L3 - Conception et développement avancé d'applications CDAA

Romain Raffin
romain.raffin@u-bourgogne.fr

UB - UFR Sciences et techniques, Dijon

2022

1ère partie

Prévisions

Déroulé :

- 14 h CM (7), 18h TD (9), 16h TP (8)
- 2 à 3 évaluations pendant les TP, sur le projet
- évaluation finale sur papier (CT)

Liens utiles :

- mail : romain.raffin@u-bourgogne.fr
- contenus du cours, informations : moodle de l'UBFC
  `https://plubel-prod.u-bourgogne.fr/course/view.php?id=4095`

Plan intermédiaire

Qu'est-ce que Qt ?

- Qt[1] est une bibliothèque logicielle orientée objet (API) développée en C++ par Qt Development Frameworks, filiale de Digia.
- c'est une plateforme de développement d'interfaces graphiques GUI (Graphical User Interface) fournie à l'origine par la société norvégienne Troll Tech, rachetée par Nokia en février 2008 puis cédée intégralement en 2012 à Digia (`www.qt.io`).
- Qt fournit un ensemble de classes décrivant des éléments graphiques (*widgets*) et des éléments non graphiques : accès aux données (fichier, base de données), connexions réseaux (*socket*), gestion du multitâche (*thread*), XML/JSON, ...

[1] On peut dire « Cute »

Qu'offre Qt ?

- Qt permet la portabilité des applications (qui n'utilisent que ses composants et du C++ standard) par simple recompilation du code source. Les environnements supportés sont les Unix (dont Linux), Windows, Mac OS X, Android...
- Qt dispose d'un moteur de rendu graphique 2D performant.
- De grandes entreprises utilisent Qt : 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.

Attention, plusieurs licences sont disponibles selon vos besoins (commercial, LGPL, GPL) : `www.qt.io/licensing/`

Que permet Qt ?

Qt est principalement dédiée au développement d'interfaces graphiques en fournissant des éléments prédéfinis appelés ***widgets*** (pour « *windows gadgets* »).

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.

Plan intermédiaire

Qu'est-ce que la programmation événementielle ?

- La programmation événementielle est une programmation basée sur les événements (logiciels, clavier, souris, *timers*, OS...).
- Le programme sera principalement défini par ses réactions aux différents événements qui peuvent se produire, c'est-à-dire des changements d'état, par exemple l'incrémentation d'une liste, un mouvement de souris ou un appui sur une touche de clavier, etc.
- La programmation événementielle est architecturée autour d'une boucle principale fournie et divisée en deux sections : la première section détecte les événements, la seconde les gère.

Quel est le principe ?

Pour chaque événement à gérer, il faut lui associer une action à réaliser (le code d'une fonction ou méthode) : c'est le gestionnaire d'événement (***handler***)[2]. Pour Qt nous verrons son utilisation juste après.

Ensuite, à chaque fois que l'événement sera détecté par ***la boucle d'événement***, le ***gestionnaire d'événement*** sera exécuté.

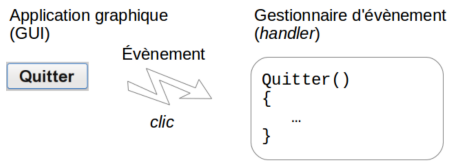


FIG. : Interaction événement/gestionnaire

[2] C'est valable quel que soit le framework.

Comment ça marche avec Qt ?

La programmation événementielle des applications Qt est basée sur un mécanisme appelé ***signal***/***slot*** : un **signal** est émis lorsqu'un événement définit se produit.

Les classes de Qt possèdent de nombreux signaux prédéfinis mais vous pouvez aussi hériter de ces classes et créer **vos propres** signaux.

Un *slot* est une fonction qui va être appelée en réponse à un signal particulier. les classes de Qt possèdent de nombreux slots prédéfinis, mais il est très courant d'hériter de ces classes et de créer ses **propres slots** afin de gérer les signaux qui vous intéressent.

L'association d'un signal à un slot est réalisée par une connexion avec l'appel ***connect()***.

