



Apprenez à programmer en C++ !

Informations sur le tutoriel



Auteurs : [M@teo21](#) et [Nanoc](#)

Difficulté :

Temps d'étude estimé : 2 mois, 15 jours

Licence :

[Plus d'informations](#)

Popularité

Visualisations : 2 140 998 600

Appréciation 17
des lecteurs : 13
63
715

2633 personnes souhaitent voir ce tutoriel publié en livre !

[Vous aussi ?](#)

Publicité

Historique des mises à jour

- Le 02/10/2010 à 12:30:32
Ajoute des tags
- Le 17/09/2010 à 14:26:59
#2934 Correction orthographique
- Le 01/09/2010 à 18:14:10
#2836

Ce cours est la suite de "[Apprenez à programmer en C](#)"

Il est nécessaire d'avoir lu au moins les parties I et II du cours de C pour pouvoir suivre celui-ci sur le C++

Après avoir découvert le langage C dans le cours "[Apprenez à programmer en C](#)", nous nous intéressons ici au C++ 😊
Le langage C++ est basé sur le C : ce que vous avez donc appris jusqu'ici va vous resservir, pour ne pas dire vous être indispensable !

Les modifications entre le C et le C++ sont nombreuses. La plus importante d'entre elles est l'introduction de la Programmation Orientée Objet, que l'on abrège couramment POO. On en entend souvent parler, mais qu'est-ce que c'est concrètement ?

Après une première partie très importante sur les notions de base du C++, nous passerons à la pratique et verrons comment créer des applications fenêtrées qui fonctionnent aussi bien sous Windows, Linux et Mac OS à l'aide de la bibliothèque Qt !



Ce cours est composé des parties suivantes :

- [\[Théorie\] La Programmation Orientée Objet](#)
 - [\[Pratique\] Créez vos propres fenêtres avec Qt](#)
 - [Annexes](#)
 -
-

Partie 1 : [Théorie] La Programmation Orientée Objet

Le langage C++ est basé sur le C : ce que vous avez donc appris dans le [cours de C](#) jusqu'ici va vous resservir, pour ne pas dire vous être indispensable !

Les modifications entre le C et le C++ sont nombreuses. La plus importante d'entre elles est l'introduction de la Programmation Orientée Objet, que l'on abrège couramment POO. On en entend souvent parler, mais qu'est-ce que c'est concrètement ?

La réponse se trouve dans cette partie du cours 😊

○

[1\) Introduction au C++](#)



- [Pourquoi avoir créé le C++ ?](#)
- [La programmation orientée quoi ?](#)

○

[2\) Premier programme C++ avec cout et cin](#)



- [Configurer l'IDE pour le C++](#)
- [Analyse du premier code source C++](#)
- [Le flux de sortie cout](#)
- [Le flux d'entrée cin](#)

○

[3\) Nouveautés pour les variables](#)



- [Le type bool](#)
- [Les déclarations de variables](#)
- [Les allocations dynamiques](#)
- [Le typedef automatique](#)
- [Les références](#)

○

[4\) Nouveautés pour les fonctions](#)



- [Des valeurs par défaut pour les paramètres](#)
- [La surcharge des fonctions](#)
- [Les fonctions inline](#)

○

[5\) La magie de la POO par l'exemple : string](#)



- [Des objets... pour quoi faire ?](#)
- [Lire et écrire dans une chaîne via string](#)
- [Opérations sur les string](#)

○

[6\) Les classes \(Partie 1/2\)](#)



- [Créer une classe](#)
- [Droits d'accès et encapsulation](#)
- [Séparer prototypes et définitions](#)

○

[7\) Les classes \(Partie 2/2\)](#)



- [Constructeur et destructeur](#)
- [Associer des classes entre elles](#)
- [Action !](#)
- [Méga schéma résumé](#)

○

[8\) Classes et pointeurs](#)



- [Pointeur d'une classe vers une autre classe](#)
- [Gestion de l'allocation dynamique](#)
- [Le constructeur de copie](#)
- [Le pointeur this](#)

○

[9\) La surcharge d'opérateurs](#)



- [Petits préparatifs](#)
- [Les opérateurs arithmétiques \(+, -, *, /, %\)](#)
- [Les opérateurs de comparaison \(==, >, <, ...\)](#)
- [L'opérateur d'affectation \(=\)](#)
- [Les opérateurs de flux \(<<, >>\)](#)

○

[10\) TP : La POO en pratique avec ZString](#)



- [Notre objectif](#)
- [Quelques préparatifs](#)
- [Constructeurs et destructeur](#)
- [La surcharge des opérateurs](#)
- [Récapitulatif](#)
- [Aller \(encore\) plus loin](#)

○

[11\) L'héritage](#)



- [Exemple d'héritage simple](#)
- [La dérivation de type](#)
- [Héritage et constructeurs](#)
- [La portée protected](#)

o

12) Eléments statiques et constants



- [Les méthodes constantes](#)
- [Les méthodes statiques](#)
- [Les attributs statiques](#)

•

Partie 2 : [Pratique] Créez vos propres fenêtres avec Qt

Vous l'avez compris en lisant la partie I : la POO, ce n'est pas évident à maîtriser au début, mais ça apporte un nombre important d'avantages : le code est plus facile à réutiliser, à améliorer, et... quand on utilise une bibliothèque là c'est carrément le pied 😊



Le but de la partie II est entièrement de pratiquer, pratiquer, pratiquer. Vous n'apprendrez pas de nouvelles notions théoriques ici, mais par contre vous allez apprendre à maîtriser le C++ par la pratique, et ça c'est important.

Qt est une bibliothèque C++ très complète qui vous permet notamment de créer vos propres fenêtres, que vous soyez sous Windows, Linux ou Mac OS. Tout ce que nous allons faire sera très concret : ouverture de fenêtres, ajout de boutons, création de menus, de listes déroulantes... bref que des choses motivantes ! 😊

o

1) Introduction à Qt



- [Dis papa, comment on fait des fenêtres ?](#)
- [Présentation de Qt](#)
- [Installation de Qt](#)

o

2) Compiler votre première fenêtre Qt



- [Codons notre première fenêtre !](#)
- [Compiler un projet Qt : la théorie](#)
- [Compiler un projet Qt : la pratique](#)
- [Exécuter le programme](#)

o

3) Personnaliser les widgets



- [Modifier les propriétés d'un widget](#)
- [Qt et l'héritage](#)
- [Un widget peut contenir un autre](#)
- [Hériter un widget](#)

o

4) Les signaux et les slots



- [Le principe des signaux et slots](#)
- [Connexion d'un signal à un slot simple](#)
- [Des paramètres dans les signaux et slots](#)

- [Créer ses propres signaux et slots](#)

○

5) Les boîtes de dialogue usuelles



- [Afficher un message](#)
- [Saisir une information](#)
- [Sélectionner une police](#)
- [Sélectionner une couleur](#)
- [Sélection d'un fichier ou d'un dossier](#)

○

6) Apprendre à lire la documentation de Qt



- [Où trouver la doc ?](#)
- [Les différentes sections de la doc](#)
- [Comprendre la documentation d'une classe](#)

○

7) Positionner ses widgets avec les layouts



- [Le positionnement absolu et ses défauts](#)
- [L'architecture des classes de layout](#)
- [Les layouts horizontaux et verticaux](#)
- [Le layout de grille](#)
- [Le layout de formulaire](#)
- [Combiner les layouts](#)

○

8) Les principaux widgets



- [Les fenêtres](#)
- [Les boutons](#)
- [Les afficheurs](#)
- [Les champs](#)
- [Les conteneurs](#)

○

9) TP : ZeroClassGenerator



- [Notre objectif](#)
- [Correction](#)
- [Des idées d'améliorations](#)

○

10) La fenêtre principale



- [Présentation de QMainWindow](#)
- [La zone centrale \(SDI et MDI\)](#)
- [Les menus](#)
- [La barre d'outils](#)
- [Les docks](#)
- [La barre d'état](#)

o

11) Traduire son programme avec Qt Linguist



- [Les étapes de la traduction](#)
- [Préparer son code à une traduction](#)
- [Créer les fichiers de traduction .ts](#)
- [Traduire l'application sous Qt Linguist](#)
- [Lancer l'application traduite](#)

o

12) Modéliser ses fenêtres avec Qt Designer



- [Présentation de Qt Designer](#)
- [Placer des widgets sur la fenêtre](#)
- [Configurer les signaux et les slots](#)
- [Utiliser la fenêtre dans votre application](#)

o

13) TP : zNigo, le navigateur web des Zéros !



