Qt ? Qu'est-ce ? <1>

- Qt (cute) est un ensemble de bibliothèques (librairies), permettant (en C++) de développer des applications avec des composants graphiques (GUI)
- Qt est complètement orienté objet
- Son développement à débuté en 1994, et sa commercialisation en 1995-1996 par la firme *Trolltech*
- Qt est portable et disponible sur plusieurs plateformes :
  - **MS/Windows** – 95, 98, NT 4.0, ME, 2000, et XP
  - **Unix/X11** – Linux, Sun Solaris, HP-UX, Compaq Tru64 UNIX, IBM AIX, SGI IRIX, ...
  - **Macintosh** – Mac OS X
  - **Embedded** – Linux platforms with framebuffer support
- Racheter fin 2008 par Nokia

Qt ? Qu'est-ce ? <2>

- Qt est disponible sous plusieurs licences :
  - **Qt Enterprise Edition** et **Qt Professional Edition** destinés aux développements à but commerciaux
  - **Qt Open Source Edition** disponible pour Unix/X11, Macintosh et Embedded Linux. Cette version est distribuée avec la *Q Public License* et la *GNU General Public License* pour les développements libres.
- Son plus bel exemple d'application dans le monde du libre est sans doute l'environnement **KDE** (`http://www.kde.org`)
- Il existe quelques «bindings» pour d'autres langages tels que Perl et Python et C#

Où se documenter ?

- Les livres :
  - Le livre de référence en anglais <<C++ GUI Programming with Qt 4, Second Edition»
  - La première version et la seconde version du livre sont disponible en ligne sur : `http://www.qtrac.eu`
  - Un livre en français : «Qt4 et C++ : Programmation d'interfaces GUI»
- Sur le Web :
  - Le site officiel de Nokia : `http://doc.qt.nokia.com/`
  - Un site d'aide et de tutoriaux en français : `http://www.qtfr.org`
  - Un forum d'aide : `http://www.qtforum.org`

Un survol des possibilités de Qt <1>

- Les applications Qt possèdent un «look and feel» natif sur toutes les plateformes supportées
- Pour le portage entre les systèmes les plus connus (Linux, Windows, Mac OS X, ...) une simple recompilation est nécessaire
- Le portage peut aussi être effectué pour des environnements embarqués
- Qt intègre le support de la 3D grâce à des composants OpenGL
- Qt utilise l'Unicode et possède des composants dédiés à l'internationalisation (*QString*, *Qt Linguist*)

Un survol des possibilités de Qt <2>

- Qt permet l'utilisation de SGBD, indépendamment de la plateforme d'utilisation. Il est compatible notamment avec : Oracle, SQLite, PostgreSQL, MySQL, ...
- Qt inclut un grand nombre de classes spécialisées pour un domaine ; il permet notamment la gestion directe du XML grâce à des parseurs SAX et DOM
- Qt inclue aussi des classes dédiées aux fonctionnalités réseaux et supporte les protocoles standard
- Qt fournit sa propre implémentation de la STL pour les compilateurs qui n'en seraient pas pourvus
- Pour faciliter la création des applications, le programme **Qt Designer** permet de créer rapidement les interfaces élémentaires

Un survol des possibilités de Qt <3>

- Pour étendre Qt à de nouveaux composants et fonctionnalités, Qt propose un système d'extension (**Meta Object System**) et un compilateur (**Meta Object Compiler (MOC)**) qui permettent d'augmenter la puissance de la bibliothèque
- Pour faciliter la création des Makefiles et autres fichiers de compilation, Qt propose l'outil **qmake** qui génère tous les fichiers nécessaires à compilation «propre»

Le modèle objet de Qt

- Qt est basé autour du modèle d'objet de Qt
- Son architecture est entièrement fondée sur la classe `QObject` et de l'outil **moc**
- En dérivant des classes de `QObject`, un certain nombre d'avantages sont hérités :
  - Gestion facile de la mémoire
  - Signaux et Slots
  - Propriétés
  - Introspection
- Qt, c'est uniquement du C++ standard avec quelques macros (d'où la portabilité)

La gestion «facile» de la mémoire

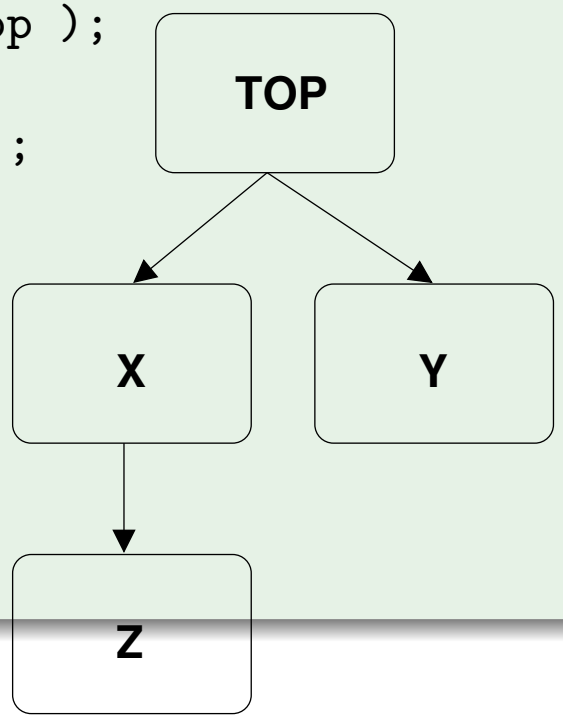
- En créant une instance d'une classe dérivée de `QObject`, il est possible de passer un pointeur vers un objet parent au constructeur
- Quand un parent est supprimé, ses enfants sont supprimés aussi
- **Il ne faut donc pas détruire manuellement les enfants !**

Exemple

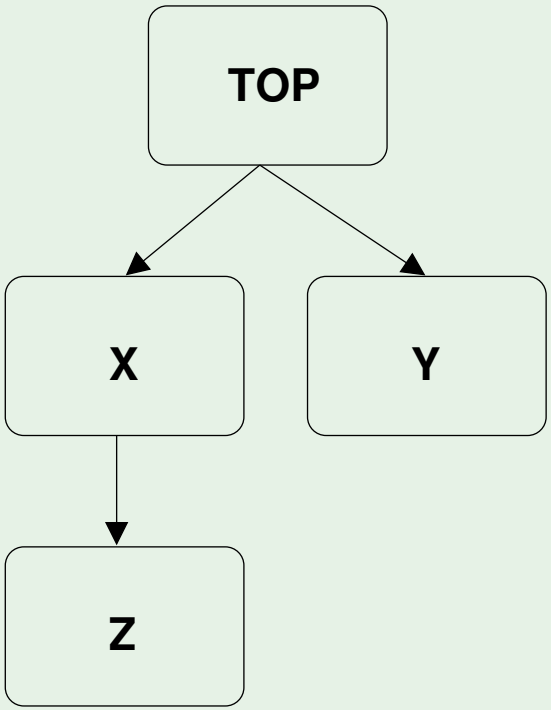
```
class VerboseObject : public QObject
{
 public:
   VerboseObject(QObject *parent=0)
   : QObject(parent)
     { std::cout << "Created" << std::endl; }
   ~VerboseObject()
     { std::cout << "Deleted: " << objectName().toStdString() <<
       std::endl; }
   void doStuff()
     { std::cout << "Do stuff: " << objectName().toStdString() <<
       std::endl; }
};
```