Une figure d'exemple

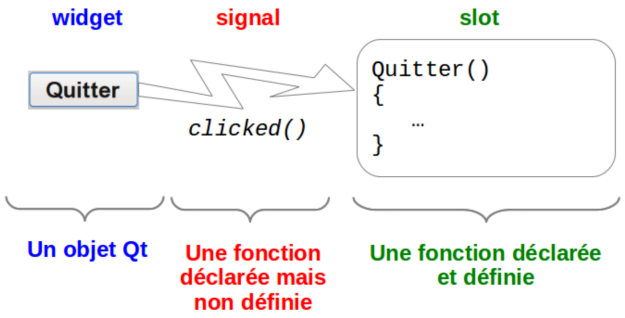


FIG. : Illustration du signal/slot

Plan intermédiaire

Qt utilise massivement la POO

Pour chaque usage, on instancie des classes, on hérite de classes parentes de Qt.

?? Cela pose problème ??

Vous devez être au clair les principes de classes, instance (et `new/delete`, référence, pointeur, accesseur, modificateurs, constructeur, surcharge d'opérateur, `const` de variable ou de fonction, containers `std::`...

Cela se fait de manière simple, la hiérarchie des éléments est bien établie.

Nous verrons également cette programmation en utilisant le C++, vous aurez donc à manipuler des classes, à en créer vous-mêmes pour vos structures ou vos traitements.

Organisation des classes de Qt

L'API Qt est constituée de classes aux noms préfixés par un `Q` et dont chaque mot commence par une majuscule (`QLineEdit`, `QLabel`...).

L'ensemble des classes est basé sur l'**héritage**. La classe **`QObject`** est la classe principale de toutes les classes Qt.

En dérivant de `QObject`, un certain nombre de spécificités Qt sont héritées, notamment : le mécanisme signal/slot pour la communication entre objets, une gestion simplifiée de la mémoire...

Les objets Qt(ceux héritant de `QObject`) peuvent s'organiser sous forme d'**arbre d'objets**. Ainsi, lorsqu'un widget est instancié, on peut lui rattacher un widget parent.

Comment écrire une classe Qt ?

Pour bénéficier des spécificités de Qt, il faudra hériter de `QObject` ou d'une classe fille de `QObject` :

```
1  class MaClasse : public QObject
2  {
3  Q_OBJECT
4  public:
5    MaClasse( QObject *parent=0 ) : QObject(parent) {}
6  private:
7
8  signals:
9    void send( int ); //un signal (défini seulement ici)
10
11 public slots:
12   void receive( int ); //un slot (à définir)
13 };
```


Pourquoi Qt est multiplateforme ?

Comme nous l'avons vu, Qt permet de définir des classes, héritant de `QObject`, et impose aussi de nouveaux mots-clés :

`QObject`, `Q_OBJECT`, `signals`, `slots`.

Ces mots-clés ne font pas partie de C++, Qt fournit donc une interface de **pré-compilation** du code source (qui est en fait un pseudo-C++) vers des sources 100% compatibles avec :

- votre compilateur (et ses versions),
- votre système et son architecture,
- votre environnement graphique (car il faut adapter les passages de messages et l'affichage au moins),
- ...

La compilation de sources Qt s'effectue en 2 étapes :

1. compilation des sources pseudo-C++ et création d'un **modèle intermédiaire** (**MOC**)
2. compilation du **modèle MOC** par le compilateur C++ classique

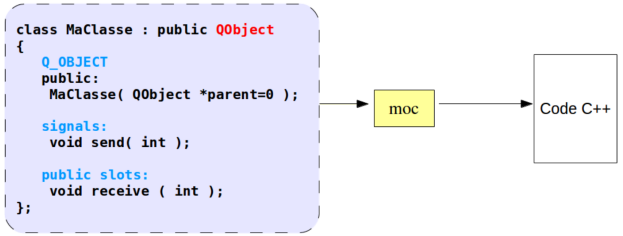


FIG. : Utilisation du MOC (étape 1)

Compilation et projet

Le « ***Meta Object Compiler*** » qui produit le modèle intermédiaire est fourni/maintenu par Qt, il fait partie des programmes installés par votre kit de développement.