- [Les navigateurs et les moteurs web](#)
- [Organisation du projet](#)
- [Génération de la fenêtre principale](#)
- [Les slots personnalisés](#)
- [Conclusion et améliorations possibles](#)

o

14) L'architecture MVC avec les widgets complexes



- [Présentation de l'architecture MVC](#)
- [L'architecture simplifiée modèle/vue de Qt](#)
- [Utilisation d'un modèle simple](#)
- [Utilisation de modèles personnalisables](#)
- [Gestion des sélections](#)

o

15) Communiquer en réseau avec son programme



- [Comment communique-t-on en réseau ?](#)
- [L'architecture du projet de Chat avec Qt](#)
- [Réalisation du serveur](#)
- [Réalisation du client](#)
- [Test du Chat et améliorations](#)

•

Partie 3 : Annexes

Besoin d'aller encore plus loin ?
Lisez donc les annexes !

o

1) Ce que vous pouvez encore apprendre



- [... sur le langage C++](#)
 - [... sur la bibliothèque standard](#)
 - [... sur la bibliothèque Qt](#)
-

Partie 1 : [Théorie] La Programmation Orientée Objet

Le langage C++ est basé sur le C : ce que vous avez donc appris dans le [cours de C](#) jusqu'ici va vous resservir, pour ne pas dire vous être indispensable !

Les modifications entre le C et le C++ sont nombreuses. La plus importante d'entre elles est l'introduction de la Programmation Orientée Objet, que l'on abrège couramment POO. On en entend souvent parler, mais qu'est-ce que c'est concrètement ?

La réponse se trouve dans cette partie du cours 😊

Introduction au C++

Le C++, enfin on y arrive ! 😊

J'espère que vous êtes encore en forme, car vous avez des tonnes de choses à apprendre (et pas des plus faciles 😱). Mais courage, si vous êtes arrivés là, vous serez capable de réussir à comprendre la suite ça ne fait aucun doute 😊

Le C++ ressemble au C en apparence, mais vous allez vite vous rendre compte qu'il est en fait bien différent.

Ce chapitre servira d'introduction. Nous allons commencer par voir ce qui différencie le C du C++ et quels sont les défauts du C qui ont conduit à la réalisation du C++.

Rappel IMPORTANT pour ceux qui n'auraient pas lu l'avertissement sur le sommaire : ce cours est la suite du [cours de C](#). [Vous devez avoir lu au moins les parties I et II du cours de C](#) avant de vous attaquer à la lecture de ce cours sur le C++, sinon vous n'allez rien comprendre !

Pourquoi avoir créé le C++ ?

Je vous l'ai dit dès le tout début du premier chapitre du cours, et je ne vous ai pas menti : le langage C n'est pas limité. Vous pouvez faire tout ce que vous voulez avec, des fenêtres (avec GTK+ par exemple), de la 2D (avec SDL), de la 3D (avec OpenGL)... Le tout est de trouver la bibliothèque qui propose ce dont vous avez besoin.

Alors du coup on peut se demander... pourquoi un gars s'est levé un jour et a dit : "Aujourd'hui je vais inventer un nouveau langage" ?

Le type en question s'appelle Bjarne Stroustrup, il est danois et il est l'auteur d'un des langages de programmation les plus utilisés dans le monde aujourd'hui. Et il n'est pas fou : il n'a pas fait ça pour le plaisir (enfin j'espère) !

Le C a des avantages

Maintenant que vous programmez en C, vous avez pu vous rendre compte certainement que celui-ci a un grand nombre d'avantages :

- C'est un langage très rapide car assez bas niveau. Si on a des calculs importants à faire, un programme écrit en C les exécute en moins de temps qu'il n'en faut pour dire "ouf" (sauf si vous avez codé avec les pieds bien entendu 🤣)
- C'est un langage portable. Le même code source peut être compilé aussi bien sous Windows, Linux, Mac OS et ne dépend d'aucun type de processeur particulier.
- Le langage est libre, ce qui permet à n'importe qui de (très) motivé d'écrire son propre compilateur. Cela explique donc la grande diversité des compilateurs aujourd'hui : GCC, mingw, MS Visual C++, Borland, et j'en passe.

Bref, autant de raisons valables d'aimer le C 😊

Mais le C a aussi des défauts !

Si le C n'avait que des avantages, ce serait le langage parfait. Or, les programmeurs savent très bien que le langage parfait n'existe pas. Tout langage a des défauts.

Donc le C n'est pas parfait. Que lui reproche-t-on ?

Déjà, certains concepts de programmation plus récents manquent :

- Les références, qui permettent d'éviter quelques prises de tête avec les pointeurs.
- Les exceptions, une technique puissante pour gérer les erreurs de ses programmes.
- ... et il y en a d'autres mais ça ne sert à rien de rentrer dans le détail de suite.

Mais le vrai problème du langage C est qu'il n'est pas prévu pour faire de la Programmation Orientée Objet, une technique de programmation particulièrement efficace apparue récemment.

La programmation orientée quoi ?

Non, ce n'est pas une insulte.

Bon d'accord, il faut avouer que quand quelqu'un nous dit : "Je fais de la programmation orientée objet", on a tendance à s'éloigner un peu de peur que ça soit contagieux 😊

Beaucoup de gens parlent ou ont entendu parler de Programmation Orientée Objet (que j'abrégerai maintenant tout le temps POO, ça va plus vite 🤪). Mais concrètement, la POO c'est quoi ?

La POO est un concept de programmation, une façon de programmer. Ce n'est pas un langage.

Le C++, lui, est un langage. Il a été principalement inventé pour faciliter l'utilisation de la POO.

La POO n'est pas utilisée qu'en C++. De nombreux langages, encore plus récents, exploitent au maximum les concepts de la POO. Je pense en particulier à Java et Python, mais il y en a bien d'autres.

Par ailleurs, il est possible de faire de la POO en C, mais c'est assez compliqué.

Bon, à quoi ça sert la POO ?

C'est une façon de programmer qui permet de rendre un code source plus facilement réutilisable, plus facile à modifier.

Comment ça ? Les programmes écrits en C sont difficiles à modifier ?

Ah non, je n'ai pas dit ça 😊

Simplement, quand le programme devient gros, il faut avouer qu'on finit assez facilement par se perdre dans toutes les fonctions qu'on a créées. La POO nous permet de mieux organiser notre code source, de lui donner une certaine logique. Les avantages de cette meilleure organisation sont nombreux, vous les découvrirez progressivement.

En fait, c'est un peu comme les pointeurs : vous n'en avez pas forcément compris l'intérêt tout de suite, mais je suis sûr que maintenant vous ne pouvez plus vous en passer ! 😊

L'idée à la base de la POO

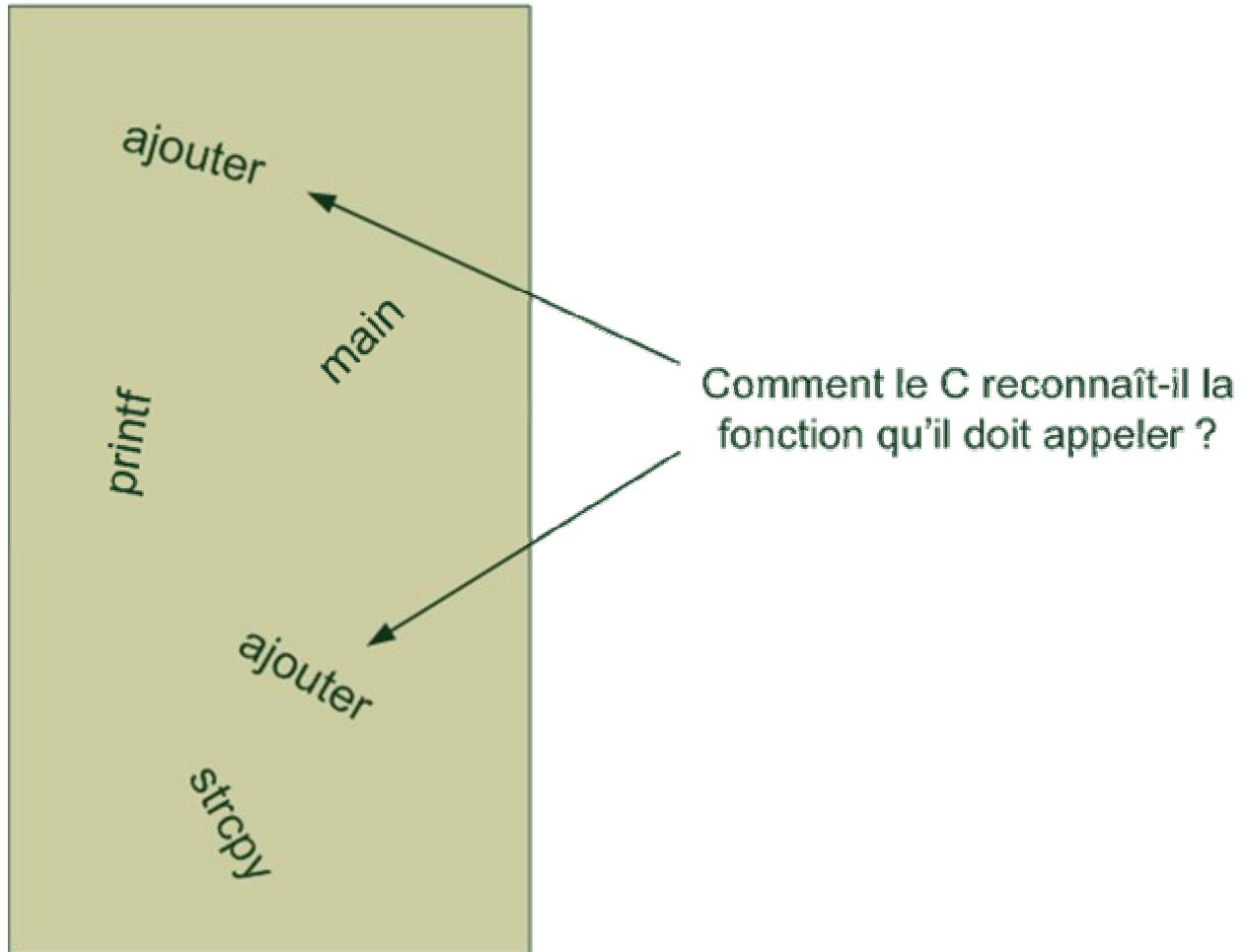
En C, vos programmes ne sont au final qu'un ensemble de fonctions accessibles de partout qui manipulent des tonnes de pointeurs. Même si vous pouvez faire plusieurs fichiers, cela ne suffit pas toujours à organiser correctement votre programme.

