

Chapitre 1

Programmation événementielle avec C++ et Qt

Ressource R2.02 : Développement d'application avec IHM

Institut Universitaire de Technologie de Bayonne – Pays Basque
BUT Informatique – Semestre 2 - P. Dagorret

v1.0 

1.- Présentation de Qt

Bref historique

- ♦ 1988 : Idée de création en C++ d'une Qt **bibliothèque logicielle orientée objet** (*API-Application Programming Interface*) par Haavard Nord et Eirik Chambe-Eng.
- ♦ 1991 : début de développement de la bibliothèque.
- ♦ 1993 : le noyau est prêt et permet de créer des composants graphiques sous Windows et Unix à partir de la même API.
Les 2 auteurs créent la société (Quasar Technologies, puis TrollTech) pour commercialiser « le meilleur framework GUI en langage C++ ».
- ♦ Au fil des années, la bibliothèque se complète : conception d'un environnement KDE, exécution sur des périphériques embarqués, plateforme pour applications mobiles, ...
- ♦ Aujourd'hui, Qt est un *framework* - **plateforme de développement d'interfaces graphiques GUI** (*Graphical User Interface*), appartenant depuis 2012 à Digia, après avoir été rachetée en 2008 par Nokia (www.qt.io).

Qu'est-ce Qt ?

- ♦ Qt est un *framework* - **plateforme** de **développement d'interfaces graphiques GUI** (*Graphical User Interface*).
- ♦ Qt fournit un **ensemble de classes** décrivant
 - des éléments **graphiques** (*widgets*, pour *windows gadgets*)
 - et des éléments **non graphiques** : accès aux données (fichier, base de données), connexions réseaux (*socket*), gestion du multitâche (*thread*), XML, etc.
- ♦ Qt facilite la **portabilité des applications** sur Unix (dont Linux), Windows et Mac OS X, Android, iOS :
 - Son API est **la même** pour toutes ces plateformes : pas de nécessité d'apprendre les APIs spécifiques à chaque OS cible
 - Le portage est fait par simple **recompilation du code source** (s'il n'utilise que ses composants)
 - Gain de temps pour les programmeurs
- ♦ Qt dispose d'un **moteur de rendu graphique 2D**.
- ♦ Autres bibliothèques multi-plateformes équivalentes connues :
 - **GTK+**, utilisée par l'environnement graphique GNOME
 - **WxWidgets**, utilisé pour développer FileZilla, CodeBlocks, Audacity, iMule... et jusqu'en 2021, ici à l'IUT, dans le module « programmation C++ avec interface graphique » ☺

Utilisation de Qt

Clients

- ♦ Google, Adobe Systems, Asus, Samsung, Philips, ou encore la NASA et bien évidemment Nokia
- ♦ Qt est notamment connu pour être la bibliothèque sur laquelle repose l'environnement graphique KDE, l'un des environnements de bureau les plus utilisés dans le monde Linux

Licences (www.qt.io/licensing/)

- ♦ GNU GPL : gratuite avec obligation de gratuité du code produit
- ♦ GNU LGPL : possibilité d'utiliser gratuitement certains outils de Qt (ex., la bibliothèque graphique) dans du code propriétaire sans contraindre à rendre le code produit libre
- ♦ Licence commerciale : nécessaire si le développeur souhaite produire du code propriétaire
- ♦ A l'IUT : Utilisation d'une licence commerciale mention *Education* : *chaque utilisateur s'engage à ne pas utiliser la plateforme à des fins commerciales*