Pour faciliter la compilation et éviter à l'utilisateur d'utiliser une série d'analyseurs (QT, C++, projet), Qt fournit ***qmake*** et la définition des besoins/contraintes du projet en fichier texte (extension `.pro`).

Cela permet de spécifier les sources à utiliser, les phases de compilation, les outils au besoin et les cibles de la compilation (système d'exploitation, versions...). Cela fait rapidement de vos sources avec Qt des applications multiplateformes.

Mettons sur pause quelques instants...

Il devient nécessaire à ce point de faire quelques rappels de POO C++

Plan intermédiaire

L'ultra-simple


```cpp
// R. Raffin, 2022
/* Exemple primaire de C++
 * utile seulement pour std : :cout (et std : :endl) et les commentaires
 * compilation :
 * g++ -c -Wall simple.cpp -o simple.o
 * g++ -Wall simple.o -o simple
 * ./simple
*/

#include <iostream> //pour std : :cout et std : :endl


int main()
{
    std : :cout << "Cela fonctionne" << std : :endl;

    return 0;
}
```

Documentation de référence :

- https://fr.cppreference.com/w/
- https://fr.wikibooks.org/wiki/Programmation_C-C++

Un container pour stocker de la donnée

C++ fournit des classes de stockage (et de traitements)[3] : les conteneurs. Ils sont de divers types :

- pile,
- file,
- liste,
- collections,
- *map*,
- ...

Les stockages peuvent être séquentiels, associatifs, ordonnés ou non.

[3] https ://fr.cppreference.com/w/cpp/container

Leur utilisation est basée sur les mêmes interfaces, ici une liste (doublement chaînée) :

```cpp
#include <iostream> //pour std : :cout et std : :endl
#include <list> //pour std : :list

void fillList(std : :list<float> & list)
{
        list.push_back(1.0);
        list.push_back(2.0);
        list.push_back(3.0);
}

void displayList(const std : :list<float> & list)
{
        for (auto & v : list)
                std : :cout << v << " ";

        std : :cout << std : :endl;
}

int main()
{
        std : :list<float> lf;

        fillList(lf);

            std : :cout << "nb d'enregistrements " << lf.size()
                    << std : :endl;

        displayList(lf);

        lf.clear();

    return 0;
}
```

Note : pour aller plus loin, les *template* permettent de définir des objets typés à l'instanciation.

Stockages plus complexe

Lorsque l'on veut stocker des attributs composés ou des tuples, on utilise `enum` ou `struct`.

- `enum`[4] : énumération de valeurs, constantes une fois déclarée :

```cpp
enum Direction {nord, est, sud, ouest};

Direction d = nord;
```

[4] `https://en.cppreference.com/w/cpp/language/enum`

Stockages plus complexe

- `struct` : type complexe, composé de plusieurs variables (champs) :

```cpp
//définition
struct Personne {
        std : :string nom;
        uint anneeNaissance;
}

void main() {

//déclaration
struct Personne p;

//utilisation
p.nom = "Jean";
p.anneeNaissance = 2022;

}
```


Pointeurs et adresses

C++ permet de distinguer les manières d'accéder à une variable, selon sa localisation dans l'espace mémoire.

Les pointeurs permettent de parcourir l'adressage virtuel mis à disposition par le système sans être limité à la pile.

Un pointeur sur une variable de type `float` est utilisé comme ceci :

```cpp
float * pf = new float;

        *pf = 12.3;

delete pf;
```

Exemples de pointeurs

On peut « faire pointer » n'importe quoi :

- un tableau de `float` :

```
1  float * pf = new float[1000];
2  pf[10] = 12.3;
3  delete [] pf;
```

- une structure :

```
1  struct data {
2      int i;
3      std::string s
4  };
5
6  struct data * d = new struct data;
7  d->i = 12;
8  d->s = "bonjour"
9
10 delete d;
```


La compilation avec g++, en ligne de commande

Pour illustrer plus facilement la compilation, on utilisera ici les étapes dans une ligne de commande. Le compilateur choisi est `g++`[5]

Si on dispose de 3 sources : `princ.cpp`, `mod.cpp`, `mod.hpp` (les extensions `.cpp` contiennent des définitions, les `.h` ou `.hpp` des déclarations) et que `princ.cpp` contient l'appel à la fonction `main()`.

