**Lesson 0**

Qt is a cross-platform application development framework for desktop, embedded and mobile. In this subject it is compiled using C++.

Qt Class Hierarchy

QTextStream

QString

QPaintDevice

QObject

QLayoutItem

QCoreApplication

QLayout

QImage

QWidget

QCalendarWidget

QDialog

QGuiApplication

QBoxlayout

QApplication

QInputDialog

Directives/Header Examples

|  |  |  |
| --- | --- | --- |
| QT += core: **Streams** | | |
| **QTextStream**  #include <QTextStream> | interface for reading and writing text | cin(stdin); cout(stdout);  cin >> d; cout << “Hello World”; |
|  |  |  |
| **QDebug**  #include <QDebug> | output stream for debugging information. | qDebug() << "Hello World"; |
| QT += widgets: **Widgets** | | |
| **QApplication**  #include <QApplication> | Execute application | a(argc, argv);  a.exec(); |
| **QMessageBox**  #include <QMessageBox> | Modal dialog | QMessageBox::StandardButton reply;  reply = QMessageBox::question(0, "Coordinate", "Enter another coordinate?",QMessageBox::Yes|QMessageBox  QMessageBox::information(0, "Output", "String format: " + Outputlist.toString() + "\nDecimal format: " + QString::number(Outputlist.toDecimal()) + "\n");  //Assign MessageBox reply to variable  reply = QMessageBox::question(0, "Coordinate", "Enter another coordinate?", QMessageBox::Yes|QMessageBox::No); |
| **QInputDialog**  #include <QInputDialog> |  | QInputDialog::getText(0,"Input","Please enter stuff”), QLineEdit::Normal, QString(""), &ok); |
| **QCalendarWidget**  #include <QCalendarWidget> | Allows user to select/input a date | QCalendarWidget \*calendar;  calendar->setGridVisible(true); |
| QT += core: **Strings** | | |
| **QString**  #include <QString> | Strings + string manipulation | QString::number(degrees);  myString.toUpper();  myString.toLower();  myString.toInt();  myString.toDouble();  //Split string according to delimiter  QString.split(" ") |
| **QStringList**  #include < QStringList> | String Objects | inputCoordinates.split(" ")  inputCoordinates.split("/") |
| **QChar**  #include < QChar> | Characters | QChar(0260); |

**Lesson 1**

Windows

Widgets

Layouts

Signals

Slots

**Widgets**

A user interface object that can process input, emit signals and

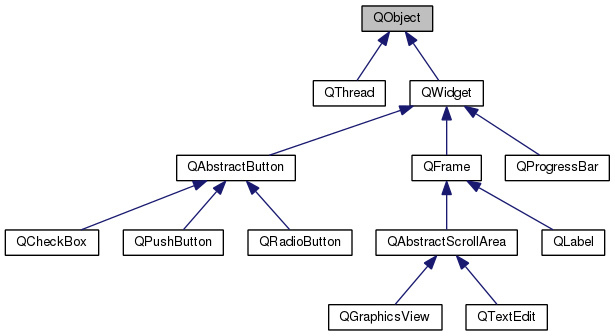
draw graphics

– A widget can be styled to have a vastly different appearance than

its default

– Most widgets generate signals that can be received by pieces of

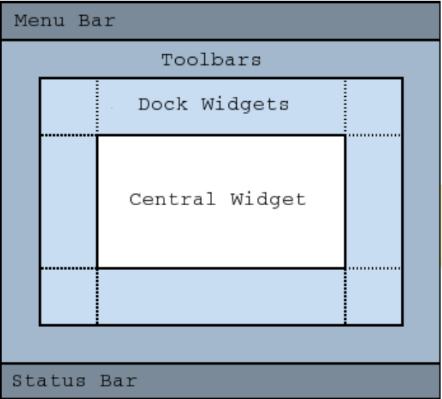
your code called slots*is possible to display widgets like a QPushButton without a layout*