EDI et principaux outils

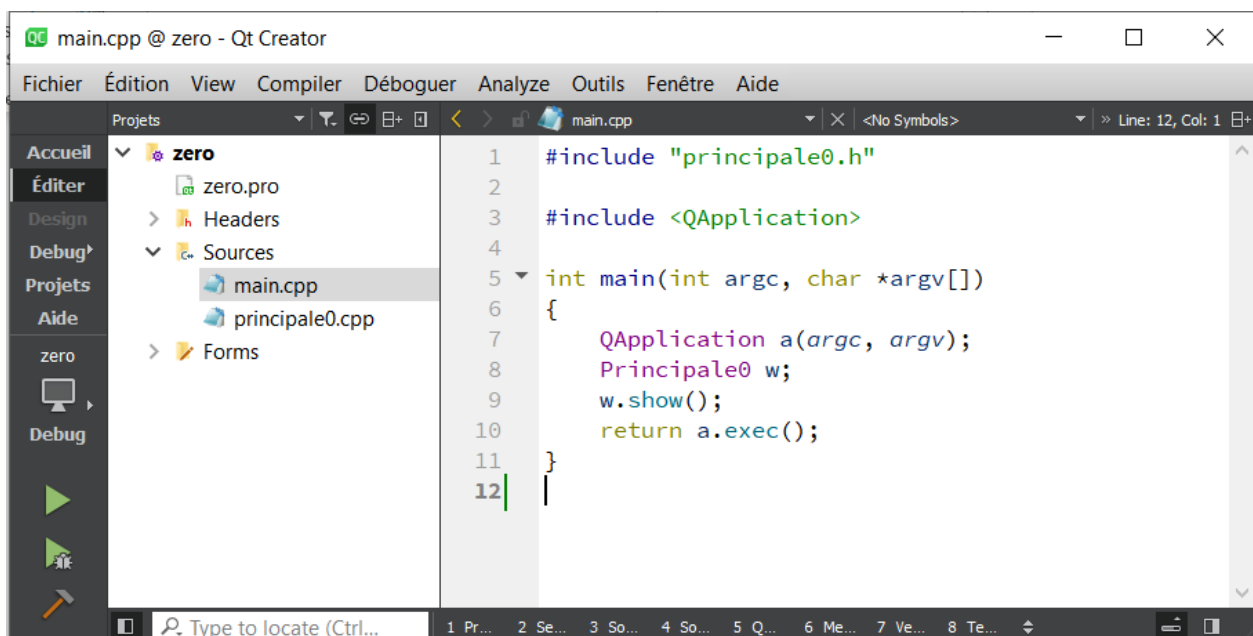
Qt Creator

- ♦ C'est l'environnement de développement intégré dédié à Qt et facilite la gestion d'un projet Qt.
- ♦ Son éditeur de texte offre les principales fonctions que sont la **coloration syntaxique**, l'**autocomplétion** (ou **complètement automatique**), l'**indentation**, etc...
- ♦ Qt Creator intègre les outils :
 - **Qt Designer** : outil de dessin d'interfaces graphiques.
 - **Qt Linguist** : il permet la mise en œuvre rapide de l'internationalisation, c'est-à-dire le développement d'applications multilingues (sans multiplier le développement)
 - **Qt Assistant** : assistant à la création de projets Qt
 - Un mode débogage et beaucoup d'autres *plugins*.



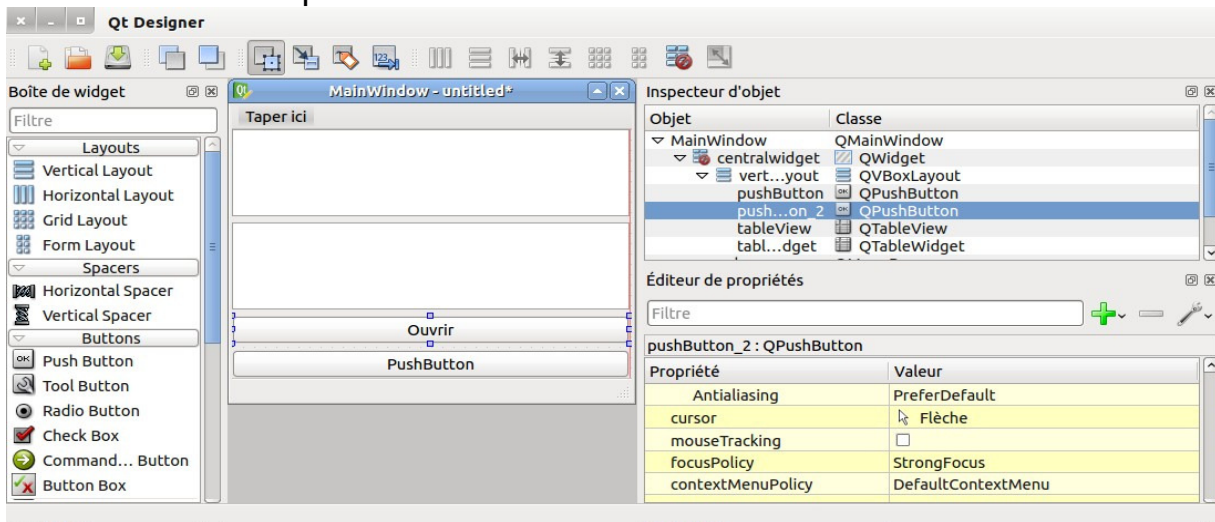
Qt Creator

- ♦ Même si **Qt Creator** est présenté comme l'environnement de développement de référence pour Qt, il existe des modules Qt pour les environnements de développement Eclipse et Visual Studio.
- ♦ Il existe d'autres EDI dédiés à Qt, comme QDevelop et Monkey Studio.a



