Librairies de développement
Qt

Yannick Kergosien

Polytech Tours
Université François Rabelais de Tours

Janvier 2016

Sommaire

Développement d'applications complexes

La STL

- Formalisme objet - Containers, Iterators, Algorithm
- Réutilisabilité grâce aux templates
- Nombreux algorithmes déjà implémentés
- Couvre les besoins de base en programmation orientée objet
- J'encourage **vivement et fortement** à utiliser la STL

Limites

- Pas d'algorithmes complexes
- Tous les problèmes du C++ ne sont pas réglés pour autant

Problèmes récurrents

Hiérarchie des objets

- Lorsque que l'on détruit un objet, on aimerait, parfois, détruire toute une liste d'objets qui dépendent de lui.
- Typiquement, si vous avez un objet " Fenêtre ", lorsque vous détruisez la fenêtre, vous aimeriez que tous les objets " Widgets " soient aussi détruits.

Communication entre les objets

- Envie d'un mécanisme de type signaux
- La modification d'un objet peut être à répercuter sur un autre

Qt en quelques mots

- Qt est à la base un toolkit graphique.
- Boîte à outil complète (GUI, Accès BDD, ...)
- Multiplateforme
  - client lourd (Windows, GNU/Linux, Mac OS X)
  - mobile / embarqué (Meego, Symbian, ...)
- Nouveaux modèles d'objets et de communication
- Quelques exemples d'outils développés avec Qt : KDE, Opera, VLC, Google Earth, Skype, Mumble...
- GNU LGPL 3 à partir de Qt 5.7

Historique de Qt en bref

- Qt1 en 1995 par Trolltech (entreprise norvégienne)
- Qt2 en 1999 et Qt3 en 2001, quelques améliorations (dont l'embarqué)
- Qt4 en 2005 (jusqu'à 4.8), de nombreux ajouts de modules et bibliothèques
- Achat en 2008 par Nokia puis Digia en 2011
- Qt5 fin 2012, amélioration de la performance, HTML5 avec QtWebKit 2, développement mobile...

L'environnement de Qt (depuis Qt4)

Différentes bibliothèques séparées en modules :

- **QtCore** : fonctionnalités non graphiques utilisées par les autres modules
- **QtWidgets** : pour les composants graphiques
- QtNetwork : pour la programmation réseau
- QtOpenGL : pour l'utilisation d'OpenGL
- QtSql : pour l'utilisation de base de données SQL
- QtXml : pour la manipulation et la génération de fichiers XML
- QtWebkit : pour le rendu HTML
- QtAssistant : pour l'utilisation de l'aide de Qt
- Qt3Support : pour assurer la compatibilité avec Qt 3
- ...

L'environnement de Qt (depuis Qt4)

C'est aussi :

- Qt Jambi : les possibilités de Qt pour le langage JAVA
- Qtopia : une version de Qt destinée aux dispositifs portables (téléphones, PocketPC, etc..) et aux systèmes embarqués (ex Qt/Embedded)
- QSA : Qt Script for Applications, ajout de scripts à ses applications.
- ...

Environnement de Développement Intégré

Qt Creator :

- Environnement de développement intégré dédié à Qt (il en existe d'autre, comme QDevelop et Monkey Studio, ou par des modules de Qt avec Eclipse ou Visual Studio)
- Gestion de projet
- Editeur (coloration syntaxique, complétion, l'indentation, ...)
- Mode de débuggage
- Outil Qt Designer
- ...

Qt Designer

- Pour créer des interfaces graphiques Qt
- Convivial : *Drag&drop*, propriétés facilement accessibles, ...
- Fichiers d'interface graphique formatés en XML (extension *.ui*), convertis en classe C++ par l'utilitaire *uic* lors de la compilation

La compilation

Génération d'une application en Qt :