**Layouts**

A layout describe how widgets are organized and positioned in a

user interface. The jobs of a QT layout:

- Positioning of widgets in GUI

- Font size, text or other widget changes

- Add or removal of new widgets

- Showing and hiding of existing widgets

Four different layouts are commonly used

– QVBoxLayout

– QHBoxLayout

– QFormLayout

– QGridLayout

QT layouts and widgets share numerous parent/child

relationships

– Widgets can contain other widgets

– Widgets can have layouts

– Layouts can contain widgets

– Layouts can contain other layouts

– There can be a gigantic graph of parent and child relationships in a GUI

**Lesson 2**

Event Handling

In Qt, this is done by signals and slots. All widgets are capable of firing events and receiving events. Signals & slots can have M-to-N connections.

**Signal**

* Signals are used to notify (emit) widgets of an event
* Emitted when a particular event occurs (e.g., clicked())
* Qt widgets: predefined signals
* Developer can create custom signals

**Slot**

* Slots are used to receive (listen for) widget events
* Function called in response to a signal
* Qt widgets: predefined slots
* Developer can create custom slots

Example: ASS3 itemdialog.cpp

class ItemDialog : public QDialog {

Q\_OBJECT

private:

...

public:

...

private slots:

void fixItem();

};

**Connection:**

* connect is used to tie together a signal & a slot
* established by developer
* handled by Qt framework

Example: ASS3 itemdialog.h

ItemDialog::ItemDialog(QWidget \* p, QString t, Item \* i)

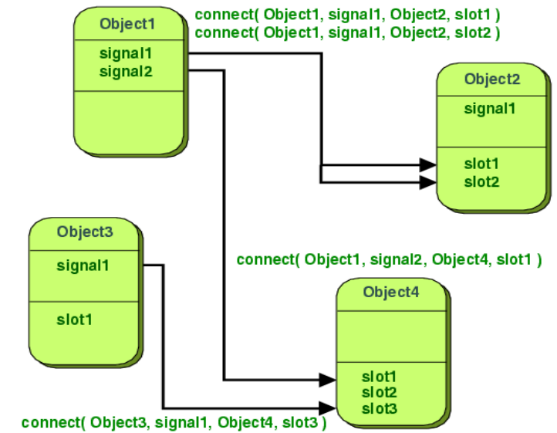
: QDialog(p), item(NULL), barcodeLineEdit(new QLineEdit()), descriptionLineEdit(new QLineEdit()), stockSpinBox(new QSpinBox()), priceSpinBox(new QDoubleSpinBox()) {

//Connect clicked() signals to slots

connect(cancelPushButton, SIGNAL(clicked()), this, SLOT(close()));

connect(OKPushButton, SIGNAL(clicked()), this, SLOT(fixItem()));

}



**Lesson 3**

Class Definitions

A default constructor does not have any parameter, but if you need, a constructor can have parameters. This helps you to assign initial value to an object at the time of its creation

**The Class Constructor**

A class constructor is a special member function of a class that is executed whenever we create new objects of that class.

GPSCoord::GPSCoord():

latitude(new Coordinate(0, 0, 0, 'N')),

longitude(new Coordinate(0, 0, 0, 'E')) {

}

**The Class Destructor**

A destructor is a special member function of a class that is executed whenever an object of it's class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class.

GPSCoord::~GPSCoord(){

delete latitude;

delete longitude;

}

**Copy constructor**

The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. The copy constructor is used to −

•Initialize one object from another of the same type.

•Copy an object to pass it as an argument to a function.

•Copy an object to return it from a function.

GPSCoord & GPSCoord::operator=(const GPSCoord & g) {

\*latitude = \*g.latitude;

\*longitude = \*g.longitude;

return \*this;

}

Difference between copy constructor and assignment

A copy constructor is used to initialize a newly declared variable from an existing variable. This makes a deep copy like assignment, but it is somewhat simpler:

1.There is no need to test to see if it is being initialized from itself.