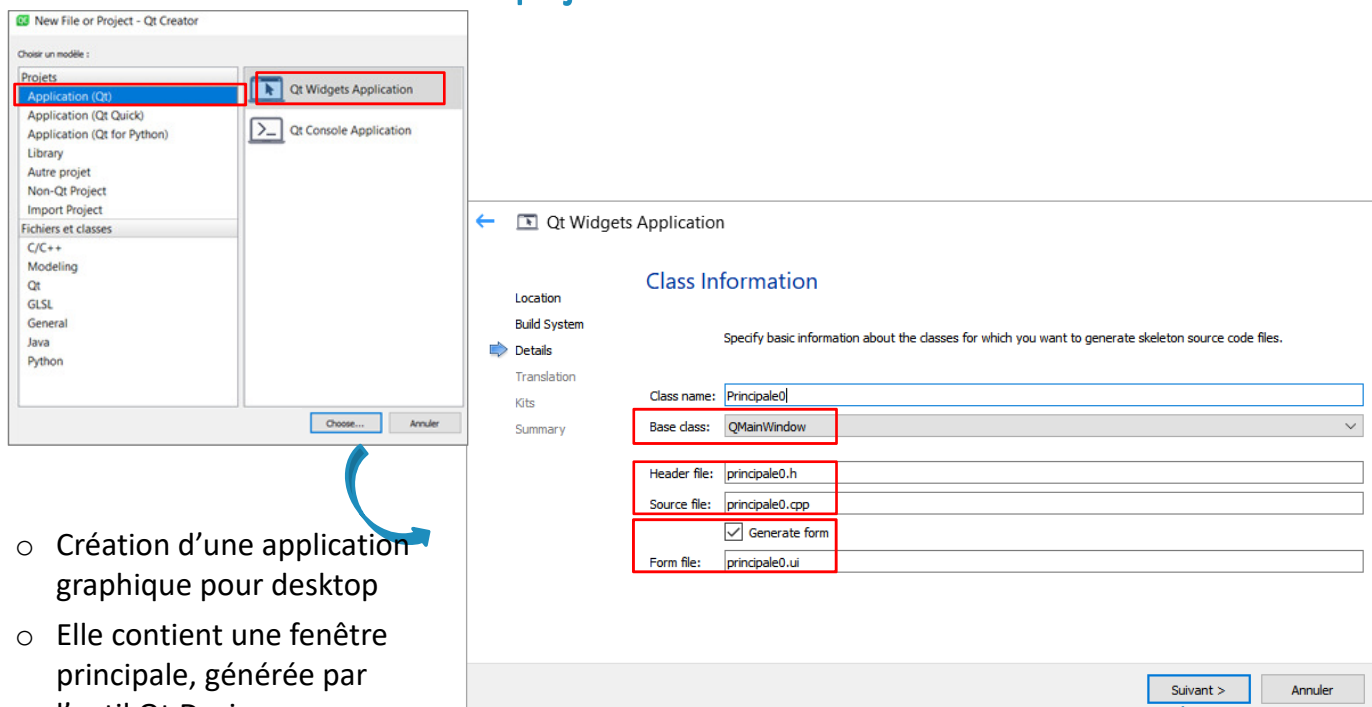
Qt Designer

- ♦ Qt Designer est l'outil de dessin d'interfaces graphiques.
- ♦ Les éléments graphiques sont placés par glisser-déposer, et leurs propriétés précisées via des menus.
- ♦ L'interface graphique est sauvegardée sous la forme d'un fichier XML d'extension **.ui**.
- ♦ Lors de la compilation, l'utilitaire **uic**, fourni par Qt, génère les fichiers C++ avec les classes correspondant au contenu du fichier XML.



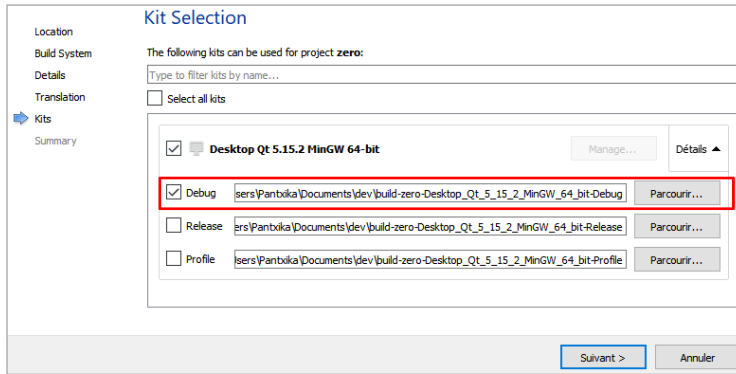
Qt Assistant – assistant nouveau projet (1/2)

- ♦ Aide à la création des 'bons' fichiers correspondant au type d'application souhaitée
- Fichier → Nouveau fichier ou projet ...**



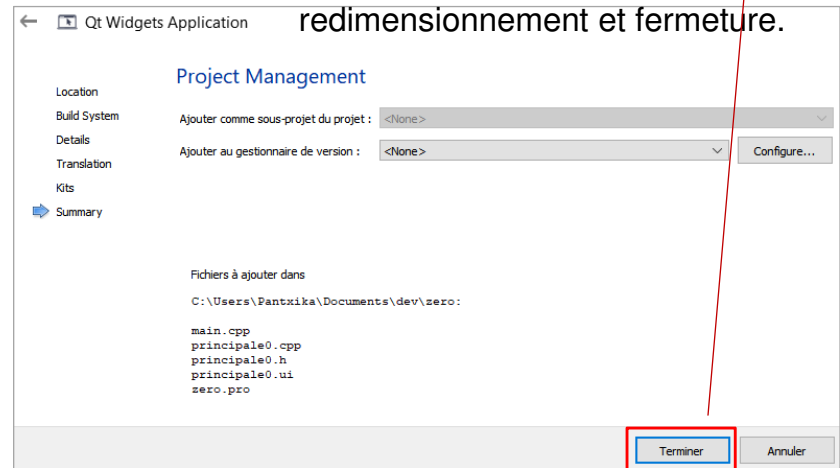
- Création d'une application graphique pour desktop
- Elle contient une fenêtre principale, générée par l'outil Qt Designer

Qt Assistant – assistant nouveau projet (2/2)



- L'application finale : une fenêtre principale vide, comportement classique : redimensionnable, déplaçable, avec barre de titre, menu système et boutons de redimensionnement et fermeture.