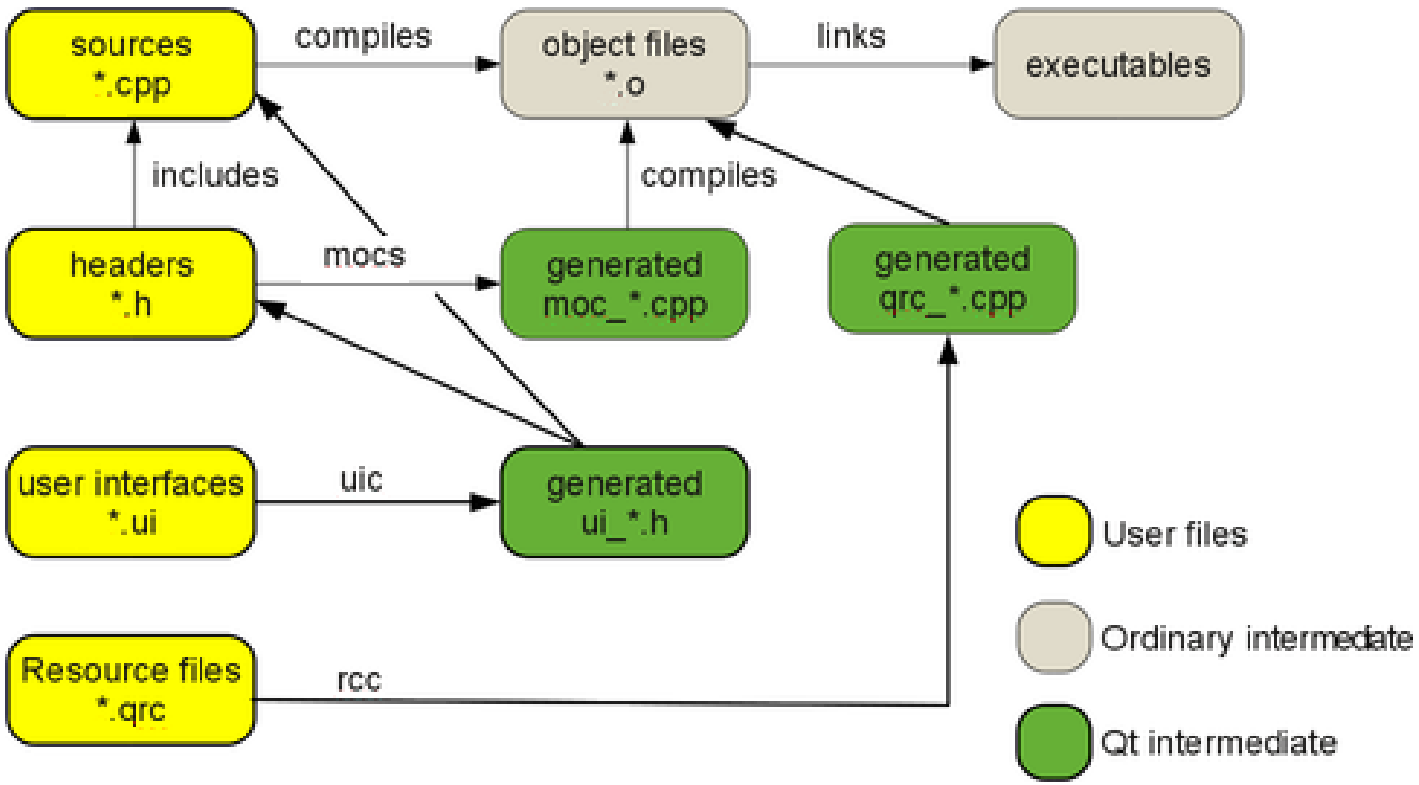
- "Moteur de production spécifique" : Qmake
- S'appuis sur un fichier descripteur du projet *.pro* pour génèrer un Makefile
- Utilise le Moc (Meta Object Compiler) : préprocesseur qui génère du code supplémentaire pour la prise en compte d'élément que le C++ ne permet pas (signaux et slots)
- S'appuis sur un fichier *.qrc* pour les ressources de l'application (images)

Exemple d'un *.pro*

Exemple

```
1  QT+= core gui \
2       ...
3       xml\
4       webkit
5  CONFIG += exceptions
6  TARGET = APORHAD
7  TEMPLATE = app
8  INCLUDEPATH += ./dmd \
9                 ...
10 SOURCES += main.cpp\
11              ...
12 HEADERS += mainindow.h \
13              ...
14 FORMS += mainwindow.ui \
15            ...
16 RC_FILE += MyApp.rc
```

La compilation



Sommaire

QObject

- Qt étend le modèle d'objets C++ (point fort de Qt)
- En partie pour gérer :
  - La hiérarchie des objets
  - La communication entre les objets
- Chaque QObject à un nom qui lui sert d'identité
- Un QObject n'est pas assignable, pas de constructeur de recopie

Hiérarchie des objets

Un unique constructeur

```
1   QObject::QObject ( QObject * parent = 0 )
```

- Lorsque vous instanciez un objet, vous lui donnez un “parent”
- Exemple : un bouton “appartient” à sa fenêtre
- **Si un parent est détruit, tous ses enfants sont aussi détruits**
  - On ne détruira (opérateur *delete*) donc que les QObject créés par l'opérateur *new* qui n'ont pas de parent.