Exemple

```
int main( int argc, char **argv )
{
 QApplication a( argc, argv );
 VerboseObject top( 0 );
 top.setObjectName("top");
 VerboseObject *x = new VerboseObject( &top );
 x->setObjectName("x");
 VerboseObject *y = new VerboseObject( &top );
 y->setObjectName("y");
 VerboseObject *z = new VerboseObject( x );
 z->setObjectName("z");
 top.doStuff();
 x->doStuff();
 y->doStuff();
 z->doStuff();
 return 0;
}
```



Exemple



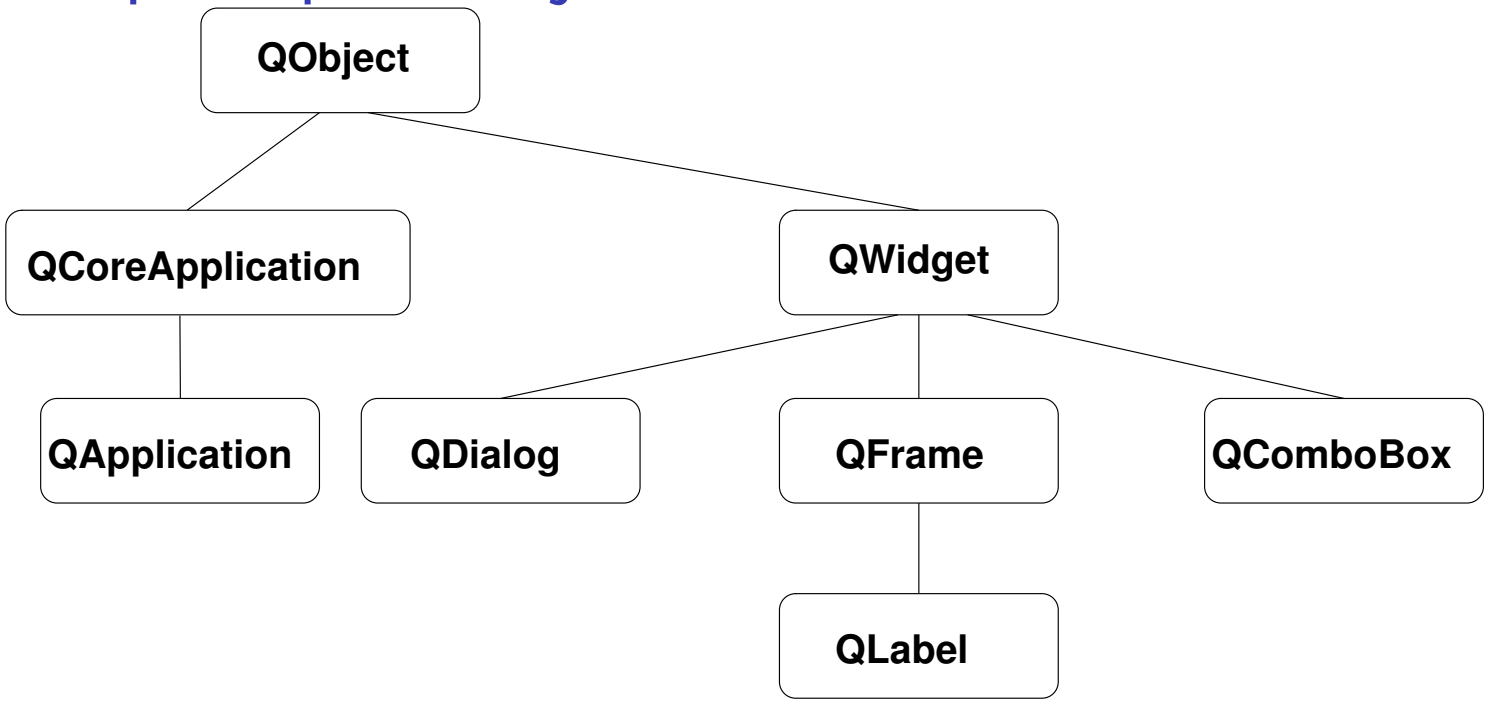

```
$ ./mem
Created
Created
Created
Created
Do stuff: top
Do stuff: x
Do stuff: y
Do stuff: z
Deleted: top
Deleted: x
Deleted: z
Deleted: y
```

- Si la référence du parent est enlevée de `x` et de `y`, une fuite de mémoire se produit

L'arbre d'objets

- Tous les `QObjects` s'organisent sous forme d'arbre
- Lors de la création d'un `QObject` avec un parent, le fils est ajouté à la liste `children()` du parent (destruction en cascade)
- La classe `QWidget` est la base de tous les composants affichables sur l'écran
  - Un `QWidget` contenant tous ses enfants : le fils d'un `QWidget` ne peut pas sortir des limites graphiques de son parent
- Les fonctions statiques `QObject::dumpObjectTree()` et `QObject::dumpObjectInfo()` sont souvent utiles au débogage de l'application

Arbre des principaux objets



- La classe `QApplication` permet interfacer l'application avec le système : **chaque application en possède une et une seule !**
- `QApplication` implémente la boucle principale d'évènements
- L'objet `QApplication` a besoin d'un composant principal sur lequel renvoyer les évènements capturés

Utilisation des «widgets»

- Qt fournit un panel complet de widgets
- Les widgets de Qt ne sont pas séparés en «controls» et «containers» ; chaque widget peut être utilisé comme «control» et «container»
- Tout widget adapté (personnalisé) est créé par héritage d'un widget existant
- Un widget peut contenir plusieurs widgets fils :
  - Les widgets fils ne peuvent s'afficher en dehors du parent
- Un widget sans parent est un «top-level» (racine) widget
- Un widget peut être positionné de manière automatique, grâce aux «layout managers» (gestionnaire de placement) ou manuellement
- Quand un widget est désactivé, caché ou détruit, la même action est appliquée récursivement à ses enfants

Premier exemple d'application

Exemple (Le «hello world !»)

```
#include <Qt/QtGui>
int main( int argc, char **argv )
{
 QApplication app( argc, argv );
 QLabel hello( "<font color=blue>Hello <i> \
               world!</i> </font>", 0 );
 hello.show();
 return app.exec();
}
```

Remarques sur l'exemple

- L'application créée sa propre fenêtre et appelle la boucle infinie de capture d'évènements
  - La boucle de capture des évènements est incluse dans l'appel à `app.exec()`
- Les évènements sont capturés par l'application, il ne reste qu'à leur associer des réactions aux widgets
- Remarques sur les widgets :
  - Chaque classe commence par un 'Q'
  - Pour chaque classe correspond un fichier de déclarations à inclure
  - Conséquence : la liste des fichiers à inclure augmente rapidement ...

Les widgets utilisateur courants <1>

- Pour concevoir une interface graphique, il faut connaître les widgets courants de la librairie graphique utilisée :

Widget	Description
`QLabel`	Affiche du texte et des images
`QCheckBox`	Une boîte à cocher avec une étiquette
`QLCDNumber`	Afficheur digital
`QLineEdit`	Une simple ligne d'édition
`QTextEdit`	Version plus riche, multi-ligne du widget précédent
`QListView`	Une liste d'éléments à une colonne (peut être «scrollée»)
`QMenu`	Tous les types de menus (normaux, pop up)
`QProgressBar`	Barre de progression pour les opérations longues
`QPushButton`	Bouton à cliquer (avec du texte ou une image)

Les widgets utilisateur courants <2>

Widget	Description
`QRadioButton`	Bouton radio (on/off en cercle) avec étiquette
`QScrollBar`	Barre de défilement horizontale ou verticale
`QSlider`	Barre de niveau horizontale ou verticale
`QSpinBox`	Boîte de saisie avec des flèches haut/bas
`QToolButton`	Un bouton à cliquer formaté pour les barres d'outils
`QWhatsThis`	Fournit une fenêtre d'informations sur les widgets