2.There is no need to clean up (eg, delete) an existing value (there is none).

3.A reference to itself is not returned.

**Static members**

We can define class members static using static keyword. When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member.

A static member is shared by all objects of the class. All static data is initialized to zero when the first object is created, if no other initialization is present. We can't put it in the class definition but it can be initialized outside the class as done in the following example by redeclaring the static variable, using the scope resolution operator :: to identify which class it belongs to.

**References**

A reference variable is an alias, that is, another name for an already existing variable. Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable.

References vs Pointers

References are often confused with pointers but three major differences between references and pointers are −

•You cannot have NULL references. You must always be able to assume that a reference is connected to a legitimate piece of storage.

•Once a reference is initialized to an object, it cannot be changed to refer to another object. Pointers can be pointed to another object at any time.

•A reference must be initialized when it is created. Pointers can be initialized at any time.

**Lesson 4**

Access Control & Inheritance

The public Members

A public member is accessible from anywhere outside the class but within a program.

The protected Members

A protected member variable or function is very similar to a private member but it provided one additional benefit that they can be accessed in child classes which are called derived classes.

The private Members

A private member variable or function cannot be accessed, or even viewed from outside the class. Only the class and friend functions can access private members.

|  |  |  |  |
| --- | --- | --- | --- |
| Access | Public | Protected | Private |
| Same Class | Yes | Yes | Yes |
| Derived Classes | Yes | Yes | No |
| Outside Classes | Yes | No | No |

Type of Inheritance

We hardly use protected or private inheritance, but public inheritance is commonly used. While using different type of inheritance, following rules are applied:

**Public Inheritance**

When deriving a class from a public base class, public members of the base class become public members of the derived class and protected members of the base class become protected members of the derived class. A base class's private members are never accessible directly from a derived class, but can be accessed through calls to the public and protected members of the base class.

**Protected Inheritance**

When deriving from a protected base class, public and protected members of the base class become protected members of the derived class.

**Private Inheritance**

When deriving from a private base class, public and protected members of the base class become private members of the derived class.

**Lesson 5**

Relationships

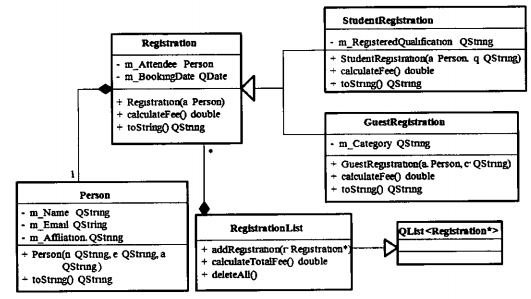
**UML Diagrams**

A screenshot of a cell phone

Description automatically generatedA close up of a logo

Description automatically generated

Example:



Example: May 2013 Q2.1

Explain the relationship between Person and Registration

Composition relationship. At least one data member of Registration (containing class), i.e. m\_Attendee is of type Person (contained class). Instance of Registration is responsible for the destruction of instances of Person.

Example: May 2013 Q2.4

Write the first line of the class definition (class Head) of StudentRegistration

class StudentRegistration : public Registration {

**Lesson 6**

Desgin Patterns (QObject)

**QObject’s Child Management**

This is the base class of all Qt Objects. Think of this as an internal Child-management system that Qt uses to handle things like QWidgets. This is so that each created QObject is unique. If you create a class that handles the visual aspects of a program (widgets), it must inherit QObject.