Hiérarchie des objets

Exemple simple

```
1    class MonObjet : public QObject
2    {
3        public:
4            MonObjet( QObject *parent=0, char *name=0 ) :
                 QObject( parent ) {
5                this->setObjectName(name);
6                std::cout << "Cree: " << qPrintable(
                     objectName()) << std::endl;
7            }
8};
```

Hiérarchie des objets

Exemple simple

```
1   class MonObjet : public QObject
2   {
3       public:
4           MonObjet( QObject *parent=0, char *name=0 ) :
                QObject( parent ) {
5               this->setObjectName(name);
6               std::cout << "Cree: " << qPrintable(
                    objectName()) << std::endl;
7           }
8           ~MonObjet() {
9               std::cout << "Detruit: " << qPrintable(
                    objectName()) << std::endl;
10          }
11  };
```

Hiérarchie des objets

Exemple simple

```
1   class MonObjet : public QObject
2   {
3       public:
4           MonObjet( QObject *parent=0, char *name=0 ) :
                QObject( parent ) {
5               this->setObjectName(name);
6               std::cout << "Cree: " << qPrintable(
                    objectName()) << std::endl;
7           }
8           ~MonObjet() {
9               std::cout << "Detruit: " << qPrintable(
                    objectName()) << std::endl;
10          }
11          void traitement() {
12              std::cout << "Traitement: " << qPrintable(
                    objectName()) << std::endl;
13          }
```

Hiérarchie des objets

Exemple simple

```
1  int main( int argc, char **argv )
2  {
3     QApplication a( argc, argv );//Décrit plus loin
4
5     MonObjet cTop(0, "Top");
6     MonObjet *co1 = new MonObjet(&cTop, "MonObjet 1");
7     MonObjet *co2 = new MonObjet(&cTop, "MonObjet 2");
8     MonObjet *co3 = new MonObjet(co1, "MonObjet 3");
9
10    cTop.traitement();
11    co1->traitement();
12    co2->traitement();
13    co3->traitement();
14
15    return 0;
16 }
```

Hiérarchie des objets

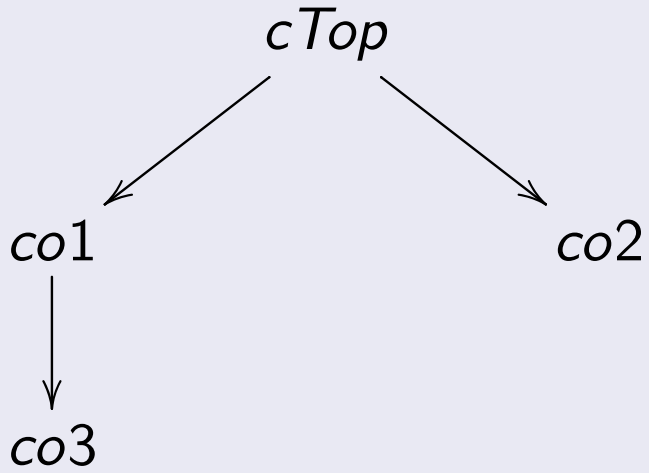
Sortie

```
Cree: Top
Cree: MonObjet 1
Cree: MonObjet 2
Cree: MonObjet 3
Traitement: Top
Traitement: MonObjet 1
Traitement: MonObjet 2
Traitement: MonObjet 3
Detruit: Top
Detruit: MonObjet 1
Detruit: MonObjet 3
Detruit: MonObjet 2
```

Code

```
MonObjet cTop(0, "Top");
MonObjet *co1 = new MonObjet(&cTop,...
MonObjet *co2 = new MonObjet(&cTop,...
MonObjet *co3 = new MonObjet(co1,...
```

Graphe



Hiérarchie - Conclusion

- Simplifie la gestion des appels aux destructeurs
- Formalisme finalement assez logique
  - Cadre des GUI: widgets d'une fenêtre à détruire si l'on détruit la fenêtre
  - Se généralise assez bien
- Mais comment les enfants ont-ils été prévenus que le père avait été détruit ?