Par exemple, vous ne pouvez pas avoir deux fonctions nommées **ajouter** dans votre programme. Même si ces fonctions sont dans deux fichiers différents !

Ainsi, si vous avez une fonction `ajouter` permettant d'ajouter des heures et ailleurs une autre fonction `ajouter` permettant d'ajouter des euros, le langage C sera perdu et vous dira qu'il ne sait pas quelle fonction appeler quand vous demandez la fonction `ajouter`.

Imaginez que toutes les fonctions d'un programme en C nagent dans une seule et même grande piscine. C'est ce qu'on appelle l'espace global. Les fonctions évoluent toutes dans un même espace.

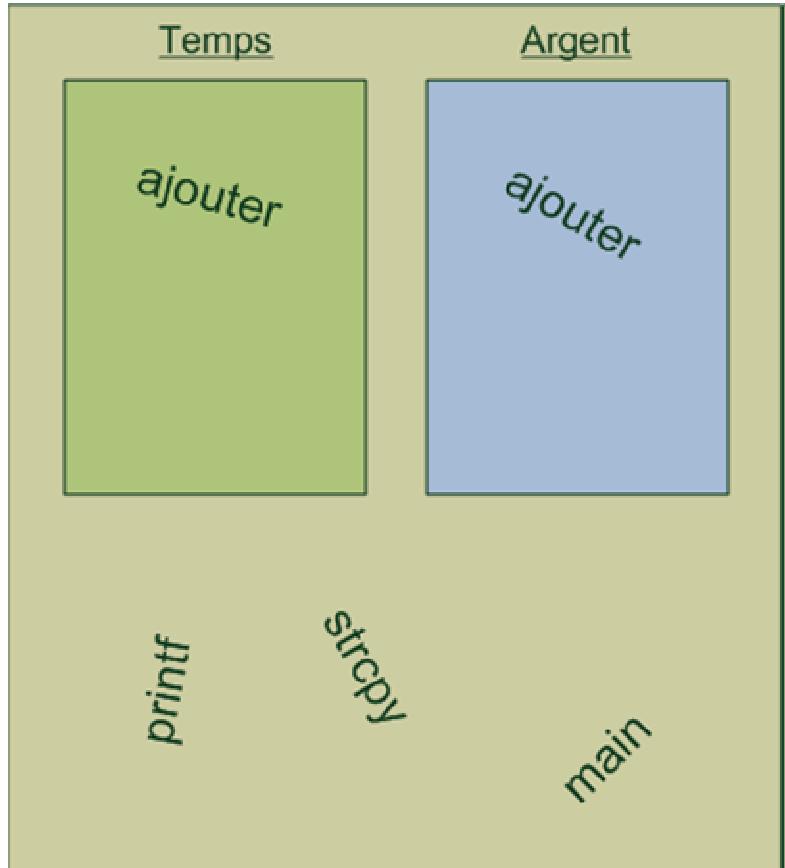
Votre programme



Imaginez maintenant si on séparait ces fonctions. On place la fonction `ajouter` spécialisée dans les heures dans une piscine, et on place la fonction `ajouter` spécialisée dans les euros dans une autre piscine. Du coup, vous n'avez plus qu'à vous rendre devant la piscine qui vous intéresse (par exemple la piscine spécialisée dans les heures) et vous pouvez alors appeler la fonction `ajouter`. Il n'y a pas de risque de conflit cette fois, parce que les fonctions sont confinées dans des espaces différents ! On ne les mélange plus



Votre programme



En C++, les fonctions sont compartimentées dans des espaces différents

Dans le schéma ci-dessus, j'ai créé 2 espaces pour séparer les fonctions ajouter. J'ai nommé ces espaces pour les identifier. On a maintenant :

- L'espace spécialisé dans la gestion du temps,
- L'espace spécialisé dans la gestion de l'argent.

Vous noterez que les autres fonctions comme main nagent encore dans l'espace global. Il est en effet toujours possible de mettre des fonctions dans l'espace global en C++.

D'autres langages interdisent carrément ce genre de choses (je pense à Java par exemple). Java est un langage complètement orienté objet : toutes les fonctions sont obligatoirement confinées dans des espaces particuliers et non dans l'espace global. Le C++ autorise encore de mettre des fonctions dans l'espace global, et c'est d'ailleurs en partie ce que certains lui reprochent (je vous rappelle que le langage parfait n'existe pas 😊).

Allons un peu plus loin

Est-ce qu'on ne peut mettre qu'une seule fonction par espace ?

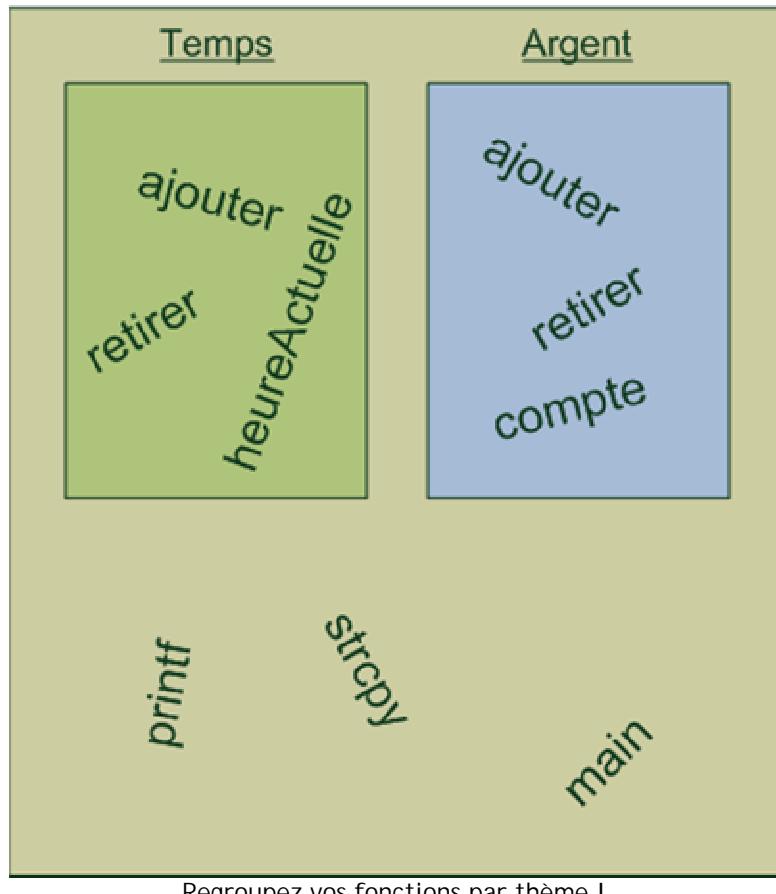
Non, bien sûr ! Le but, c'est de regrouper les fonctions de notre programme par "thème".

Prenons par exemple le thème Temps.
On peut ajouter des heures, mais on pourrait aussi créer la fonction qui retire des heures, une autre qui donne l'heure actuelle, etc.

De même pour le thème Argent. Disons que ça c'est l'argent que vous avez.
Vous pouvez aussi retirer de l'argent, créer la fonction qui vous donne la quantité d'argent que vous avez sur votre compte en banque, etc.

Ajoutons ces fonctions dans notre schéma :

Votre programme



Vous noterez qu'il y a 2 fonctions retirer : là encore ça ne pose pas de problème car elles sont dans 2 espaces différents.

Eh bien, croyez-moi si vous voulez, mais si vous avez compris ce que je viens d'expliquer à l'instant, vous avez déjà une bonne petite idée de ce qu'est la programmation orientée objet !