Example: ASS 3

class ItemDialog : public QDialog {

Q\_OBJECT

private:

ItemDialog(QWidget \* p, QString t, Item \* i);

Item \* getItem() const;

public:

static Item \* changeItem(QWidget \* p, Item \* i);

static Item \* addItem(QWidget \* p);

private:

Item \* item;

QLineEdit \* barcodeLineEdit;

QLineEdit \* descriptionLineEdit;

QSpinBox \* stockSpinBox;

QDoubleSpinBox \* priceSpinBox;

private slots:

void fixItem();

};

* QObject’s copy constructor is not public (QObjects cannot be copied)
* Copying a QObject’s data members into another QObject is still possible, but the two objects are still considered unique
* Each QObject can have (at most) one parent object and an arbitrarily large container of QObject\*children
* Each QObject stores pointers to its children in a QObjectList

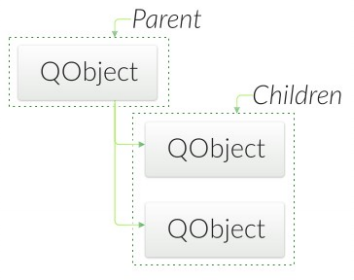
class Person : public QObject {

public:

Person(QObject\* parent, QString name);

virtual ~Person();

};



**QObject’s Child Management:** Parent-child hierarchy

NB: Parent-child relationship IS NOT inheritance!

Parent-child relationship: is meant to describe containment, or management, of objects at runtime.

Base class: The base-derived relationship is a static relationship between classes determined at compile-time.

* Initialization with pointer to parent QObject parent adds new object to list of its children
* Delete parent: automatically deletes all its children (recursively)
* Delete child: automatically removed from parent’s child list
* Some memory management handled by Qt, only delete objects created with new without parent Widgets
* Parent has additional meaning: child widgets shown within parent’s area

**Lesson 7**

Dessig n Patterns (Composite & Observer Patterns)

**Composite Pattern:** Parents and Children

Is intended to facilitate building complex (composite) objects from simpler (component) parts by representing the part-whole hierarchies as tree-like structures.

A composite object is something that can contain children.

A component object is something that can have a parent

**Observer Pattern (Publish/Subscribe Pattern)**

A design pattern that enables such a message-passing mechanism. When a particular subject object changes state, it needs an indirect way to alert (and perhaps send additional information to) all the other objects that are listening to state-change events, known as observers.

**Command Pattern**

Because an application might provide a variety of different ways for the user to issue the same command (e.g. menus, toolbar buttons, keyboard shortcuts), encapsulating each command as an action helps to ensure consistent, synchronized behavior across the application. QAction is, therefore, an ideal base class for implementing the Command pattern.

*Implement when a program/object is not sure whether classes are available or when it would be appropriate to execute such member functions.*

*Calls to such member functions are represented by objects themselves, so that they can be executed at the appropriate time (e.g. events or actions)*

QAction

commands can be invoked via menus, toolbar buttons, and keyboard shortcuts. Each of these commands is represented as an action with QAction.

Once a QAction has been created it should be added to the relevant menu and toolbar, then connected to the slot which will perform the action.

QActionGroup can be used to group these together for context awareness. For example , if you have a Left Align action, a Right Align action, a Justify action, and a Center action, only one of these actions should be active at any one time.

Example: ASS3 Different buttons on a GUI to perform the same action

MyMainWindow::MyMainWindow() : textEdit(new QTextEdit(this)) {

QActionGroup \* fileGroup = new QActionGroup(this);

fileGroup->addAction(new QAction(QIcon("/icons/o.png"), "&Open", this));

fileGroup->addAction(new QAction(QIcon("/icons/s.png"), "&Save", this));

fileGroup->addAction(new QAction(QIcon("/icons/x.png"), "E&xit", this));

}

**Lesson 3**

QT Classes

**Parameterized Constructor**

**Lesson 8**

OOP

**Containers**

Containers are objects that can contain other objects.

To use a generic container, the declaration should answer the question: “Container of what?” For example:

QList<double> doubleList;

You can design a container class in one of three different ways:

1) class doubleList: public QList;

2) You can create a QList as a member function.