- Compilation en mode Debug uniquement, pour faciliter la mise au point du programme
- Un répertoire est créé pour y ranger les ressources générées par l'Assistant : fichiers .h .cpp et .ui
- La description des ressources du projet est rangée dans le fichier .pro



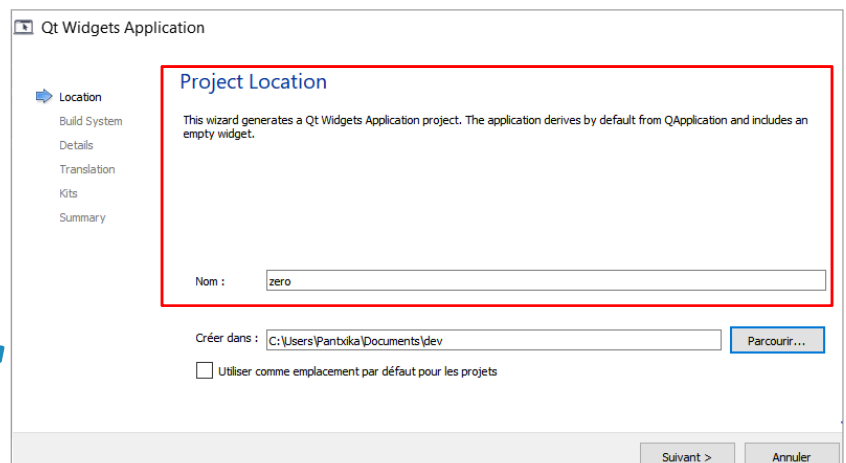
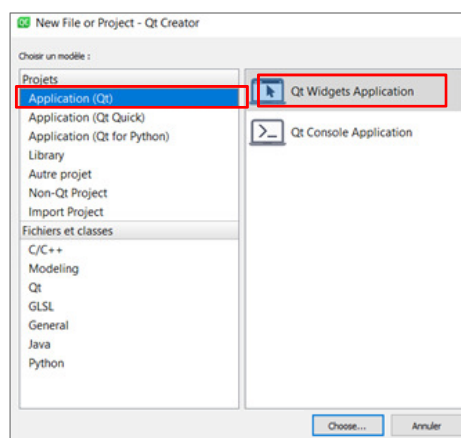
3.- Projet Qt

Notion de projet (1/2)

Projet Qt

- ◆ Qt Creator enregistre dans un fichier **projet** (.pro) toutes les informations / chemins d'accès vers les ressources participant à la création d'une application : modules de Qt utilisés, *chemins d'accès aux fichiers sources* (*), dépendances entre fichiers, paramètres passés au compilateur, répertoires de déploiement...

(*) Ainsi, un même fichier source (par exemple, un module composé de `pile.h` et `pile.cpp`) peut être utilisé dans le développement de plusieurs applications sans nécessité de dupliquer son code

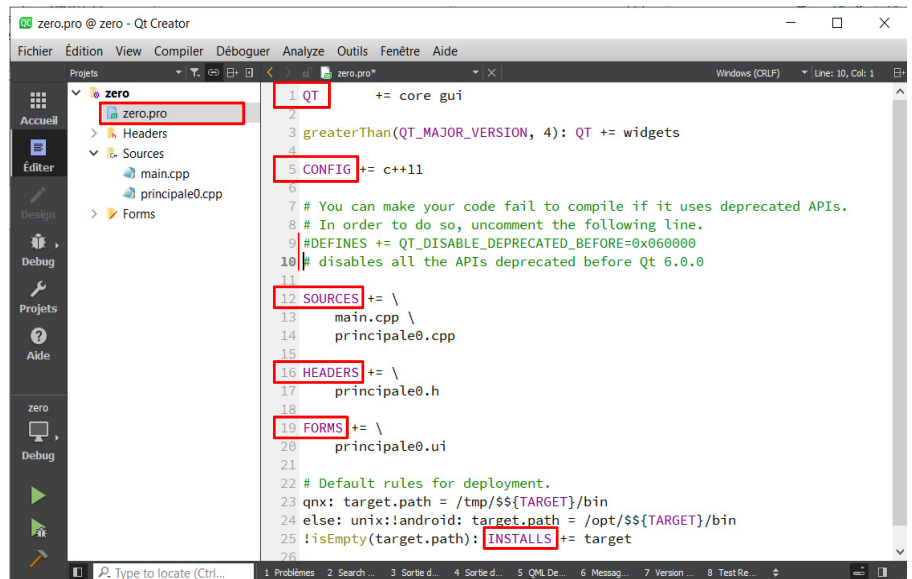


Notion de projet (2/2)

- Le fichier `.pro` est éditable : il peut être créé 'à la main', ou bien généré par Qt Assistant. Le paramétrage est fait par affectation de variables.

qmake

- Qt fournit le moteur de production (*Build system*) **qmake** permettant de créer la commande de compilation spécifique à la plateforme à partir du contenu du fichier `.pro`. Ainsi, sous les systèmes UNIX/Linux, qmake produira un **Makefile**
- Qt Creator fournit aussi des assistants (*wizard*) pour créer des modèles de projets servant à créer rapidement de nouveaux projet par duplication des modèles.



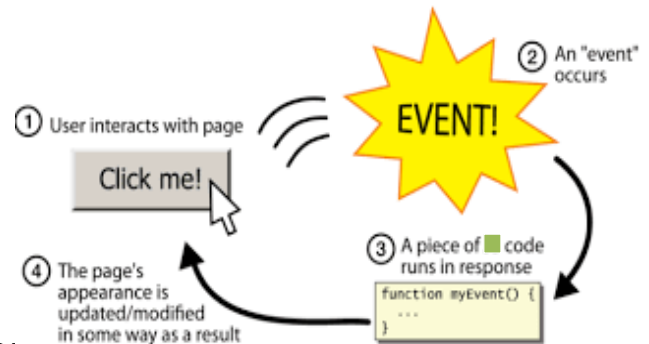
Vocation première de Qt

- Qt est principalement dédiée au développement d'interfaces graphiques en fournissant des éléments prédéfinis appelés **widgets**.
- Les **widgets** peuvent être utilisés pour créer ses propres **fenêtres** et **boîtes de dialogue** complètement prédéfinies. (ouverture/enregistrement de fichiers, progression d'opération, etc)
- Les interactions avec l'utilisateur sont gérées par un mécanisme appelé **signal/slot**. Ce mécanisme est la base de la **programmation événementielle** des applications basées sur Qt.