Les widgets utilisés pour grouper des widgets

Widget	Description
`QFrame`	Classe de base pour les widgets avec frames
`QGroupBox`	Groupe horizontalement ou verticalement les éléments dans un cadre (ligne)
`QHBoxLayout`	Groupe horizontalement les éléments sans frame
`QVBoxLayout`	Groupe verticalement les éléments sans frame
`QMenuBar`	Barre contenant les menus
`QToolBar`	Barre contenant des boutons spécifiques
`QMainWindow`	Grande fenêtre principale plus menus et barre de boutons
`QDialog`	Frame qui apparait en popup contenant d'autres widgets

Gestionnaire de placement - « Layout Manager »

- Un des problèmes lors de la conception d'une interface graphique concerne le placements des widgets :
  - Le placement initial
  - Le comportement lorsque la fenêtre s'agrandit ou se rétrécit
- Pour éviter ces problèmes : utiliser un « Layout Manager »
- Qt propose deux types de layout managers :
  - `QBoxLayout` qui range les widgets soit verticalement (`QVBoxLayout`) soit horizontalement (`QHBoxLayout`)
  - `QGridLayout` qui range les widgets comme dans un tableau
- Les classes `QXXLayout` dérivent directement de `QObject`, ce ne sont pas des widgets !

La classe `QBox` permet de ranger les widgets, mais lors d'un agrandissement ou rétrécissement de la fenêtre, les widgets ne sont pas replacés.

Les « QBoxLayout » <1>

- Reçoit la place que prend le widget parent et le divise en petit carrés destinés à contenir les widgets à placer
- `QBoxLayout` est très rarement utilisé, on utilise plutôt les classes `QHBoxLayout` et `QVBoxLayout` qui en héritent directement
- Les widgets sont rangés selon l'ordre dans lequel ils sont ajoutés au layout
- Il est possible de passer au constructeur un entier qui initialisera le nombre de cases prévues au départ
- Les layouts peuvent être imbriqués grâce à la méthode `addLayout` ou encore en l'initialisant avec un layout comme parent

Les « QBoxLayout » <2>

Exemple

```
#include <Qt/QtGui>
int main (int argc, char* argv[])
{QApplication app(argc, argv);
 QWidget w;
 QHBoxLayout * hbox_1 = new QHBoxLayout(&w);
 QPushButton *b1 = new QPushButton("Un", &w);
 QPushButton *b2 = new QPushButton("Deux", &w);
 hbox_1->addWidget(b1);
 hbox_1->addWidget(b2);
 w.show();
 return app.exec();
}
```

Le « QGridLayout »

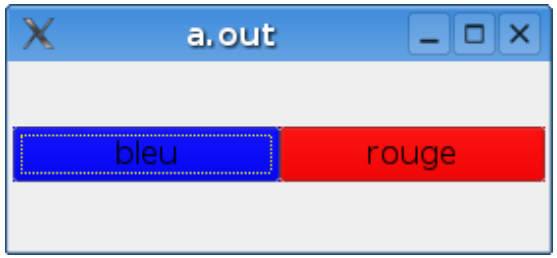
- Un `QGridLayout` divise l'espace comme un tableau où chaque cellule est identifiable par son numéro de ligne et de colonne (débutant à 0)
- Le nombre de lignes et de colonnes s'adapte en fonction des widgets placés
- Il existe plusieurs constructeurs permettant notamment de définir initialement le nombre de lignes et de colonnes
- Les cases vides sont acceptées
- La largeur minimale d'une colonne peut être définie avec : `setColumnMinimumWidth( int col, int space )`
- La hauteur minimale d'une ligne peut être définie avec : `setRowMinimumHeight( int row, int space )`

Exemple

```
#include <Qt/QtGui>
int main (int argc, char* argv[])
{QApplication app(argc, argv);
 QWidget w;
 QGridLayout *grid = new QGridLayout(&w);
 QPushButton *b1 = new QPushButton("Un", &w);
 QPushButton *b2 = new QPushButton("Deux", &w);
 QPushButton *b3 = new QPushButton("Trois", &w);
 grid->addWidget(b1,0,0);
 grid->addWidget(b2,0,1);
 grid->addWidget(b3,1,0,1,2);
 w.show();
 return app.exec();
}
```

Les propriétés des widgets utilisées par le layout manager <1>

- Les widgets intègrent des propriétés qui déterminent leur comportement lors de leur placement et dimensionnement
  - Exemple : *la taille des boutons*



- Les propriétés de redimensionnement (taille) sont personnalisables grâce à la classe `QSizePolicy`
- Les espacements entre widgets peuvent être modifiés :
  - Soit en spécifiant des marges dans les layouts (`setMargin(int)` et `setSpacing(int)`)
  - Soit en utilisant ponctuellement les méthodes `addSpacing(int)` et `insertSpacing(int,int)` ajoutant un espace dans le layout

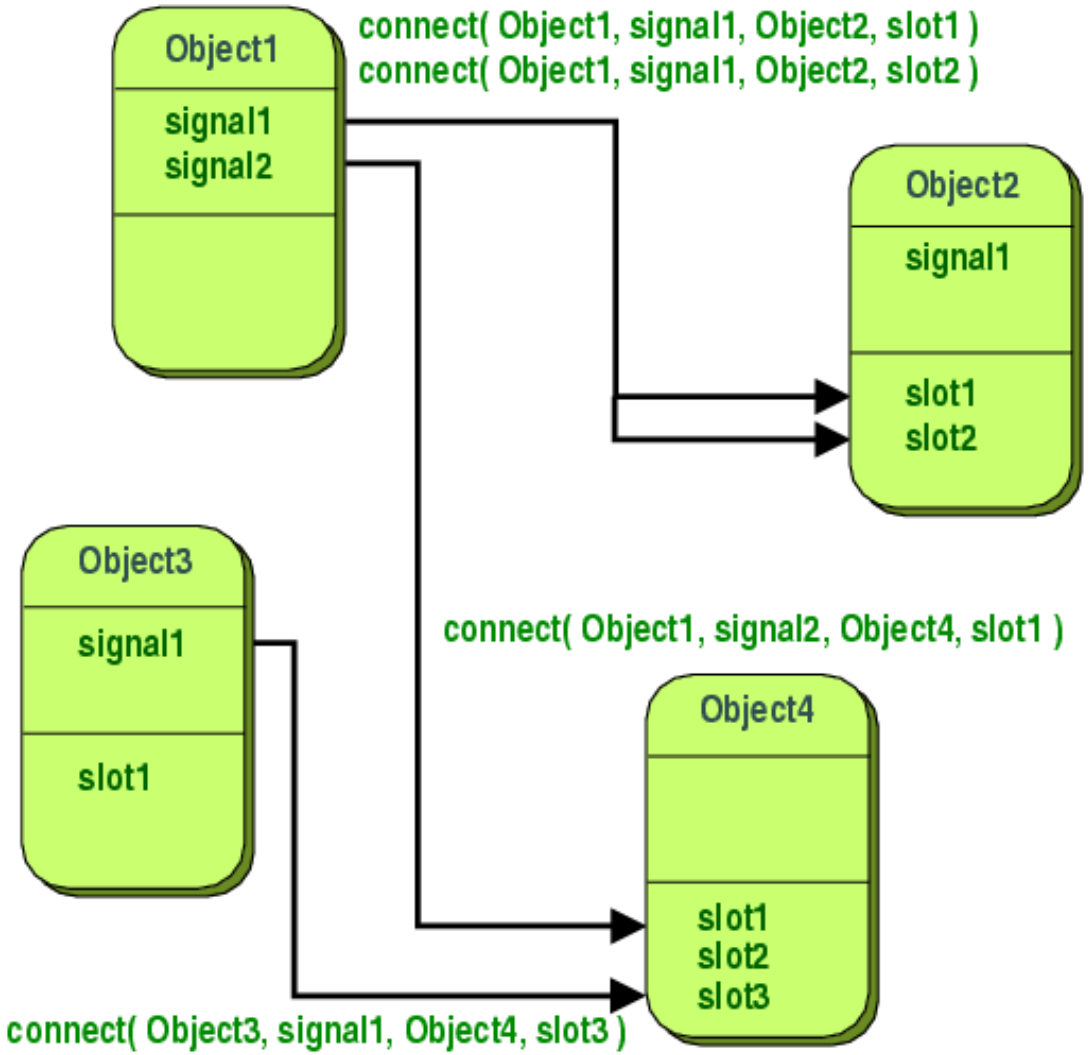
Les propriétés des widgets utilisées par le layout manager <2>