Signaux et slots

- Qt donne un mécanisme complet de gestion des signaux
- Objectif : faire communiquer facilement des objets les uns avec les autres
- Exemple : lorsque l'on clique sur un bouton Fermer, on souhaite appeler la fonction `close` de la fenêtre dans laquelle le bouton est situé...
- Généralement cela s'effectue par l'intermédiaire d'un *callback*

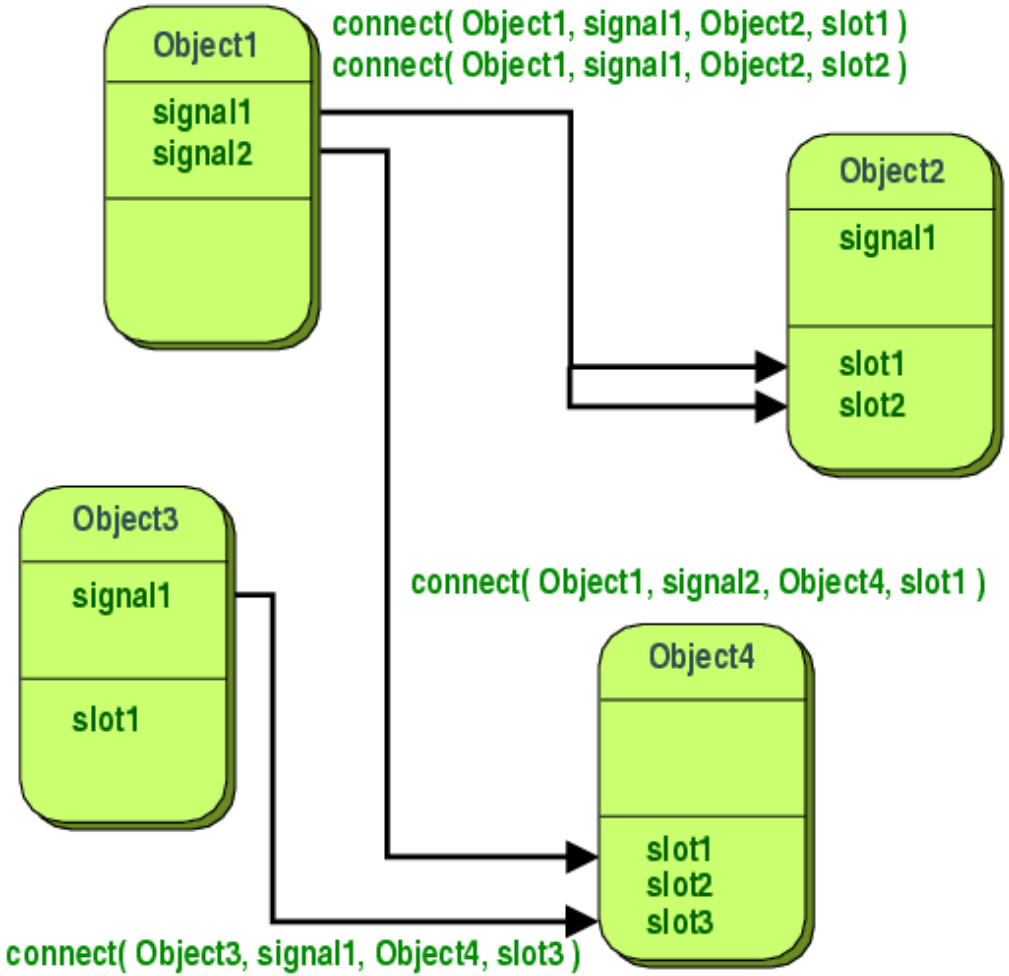
Callbacks (fonction de retour)

- Vous souhaitez appeler une fonction (C) depuis une autre fonction (T)
- L'idée est de passer un pointeur sur C à la fonction T
- Pas terrible :
  - Si on change la fonction (C)... (pas de centralisation)
  - Réutilisabilité médiocre

Signaux et slots en Qt

- signal / slot (héritage de QObject)
- Un signal est connecté à un slot
- L'émetteur d'un signal ne connait pas le ou les objets destinataires (peut être ignoré)
- Un objet interceptant un signal ne connait pas le ou les émetteurs
- Gestion par événements : communication asynchrone : une méthode qui émet un signal ne sait pas s'il a été reçu ou traité
- Emettre un signal dans le vide n'est pas grave
- La signature d'un signal est d'un slot doit concorder : même type de paramètres (mais un slot peut avoir moins de paramètres qu'un signal)