Programmation événementielle (1/4)

Principe

- ♦ La programmation événementielle est une **programmation basée sur les événements**.
- ♦ Le programme est principalement défini par ses **réactions** aux différents événements qui peuvent se produire, c'est-à-dire des **changements d'état**, comme par exemple :
 - L'incrément d'une liste
 - Un mouvement de souris
 - L'appui sur une touche du clavier
- ♦ Les événements peuvent être provoqués **par l'action d'un utilisateur** ou bien **par le système**.
- ♦ **L'application est toujours prête à réagir à tout type d'événement !**



Programmation événementielle (2/4)

Architecture d'une application événementielle

- ♦ Une application événementielle est organisée autour de deux sections :
 - La première section est une **boucle principale**. Elle **détecte** les événements
 - La seconde section **traite** les événements qui ont été détectés
- ♦ A chaque **événement** que l'application doit prendre en compte, il faut lui **associer** une **action à réaliser**.
- ♦ Cette action (sous-programme, méthode qui traite un événement), s'appelle un **gestionnaire d'évènement** (*event handler*).

Programmation événementielle (3/4)

Fonctionnement d'une application événementielle **non graphique**

- ◆ Une fois lancée, l'application **se met en attente** des événements qui la concernent.
 - Cette attente constitue la **boucle principale**.
 - Les événements proviennent du système d'exploitation (par ex. horloge), d'objets de l'application, ou d'autres sources extérieures à l'ordinateur (ex. capteurs)
- ◆ Lorsqu'un **événement est détecté** par la boucle principale, le **gestionnaire d'événement** correspondant **est exécuté**.

Programmation événementielle (4/4)

Fonctionnement d'une application à interface **graphique**

- ◆ Une application graphique **est** une application événementielle
- ◆ L'application **crée** la **fenêtre principale** à l'intérieur de laquelle auront lieu les interactions avec l'utilisateur.
- ◆ Puis l'application **se met en attente** des événements qui la concernent.
 - Cette attente constitue la **boucle principale**.
 - Les événements proviennent du système d'exploitation, des objets graphiques de l'application, des mouvements de la souris, ou d'autres sources extérieures à l'ordinateur
- ◆ Lorsqu'un **événement est détecté** par la boucle principale, le **gestionnaire d'événement** correspondant **est exécuté**.

Application graphique
(GUI)

Quitter

Évènement
 clic

Gestionnaire d'évènement
(handler)

```
Quitter()  
{  
    ...  
}
```


Mise en œuvre avec Qt

Signal - Slot

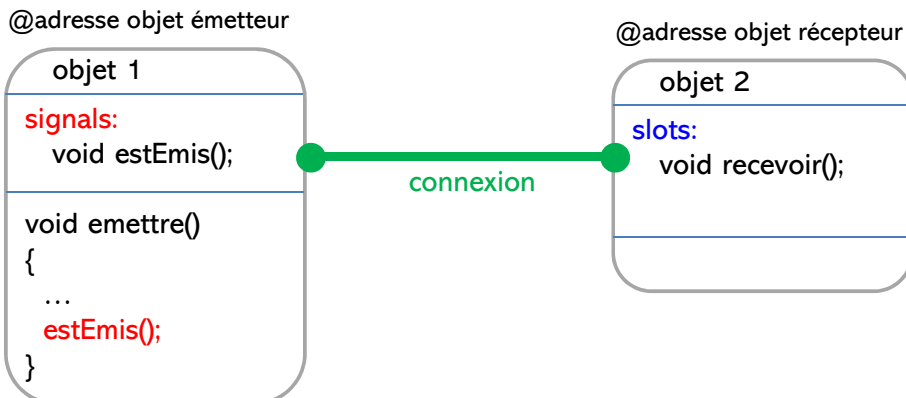
- ♦ La programmation événementielle des applications Qt est basée sur un **mécanisme** appelé **signal / slot** :
 - un **signal** est émis lorsqu'un **événement** particulier se produit.
 - un **slot** est un **gestionnaire d'événement**, c'est-à-dire un **sous-programme** qui sera appelé / exécuté en réponse à un **signal** particulier.
- ♦ L'association d'un **signal** à un **slot** est réalisée par une **connexion** grâce à la méthode (statique) **connect()**.
- ♦ Cette connexion relie 2 objets :
 - Un objet **émetteur**, chargé de produire le **signal**
 - Un objet **récepteur**, chargé d'exécuter le **slot**
- ♦ **Les classes graphiques de Qt possèdent de nombreux signaux et slots prédéfinis.** Il est possible d'en créer des supplémentaires en utilisant le mécanisme d'héritage.



Exemples (1/2)

Application non graphique

- ♦ Connexion entre l'événement `estEmis()` d'un objet et la méthode `recevoir()` d'un autre objet



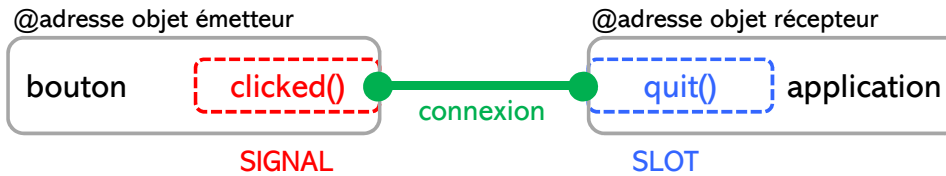
- ♦ **Effet à l'exécution**
Lors de l'appel `objet1.emettre()`, un signal est émis vers l'objet2, qui, du coup, exécute la méthode `recevoir()`.