- Lorsque l'on souhaite que ce soit les espacements qui varient et non pas la taille des widgets on peut utiliser des élastiques
- Les méthodes `addStretch(int)` et `insertStretch(int,int)` permettent d'ajouter un élastique à un widget avec un poids passé en paramètre
- Toutes les commandes d'élastiques vues s'adaptent aussi sur les `QGridLayout` en utilisant les méthodes : `setRowStretch`, et `setColumnStretch`

Introduction

- Toute application (graphique) doit permettre de traiter les évènements
  - Exemple : le clic sur un bouton déclenche la fermeture de l'application
- Pour cela, Trolltech a créé pour Qt une solution appelée « signals and slots »
- Les signaux et slots sont des mécanismes génériques de communication inter-objets
- Ils sont utilisés pour remplacer les « anciens » mécanismes de « callbacks » :
  - Un callback consiste en un pointeur sur une fonction passé à un bouton (par exemple) qui est appelée lorsque que le bouton est cliqué
  - Aucune vérification du typage par le compilateur !
  - Il est difficile de développer des classes génériques indépendantes des évènements qu'elles traitent

Fonctionnement <1>

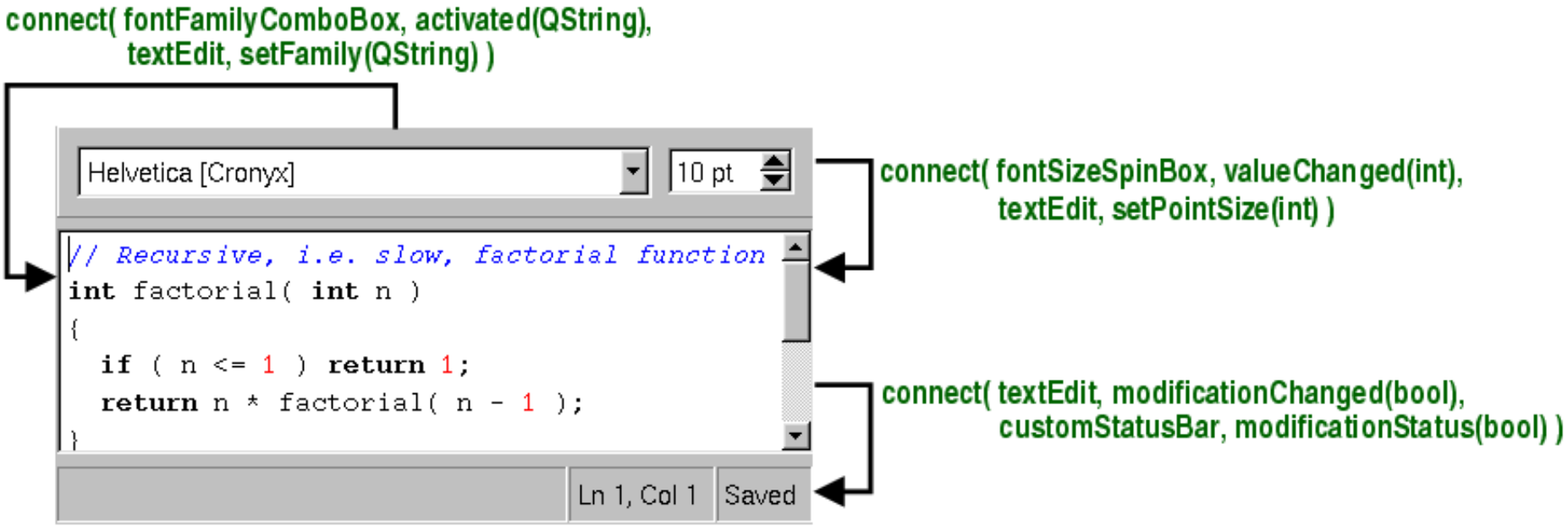


Fonctionnement <2>

- Un signal est émis quand un évènement se produit
- Un slot est une méthode (ou fonction) appelée en réponse à un signal particulier
- Qt intègre un grand nombre de slots prédéfinis, mais il est possible de créer ses propres slots (et signaux)
- Le mécanisme des signaux et slots est « type-safe », et les erreurs de typage sont reportées comme avertissement lors de la compilation
- La méthode `QObject::connect` permet d'associer un slot à un signal
- Les connexions peuvent être ajoutées et enlevées à n'importe quel moment durant l'exécution

Fonctionnement <3>

- Toutes les classes héritant de `QObject` peuvent contenir des signaux et des slots
- Il est possible de connecter un signal à plusieurs slots, et plusieurs signaux à un seul slot :



Exemple

Exemple

```
#include <Qt/QtGui>
int main (int argc, char* argv[])
{
 QApplication app(argc, argv);
 QPushButton bouton("Fermer", 0);
 QObject::connect( &bouton, SIGNAL( clicked() ), &app,
                   SLOT( quit() ));
 bouton.show();
 return app.exec();
}
```

Les signaux

- Les signaux sont émis par un objet lorsque son état change
- Seules les classes définissant un signal peuvent émettre **ce** signal

Remarque

Les signaux sont déclarés, mais ils ne sont jamais implémentés dans le .cpp ! Leur implémentation est générée automatiquement par le moc (Meta Object Compiler).

- Le concepteur peut émettre un signal particulier grâce au mot clef `emit`
- Quand un signal est émis, les slots connectés sont exécutés immédiatement
- Si plusieurs slots sont connectés à un signal, leur ordre d'exécution est **arbitraire** !

Les slots

- Un slot est appelé dès que le signal connecté est émis
- Comme les slots sont des méthodes normales, ils possèdent les mêmes droits que d'autres méthodes classiques
- Comme les autres attributs d'une classe, les slots peuvent être déclarés `private`, `protected` et `public`
- Les slots peuvent aussi être virtuels (`virtual`)
- Les slots sont plus lents que les fonctions de « callback », mais la différence n'est pas décelable par l'utilisateur

Créer ses propres signaux et slots <1>

- Les signaux et slots définis dans Qt ne sont pas suffisants
- Qt propose des solutions pour définir ses propres signaux et slots, mais attention
  - Il n'est pas possible de définir une classe qui ne contient que des slots ou des signaux
  - Les signaux et slots ne peuvent être déclarés que dans des classes qui héritent de `QObject`
  - Dans toute classe définissant des signaux ou slots, il faut placer la macro `Q_OBJECT`
  - Les paramètres par défaut ne sont pas utilisables pour les signaux et les slots