Maintenant je vous rassure : ça c'est vraiment la base de la base. La POO implique pas mal de règles, et l'organisation est parfois un peu déroutante. On va parler de POO dans pratiquement toute cette partie du cours, vous aurez donc le temps de vous rendre compte à quel point le sujet est riche 😊

Comme la POO est un vaaaaste sujet, nous n'allons pas l'aborder immédiatement (je veux pas vous tuer de suite 😱). Comme je vous l'ai dit au début, il y a de nombreuses différences entre le C et le C++. La plus importante d'entre elles est l'introduction de la POO, mais ce n'est pas la seule. Il y a aussi une foule de petites nouveautés pas bien compliquées à comprendre.

Voilà ce qu'on va faire :

1. Dans un premier temps nous allons découvrir toutes ces petites nouveautés qui n'ont aucun rapport avec la POO (ça prendra environ 3 chapitres).
2. Ensuite nous attaquerons la POO et je vous en ferai manger jusqu'à la fin de cette partie du cours 😊

On attaque notre premier programme C++ dès le chapitre suivant. Nous y découvrirons la notion de flux d'entrée / sortie. Une sorte de... mise en bouche quoi 😊

Premier programme C++ avec cout et cin

Après un bref chapitre d'introduction, nous pouvons commencer à coder nos premières lignes de C++ 😊

Ce chapitre sera assez simple : nous verrons quelles techniques on utilise en C++ pour afficher du texte à l'écran (dans une console) et comment on récupère du texte saisi au clavier. Vous allez voir que c'est assez différent ce qu'on connaît en C avec printf et scanf.

Nous réutiliserons cela dans tout le cours de C++. Soyez donc attentifs, et ça ne devrait pas vous poser de problème 😊

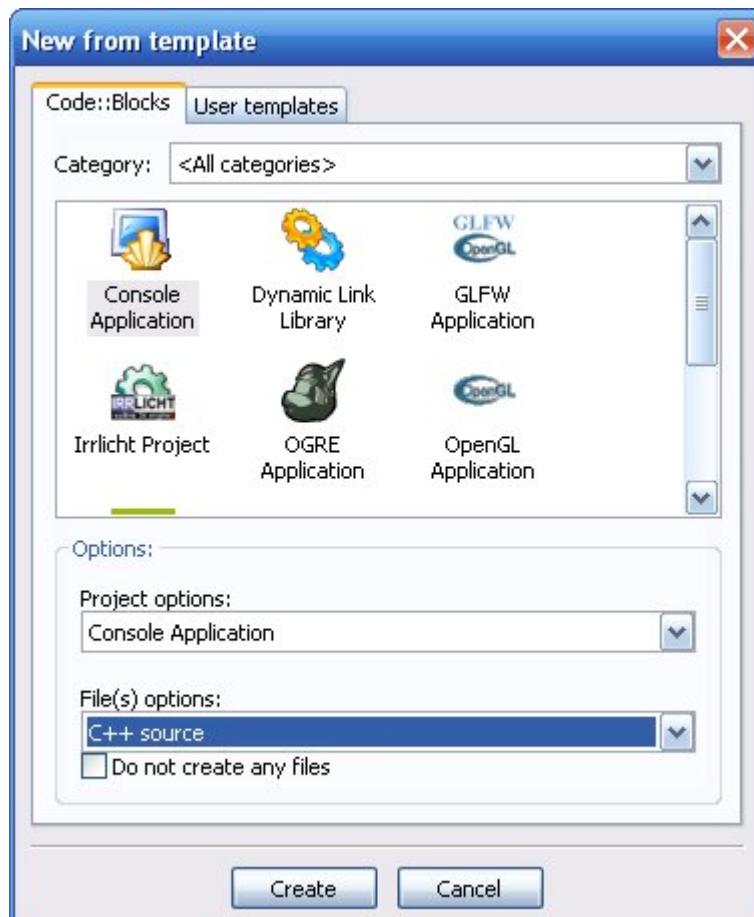
Configurer l'IDE pour le C++

Jusqu'ici, vous n'avez créé dans votre IDE que des projets en C.

Or, en C++ on utilise un autre compilateur. Par exemple, il y a gcc qui est un compilateur C, et g++ qui est un compilateur C++. Il va donc falloir dire à votre IDE que vous allez faire du C++, sinon il appellera le mauvais compilateur (et là ça risque pas de marcher 😐).

Lancez donc votre IDE favori. Pour ma part, vous l'aurez compris dans les chapitres précédents, je travaille principalement sous Code::Blocks. Si vous avez un autre IDE comme Visual C++ ou Dev C++ ça marche aussi 😊

Créez un nouveau projet de type console (eh oui, on retourne à la console pour faire nos expériences) et pensez à bien sélectionner C++.



Création d'un nouveau projet. Veillez à bien sélectionner C++

Si vous avez Dev C++ la manipulation est la même, je ne vous refais pas de screenshot vous êtes grands maintenant 😊
Sous Visual C++, le projet est par défaut compilé en C++, vous n'aurez donc pas besoin de spécifier quoi que ce soit.

Cliquez sur "Create" pour créer le nouveau projet.

Code::Blocks crée un premier fichier nommé main.cpp dans le projet avec quelques premières lignes de code C++.

En C++, vos fichiers .c ont l'extension .cpp. Les fichiers .h, eux, gardent, l'extension .h.

Certains trouvent cela illogique et ont choisi à la place d'utiliser .cc (pour les sources C++) et .hh (pour les headers C++). Ne soyez donc pas surpris par ce type de notation 😊

Voici le code de base que nous propose Code::Blocks dans notre nouveau projet :

Code : C++ - Sélectionner

```
#include <iostream>

int main()
{
    std::cout << "Hello world!" << std::endl;
    return 0;
}
```

Si vous avez un autre IDE, supprimez le code qui a été généré et utilisez celui-ci à la place pour qu'on soit sûrs de travailler sur le même code 😊

Analyse du premier code source C++

Intéressons-nous maintenant à chacune de ces lignes de code et voyons ce qui change pour le moment par rapport au C.

Include

Code : C++ - Sélectionner

```
#include <iostream>
```

On reconnaît là une bonne vieille directive de préprocesseur.

2 choses :

- Ce qui choque tout d'abord, c'est qu'il n'y a pas d'extension .h. En effet, en C++ les fichiers d'en-tête standard ne possèdent plus d'extension .h, mais vous verrez qu'il y a des exceptions (des gens qui n'ont pas encore fait l'évolution 🤪).
- D'autre part, la directive d'inclusion n'est plus la même. Ici, elle s'appelle iostream, ce qui signifie "flux d'entrée-sortie". Oubliez stdlib et stdio, ce sont des en-têtes du C, on ne les utilise plus en C++.

On inclut donc ici la librairie iostream qui contient les outils nécessaires pour afficher du texte dans la console et récupérer la saisie au clavier.

Fonction main()

Code : C++ - Sélectionner

```
int main()
{
    // ...
    return 0;
}
```

On retrouve notre fonction main habituelle. Cela fonctionne comme en C : tout programme commence par la fonction `main()`. Rien de choquant ici 😊

La fonction retourne 0, ce qui est là encore logique puisque la fonction doit retourner un int.

Pour info, 2 formes de main sont possibles. Celle-ci :

```
int main()
```

... mais aussi cette forme un peu plus compliquée que vous connaissez aussi :

```
int main(int argc, char *argv[ ])
```

La seconde forme permet de récupérer les arguments d'appel du programme, ce qu'on ne fait pas toujours. Vous pouvez donc vous contenter de la première forme qui est surtout plus simple à retenir 😊

cout

Code : C++ - [Sélectionner](#)

```
std::cout << "Hello world!" << std::endl;
```

La première ligne du main est la plus intéressante, c'est d'ailleurs la seule qui doit vraiment vous surprendre. En effet, ça ne ressemble pas à un appel de fonction, il y a plein de signes bizarres, pas de parenthèses comme on a l'habitude dans un appel de fonction. Bon sang de bonsoir qu'est-ce que c'est ? 😬

On va découvrir ça maintenant plus en détails 😊

Le flux de sortie cout

cout n'est pas une fonction mais un flux, un élément nouveau introduit en C++. Notez que ça n'a rien à voir avec la POO pour le moment 😐

Rassurez-vous, les fonctions existent toujours en C++ (d'ailleurs vous avez vu qu'il y a un main()), mais vous vous rendrez compte petit à petit qu'on ne les utilise plus de la même manière.

Le flux cout est l'équivalent de la fonction printf... mais en mieux, en plus simple 😊

Revoyons cette fameuse ligne de code :

Code : C++ - [Sélectionner](#)

```
std::cout << "Hello world!" << std::endl;
```

Il y a deux mots-clé particuliers dans cette ligne : cout et endl. Vous noterez qu'ils ont tous les deux le préfixe std:: : Tous les mots-clé de la librairie standard du C++ utilisent ce préfixe. Théoriquement, on est obligé de le mettre à chaque fois, mais c'est un peu lourd.

On a heureusement une solution pour se simplifier la vie. Rajoutez cette ligne de code avant le main() :

Code : C++ - [Sélectionner](#)

```
using namespace std;
```

Du coup, vous pouvez virer tous les préfixes std:: :

Ca rend déjà notre code un peu plus facile à lire :

Code : C++ - [Sélectionner](#)

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```

Voilà un premier point de réglé. Nous ne rentrerons pas dans le détail de ce "using namespace" pour le moment (pour ne pas compliquer inutilement les choses).