Exemples (2/2)

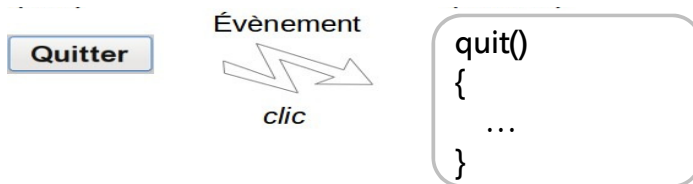
Application graphique

- ◆ Connexion entre l'événement clic d'un bouton situé à l'écran, et la méthode faisant arrêter l'application



- ◆ Effet à l'exécution

Le clic sur le bouton a pour effet de fermer l'application



Classe QObject – Tout est QObject – Tout est Q...

Classe QObject

- ◆ Toutes les classes de prédéfinies de l'API Qt ont un nom :
 - Commençant par la lettre **Q**
 - Puis suivi par un nom dont chaque mot commence par une majuscule.
 Exemples : **Q**Label, **Q**Push**B**utton, ...
- ◆ L'API Qt est basée sur l'**héritage** : la classe **QObject** est la **classe mère de toutes les classes Qt**.
- ◆ La classe **QObject** fournit à ses objets un certain nombre de capacités, comme :
 - La capacité de **communiquer entre eux** via le mécanisme *signal/slot*
 - Une gestion simplifiée de la mémoire
- ◆ Ainsi, **par héritage**, **toutes les classes de Qt possèdent ces capacités**.
- ◆ Les objets Qt (ceux héritant de **QObject**) peuvent s'organiser sous forme d'**arbres d'objets**. Ainsi, lorsqu'une classe est instanciée, on peut lui définir un **objet parent**.

Exemple de classe issue de QObject (1/2)

- ◆ Pour bénéficier de l'API Qt, il faut hériter de **QObject** ou d'une classe fille de **QObject**
- ◆ ☹ Le fichier contient des **macros**. La compilation nécessitera un outil spécifique fourni par Qt

maclasse.h

```

1  #ifndef...
2  #include <QObject>
3
4  class MaClasse : public QObject
5  {
6      Q_OBJECT
7      public:
8          MaClasse(QObject *parent = nullptr);
9      public slots:
10         void recevoir(int);      // slot : corps à définir dans .cpp
11     private:
12         signals:
13         void estEmis(int) ;      // signal : pas de corps à définir dans .cpp
14         //...
15     };
16 #endif // MACLASSE_H

```



Exemple de classe issue de QObject (2/2)

- ◆ L'outil **moc** fourni par Qt, transforme les fichiers sources en code 100% C++
- ◆ Cette particularité constitue un point faible de Qt...

```

maclasse.h
1  #ifndef...
2  #include <QObject>
3
4  class MaClasse : public QObject
5  {
6      Q_OBJECT
7      public:
8          MaClasse(QObject *parent = nullptr);
9      public slots:
10         void recevoir(int);      // slot : corps à définir dans .cpp
11     private:
12         signals:
13         void estEmis(int) ;      // signal : pas de corps à définir dans .cpp
14         //...
15     };
16 #endif // MACLASSE_H

```



```

maclasse.h
1  #ifndef...
2  #include <QObject>
3
4  class MaClasse : public QObject
5  {
6
7
8
9
10
11
12
13
14     //...
15 };
16 #endif // MACLASSE_H

```

100% C++



Classes QApplication et QApplication (1/2)

Classe **QCoreApplication**

- ◆ Cette classe hérite de la classe **QObject**
- ◆ Dans une application Qt non graphique, il **doit y avoir une et une instance de cette classe; elle représente l'application.**
- ◆ Cette classe fournit la **boucle principale d'événements**, où tous les événements provenant du système d'exploitation (horloge, événement réseau,...) ou d'autres sources sont expédiés pour être traités.
- ◆ Sa méthode **exec()** se charge d'exécuter la boucle principale d'événements jusqu'à la fermeture du dernier objet de l'application.
- ◆ La classe **QCoreApplication** gère aussi l'initialisation et la finalisation de l'application, ainsi que ses paramètres.



Classes QApplication et QApplication (2/2)

Classe **QApplication**

- ◆ Cette classe hérite de la classe **QCoreApplication**
- ◆ Tout comme la classe **QCoreApplication**
 - Dans une application Qt graphique, il doit y avoir **une et une instance de cette classe**, quel que soit le nombre de fenêtre graphiques contenues dans l'application; **elle représente l'application.**
 - Elle fournit la **boucle principale d'événements**, où tous les événements provenant du système d'exploitation (horloge, événement réseau,...) ou d'autres sources sont expédiés pour être traités.
 - Sa méthode **exec()** se charge d'exécuter la boucle principale d'événements jusqu'à la fermeture du dernier objet de l'application.
 - Elle gère aussi l'initialisation et la finalisation de l'application, ainsi que ses paramètres.
- ◆ **Ses particularités**
 - **L'instance de QApplication doit être créée avant tout objet graphique.**
 - Cette classe prend en charge des fonctionnalités spécifiques aux applications graphiques, notamment liées à l'initialisation et destruction des objets graphiques.