Créer ses propres signaux et slots <2>

Définition

```
class MyClass : public QObject
{
      Q_OBJECT // Macro nécessaire

      signals: // Signaux définis manuellement
            void signal1( int, const char * );

      private slots: // Slots privés définis manuellement
            void slot1( QString &qStr );
      public slots: // Slots publics définis manuellement
            void slot2( );

      private: // Méthodes et attributs privés
      public: // Méthodes et attributs publiques
};
```

Créer ses propres signaux et slots <3>

- Cette classe est différente d'une classe C++ standard
  - Elle hérite obligatoirement de `QObject`
  - Elle fait appel à la macro `Q_OBJECT`
  - De nouveaux mots clefs apparaissent : `signals`, `private slots`, `...`
  - Les méthodes définies dans les sections slots retournent **obligatoirement** le type void
- Ce type de classe ne peut pas être compilé avec le compilateur C++ habituel :
  - Il est nécessaire d'utiliser en premier lieu le compilateur `moc`
  - `moc` remplace tous les mots clefs spécifiques par des commandes et appels C++ implémentant le comportement voulu de l'objet

Créer ses propres signaux et slots

Exemple

Exemple

Fichier .h

```
class Foo : public QObject
{
 Q_OBJECT
 public:
   Foo();
   int value() const {return val;}
 public slots:
   void setValue( int );
 signals:
   void valueChanged( int );
 private:
   int val;
};
```

Fichier .cpp

```
void Foo::setValue(int v)
{
 if ( v != val ) {
   val = v;
   emit valueChanged(v);
 }
}
```

Créer ses propres signaux et slots

Exemple (suite)

Exemple

```
Foo a, b;
connect(&a, SIGNAL(valueChanged(int)), &b, SLOT(setValue(int)));
b.setValue( 11 ); // a == indéfini b == 11
a.setValue( 79 ); // a == 79 b == 79
b.value(); // renvoie 79
```

Présentation des outils

- Qt est fourni avec un grand nombre d'outils graphiques ou en ligne de commande pour accélérer les phases de développement :
  - **qtconfig** - permet de modifier des paramètres communs à toutes les applications `Qt`
  - **Qt Designer** - permettant de dessiner les widgets en mode graphique
  - **Qt Linguist, lupdate et lrelease** - utilisés pour la traduction des messages de l'application développée
  - **Qt Assistant** - un outil graphique d'aide sur Qt
  - **qmake** - chargé de créer des *Makefiles* appropriés
  - **qembed** - permettant la conversion de fichiers images (par exemple) en code C++
  - **rcc** - permet d'incorporer des données extérieures (icônes, images, ...) à la compilation
  - **moc** - le Meta Object Compiler
  - **uic** - le User Interface Compiler

Les fonctionnalités de déboggage <1>

- Messages d'avertissements :
  - `qDebug( const char * msg, ... )` qui affiche un message de déboggage
  - `qWarning( const char * msg, ... )` qui affiche un avertissement
  - `qFatal( const char * msg, ... )` qui affiche un message d'erreur puis quitte l'application
- Toutes ces fonctions affichent leur message sur la sortie d'erreur standard (sous Unix) : `stderr`
- Il est aussi possible de rediriger et reformater les sorties de ces fonctions grâce à l'utilisation de `qInstallMsgHandler`

Exemple

```
void myMessageOutput( QtMsgType type, const char *msg )
{ switch ( type ) {
  case QtDebugMsg:
      cerr << "Debug: " << msg << endl;
      break;
  case QtWarningMsg:
      cerr << "Warning: " << msg << endl;
      break;
  case QtFatalMsg:
      cerr << "Fatal: " << msg << endl;
      abort(); // core dump délibéré
  }
}
...
qInstallMsgHandler( myMessageOutput );
qDebug("Teste déboggage !!");
```

- `qInstallMsgHandler(0)` restaure le traitement initial

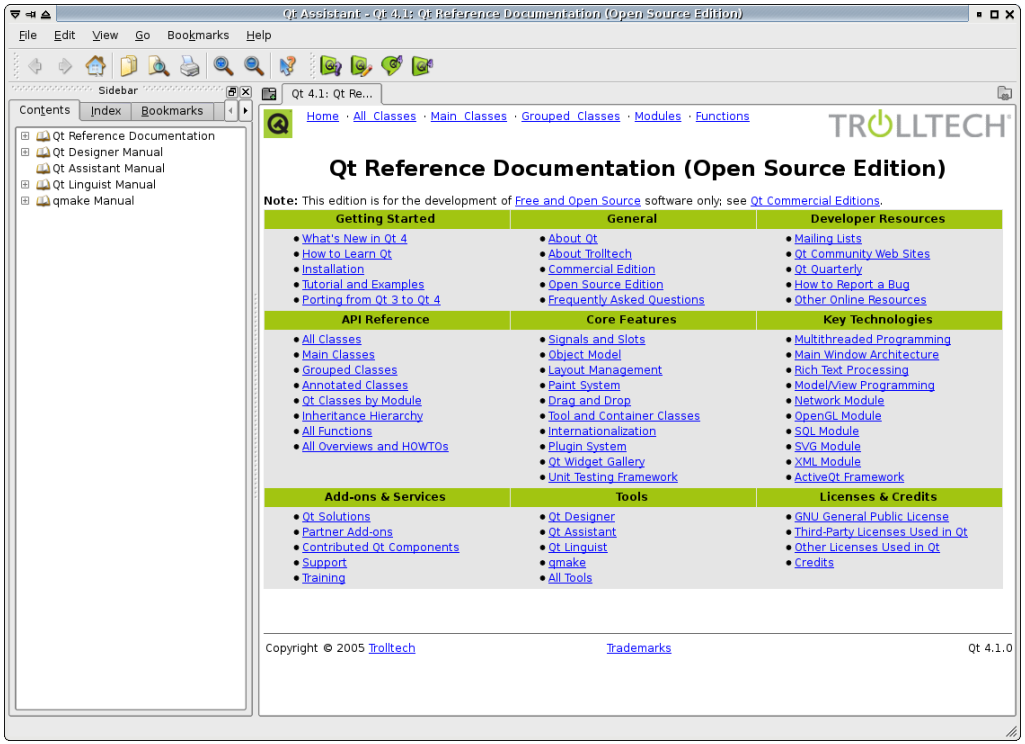
Les fonctionnalités de déboggage <3>

- Qt fournit aussi des macros : `Q_ASSERT(b)` et `Q_CHECK_PTR(p)`
  - `Q_ASSERT(b)` ou `b` est un booléen écrit un assert si `b` est faux
  - `Q_CHECK_PTR(p)` ou `p` est un pointeur écrit un avertissement si `p` est nul

Exemple

```
char *alloc( int size )
{
 Q_ASSERT( size > 0 );
 char *p = new char[size];
 Q_CHECK_PTR( p );
 return p;
}
```

Qt Assistant et Qt Linguist <1>



- Qt Assistant est un outil d'aide pour la documentation de Qt
- Qt Linguist et ses petits outils (`lrelease` et `lupdate`) permettent de traduire simplement le texte d'une application Qt

Qt Assistant et Qt Linguist <2>

1. *Premièrement, il faut créer un fichier projet.pro contenant la liste des fichiers de langue à traiter*
2. *Puis il faut utiliser l'outil `lupdate` qui extrait du code source le texte à traduire*
3. *Ensuite, à l'aide Qt linguist, traduire le texte*
4. *Enfin exécuter la commande `lrelease` qui génère les fichiers traduits prêts à l'emploi*