Signaux et slots



Les signaux

- Un signal est émis sur occurence d'un évènement particulier
  - "Automatique" si le signal existe déjà (exemple : QPushButton::clicked())
  - Emettre le signal soit même : emit monSignal(parametreSignal);
- Définir son signal :
  - pas de valeur de retour (void)
  - pas de définition de la méthode (pas corps)

Exemple (peut être public ou private ou protected)

```
1 public signals :
2         void monSignal(int parametreSignal);
```

Les slots

- Un slot est une fonction membre d'un objet qui est appelée sur occurence d'un signal (ou comme une méthode normale)
- Doivent avoir un corps mais peuvent être virtuels ou surchargés

Exemple (peut être public ou private ou protected)

```
1 public slots :
2         int monSlot(int parametre);
```

Connexions des Signaux et des slots

- Grâce à la méthode connect/disconnect de QObject
- Un signal peut être connecté à plusieurs slots (activés dans un ordre arbitraire).
- Plusieurs signaux peuvent être connectés à un seul slot.

Exemple

```
1  QObject::connect(ObjEmetteur, SIGNAL(monSignal(int)),
2                   ObjRecepteur, SLOT(monSlot(int)));
3  ...
4  QApplication app(argc, argv);
5  ...
6  QPushButton * bExit= new QPushButton("Quit",this);
7  ...
8  QObject::connect(bExit, SIGNAL(clicked()),
9                              &app, SLOT(quit()));
```

Signaux et slots - Exemple

Objectif : synchronisation d'une valeur d'un attribut de plusieurs mêmes objets

Code

```
1 class SyncVal
2 {
3 public:
4     SyncVal() { my_value = 0; }
5     int value() { return my_value; }
6     void setValue(int value);
7 private:
8     int my_value;
9 };
```

Signaux et slots - Exemple

Code

```
1  #include <QObject>
2  class SyncVal : public QObject
3  {
4      Q_OBJECT
5  public:
6      SyncVal() { my_value = 0; }
7      int value() { return my_value; }
8  public slots:
9      void setValue(int value);
10 public signals:
11     void valueChanged(int newValue);
12 private:
13     int my_value;
14 };
```

Signaux et slots - Exemple

Code

```
1 void SyncVal::setValue(int value)
2 {
3     if (value != my_value) {
4         my_value = value;
5         emit valueChanged(value);
6     }
7 }
```

Signaux et slots - Exemple

Code

```
1  SyncVal a, b;
2  QObject::connect(&a, SIGNAL(valueChanged(int)),
3                   &b, SLOT(setValue(int)));
4
5  a.setValue(12);
6  b.setValue(48);
7  QObject::connect(&b, SIGNAL(valueChanged(int)),
8                   &a, SLOT(setValue(int)));
9  b.setValue(42);
```

Signaux et slots - Exemple

Code

```
1  SyncVal a, b;
2  QObject::connect(&a, SIGNAL(valueChanged(int)),
3                   &b, SLOT(setValue(int)));
4
5  a.setValue(12); // a.value = 12, b.value = 12
6  b.setValue(48); // a.value = 12, b.value = 48
7  QObject::connect(&b, SIGNAL(valueChanged(int)),
8                   &a, SLOT(setValue(int)));
9  b.setValue(42); //infini ou a.value=42, b.value=42 ?
```

Quand utiliser les signaux et slots

- Pour les interfaces graphiques
- Lorsque vous concevez vos propres objets :
  - Dès que vous connaissez une modification qui est susceptible d'intéresser un autre objet, vous emettez un signal
  - Un autre programmeur qui utiliserait votre objet peut ainsi utiliser les signaux que vous avez émis très simplement, en les branchant sur des slots de son propre objet

Signaux et slots - Conclusion

- Grande flexibilité/réutilisabilité
- Aucune des extrêmités n'a besoin de connaître la logique interne de l'autre
- Formalisme simple, implémentation complexe
- Application Qt = 100% C++ mais...
  - Nécessite un outil supplémentaire lors de la compilation : le "MOC" (Meta Object Compiler) qui génère du code C++ compatible
  - Utilise lors de la précompilation des macros (SIGNAL, SLOT, Q_OBJECT, etc.)
  - Génère du code supplémentaire (tables de signaux /slots)
    - exemple : un fichier en-tête *UneClass.h* d'une class utilisant des signaux/slots sera précompilé par l'utilitaire MOC et produira un fichier cpp *moc_uneClass.cpp*.