Exemple d'application Qt non graphique

main.cpp

```

1  #include <QCoreApplication>
2
3  int main(int argc, char **argv)
4  {
5      QCoreApplication app(argc, argv); // l'application
6
7      int retour = app.exec(); // exécute la boucle principale
8                          // d'événements
9
10     return retour;
11 }
```

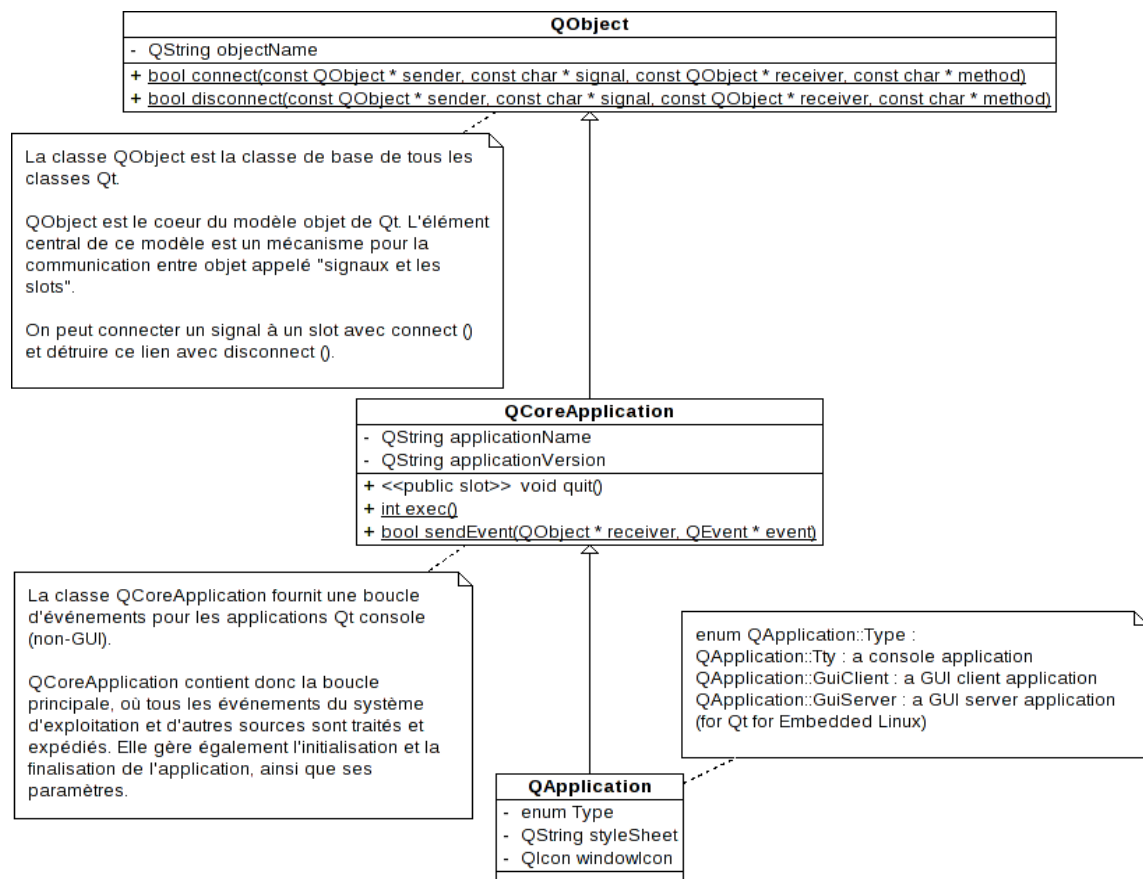
Exemple d'application Qt graphique

main.cpp

```

1  #include <QApplication>
2  #include "principale.h"
3
4  int main(int argc, char **argv)
5  {
6      QApplication app(argc, argv); // l'application
7
8      Principale ihm;           // ma fenêtre principale
9      ihm.show();               // affichage
10
11     int retour = app.exec(); // exécute la boucle principale
12                          // d'événements
13
14     return retour;
15 }
```

Hiérarchie des classes Qt – schéma partiel 1

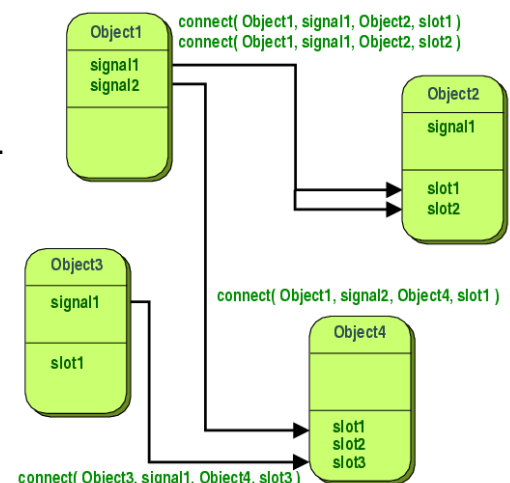


6.- Mécanisme signal / slot

Signal et Slot

- ◆ Les signaux et slots constituent le mécanisme de communication entre objets sous Qt
- ◆ Toute classe qui hérite de la classe **QObject** peut utiliser des signaux et des slots
- ◆ La déclaration de la classe doit contenir la macro **Q_OBJECT**
- ◆ Le mécanisme signal/slot est flexible et modulaire.
Possibilité de :
 - Connecter plusieurs signaux à un même slot
 - Connecter un signal à plusieurs slots.