On compile en 3 phases :

1. mod.cpp -> mod.o : `g++ -c -Wall mod.cpp -o mod.o`
2. princ.cpp -> princ.o : `g++ -c -Wall princ.cpp -o princ.o`
3. éditions de liens : `g++ -Wall princ.o mod.o -o monBinaire`

[5] issu de « *Gnu Compiler Collection* » `https://gcc.gnu.org/`
`https://gcc.gnu.org/projects/cxx-status.html`

L'ultra-simple classe

Pour illustrer la programmation objet avec C++, nous allons créer une classe « *simpleClass* » qui contient des attributs : un **nom** et un **âge**.

> *La classe minimale : constructeur sans et avec paramètres, destructeur même vide, accesseurs et mutateurs des attributs.*

On utilise `std::string` pour gérer la chaîne de caractère du nom et `float` pour gérer l'âge [6].

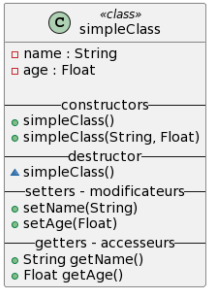
Le programme devra pouvoir remplir des instances de cette classe et afficher les données extraites.

Nous allons découpler les traitements de la classe : elle ne saura pas afficher son contenu et ce sera le rôle d'une fonction indépendante.

[6] qui est donc fluctuant, ce n'est pas une très bonne idée mais cela évite de gérer des dates

1er diagramme de classe

Le diagramme de classe correspondant serait :



(vous les reverrez dans les CM/TD/TP de CDAA mais surtout en Modélisation UML[7] cette année)

Note : les diagrammes sont rendus grâce à `https://plantuml.com/`, ce qui permet de spécifier en texte simple les éléments de modélisation.

[7] `https://fr.wikibooks.org/wiki/Programmation_UML`

L'ultra-simple classe

Listing 1 : simpleClass.cpp

```cpp
#include "simpleClass.hpp"

simpleClass : :simpleClass()
{
        setAge(0.1);
        setName("");
}

simpleClass : :simpleClass(const std : :string & n, const float & f)
{
        setAge(f);
        setName(n);
}

void simpleClass : :setName(const std : :string & n)
{
        name = n;
}

void simpleClass : :setAge(const float & a)
{
        if ( a > 0.0)
                age = a;
} //utiliser une fonction de validation d'un age, lancer une exception
```

Listing 2 : simpleClass.hpp

```cpp
#include <string>

class simpleClass {
        private :
                std : :string name;
                float age;

        public :
        //constructors
        simpleClass();
        simpleClass(const std : :string & n, const float & f);

        //destructor
        ~simpleClass() {;}; //inline definition

        //setters – modificateurs
        void setName(const std : :string & n);
        void setAge(const float & a);

        //getters – accesseurs
        std : :string getName() const { return name; };
        float getAge() const {return age;};
};
```

Programme principal

Listing 3 : main.cpp

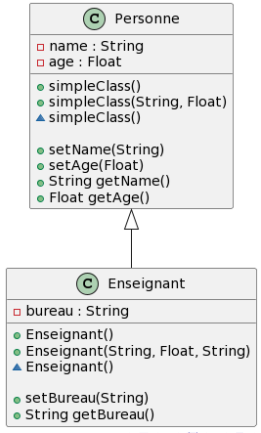
```cpp
#include <iostream>

#include "simpleClass.hpp"

std : :string extractStringFromSimpleClass(const simpleClass& c)
{
        std : :string s="";

        s += c.getName();
        s += " ";
        s += std : :to_string(2022.0 – c.getAge());

        return s;
}

int main()
{
        simpleClass sc("Robert", 12.3);

        std : :cout << extractStringFromSimpleClass(sc) << std : :endl;

        return 0;
}
```

hiérarchie (public, privé, multiple - polymorphisme)

On veut modéliser un « Enseignant » à partir de la classe précédente (que l'on appelera « Personne »).