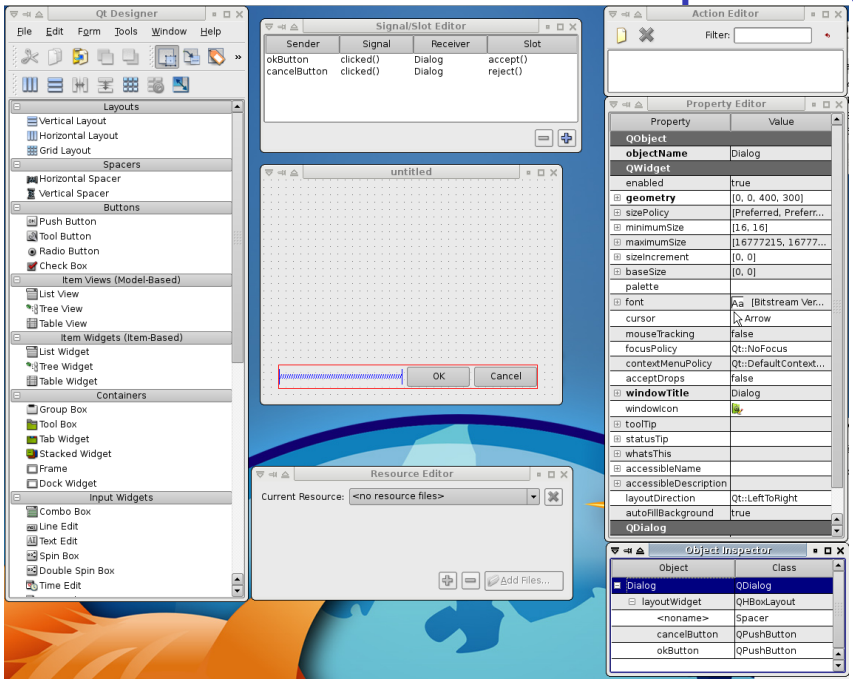
Exemple

```
#include <Qt/QtGui>
int main( int argc, char **argv )
{ QApplication app( argc, argv );
 QTranslator translator( 0 );
 translator.load( "tt1_la", "." );
 app.installTranslator( &translator );
 QPushButton hello( QPushButton::tr("Hello world!"), 0 );
 hello.show();
 return app.exec();
}
```

Exemple (Fichier projet : traduc.pro)

```
SOURCES                 = traduc.cpp
TRANSLATIONS            = tt1_la.ts
```

Qt Designer et le User Interface Compiler (uic)



- Qt Designer permet de dessiner à l'aide de la souris l'interface graphique de l'application
- Le résultat est sauvegardé dans un fichier xml à l'extension .ui
- Le User Interface Compiler (uic) transforme les fichiers .ui généré par Qt designer pour les transformer en fichiers sources ou entêtes C++

le Meta Object Compiler (moc) <1>

- Le Meta Object Compiler (moc) permet de pré-compiler les extensions développées pour Qt
- Il pré-compile tout fichier contenant des instructions du « meta object system », i.e :
  - **Le fichier doit contenir la macro `Q_OBJECT`**
  - Possibilité d'ajouter des slots ou des signaux aux classes héritant de QObject
  - Possibilité d'ajouter des propriétés aux objets aux classes héritant de QObject
  - Possibilité d'utiliser `tr()` (internationalisation) directement sur la classe `QObject`
  - Utilisation des macros : `Q_PROPERTY`, `Q_ENUMS`, `Q_CLASSINFO`, `Q_INTERFACES` et `Q_FLAGS`
  - ...

le Meta Object Compiler (moc) <2>

- Le résultat produit par le moc doit être lié avec le reste des fichiers compilés pour produire le programme exécutable
- **Méthode A :** les instructions du « meta object system » se trouve dans un fichier myclass.h. Le résultat de l'exécution du moc doit être écrit dans un fichier nommé moc_myclass.cpp qui devra être compilé et lié comme d'habitude
- **Méthode B :** les instructions du « meta object system » se trouve dans un fichier myclass.cpp. Le résultat de l'exécution du moc doit être écrit dans un fichier nommé myclass.moc. Ce fichier doit être inclus à la fin du fichier myclass.cpp (`#include "myclass.moc"`) qui sera compilé comme d'habitude
- **La méthode normale est bien sûr la méthode A !**
- Il est conseillé de lire le manuel du moc car il existe des limitations et des cas particuliers

qmake <1>

- qmake est un outil permettant d'automatiser la génération des makefiles pour des applications Qt :
  - Prise en compte du moc, du uic et d'autres spécificités Qt
- qmake utilise un fichier d'information (.pro) pour générer les makefiles appropriés

Exemple (fichier monprojet.pro)

```
SOURCES = hello.cpp
SOURCES += main.cpp
HEADERS = hello.h
CONFIG += qt warn_on release
```

- Exécution : `qmake -o Makefile monprojet.pro`

qmake <2>

Exemple (fichier monprojet.pro)

```
SOURCES = hello.cpp
SOURCES += main.cpp
HEADERS = hello.h
CONFIG += qt warn_on release
```

- `qt` indique à qmake que l'application est construite pour Qt (ajout des librairies et des répertoires d'includes)
- `warn_on` indique à qmake que les warnings du compilateur seront affichés
- `release` indique à qmake que l'application compilée est dédiée à la diffusion. Le programmeur peut remplacer `release` par `debug`
- Il faut toujours utiliser le `+=` pour la partie `CONFIG` car il ne faut pas effacer les options déjà existantes

qmake <3>

- qmake permet d'ajouter des parties spécifiques à chaque plateforme et de tester certaines conditions comme l'existance d'un fichier

Exemple (fichier hello.pro)

```
CONFIG += qt debug
HEADERS += hello.h
SOURCES += hello.cpp
SOURCES += main.cpp
win32 {
     SOURCES += hellowin.cpp
} unix {
     SOURCES += hellounix.cpp
}
!exists( main.cpp ) {
     error( "No main.cpp file found" )
}
```

Les fichiers <1>

- Qt peut charger et sauvegarder des fichiers au format texte, xml et binaires
- Qt gère les fichiers locaux grâce à ses propres composants et les fichiers distants en utilisant les protocoles ftp et http
- Qt manipule les fichiers comme des objets de la classe `QFile`
- `QFile` peut tester l'existence d'un fichier, l'ouvrir, tester si il est déjà ouvert, le fermer et l'effacer
- Les modes d'ouvertures sont : `QIODevice::ReadOnly`, `QIODevice::WriteOnly`, `QIODevice::ReadWrite`, `QIODevice::Append`, ...

Exemple

```
#include <Qt/QtGui>
#include <iostream>
int main( int argc, char **argv )
{QApplication a( argc, argv );
 QFile f( "f.txt" );
 if( !f.exists() )
 { std::cout << "Le fichier n'existe pas." << std::endl;
   return 0;
 }
 if( !f.open( QIODevice::ReadOnly ) )
 { std::cout << "Échec lors de l'ouverture." << std::endl;
   return 0;
 }
 std::cout << "Ca marche." << std::endl;
 f.close();
 return 0;
}
```

Les fichiers et les flux

- Un fichier seul ne sert pas à grand chose, Qt propose des classes de flux prenant un fichier comme entrée :
  - `QTextStream` : permet de gérer des fichiers textes
  - `QDataStream` : permet de gérer des fichiers binaires
- `QDataStream` peut aussi être utilisé pour « sérialiser » des objets
- Les classes `QTextStream` et `QDataStream` peuvent agir avec n'importe quelle objet issu d'une sous-classe de `QIODevice` : `QFile, QBuffer et QTcpSocket`