Sommaire

Panorama

- Le framework Qt est constitué/découpé en plusieurs modules
- Chaque module contient des classes soit "nouvelles" soit "réimplémentées" préfixés par Q (ça reste du C++) en UpperCamelCase
- Deux modules de bases :
  - QtCore : Classes de base non graphiques utilisées par les autres modules (structures de données, manipulation de fichiers, exceptions, etc.)
  - QtWidgets : Composants d'interfaces graphiques (Graphical User Interface)

  A cheval sur ces deux modules : le modèle **MVC** !

- Et c'est aussi : QtMultimedia, QtNetwork, QtOpenGL, QtOpenVG, QtScript, QtScriptTools, QtSql, QtSvg, QtWebKit, QtXml, QtXmlPatterns, QtDeclarative, Phonon, Qt3Support, QtDesigner, QtUiTools, QtHelp, QtTest, QAxContainer, QAxServer et QtDBus.

QtCore

Les classes importantes :

- QObject : classe de base
- QVariant :
  - $\approx$ union au sens C++ de plusieurs types de base
  - permet de stocker n'importe quel type de variable sous une forme abstraite
  - Possibilité d'étendre QVariant avec ses propres types
- QtConcurrent::Exception : les exceptions

QtCore

Les classes provenant de la STL (Conteneurs, Iterateurs et Algorithmes) et même plus :

- Tableau : QVector, QBitArray, QByteArray
- Liste : QList, QLinkedList
- File de priorités : QQueue, QStack
- Tableau associatif : QMap, QMultiMap, QHash, QMultiHash
- Ensemble : QSet
- Chaîne de caractères : QString, QStringList, QConstString, QChar

QtCore

Les classes pratiques et utiles :

- Dates et heures : QDate, QDateTime, QTime
- QDebug : Flux de sortie pour le débogage d'informations
- QUrl : Interface de commodité pour travailler avec des URL
- Créer/Gérer des threads : Qthread, QThreadPool, QWaitCondition, Qsemaphore, Qmutex

QtCore

Les classes pratiques et utiles :

- Accès aux fichiers et dossiers : QFile, QDir, QFileInfo, QDirIterator, ...
- Minuterie : QTimer, QTimerEvent, ...
- Les flux : QTextIStream, QTextOStream, QTextStream, ...
- Pointeurs intelligents : QPointer, QSharedDataPointer, QExplicitlySharedDataPointer, ...

QtWidgets

Avant tout : QApplication :

- gére l'ensemble des paramètres et des affichages d'une application
- contient entre autre la boucle principale de traitement des événements
- doit être créé avant tout objet graphique et reçoit tous les paramètres transmis à la fonction main

Code

```
1 int main(int argc, char *argv[])
2 {
3     QApplication a(argc, argv);
4 ....//affichage de fenêtres graphiques
5     return a.exec();//Boucle d'attente des événements
          jusqu'à la fermeture du dernier widget
6 }
```

QtWidgets

Composant graphique : la classe QWidget

- elle hérite de QObject
- C'est la classe mère de toutes les classes servant à réaliser des interfaces graphiques
- Capable : de recevoir les événements souris/clavier, d'être peint, placé selon un axe z, ...
- Possède les métodes : show, close, repaint, ...

Code

```
1int main(int argc, char *argv[])
2{
3    QApplication a(argc, argv);
4    QLabel hello("Test Qt !");
5    hello.show();
6    return a.exec();
7}
```

QtWidgets

Code

```
1int main(int argc, char *argv[])
2{
3    QApplication a(argc, argv);
4    QWidget box;
5    QPushButton *quitBtn =new QPushButton("Quit",&box);
6    QObject::connect(quitBtn, SIGNAL(clicked()), &a,
         SLOT(quit()));
7    box.show();
8    return a.exec();
9}
```

QtWidgets

Les *layouts* :

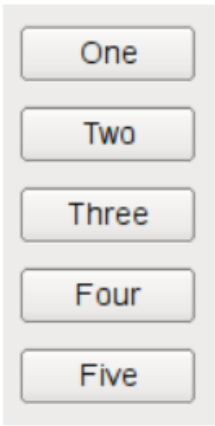
- système de disposition (organisation et le positionnement automatique des widgets enfants dans un widget)
- Ensemble de classes QxxxLayout
- Doit assurer :
  - Positionnement des widgets enfants
  - Gestion des tailles (minimale, préférée)
  - Redimensionnement
  - Mise à jour automatique lorsque le contenu change