Intéressons-nous de plus près à cette ligne avec cout, désormais plus lisible :

Code : C++ - [Sélectionner](#)

```
cout << "Hello world!" << endl;
```

Vous voyez qu'il y a un nouveau symbole : le chevron <
On le rencontre d'ailleurs toujours par paire, comme ceci : <<

Imaginez que ces chevrons représentent en fait des flèches. Du coup, la ligne se lit de droite à gauche :

1. On prend le mot-clé endl, qui signifie retour à la ligne.
2. On l'insère à la fin de la chaîne "Hello world!"
3. On insère le résultat obtenu dans cout.

cout est la contraction de "c-out" (console out = sortie console). En bon français, vous devez prononcer cela "Ci Aoute" 🎉
cout représente la sortie en C++ (out = sortie). La sortie d'un programme, ben c'est tout simplement l'écran. Donc cout représente l'écran 😊

Résultat des courses, ce code signifie que le texte "Hello world!" suivi d'un retour à la ligne est envoyé vers l'écran. Il faut bien imaginer que les chevrons << indiquent le sens dans lequel les données sont envoyées.

Cette ligne de code commande donc un affichage de texte à l'écran. Compilez et exécutez le programme pour voir :

Code : Console - [Sélectionner](#)

```
Hello world!
```

Revenons un peu sur endl.

endl est un mot-clé qui signifie "fin de ligne" (end line en anglais). En fait, c'est tout bêtement un mot qui remplace le \n que vous connaissez du C, qu'on utilisait pour faire des sauts de ligne.

Ah bon, on ne peut plus utiliser \n en C++ ?

Si si. En fait, le mot-clé endl a été entre autres introduit pour améliorer la lisibilité du code source (pour ne pas qu'on mélange le "n" avec le texte à afficher).

Vous pouvez d'ailleurs tester, vous verrez que l'\n fonctionne toujours :

Code : C++ - [Sélectionner](#)

```
cout << "Hello world!\n";
```

Le résultat à l'écran est exactement le même :

Code : Console - [Sélectionner](#)

```
Hello world!
```

Désormais, j'utiliserai endl à la place de \n dans la suite du cours.

L'intérêt de cout

Pour le moment, vous devez vous dire que cout ressemble étrangement à la fonction printf, avec juste des symboles << en plus pour vous embrouiller l'esprit 😊

En fait, c'est encore plus facile à utiliser que printf. L'intérêt se voit notamment lorsqu'on veut afficher le contenu d'une variable. Regardez ce code, c'est super simple :

Code : C++ - [Sélectionner](#)

```
int main()
{
    int age = 21;
    cout << "Salut, j'ai " << age << " ans" << endl;
    return 0;
}
```

Code : Console - [Sélectionner](#)

```
Salut, j'ai 21 ans
```

Voilà, c'est assez intuitif pour que je n'aie pas besoin de vous expliquer comment ça marche 😊

Ce qui est génial, c'est qu'on n'a plus besoin de s'embêter avec la syntaxe des printf : %d, %lf, %s, %c etc... Ici, le langage est plus intelligent, il reconnaît le type de variable qui lui est envoyé.

Sceptiques ? Ok, essayez d'ajouter une variable de type chaîne de caractère dans ce cout :

Code : C++ - [Sélectionner](#)

```
int main()
{
    int age = 21;
    char pseudo[] = "M@teo21";
    cout << "Salut, j'ai " << age << " ans et je m'appelle " << pseudo << endl;
    return 0;
}
```

Code : Console - [Sélectionner](#)

```
Salut, j'ai 21 ans et je m'appelle M@teo21
```

Vous voyez, on a envoyé d'un coup à cout un entier et une chaîne de caractères, sans préciser le type de variable, et il n'a pas bronché 😊

Pour rappel, en C on aurait dû faire :

Code : C - [Sélectionner](#)

```
printf("Salut, j'ai %d ans et je m'appelle %s\n", age, pseudo);
```

Non seulement il fallait se souvenir des codes %d et %s, mais en plus les variables utilisées sont indiquées à la fin. En C++ avec cout, comme vous avez pu le constater, la variable est placée au milieu, ce qui rend le code plus facile à lire 😊

Bien entendu, vous n'êtes pas limité à un seul cout par programme 😊
Vous pouvez tout à fait faire plusieurs cout si vous le voulez :

Code : C++ - [Sélectionner](#)

```
cout << "Salut, j'ai " << age << " ans" << endl;
cout << "Je m'appelle " << pseudo << endl;
```

Résultat :

Code : Console - [Sélectionner](#)

```
Salut, j'ai 21 ans
Je m'appelle M@teo21
```

Le endl à la fin de chaque cout n'est pas obligatoire. Si vous l'enlevez, il n'y aura juste pas de retour à la ligne.
Entraînez-vous un peu avec cout, vous devriez vous y habituer rapidement !

Le flux d'entrée cin

Après avoir vu comment afficher du texte, voyons voir maintenant comment récupérer du texte saisi au clavier.
Là encore, ça fonctionne avec un système de flux, et vous allez voir que c'est un vrai régal de simplicité 😊

Le mot-clé à connaître ici est cin. Il vient en remplacement de la pratique (mais complexe) fonction scanf du langage C.
cin est la contraction de "c-in", ce qui signifie entrée console. Prononcez ça comme il faut siouplait : "Ci in" 🍪

cin représente l'entrée en C++. Et qu'est-ce que l'entrée ? C'est le clavier ! Eh oui, c'est par le clavier qu'on entre les données.

cin représente donc le clavier et permet de récupérer du texte saisi par l'utilisateur.
Tenez, on va faire un truc super original : on va lui demander son âge 🍪

Regardez comment ça fonctionne :

Code : C++ - [Sélectionner](#)

```
cin >> age;
```

Si vous êtes un peu observateur, 2 choses doivent vous avoir choqué :

- Les flèches ont changé de sens ! Eh oui, la lecture se fait ici de gauche à droite. cin représente le clavier et envoie les données dans la variable age. Il faut donc imaginer que les données transitent du clavier vers la variable age. C'est ce qu'on appelle un flux 🍪
- Il n'y a pas de symbole & devant age ! En C, on aurait dû écrire &age pour envoyer l'adresse de la variable à la fonction pour qu'elle sache où écrire en mémoire. En C++, c'est plus la peine ! Il y a en effet un mécanisme qui remplace un peu les pointeurs qu'on appelle les références. On étudiera ça dans le prochain chapitre plus en détails.

Ne confondez pas le sens des flèches entre cout et cin. Le principe c'est que les données transitent dans un sens précis : de la

mémoire vers l'écran (cout) ou du clavier vers la mémoire (cin).

Les flèches sont donc dans le sens de déplacement des informations. Avec un peu de bon sens, vous ne devriez pas vous tromper 😊

Programmes de test de cin

Testons maintenant cin dans un petit programme. Voici le code complet :

Code : C++ - [Sélectionner](#)

```
#include <iostream>

using namespace std;

int main()
{
    int age = 0;
    cout << "Quel age avez-vous ?" << endl;
    cin >> age;
    cout << "Ah ! Vous avez donc " << age << " ans !" << endl;
    return 0;
}
```

Code : Console - [Sélectionner](#)

Quel age avez-vous ?

21

Ah ! Vous avez donc 21 ans !

Désolé si je me répète, mais je trouve cela d'une simplicité effarante 😊

Finis les %d qui nous agacent, finis les oubli de symbole & dans les scanf, finis 😊

Bon d'accord, cette nouvelle syntaxe surprend un peu quand on a fait pas mal de C avant, mais on s'y fait vite rassurez-vous.

On peut aussi s'entraîner à demander le pseudonyme de l'utilisateur :

Code : C++ - [Sélectionner](#)

```
#include <iostream>

using namespace std;

int main()
{
    char pseudo[50];
    cout << "Quel est votre pseudo ?" << endl;
    cin >> pseudo;
    cout << "Salut " << pseudo << endl;
    return 0;
}
```

Code : Console - [Sélectionner](#)

Quel est votre pseudo ?

M@teo21

Salut M@teo21

Alors, premières impressions du C++ ? 😊

"Ca change" ? Ah ben oui un peu, c'est sûr 😊

En effet, la syntaxe des flux cout et cin surprend un peu... mais je suis sûr que vous serez convaincu comme moi que grâce à ce système de nombreuses choses sont simplifiées !

On va continuer notre tour d'horizon des ajouts-au-C++-qui-n'ont-rien-à-voir-avec-la-POO dans le chapitre suivant.

Au programme, nous allons découvrir les changements au niveau de la gestion des variables. Nous verrons en particulier ce que sont ces mystérieuses références dont vous avez entendu parler.