Un Enseignant a **aussi** un bureau (String).

La modélisation UML est :



Classe Personne

Listing 4 : personne.cpp

```cpp
#include "personne.hpp"

Personne : :Personne()
{
        setAge(0.1);
        setName("");
}

Personne : :Personne(const std : :string & n, const float & f)
{
        setAge(f);
        setName(n);
}

void Personne : :setName(const std : :string & n)
{
        name = n;
}

void Personne : :setAge(const float & a)
{
        if ( a > 0.0)
                age = a;
} //utiliser une fonction de validation d'un age, lancer une exception
```

Listing 5 : personne.hpp

```cpp
#pragma once

#include <string>

class Personne {
        private :
                std : :string name;
                float age;

        public :
        //constructors
        Personne();
        Personne(const std : :string & n, const float & f);

        //destructor
        ~Personne() {;}; //inline definition

        //setters – modificateurs
        void setName(const std : :string & n);
        void setAge(const float & a);

        //getters – accesseurs
        std : :string getName() const { return name; };
        float getAge() const {return age;};
};
```

Classe Enseignant

Listing 6 : enseignant.cpp

```cpp
#include <string>

#include "enseignant.hpp"

        //constructors
Enseignant : :Enseignant()
{
        Personne();
}
Enseignant : :Enseignant(const std : :string & n, const float & f, const
    std : :string & b) :Personne(n,f)
{
        setBureau(b);
}

std : :string Enseignant : :getBureau() const
{
        return bureau;
}

void Enseignant : :setBureau(const std : :string & b)
{
        bureau = b;
}
```

Listing 7 : enseignant.hpp

```cpp
#include <string>
#include "personne.hpp"

class Enseignant : public Personne {
        private :
                std : :string bureau;

        public :
        //constructors
        Enseignant();
        Enseignant(const std : :string & n, const float & f, const std : :string &
            b);

        //destructor
        ~Enseignant() {;};

        std : :string getBureau() const;
        void setBureau(const std : :string &);

};
```


Programme principal

Listing 8 : main.cpp

```cpp

#include "personne.hpp"

std : :string extractStringFromEnseignant(const Enseignant& e)
{
        std : :string s="";

        s += e.getName();
        s += " ";
        s += std : :to_string(2022.0 – e.getAge());
        s += " ";
        s += e.getBureau();

        return s;
}

int main()
{
        Enseignant sc("Robert", 12.3, "B06");

        std : :cout << extractStringFromEnseignant(sc) << std : :endl;

        return 0;
}
```


Plan intermédiaire

Comment accéder aux propriétés d'une classe Qt ?

Un objet Qt peut avoir des propriétés. Toutes les propriétés sont des attributs de la classe que l'on peut lire et éventuellement modifier.

Qt suit cette convention pour le nom des accesseurs : `propriete()` pour lire la propriété et `setPropriete()` pour la modifier.

Pour connaître l'ensemble des classes, méthodes et propriétés, signaux et slots de Qt, il faut consulter la documentation en ligne :

`http://doc.qt.io/`

Pour pouvoir utiliser une classe, il faut inclure sa déclaration, par exemple :

`#include <QLineEdit>`

Exemple d'utilisation

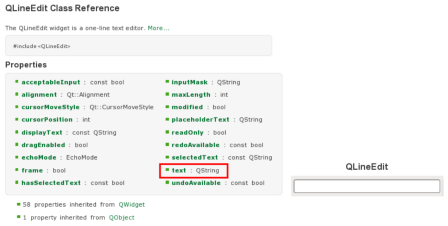


FIG. : Aide sur QLineEdit

```qt
//Pour lire la chaîne de caractères saisie dans un QLineEdit
QString unTexte = monLineEdit.text();

//Pour modifier la chaîne de caractères d'un QLineEdit
monLineEdit.setText("mon texte");
```

Qu'est-ce que les modules ?