QtWidgets

Les *layouts* :

- Ajouter des widgets dans un layout :
  void QLayout::addWidget(QWidget *widget)
- Associer un layout à un widget :
  void QWidget::setLayout (QLayout *layout)

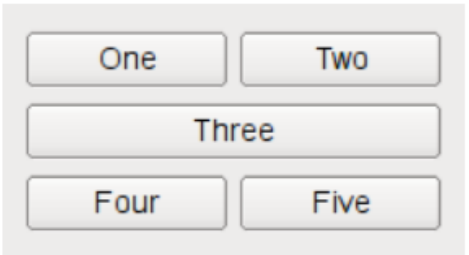
QHBoxLayout



QVBoxLayout

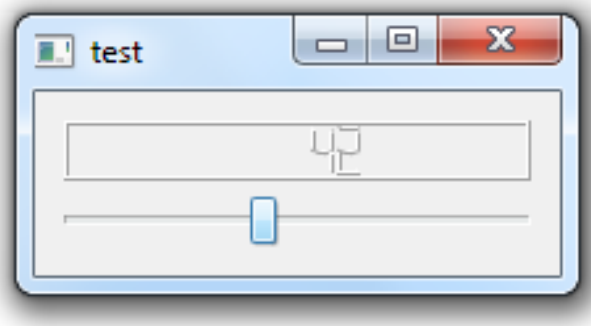
QFormLayout

QGridLayout



QtWidgets

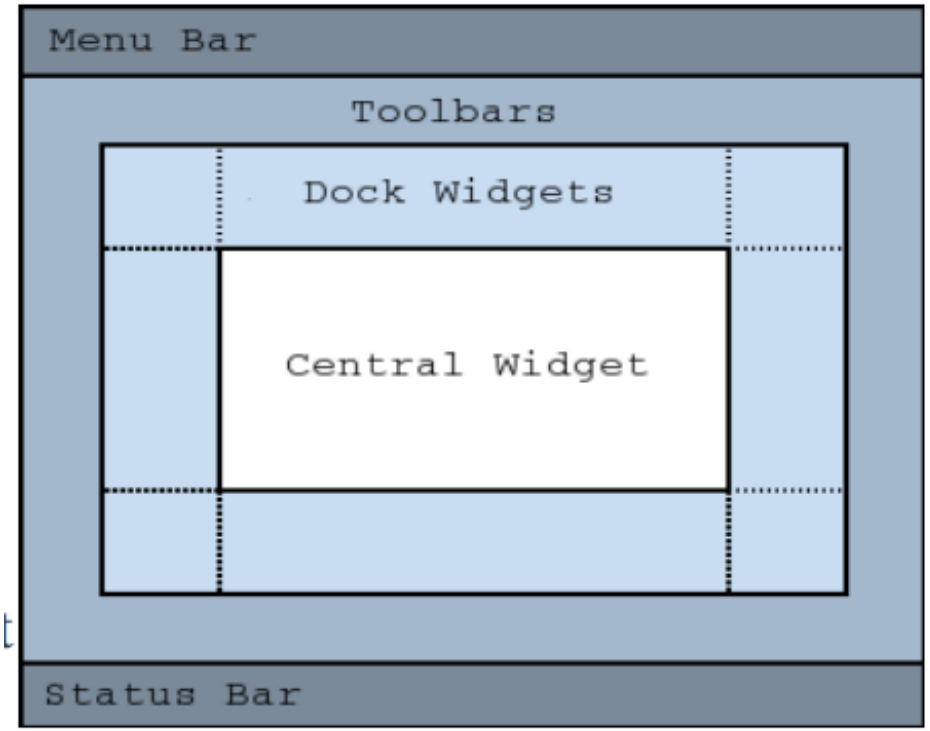
```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QWidget box;
    QLCDNumber* lcd = new QLCDNumber( &box);
    QSlider * slider = new QSlider( Qt::Horizontal, &box );
    QVBoxLayout *mainLayout = new QVBoxLayout(&box);
    mainLayout->addWidget(lcd);
    mainLayout->addWidget(slider);
    box.setLayout(mainLayout);
    QObject::connect( slider, SIGNAL(valueChanged(int)), lcd, SLOT(display(int)) );
    box.show();
    return a.exec();
}
```



QtWidgets

Fenêtre d'application principale : QMainWindow

- Propre mise en page à laquelle vous pouvez ajouter QToolBars, QDockWidgets, un QMenuBar, et un QStatusBar.