Nouveautés pour les variables

Nous continuons notre tour d'horizon des nouveautés du C++ dans ce chapitre. Nous n'allons pas encore voir ici la POO, mais patience, ça ne saurait tarder 😊

Nous nous intéresserons aux nouveautés relatives aux variables, c'est-à-dire à la gestion de la mémoire. Nous découvrirons entre autres le type bool, les modifications par rapport aux définitions de variables, les allocations dynamiques en C++ et les références.

Rien de bien difficile au programme donc, mais il s'agit de nouveautés importantes, donc à ne pas négliger.

Le type bool

On a découvert dès le début du cours de C qu'il existait un grand nombre de types de variable différents :

- int
- long
- float
- double
- char
- etc.

Le problème s'est posé lorsqu'on a voulu stocker des booléens. Faute d'avoir un type de donnée spécialisé dans le stockage des booléens, on a fait comme la plupart des programmeurs C font : on a utilisé un type entier, comme int.

Rappel. Un booléen est une variable qui peut prendre 2 valeurs : vrai ou faux. Par exemple, "majeur" est un booléen : soit on est majeur, soit on ne l'est pas 🍫

On dit que le nombre 0 représente "Faux", tandis que le nombre 1 représente "Vrai" (en fait, tout nombre différent de 0 signifie "Vrai").

Or, si on utilise int pour les booléens, on risque de les confondre avec des variables destinées à stocker des nombres, puisque int est à la base fait pour stocker des nombres !

Petit exemple tout simple :

Code : C - Sélectionner

```
int majeur = 1;
int age = 21;
```

La variable majeur est un booléen, car elle signifie soit vrai soit faux.

La variable age, elle, est un nombre. Elle peut valoir par exemple 21.

Mais comment fait-on pour savoir laquelle de ces variables est un booléen et laquelle est un nombre ?

On peut se baser sur le nom de la variable, c'est sûr, mais il aurait été plus pratique et plus clair d'avoir un type spécial pour les booléens.

Ca tombe bien ! Il y a justement en C++ un nouveau type de base : le type bool. Toute variable de ce type peut prendre 2 valeurs :

- true, qui signifie vrai.
- false, qui signifie faux.

(je vous conseille de retenir ces 2 valeurs par coeur, vous en aurez besoin 😊)

Du coup, le code qu'on a vu plus haut s'écrirait comme ceci en C++ :

Code : C++ - Sélectionner

```
bool majeur = true;  
int age = 21;
```

Voilà une bonne chose qui nous permettra d'éviter des ambiguïtés dans nos programmes 😊

Il n'y a pas de guillemets autour de true, car c'est un mot-clé du langage C++. Ce n'est pas une chaîne de caractères !

Rappel : les booléens dans les conditions

Je tiens juste à vous faire un petit rappel. Si vous avez bien suivi le cours de C, ça ne devrait pas vous choquer 😊

En théorie, on peut tester un booléen comme ceci dans une condition :

Code : C++ - Sélectionner

```
if (majeur == true) // S'il est majeur (forme longue)  
{  
    // ...  
}
```

Mais en général, si la variable a un nom clair, on préfèrera enlever la partie == true. C'est tout à fait possible et l'ordinateur le comprend très bien :

Code : C++ - Sélectionner

```
if (majeur) // S'il est majeur (forme courte)  
{  
    // ...  
}
```

Ce code est plus lisible et plus court que le précédent. On comprend bien que la condition est "S'il est majeur".

Par ailleurs, le point d'exclamation sert à exprimer la négation. Dans notre cas, ce code signifierait "S'il n'est pas majeur" :

Code : C++ - Sélectionner

```
if (!majeur) // S'il n'est PAS majeur  
{  
    // ...  
}
```

Ce n'est pas une nouveauté du C++ car ça existait déjà en C, mais je tenais juste à vous informer que cette technique fonctionnait toujours avec le type bool 😊

Les déclarations de variables

En C, les variables devaient être déclarées (= créées) au début des fonctions. Vous avez vu cela dans le chapitre sur les variables au tout début du cours 😊

Vous deviez donc faire toutes vos déclarations avant de commencer les instructions :

Code : C - Sélectionner

```
void maFonction()
{
    // D'abord on déclare les variables
    double prixOrigine = 0.0;
    double prixAchat = 0.0;
    double difference = 0.0;
    FILE* fichier = NULL;

    // Ensuite on peut exécuter des instructions, des appels de fonction, etc.
    fichier = fopen("exemple.txt", "r");

    if (fichier != NULL)
    {
        fonction();

        fscanf(fichier, "%lf", &prixOrigine);
        fscanf(fichier, "%lf", &prixAchat);

        difference = prixAchat - prixOrigine;
        // etc.
    }
}
```

La nouveauté en C++, c'est que l'on peut désormais déclarer des variables n'importe où dans une fonction. C'est plus pratique lorsqu'on programme, ça nous évite d'avoir à remonter au début de la fonction si on n'a besoin d'une variable qu'à un moment de la fonction. Cela peut aussi améliorer la lisibilité du code surtout dans de grosses fonctions.

On pourrait donc écrire le code précédent comme ceci en C++ :

Code : C++ - Sélectionner

```
void maFonction()
{
    FILE* fichier = NULL;
    fichier = fopen("exemple.txt", "r");

    if (fichier != NULL)
    {
        fonction();

        double prixOrigine = 0.0; // Déclaration au milieu
        fscanf(fichier, "%lf", &prixOrigine);

        double prixAchat = 0.0; // Autre déclaration au milieu
        fscanf(fichier, "%lf", &prixAchat);

        double difference = prixAchat - prixOrigine; // Encore autre déclaration au milieu
        // etc.
    }
}
```

Avec une version récente du langage C, il est aussi possible de déclarer une variable en plein milieu d'une fonction. Cependant, les

programmeurs C préfèrent en général continuer à déclarer leurs variables au début des fonctions.

Précision importante : les variables ainsi créées sont locales aux blocs où elles ont été déclarées. Je m'explique. On dit que les accolades { et } délimitent des blocs. Dans le code ci-dessus, vous devriez en voir deux : la fonction et le bloc if. Comme la variable prixAchat a été déclarée dans le bloc if, elle sera supprimée à la fin du bloc if. Si elle avait été déclarée au début de la fonction en revanche, elle aurait été accessible dans toute la fonction.

Voilà, c'est assez simple à comprendre mais il faut le savoir ! La variable est détruite à la fin du bloc dans lequel elle a été déclarée.

Code : C++ - Sélectionner

```
void maFonction()
{
    FILE* fichier = NULL;
    fichier = fopen("exemple.txt", "r");

    if (fichier != NULL)
    {
        fonction();

        double prixOrigine = 0.0; // Création de prixOrigine
        fscanf(fichier, "%lf", &prixOrigine);

        double prixAchat = 0.0; // Création de prixAchat
        fscanf(fichier, "%lf", &prixAchat);

        double difference = prixAchat - prixOrigine; // Création de difference
    } // Destruction automatique de prixOrigine, prixAchat et difference
} // Destruction automatique de fichier
```

Déclaration dans une boucle

Dans le même ordre d'idée, il y a une nouveauté vraiment très pratique (comprenez : je m'en sers tout le temps 😊). On peut déclarer une variable directement dans une instruction for.

Prenons un exemple. Vous codez votre programme, tout va bien. Puis à un moment, pour une raison ou une autre, vous avez besoin de faire une boucle qui se répète 10 fois. Vous allez sûrement faire un for. Mais pour boucler 10 fois, vous aurez besoin d'une variable de boucle qui va retenir le nombre de tours de boucle (quand on n'est pas inspiré on appelle en général cette variable i).

En C, c'est un peu embêtant parce qu'il faut remonter au début de la fonction pour rajouter la déclaration de la variable. En plus, on ne sait pas trop quand elle sera utilisée en lisant la déclaration :

Code : C - Sélectionner

```
void maFonction()
{
    int i = 0;

    /* Plein de code
     *
     */

    for (i = 0 ; i < 10 ; i++)
    {
    }
}
```

La nouveauté en C++, c'est que vous pouvez déclarer votre variable i directement dans l'instruction for. Elle sera détruite à la fin de la boucle, quand vous n'en aurez plus besoin.

Avantages : vous n'avez pas à remonter au début de la fonction pour déclarer la variable, et celle-ci est automatiquement détruite à la fin de la boucle. Pas d'utilisation inutile de la mémoire.

Le code C++ ressemblera donc à cela :

Code : C++ - Sélectionner

```
void maFonction()
{
    /* Plein de code
    ...
    ...
    */

    for (int i = 0 ; i < 10 ; i++) // Déclaration de i
    {

    } // Destruction automatique de i
}
```

Ca n'a l'air de rien, mais je vous assure qu'en pratique quand on programme, ça c'est vraiment génial 😊
Vous me verrez donc le faire la plupart du temps dans la suite du cours.

Les allocations dynamiques

Si je vous dis "malloc" et "free", ça vous rappelle de joyeux souvenirs non ? 😊

Code : C - Sélectionner

```
int main()
{
    int *variable = NULL;

    variable = malloc(sizeof(int)); // Allocation de mémoire

    free(variable); // Libération de mémoire

    return 0;
}
```

L'allocation dynamique est une technique qui permet de gérer vous-même l'allocation de mémoire pour vos variables. C'est notamment très pratique dans le cas de l'allocation de tableaux dont on ne connaît pas la taille avant compilation (revoyez le [chapitre sur l'allocation dynamique](#) au besoin !).