Exemple

```
QFile file( "splash.dat" );
if ( file.open(QIODevice::WriteOnly) )
{
 QDataStream out( &file );
 out << QString( "SplashWidgetStyle" )
     << QFont( "Times", 18, QFont::Bold ) << QColor( "skyblue" );
 file.close()
}
QString str;
QFont font;
QColor color;
QFile file2( "splash.dat" );
if ( file2.open(QIODevice::ReadOnly) )
{
 QDataStream in( &file2 );
 in » str » font » color;
 file2.close()
}
```

Les zones défilables

- Pour afficher un objet de grande taille, il faut utiliser des barres de défilement : `QScrollbar`
- Pour faciliter l'utilisation de ces barres, Qt propose la classe `QScrollArea`
- La structure d'un `QScrollArea` est la suivante :
  - Un widget à afficher
  - Deux barres de défilement (horizontale et verticale)
- Il est possible d'accéder directement aux différents éléments composant le `QScrollArea` (barre horizontale et verticale)
- Les barres de défilement peuvent apparaître au besoin ou en permanence. Cela dépend de la politique des barres de défilement

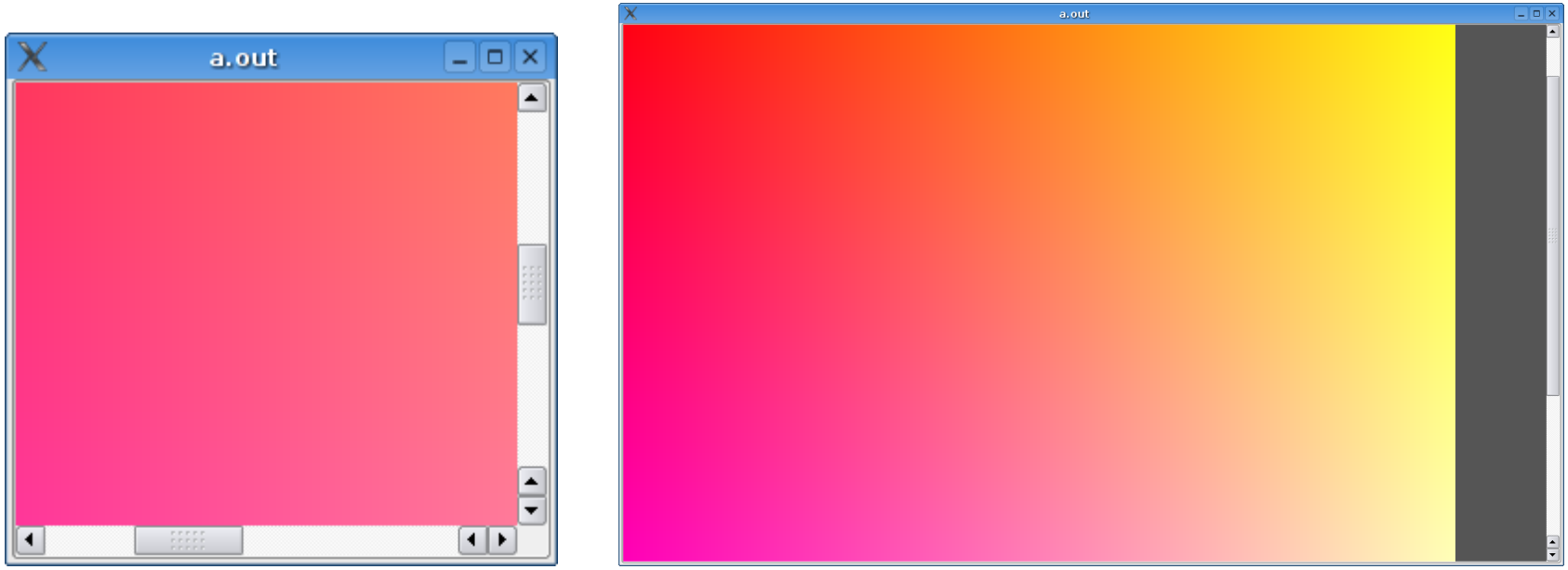
Exemple

```
#include <Qt/QtGui>
class MyWidget : public QWidget
{ public:
    MyWidget(QWidget* parent = 0) : QWidget(parent)
            { resize(1024,1024); }
 protected:
    void paintEvent(
    { QPainter p(thi
     p.setPen(Qt::No
     for (int i=0; i
       for (int j=0;
       { p.setBrush(
        p.drawRect(i
     }
    }
};
```

Exemple (Suite)

```
int main (int argc, char* argv[])
{ QApplication app(argc, argv);
  MyWidget w;
  QScrollArea scrollArea;
  scrollArea.setWidget(&w);
  scrollArea.resize(200, 200);
  scrollArea.show();
  return app.exec();
}
```

Illustration d'un QScrollArea



Quand les slots et signaux ne suffisent plus

- Lorsque l'on veut créer/étendre un widget, la solution des signaux et slots est parfois insuffisante
- Il faut gérer les évènements plus finement, au plus près du widget de manière à redéfinir son comportement
- Dans ce but, Qt permet de manipuler des objets issus de sous-classes de la classe `QEvent`
- Les évènements peuvent être reçus et gérer par tout objet issu d'une sous-classe de `QObject`, mais sont intéressant surtout dans le cas des widgets

Comment sont reçus les évènements

- Quand un évènement est reçu, Qt crée un objet instance de la sous-classe appropriée de `QEvent`
- Cet objet est ensuite passé au widget destination par l'appel à sa méthode `event()`
- Cette méthode ne traite pas directement l'évènement, mais exécute la méthode de gestion appropriée en fonction du type de l'évènement reçu
- Elle renvoie une réponse rendant compte du traitement ou du refus de l'évènement
- Certains évènements comme `QMouseEvent` et `QKeyEvent` sont émis par le système, d'autres comme `QTimerEvent` proviennent d'autres sources ou programmes

Les types d'évènements

- La plupart des évènements ont leur classe de traitement : `QResizeEvent`, `QPaintEvent`, `QMouseEvent`, `QKeyEvent`, `QCloseEvent`, ...
- Chaque sous-classe comporte des fonctions spécifiques adaptées à l'évènement traitée : `QResizeEvent` comprend les méthodes `size()` et `oldSize()` pour permettre aux widgets de connaître l'évolution de leur taille
- Certaines classes comme `QMouseEvent` peuvent gérer plusieurs évènements (le click, le double click, ...)

Gestion des évènements <1>

- Habituellement, un évènement est traité par l'appel d'une méthode virtuelle : `QPaintEvent` est géré lors de l'appel à la méthode `paintEvent()`
- Cette méthode est responsable du traitement de l'évènement, toutefois, il est parfois nécessaire de faire appel aux classes de bases pour finir la gestion d'un évènement

Exemple

```
void MyCheckBox::mousePressEvent(QMouseEvent *event)
{
     if (event->button() == Qt::LeftButton) {
        // Gère le click gauche ici
     } else {
        // Les autres clicks sont passés à la classe de base
        QCheckBox::mousePressEvent(event);
     }
}
```

Gestion des évènements <2>

- Il arrive qu'il n'y est pas de classe spécifique pour la gestion de l'évènement ou qu'elle soit insuffisante
- C'est le cas de la touche "tabulation" qui ne doit pas être interceptée par un widget de texte mais par le widget supérieur
- Dans ce cas, il est possible de redéfinir la méthode `event()` pour prendre en compte ce cas et appeler la méthode de gestion appropriée
- La méthode `event()` renvoie `true` pour indiquer que l'évènement a bien été traité