Depuis la version 4, Qt sépare sa bibliothèque en modules. Un module regroupe un ensemble de classes. Les principaux modules :

- `QtCore` : pour les fonctionnalités non graphiques utilisées par les autres modules ;
- `QtGui` : pour les composants graphiques (*qt4*), maintenant `QtWidgets` (*qt5*) ;
- `QtNetwork` : pour la programmation réseau ;
- `QtSql` : pour l'utilisation de base de données SQL ;
- `QtXml` : pour la manipulation et la génération de fichiers XML ;

et de nombreux autres ...

Il faut activer un module dans un projet Qt pour pouvoir accéder aux classes qu'il regroupe (cf. plus loin la variable `QT` d'un fichier .pro).

La classe `QApplication`

Pour créer une application Qt graphique (« ***Graphical User Interface*** » - GUI), il faut instancier un objet de la classe **`QApplication`** (et `QCoreApplication` pour les applications non graphiques).

La classe `QApplication` (qui hérite de `QCoreApplication`) fournit une boucle principale d'événements graphiques pour les applications Qt : tous les événements du système seront traités et expédiés.

C'est sa méthode `exec()` qui exécute la boucle d'attente des événements jusqu'à la fermeture du dernier objet de l'application.

La classe `QApplication` gère également l'initialisation et la finalisation de l'application, ainsi que ses paramètres.

L'instance de `QApplication` doit être créée avant tout objet graphique.

Exemple `main.cpp`

```qt
#include <QApplication>
#include "ihm.h"

int main(int argc, char **argv)
{
QApplication app(argc, argv); //un objet QApplication

IHM ihm; //ma fenêtre
ihm.show(); //affichage de ma fenêtre

int ret = app.exec(); //exécute la boucle principale d'événement
return ret;
}
```

Diagramme de classes partiel

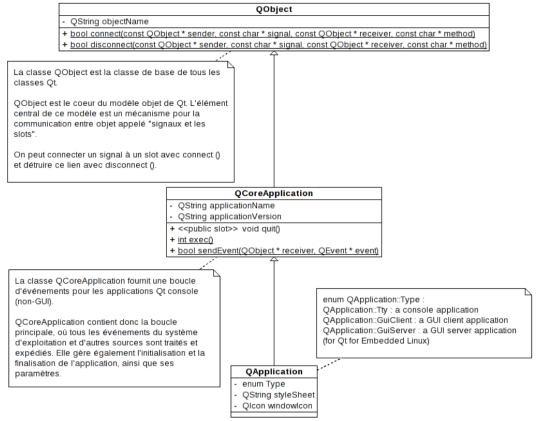


FIG. : Diagramme de classes QObject-QApplication

Les widgets

Avec Qt, les éléments (ou composants) graphiques prédéfinis sont appelés des *widgets* (pour *windows gadgets*). Les widgets sont les éléments principaux de la création d'interface graphique utilisateur (GUI).

Les widgets peuvent afficher des données et des informations sur un état, recevoir des actions de l'utilisateur et agir comme un conteneur pour d'autres widgets qui doivent être regroupés.

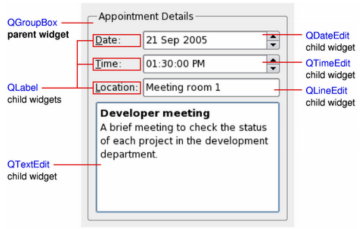


FIG. : Composition de widgets

La classe `QWidget`

La classe `QWidget` fournit la capacité de base d'affichage à l'écran et de gestion des événements. Elle est la classe mère de toutes les classes servant à réaliser des interfaces graphiques.

