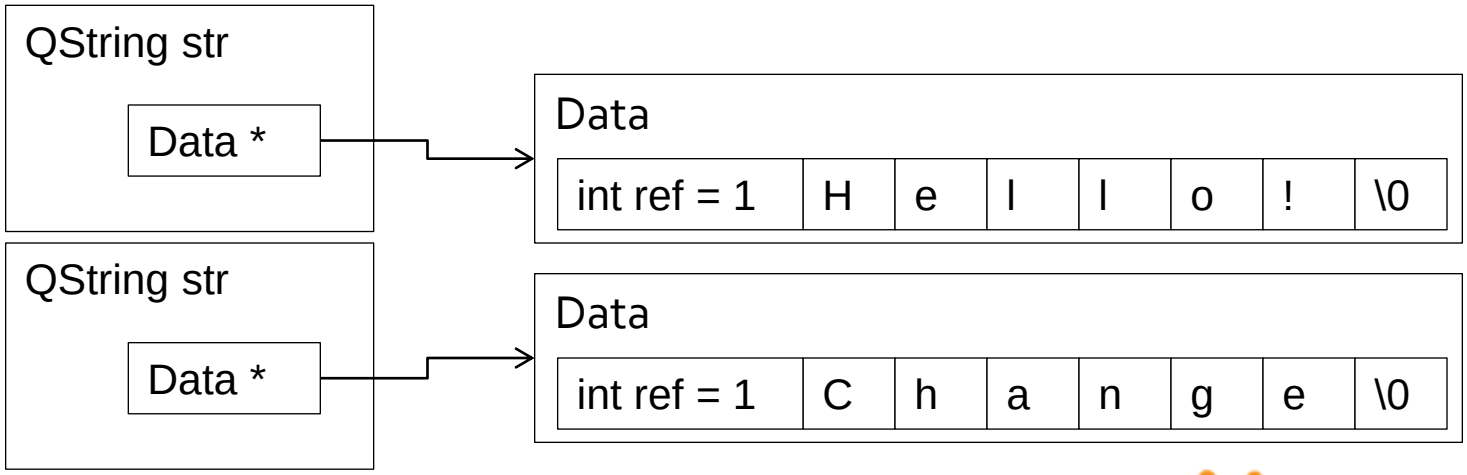
Implicit sharing

- 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

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 the two 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

<symbio>

- 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 *>

<symbio>

Object model and signals & slots

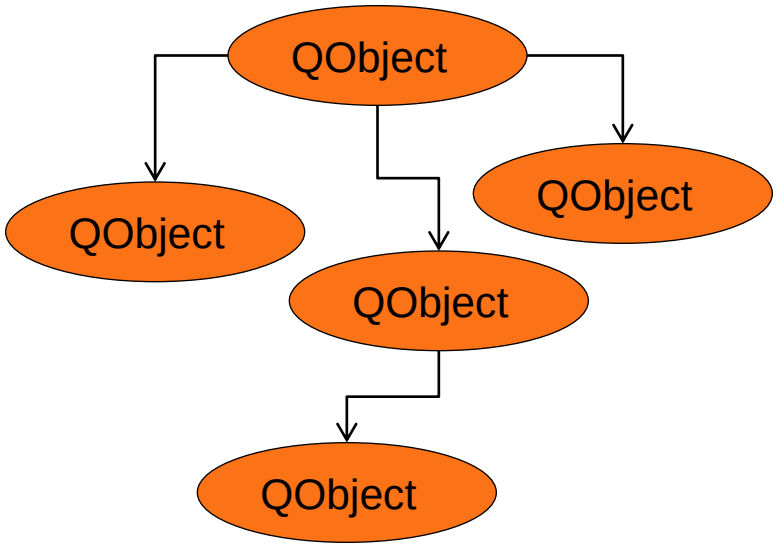
CORE FEATURES

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
  - i.e. *onSignal: slot()* in QML code

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

<symbio>

- *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

- *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

- 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();
    }
}
```

Trying it out

<symbio>

- Open the *hellosignalslot* example
- Build and run

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* objects 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

Object properties

CORE FEATURES

Object properties

- All QObject-based classes support *properties*
  - A property is *QVariant* type, which is stored in a dictionary that uses C-style zero-terminated character arrays as keys
    - i.e. name-value pair
  - Properties can be *dynamic* or *static*
    - *Dynamic properties are assigned at run-time*
    - *Static properties are defined at compile time and processed by the meta-object compiler*

Object properties

- Static properties are declared into class header using *Q_PROPERTY* macro

```
class AnimatedPixmap : public QObject, public QGraphicsPixmapItem
{
    Q_OBJECT
    Q_PROPERTY(qreal rotation READ rotation WRITE setRotation NOTIFY rotationChanged)
```

- The above statement defines a property
  - Type is *qreal*, name is *rotation*
  - When read, *rotation* function is called
  - When modified, *setRotation* function is called
  - Changes are notified via *rotationChanged* signal

Object properties

- Properties are used in QML/C++ hybrid programming
  - Object properties are mapped into QML
- Monday's topics
  - QML plug-in's
  - Exposing C++ objects to QML

SERIOUS ABOUT SOFTWARE