QtWidgets

Fenêtre d'application principale : QMainWindow

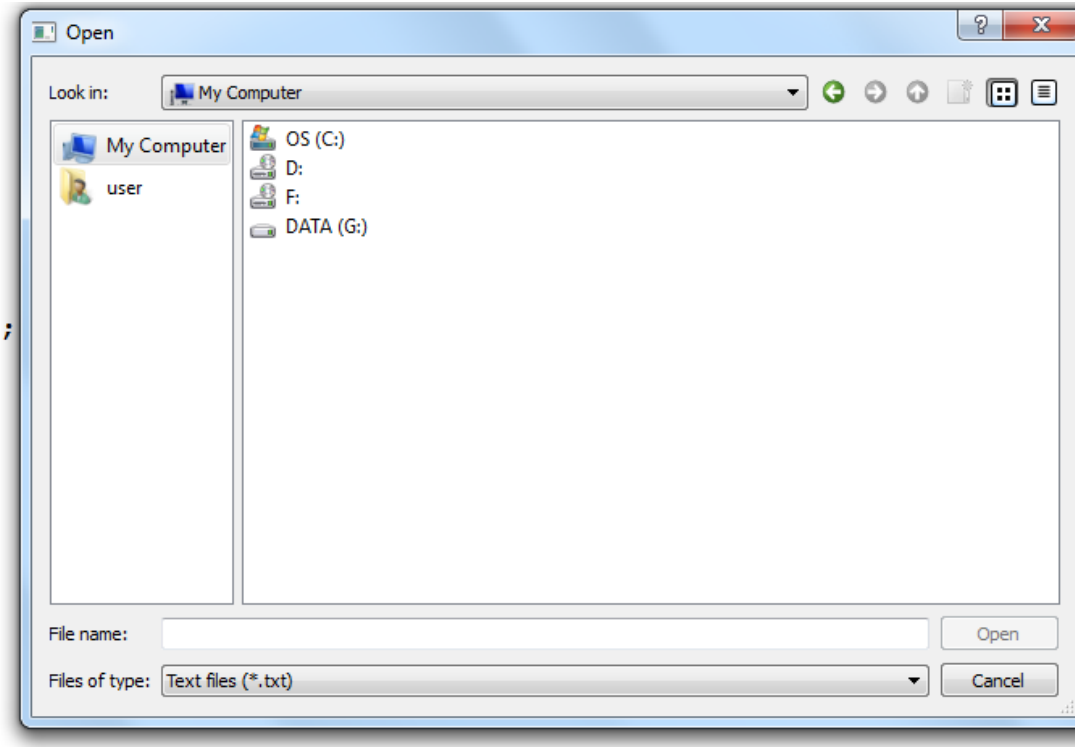
- Soit une interface unique (SDI pour Single Document Interface)
- Soit multiple interfaces (MDI pour Multiple Document Interface), dans ce cas le widget central sera QMdiArea

Mais il existe aussi des fenêtres "courantes" avec QDialog :

- QFileDialog,
- QMessageBox,
- QColorDialog,
- QFontDialog,
- QProgressDialog,
- QErrorMessage ...

QtWidgets

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QWidget box;
    QFileDialog dialog (&box);
    dialog.setFilter ("Text files (*.txt)");
    QStringList file_names;
    if (dialog.exec() == QDialog::Accepted)
    {
       file_names = dialog.selectedFiles();
       QString first_name = file_names[0];
       //..........
    }
    box.show();
    return a.exec();
}
```



QtWidgets

D'autres points :

- QStyle : classe de base abstraite qui encapsule le *look and feel* (changer de style grâce à setStyle())
- QAction : fournit une interface abstraite pour décrire une action (commandes communes peuvent être invoquées via des menus, boutons, et des raccourcis clavier, ...)
- QMenu et QMenuBar : pour faire les traditionnels menu
- QToolBar : barre d'outils qui contient un ensemble de contrôles (généralement des icônes) et située sous les menus
- QStatusBar : fournit une barre horizontale appropriée pour la présentation des informations d'état
- QWebView : afficher une page web (à partir de son URL par exemple)
- etc.

QtWidgets

Avec QtCreator (et QtDesigner), c'est simple !

Petite démo rapide.