Les widgets :

- sont créés « cachés » (`hide`/`show`)
- sont capable de se « peindre » (`paint`/`repaint`/`update`)
- sont capable de recevoir les événements souris, clavier
- sont tous rectangulaires (x,y,*width*,*height*)
- sont initialisés par défaut en coordonnées 0,0
- sont ordonnés suivant l'axe z (la profondeur)
- peuvent avoir un widget parent et des widgets enfants

Exemple

```
1  #include<QApplication>
2  #include <QWidget>
3
4  int main(int argc, char **argv)
5  {
6  QApplication app(argc, argv);
7  QWidget w; //un objet widget qui n'a pas de parent
8
9  //un widget est crée caché, il faut donc l'afficher
10 w.show(); //on obtient une fenêtre "vide"
11
12 //on exécute la boucle principale d'événements
13 int ret = app.exec();
14
15 //lorsqu'on ferme la fenêtre, on sort de la boucle
16 //et on quitte l'application
17 return ret;
18 }
```

Une première fenêtre

Un widget d'affichage qui n'est pas intégré dans un widget parent est une fenêtre.

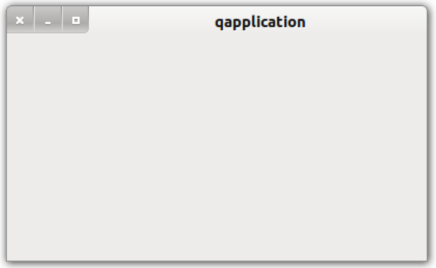


FIG. : 1ère fenêtre

Les widgets prédéfinis

Qt fournit des widgets prédéfinis permettant de composer ses propres applications graphiques :

FIG. : Quelques (autres) widgets

Les widgets prédéfinis

Qt fournit des widgets prédéfinis permettant de composer ses propres applications graphiques :

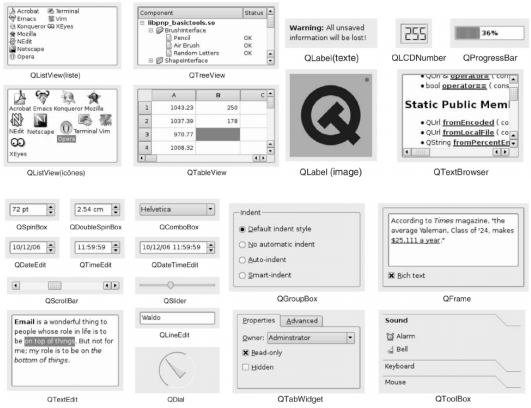


Diagramme de classes partiel de widgets

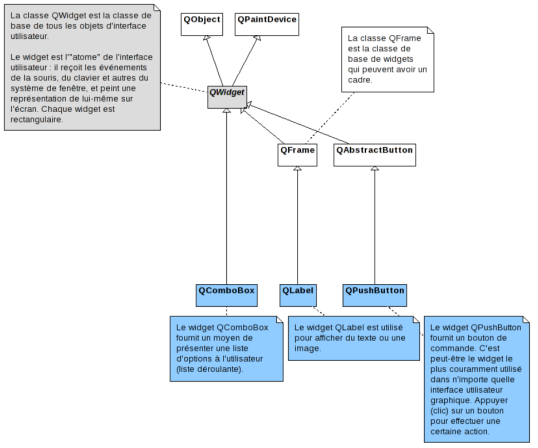


Diagramme de classes partiel de `QMainWindow`

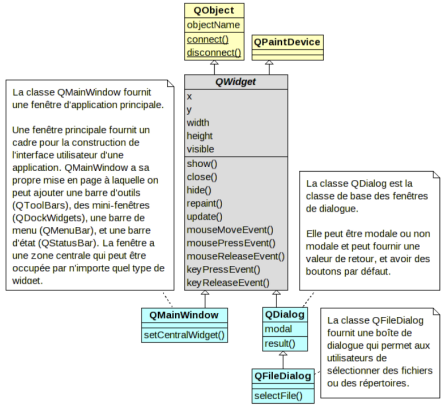


FIG. : Diagramme de classes Fenêtre

Création de widget et de fenêtre personnalisées

La création de widgets personnalisés est faite en héritant de `QWidget` ou d'une classe fille :