Gestion des évènements <3>

Exemple

```
bool MyWidget::event(QEvent *event)
{
  if (event->type() == QEvent::KeyPress) {
     QKeyEvent *ke = static_cast<QKeyEvent *>(event);
     if (ke->key() == Qt::Key_Tab) {
        // Ici, gestion particulière du tab
        return true;
     }
  } else if (event->type() == MyCustomEventType) {
     MyCustomEvent *myEvent=static_cast<MyCustomEvent *>(event);
     // Ici, gestion particulière d'un évènement
     return true;
  }
  return QWidget::event(event);
}
```

- `QWidget::event()` est appelé pour les cas non traités

Filtrer les évènements <1>

- Il est parfois utile d'intercepter les évènements destinés à un autre objet
- La méthode `installEventFilter()` permet d'installer un filtre sur un objet interceptant tous les évènements destinés à un de ses descendants
- Lorsqu'un filtre est installé, la méthode `eventFilter()` est exécutée à chaque évènement transitant par l'objet
- La méthode `removeEventFilter()` enlève un filtre existant
- La méthode `eventFilter()` renvoie `true` pour stopper la progression de l'évènement, et `false` sinon
- Si tous les `eventFilter()` renvoient `false`, l'évènement parvient à l'objet destination

Filtrer les évènements <2>

Exemple

```
bool FilterObject::eventFilter(QObject *object, QEvent *event)
{
  if (object == target && event->type() == QEvent::KeyPress) {
     QKeyEvent *keyEvent = static_cast<QKeyEvent *>(event);
     if (keyEvent->key() == Qt::Key_Tab) {
        // Cas particulier du tab
        return true;
     } else
        return false;
  }
  return false;
}
```

- Il est possible d'intercepter/filtrer tous les évènements en plaçant un filtre sur l'objet `QApplication` ou `QCoreApplication`

Créer ses évènements

- Dans beaucoup d'applications, il faut créer ses propres évènements
- Pour créer son propre évènement, il faut :
  - Lui attribuer un identifiant plus grand que `QEvent::User`
  - Sous-classer la classe `QEvent` afin d'implémenter les comportements spécifiques
- Il est possible de poster ses évènements en utilisant les méthodes de classes `QCoreApplication::sendEvent()` et `QCoreApplication::postEvent()`
- `sendEvent()` transmet l'évènement immédiatement qui est traité avant que la méthode ne se termine

Envoyer des évènements

- `postEvent()` transfère l'évènement dans la file d'évènements de l'application
  - ⇒ À la prochaine itération de la boucle principale d'évènements de Qt, l'évènement sera traité
  - ⇒ Cela permet à Qt de réaliser certaines optimisations

Exemple

Si plusieurs évènements de redimensionnement sont émis, alors Qt les compresse en un seul et appelle la méthode `paintEvent` une seule fois évitant que l'application se redessine plusieurs fois inutilement.

Remarque

La méthode `update()` appliquée à un widget << poste » l'évènement `QPaintEvent` dans la file d'évènements de l'application.

Introduction

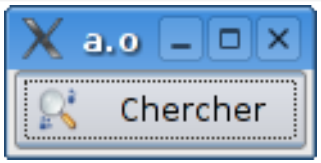
- Qt fournit un excellent support de la 2D et de la 3D
- Les classes graphiques 2D de Qt gèrent les images bitmap et vectorielles
- Elles gèrent aussi les images animées et la détection de collisions
- Qt permet d'écrire du texte au format Unicode et d'appliquer des transformations sur les objets (rotation, ...)
- Qt gère aussi la 3D en proposant des classes reliées directement sur la bibliothèque d'OpenGL

Les images

- La classe `QImage` permet de gérer en entrée et en sortie un grand nombre de format d'images dont : BMP, GIF, JPEG, MNG, PNG, PNM, XPM et XBM
- Certains widgets de Qt peuvent afficher directement des images, c'est le cas des boutons, étiquettes, menus, ...

Exemple

```
QPushButton *button = new QPushButton( "Chercher", parent );
button->setIcon( QIcon("find.png") );
```



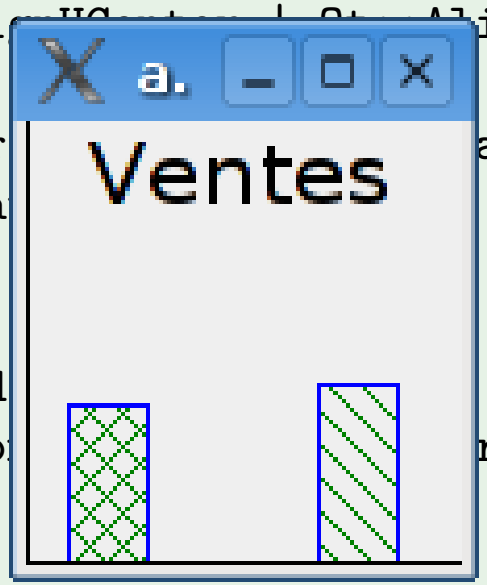
- `QImage` permet d'utiliser la transparence si elle est gérée par le format du fichier
- La classe `QMovie` permet de manipuler les images animées

Dessiner avec Qt <1>

- La classe `QPainter` fournit les primitives de dessin ainsi que des fonctionnalités avancées comme les transformations et le clipping
- Tous les objets Qt se dessinent en utilisant `QPainter`
- Fonctionnalités :
  - **Dessin :** points, lignes, polygones, ellipses, arcs de cercle, bézier, ...
  - **Transformations :** translation, rotation, mise à l'échelle, cisaillement
  - **Clipping :** union, intersection, soustraction et XOR
- Le coin en haut à gauche est localisé aux coordonnées $(0, 0)$ et le coin en bas à droite aux coordonnées $(width() - 1, height() - 1)$

Exemple

```
void paintEvent( QPaintEvent * )
{QPainter painter( this );
 draw_bar( &painter, 0, 39, Qt::DiagCrossPattern );
 draw_bar( &painter, 2, 44, Qt::FDiagPattern );
 painter.setPen( Qt::black );
 painter.drawLine( 0, 0, 0, height() - 1 );
 painter.drawLine( 0, height() - 1, width() - 1, height() - 1 );
 painter.setFont( QFont("Helvetica", 18) );
 painter.drawText(rect(),Qt::Ali                    ignTop,"Ventes");
}
void draw_bar( QPainter *painter                    arHeight,
               Qt::BrushStyle pa
{
 painter->setPen( Qt::blue );
 painter->setBrush( QBrush(Qt::d                    );
 painter->drawRect( 10 + 30 * mo                    rHeight, 20,
                    barHeight );
}
```



Les différents dispositif sur lesquels dessiner

- `QPainter` peut agir sur n'importe quel dispositif de dessin :
  - Un `QImage` est un composant non affiché à l'écran. Le dessin effectué dans un `QImage` est ensuite recopié bit à bit sur un widget, c'est la technique du « double buffering »
  - Un `QPicture` est une image pouvant être dessinée notamment avec un `QPainter`. Il peut ensuite être sauvegardé dans un fichier image
  - Un `QPrinter` représente un périphérique d'impression. Sous windows il utilise le gestionnaire d'impression windows, sous unix du PostScript est envoyé au démon d'impression
  - Un `QWidget` est aussi un dispositif de dessin comme vu dans l'exemple précédent

Le dessin 3D

- OpenGL est une API (Application Program Interface) utilisée pour dessiner des scènes 3D
- Les programmeurs Qt peuvent utiliser la 3D dans leur application de la manière suivante :
  - Utiliser (ou sous-classer) la classe `QGLWidget` qui hérite de `QWidget`
  - Dessiner avec les fonctions standard OpenGL plutôt qu'avec un `QPainter`
- Le module OpenGL utilisé par Qt est disponible sur les plateformes Windows, X11 et Macintosh et utilise soit l'installation OpenGL système soit la bibliothèque Mesa