Attention : l'ordre d'activation des slots est **arbitraire**.
- ◆ Faible couplage du mécanisme signal/slot :
 - Un émetteur ne connaît pas et n'a pas besoin de connaître le/les récepteurs
 - L'émetteur ne sait pas si le signal a été reçu
 - Le récepteur ne connaît pas l'émetteur
- ◆ Contrôle de type
 - Les **types** des paramètres d'un duo signal/slot doivent être les mêmes
 - Un slot peut avoir moins de paramètres qu'un signal



Connexion - Déconnexion

- ♦ La connexion entre un signal et un slot est réalisée par la méthode **connect()**
- ♦ C'est une **méthode statique** de la classe `QObject`

Exemple dans appli graphique : connexion entre signal / slot prédéfinis



```
bool QObject::connect( const QObject * sender, const char * signal,
                      const QObject * receiver, const char * method,
                      Qt::ConnectionType type = Qt::AutoConnection ) const ;
```

- ♦ Une connexion entre un signal et un slot peut être supprimée par la méthode **disconnect()**



5.- Organisation des classes Qt

Exemple dans application Qt graphique

main.cpp

```
1  #include <QApplication>
2  #include <QPushButton>
3
4  int main(int argc, char **argv)
5  {
6      QApplication app(argc, argv);
7      QPushButton bouton("Quitter");
8
9      // connexion du signal prédéfini clicked() de l'objet bouton
10     // au slot prédéfini quit() de l'objet app
11
12     QObject::connect(&bouton, SIGNAL(clicked()), &app, SLOT(quit()));
13     bouton.show();
14
15     return app.exec();
16 }
```



Signaux personnalisés

- ◆ Lorsque l'on crée une classe personnalisée, il est aussi possible de créer des signaux et slots propres à cette classe
- ◆ La classe doit hériter de la classe `QObject`
- ◆ **Créer** un signal personnalisé
 - Utiliser le mot-clé **signal** dans la déclaration de la classe
 - Il s'agit obligatoirement d'une méthode **void**
 - La méthode n'aura pas définition → pas de corps dans le fichier .cpp)
- ◆ **Emettre** un signal
 - Utiliser la méthode `emit()` :

```
emit nomDuSignal (parametreDuSignal);
```

Propriétés

- ◆ Un signal peut être connecté à un autre signal. Ainsi, lorsque le premier signal est émis, il entraîne automatiquement l'émission du second signal
- ◆ **L'émission d'un signal peut être automatique**
 C'est le cas des classes prédéfinies, où les signaux sont déjà définis.
Exemple : l'appui sur un bouton (classe `QPushButton`) entraîne automatiquement l'envoi du signal prédéfini `clicked()`



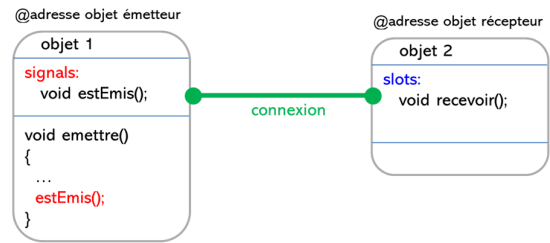
Slots personnalisés

- ◆ Lorsque l'on crée une classe personnalisée, il est aussi possible de créer des signaux et slots propres à cette classe
- ◆ La classe doit hériter de la classe `QObject`
- ◆ **Créer** un signal personnalisé
 - Utiliser le mot-clé **slot** dans la déclaration de la classe
- ◆ A part cette particularité, les slots sont des méthodes 'normales'
- ◆ Les slots peuvent donc être appelés par une autre méthode



Exemple dans application Qt **non** graphique (1/2)

- ◆ Mise en œuvre de l'exemple déjà présenté



maclasse.h

```

1  #ifndef...
2  #include <QObject>
3
4  class MaClasse : public QObject
5  {
6      Q_OBJECT
7      public:
8          MaClasse(int num = 0, QObject *parent = nullptr);
9          void emettre();          // émettra le signal
10         public slots:
11             void recevoir(int);    // slot : corps à définir dans .cpp
12         private:
13             int _numero ;          // rang de création
14         signals:
15             void estEmis(int) ;    // signal P. : pas de corps à définir dans .cpp
16     };
17 #endif // MACLASSE_H

```

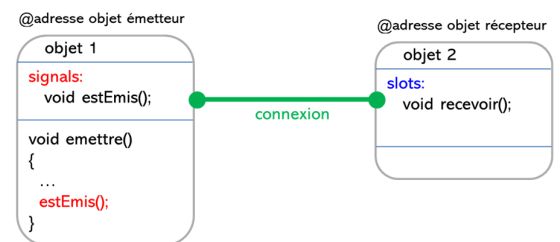


33

Exemple dans application Qt **non** graphique (2/2)

Utilisation de la classe

- ◆ Dans le main.cpp, connexion des signaux/slots de 2 objets de la classe MaClasse
- ◆ Signatures de signal / slot compatibles
- ◆ L'appel objet1.emettre() enverra un signal, qui déclenchera l'exécution du slot recevoir().



main.cpp

```

1  #include <QCoreApplication>
2  #include "maclasse.h"
3
4  int main(int argc, char *argv[])
5  {
6      QCoreApplication a(argc, argv);
7      MaClasse objet1(1);
8      MaClasse objet2(2);
9
10     QObject::connect(&objet1, SIGNAL(estEmis(int)),
11                     &objet2, SLOT(recevoir(int)));
12     objet1.emettre();
13
14     return a.exec();
15 }

```



34