En C++, les allocations dynamiques existent toujours et on en fait toujours. D'ailleurs, les fonctions malloc et free sont toujours utilisables. Cependant, le C++ dispose de nouveaux opérateurs spécialisés dans les allocations dynamiques : new et delete.

new et delete sont des opérateurs, des mots-clé du langage C++. Contrairement à malloc et free, ce ne sont pas des fonctions. new et delete font en fait eux-mêmes appel aux fonctions malloc et free (on n'a pas réinventé la roue). Cependant, ils font aussi des tests et des initialisations supplémentaires, ce qui fait qu'on préférera toujours utiliser new et delete au lieu de malloc et free. Ils sont plus adaptés en C++.

Allocation dynamique d'une variable

new et delete ne s'utilisent pas exactement de la même manière que malloc et free.

On va dans un premier temps apprendre à s'en servir pour allouer une variable simple, puis on verra ensuite le cas de l'allocation de tableaux.

On souhaite donc allouer dynamiquement une variable (de type int par exemple).
En C++, on va d'abord devoir créer le pointeur et l'initialiser à NULL, ça on n'y coupe pas :

Code : C++ - Sélectionner

```
int *variable = NULL;
```

Allocation de mémoire

L'allocation de mémoire avec new se fait comme ceci :

Code : C++ - Sélectionner

```
variable = new int; // Allocation dynamique
```

Comparé à la "version C", il n'y a pas photo 😊

On n'a plus besoin d'utiliser l'opérateur sizeof() du C. Ici, on indique juste le type de variable à créer.

Libération de mémoire

Lorsque vous avez fini d'utiliser votre variable et que vous n'en avez plus besoin, vous devez la libérer avec l'opérateur delete. Ultra-simple :

Code : C++ - Sélectionner

```
delete variable; // Libération de mémoire
```

new et delete étant des opérateurs, et non des fonctions (désolé d'insister 🤪), on ne met pas de parenthèses.

Résumé

En résumé, voici à quoi ressemble un code d'allocation / libération de mémoire en C++ :

Code : C++ - Sélectionner

```
int main()
{
    int *variable = NULL;

    variable = new int; // Allocation de mémoire

    delete variable; // Libération de mémoire

    return 0;
}
```

Allocation dynamique d'un tableau

Si on veut allouer un tableau, l'opération est là encore très simple. On n'a plus besoin de faire un calcul du type $20 * \text{sizeof}(\text{int})$ comme on devait le faire en C.

On commence par créer le pointeur :

Code : C++ - Sélectionner

```
int *tableau = NULL;
```

Allocation de mémoire

Ensuite, l'allocation se fait comme ceci :

Code : C++ - [Sélectionner](#)

```
tableau = new int[20]; // Allocation de mémoire (20 cases)
```

Dans ce cas, un tableau de 20 cases sera alloué. Bien entendu, il est aussi possible de remplacer ce nombre par une variable :

Code : C++ - [Sélectionner](#)

```
tableau = new int[taille]; // Allocation de mémoire ("taille" cases)
```

La longueur du tableau sera définie par la valeur de la variable taille.

Libération de mémoire

Lorsque vous n'avez plus besoin du tableau, vous devez le libérer... avec cette fois l'opérateur `delete[]` pour bien préciser qu'il s'agit d'un tableau. Vous n'avez pas besoin de préciser la taille entre crochets, mais n'oubliez pas ces crochets ils sont importants.

Code : C++ - [Sélectionner](#)

```
delete[] tableau; // Libération de mémoire
```

Résumé

Code : C++ - [Sélectionner](#)

```
int main()
{
    int *tableau = NULL;

    tableau = new int[20]; // Allocation de mémoire (tableau)

    delete[] tableau; // Libération de mémoire (tableau)

    return 0;
}
```

Il y a donc en tout 4 opérateurs :

- `new` s'utilise avec `delete` pour allouer une variable
- `new[]` s'utilise avec `delete[]` pour allouer un tableau

Le `typedef` automatique

Vous souvenez-vous du chapitre sur les [structures et énumérations](#) ? 😊

On y avait appris à créer nos propres types de variables. On avait notamment utilisé l'exemple d'une structure nommée `Coordonnees` :

Code : C - Sélectionner

```
struct Coordonnees
{
    int x;
    int y;
};
```

Le problème des structures en C, c'est qu'il fallait placer le mot-clé struct au début de chaque déclaration d'une variable de type personnalisé :

Code : C - Sélectionner

```
struct Coordonnees point;
```

Pour éviter d'avoir à répéter ce mot à chaque déclaration, on avait découvert l'instruction typedef qu'on utilisait comme ceci avant la définition de notre structure :

Code : C - Sélectionner

```
typedef struct Coordonnees Coordonnees; // typedef permet d'éviter d'avoir à taper "struct"

struct Coordonnees
{
    int x;
    int y;
}.
```

Du coup, on pouvait déclarer une variable sans avoir à écrire struct devant :

Code : C - Sélectionner

```
Coordonnees point; // Le mot-clé struct est inutile grâce au typedef
```

La nouveauté

En C++, qu'on se rassure, les structures existent toujours (il y a même encore mieux, mais n'anticipons pas 😊).

La nouveauté du C++, c'est que le typedef est désormais automatique. A chaque fois que l'on déclare une structure (ou une énumération), un typedef est réalisé automatiquement par le compilateur. On peut donc n'écrire que l'instruction de déclaration de la structure :

Code : C++ - Sélectionner

```
// Le typedef est réalisé automatiquement par le compilateur, pas besoin de l'écrire

struct Coordonnees
{
    int x;
    int y;
};
```

Grâce à cela, le mot-clé struct devient totalement inutile lors d'une déclaration de variable :

Code : C++ - Sélectionner

```
Coordonnees point; // Le mot-clé struct est inutile grâce au typedef automatique
```

Les références

Nous arrivons maintenant au point le plus important (et délicat) de ce chapitre. Ouvrez grandes vos oreilles (ou plutôt vos yeux ).

Le C++ introduit un nouveau concept : les références. Une référence est un synonyme d'une autre variable. On verra ce que ça veut dire un peu plus loin .

Vous allez voir que les références ressemblent beaucoup aux pointeurs. Elles ont en effet été créées pour simplifier l'utilisation des pointeurs. Attention toutefois : je vous préviens qu'au début vous risquez de confondre les références avec les pointeurs (c'est assez perturbant quand on voit ça la première fois j'avoue .

Les références à l'intérieur d'une fonction

Pour créer une référence, on doit utiliser le symbole & dans la déclaration :

Code : C++ - Sélectionner

```
int &referenceSurAge;
```

Attention à ne pas confondre !

Dans une déclaration, le symbole & signifie "Je veux créer une référence" (c'est ce qu'on découvre maintenant). Partout ailleurs, le symbole & signifie "Je veux obtenir l'adresse de cette variable" (ça on l'avait déjà vu).

On confond facilement quand on débute. Il faut dire que les programmeurs n'ont pas été très malins en réutilisant le symbole & ici, y'a rien de tel pour confondre .

Quand vous voyez un & désormais, vérifiez s'il se trouve dans une déclaration : si c'est dans une déclaration, c'est qu'on cherche à créer une référence, sinon c'est qu'on demande à obtenir l'adresse de la variable.

Bon, on a créé une référence. Et alors ?

Et alors si vous compilez le code ci-dessus, le compilateur va vous insulter poliment :

Citation : Compilateur C++

```
error: 'referenceSurAge' declared as reference but not initialized
```

Si vous lisez l'anglais (et si vous ne le lisez pas vous devriez), vous avez compris le problème : le compilateur veut qu'on initialise immédiatement la référence.

Et ça c'est très important : une référence doit être immédiatement initialisée dès le début, contrairement aux pointeurs. Et ce n'est pas tout : une fois initialisée, la référence ne pourra plus changer !

Il y a donc deux règles que j'aimerais que vous reteniez par cœur :

- Règle 1 : une référence doit être initialisée dès sa déclaration.
- Règle 2 : une fois initialisée, une référence ne peut plus être modifiée.

Initialisation d'une référence

On va donc initialiser notre référence.

Comme je vous l'ai dit un peu plus tôt, une référence est un synonyme d'une autre variable. Cela veut donc dire qu'il faut créer une autre variable pour y trouver un minimum d'intérêt .

Allez hop, il est l'heure de ressortir la bonne vieille variable qui a fait ses preuves : la variable... age !

Pourquoi ? On l'a vu : une référence ne peut pas faire référence à une nouvelle variable une fois qu'elle a été initialisée. Un pointeur, lui, peut toujours pointer vers une nouvelle variable au cours de l'exécution du programme.