3) You can have a doubleList class which has \*double objects

Example: implementation of a container class

class ItemList {

private:

//The list should not be accessible from outside the class

QMap<QString, Item\*> list;

bool changed;

public:

ItemList();

void readList(QString fname);

void writeList(QString fname);

bool getChanged() const;

void addItem(const Item & i);

bool removeItem(QString b);

Item \* findItem(QString b) const;

QStringList toStringList() const;

~ItemList();

};

**QList<T>**

The most commonly used container class. It stores a list of values of a given type (T) that can be accessed by index.

Internally, the QList is implemented using an array, ensuring that index-based access is very fast.

QStringList inherits from QList<QString>.

Add items to a list:

QList::append() insert at end

QList::prepend() inert at beginning

QList::insert() insert in the middle

**Inheritance**

#indef CHILD\_H

#define CHILD\_H

#include “parent.h”

// Sub class inheriting from Base Class(Parent)

class Child: public Parent {

public :

int i\_c;

};

#endif // CHILD\_H

#indef PARENT\_H

#define PARENT\_H

class Parent {

public :

int id\_p;

};

#endif // PARENT\_H

#indef CHILDLIST \_H

#define CHILDLIST\_H

#include “child.h”

// Container class that manages a list of child

class ChildList: public QList<Child\*>{

public :

int i\_cl;

};

#endif // CHILDLIST \_H

#include <QTextStream>

#include <QApplication>

//main function

int main()

{

QApplication a;

a(argc, argv);

cout(stdout);

Child obj1;

// child has all data members and member functions of class parent

obj1.id\_c = 7;

obj1.id\_p = 91;

cout << "Child id is " << obj1.id\_c << endl;

cout << "Parent id is " << obj1.id\_p << endl;

a.exec();

}

**Polymorphism**

Runtime Polymorphism (Virtual Functions)

Static Polymorphism (Function Overloading)

#indef PARENT\_H

#define PARENT\_H

class Parent {

public :

int id\_p;

//Constructor

Parent(int id);

//Add virtual destructor to make polymorphic (run-time polymorphism)

virtual ~Parent();

};

#endif // PARENT\_H

#include <QTextStream>

#include <QApplication>

//main function

int main()

{

QApplication a;

a(argc, argv);

cout(stdout);

Parent c;

//Run-time Type Identification (RTTI), cast to a specific type

if(dynamic\_cast<Child\*>(c)->id\_c == “C49”){

cout << “Child ID is ‘C49’\n”;

}

a.exec();

}

#include "journalarticle.h"

//constructor for child class

JournalArticle::JournalArticle(QString \_title,

QStringList \_authors,

int \_year,

QString \_refID,

QString \_journalName,

int \_volume,

int \_number):

//constructor for parent class

Reference(\_title, \_authors, \_year, \_refID),volume(\_volume), number(\_number){

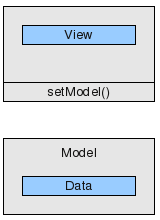
}

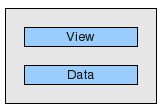
**Lesson 9**

ViewModels

Model/View is a technology used to separate data from views in widgets that handle data sets. Standard widgets are not designed for separating data from views and this is why Qt has two different types of widgets. Both types of widgets look the same, but they interact with data differently.

Standard View Model





Example: ASS3 A View (Class implemented as a container)

class ItemDialog : public QDialog {

Q\_OBJECT

private:

ItemDialog(QWidget \* p, QString t, Item \* i);

Item \* getItem() const;

public:

static Item \* changeItem(QWidget \* p, Item \* i);

static Item \* addItem(QWidget \* p);

private:

Item \* item;

QLineEdit \* barcodeLineEdit;

//...

};

Example: ASS3 A Model (Class implemented as a container)

class ItemList {

private:

QMap<QString, Item\*> list;

bool changed;

public:

ItemList();

void readList(QString fname);

void writeList(QString fname);

bool getChanged() const;

//...

~ItemList();

};