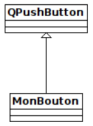


FIG. : Widget personnalisé par héritage

La création de fenêtres personnalisées est faite en héritant de `QWidget`, `QDialog` ou `QMainWindow` :

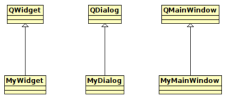


FIG. : Fenêtre personnalisée par héritage

Création de fenêtre personnalisée

Ensuite, on compose sa fenêtre personnalisée en y intégrant des widgets :

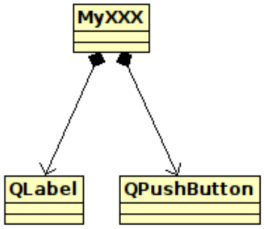


FIG. : Fenêtre personnalisée par héritage

Remarques : La classe Qt dédiée à la fenêtre principale est bien évidemment `QMainWindow`. C'est réalisé dans le constructeur de la classe « Fenêtre » (ici MyXXX).

Création de fenêtre personnalisée : `QMainWindow`

La classe `QMainWindow` propose un « squelette » permettant de réaliser ce type d'application :

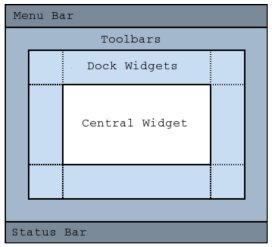


FIG. : Widgets de `QMainWindow`

Définition d'une QMainWindow

Dans MyMainWindow.h :

```
1  #include<QMainWindow>
2  #include <QLabel>
3
4  //MA classe fenêtre principale
5  class MyMainWindow : public QMainWindow
6  {
7  Q_OBJECT
8
9  public:
10   MyMainWindow( QWidget *parent=0 );
11   ~MyMainWindow();
12
13 private:
14   QLabel *label; //un pointeur sur une instance de QLabel
15 };
```


Déclaration d'une QMainWindow

Dans MyMainWindow.cpp :

```
1  #include "MyMainWindow.h"
2
3  //constructeur
4  MyMainWindow : :MyMainWindow( QWidget *parent ) : QMainWindow(parent)
5  {
6    //on instancie un QLabel en indiquant son parent (this => le widget courant)
7    label = new QLabel("Je suis un QLabel", this);
8    //...
9
10   //on fixe le widget QLabel au centre de la fenêtre
11   setCentralWidget(label);
12 }
13
14 //destructeur
15 MyMainWindow : :~MyMainWindow()
16 {
17 }
```


L'application principale

Dans main.cpp :

```
1  #include <QApplication>
2  #include "MyMainWindow.h"
3
4  int main(int argc, char **argv) {
5
6        QApplication app(argc, argv); //mon objet application
7        MyMainWindow w; //mon objet fenêtre
8
9        w.show(); //affichage
10
11 return app.exec(); //boucle
12 }
```


Résultat

FIG. : Sortie de QMainWindow

Mécanisme signal/slot

Rappel : les signaux et les slots forment un mécanisme de communication entre objets avec Qt.

Toutes les classes qui héritent de `QObject` ou d'une de ses sous-classes (par exemple, `QWidget`) peuvent contenir des signaux et des slots.

Il faut aussi ajouter la macro `Q_OBJECT` au sein de cette classe.

Les signaux et les slots sont **faiblement couplés** : une classe qui émet un signal *ne sait pas* (et *ne se soucie pas de*) quels slots vont recevoir ce signal. De la même façon, un objet interceptant un signal ne sait pas quel objet a émis le signal.

Connexion/déconnexion

Une connexion signal/slot doit être réalisée par la méthode `connect()` :

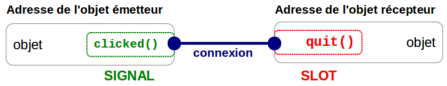


FIG. : Connexion Signal/Slot

Prototype :

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

Une connexion signal/slot peut être supprimée par la méthode `disconnect()`.