Dans certains langages récents, comme le Java, les pointeurs ont complètement disparu. On n'utilise plus que des références, ce qui limite beaucoup les risques d'erreur et simplifie les programmes. La différence, c'est qu'en Java on peut modifier les références en cours de route, contrairement au C++ 😊

Il est très courant de confondre les pointeurs et les références lorsqu'on débute (si ça peut vous rassurer, moi aussi j'ai pas mal confondu au début). Je vais donc vous donner 2 codes source : le premier utilise les pointeurs, le second les références. Si à un moment vous avez un doute et que vous vous mettez à confondre pointeurs et références, servez-vous de l'exemple ci-dessous pour vous assurer que vous faites les choses correctement :

----- Code d'exemple avec un pointeur -----	----- Code d'exemple avec une référence -----
<p>Code : C++ - Sélectionner</p> <pre>int main() { int age = 21; int *pointeurSurAge = &age; cout << *pointeurSurAge; *pointeurSurAge = 40; cout << *pointeurSurAge; return 0; }</pre>	<p>Code : C++ - Sélectionner</p> <pre>int main() { int age = 21; int &referenceSurAge = age; cout << referenceSurAge; referenceSurAge = 40; cout << referenceSurAge; return 0; }</pre>

Voilà, j'espère que ce comparatif vous permettra d'y voir plus clair 😊

Ce qu'il faut retenir dans l'histoire, c'est que les références sont là pour simplifier l'écriture du code source. Comme on n'a plus besoin d'utiliser le symbole * à chaque fois qu'on veut accéder à la variable age, on minimise les risques d'erreur dans nos programmes.

Les références vers des structures

Si vous faites une référence vers une structure, il faudra utiliser le symbole point "." et non le symbole flèche "->" lorsque vous voulez accéder à un élément d'une structure.

Code : C++ - Sélectionner

```
struct Coordonnees
{
    int x;
    int y;
};

int main()
{
    Coordonnees point;
    Coordonnees &referenceSurPoint = point;

    referenceSurPoint.x = 10;
    referenceSurPoint.y = 5;

    cout << "x : " << referenceSurPoint.x << endl;
    cout << "y : " << referenceSurPoint.y << endl;

    return 0;
}
```

Code : Console - Sélectionner

```
x : 10  
y : 5
```

Une fois de plus, vous voyez qu'une référence s'utilise exactement comme une variable 😊

Les références lors d'un appel de fonction

Les codes qu'on a vus jusqu'ici n'étaient pas très utiles. En pratique, on n'est pas suffisamment maso pour créer des références juste "pour le plaisir" si elles ne sont pas indispensables.

En fait, comme pour les pointeurs, les références révèlent toute leur utilité lorsqu'on appelle une fonction.

Voyons voir ça dans un exemple :

Code : C++ - Sélectionner

```
struct Coordonnees  
{  
    int x;  
    int y;  
};  
  
void remiseAZero(Coordonnees &pointAModifier);  
  
int main()  
{  
    Coordonnees point;  
  
    remiseAZero(point); // Pas besoin d'indiquer l'adresse de point avec un & lors de l'appel  
  
    return 0;  
}  
  
void remiseAZero(Coordonnees &pointAModifier) // La fonction indique qu'elle récupère une référence  
{  
    // La référence s'utilise exactement comme une variable  
    // On utilise donc des points "." et non des flèches "->"  
    pointAModifier.x = 0;  
    pointAModifier.y = 0;  
}
```

On transmet la référence à la fonction RemiseAZero le plus simplement du monde, sans avoir à mettre de symbole &.

Code : C++ - Sélectionner

```
remiseAZero(point);
```

Le but des références est là encore très clair : éviter d'avoir à taper des symboles en plus pour minimiser les erreurs.

La fonction doit bien préciser qu'elle reçoit une référence. On doit donc placer le symbole & dans la déclaration (et dans le prototype) :

Code : C++ - Sélectionner

```
void remiseAZero(Coordonnees &pointAModifier)
```

Ensuite, à l'intérieur de la fonction, on se sert de la référence comme si c'était une variable (dans le cas présent, on utilise donc le symbole point et non la flèche ->) :

Code : C++ - Sélectionner

```
pointAModifier.x = 0;  
pointAModifier.y = 0;
```

Comparaison pointeur / référence

Une fois de plus, je crois qu'il est utile que je vous fasse un comparatif du même code utilisant d'un côté un pointeur, de l'autre une référence.

----- Code d'exemple avec un pointeur ----- ----- Code d'exemple avec une référence -----

Code : C++ - Sélectionner

```
int main()  
{  
    Coordonnees point;  
  
    remiseAZero(&point);  
  
    return 0;  
}  
  
void remiseAZero(Coordonnees *pointAModifier)  
{  
    pointAModifier->x = 0;  
    pointAModifier->y = 0;  
}
```

Code : C++ - Sélectionner

```
int main()  
{  
    Coordonnees point;  
  
    remiseAZero(point);  
  
    return 0;  
}  
  
void remiseAZero(Coordonnees &pointAModifier)  
{  
    pointAModifier.x = 0;  
    pointAModifier.y = 0;  
}
```

Ces codes fonctionnent tous deux très bien en C++. Autant que possible, on utilisera des références en C++, sauf quand l'utilisation d'un pointeur est obligatoire.

Pour ceux qui se posent la question : on aurait tout à fait pu appeler la référence de la fonction "remiseAZero" point au lieu de pointAModifier. Il n'y a pas de risque de conflit avec la variable point du main car elle se trouve dans une autre fonction. J'ai juste changé le nom pour que vous évitez de les confondre 😊

A retenir : s'il y a un code que vous devez retenir pour les références, c'est celui de l'appel d'une fonction utilisant une référence (celui que nous venons de voir). Dans 99,99% des cas, on utilise les références lorsqu'on fait appel à une fonction.

Que de nouveautés ! C'est le moins qu'on puisse dire 😊

Et encore, vous n'avez pas tout vu ! Dans le prochain chapitre, nous découvrirons les nouveautés du C++ relatives aux fonctions.

Et après... après, je pense qu'on pourra commencer à parler de POO 😊

Tout ce que nous avons découvert dans ce chapitre est utile et sera largement utilisé par la suite. Prenez le temps de vous familiariser avec. Faites en particulier quelques tests et exercices avec les références car c'est un peu délicat au début, vu qu'on les mélange facilement avec les pointeurs. Heureusement, avec un peu d'expérience, on ne se trompe plus 😊

Nouveautés pour les fonctions

Nous avons vu que le C++ proposait de nombreuses nouveautés relatives pour les variables. Ce chapitre est la suite du précédent, mais est cette fois axé sur les nouveautés relatives aux fonctions.

Ce chapitre sera un peu plus court car il y a assez peu de changements au final. Ne vous endormez pas pour autant parce que vous allez découvrir les valeurs par défaut et les fonctions surchargées, deux éléments très importants que nous réutiliserons largement dans la suite.

Courage, c'est le dernier chapitre avant la POO (ou plutôt devrais-je dire : "Profitez-en bien mes petits ! 😊").

Des valeurs par défaut pour les paramètres

Si je vous dis "paramètre de fonction", vous voyez de quoi je parle n'est-ce pas ?
Je l'espère, parce qu'il serait temps de le savoir à votre niveau maintenant. 😊

Bon allez, un petit rappel !

Comme un petit rappel ne fait jamais de mal, voici un exemple de fonction :

Code : C++ - [Sélectionner](#)

```
int nombreDeSecondes(int heures, int minutes, int secondes)
{
    int total = 0;

    total = heures * 60 * 60;
    total += minutes * 60;
    total += secondes;

    return total;
}
```

Cette fonction calcule le nombre de secondes en additionnant les heures, minutes et secondes qu'on lui envoie. Rien de bien compliqué ! 😊

Les variables heures, minutes et secondes sont les paramètres de la fonction nombreDeSecondes. Ce sont des valeurs qu'elle reçoit, celles avec lesquelles elle va travailler.

Il est facile de reconnaître les paramètres d'une fonction, car ceux-ci se trouvent toujours écrits entre les parenthèses. 😊

Les valeurs par défaut

La nouveauté en C++, c'est qu'on peut donner des valeurs par défaut à certains paramètres de nos fonctions. Ainsi, on ne sera pas obligé d'indiquer à chaque fois tous les paramètres lorsqu'on appelle une fonction !

Pour bien voir comment on doit procéder, on va regarder le code complet. J'aimerais que vous le copiez dans votre IDE pour faire les tests en même temps que moi :

Code : C++ - [Sélectionner](#)

```

#include <iostream>

using namespace std;

// Prototype de la fonction
int nombreDeSecondes(int heures, int minutes, int secondes);

// Main
int main()
{
    cout << nombreDeSecondes(1, 10, 25) << endl;

    return 0;
}

// Définition de la fonction
int nombreDeSecondes(int heures, int minutes, int secondes)
{
    int total = 0;

    total = heures * 60 * 60;
    total += minutes * 60;
    total += secondes;

    return total;
}

```

Ce code donne le résultat suivant :

Code : Console - [Sélectionner](#)

4225

Sachant qu'1 heure = 3600s, 10 minutes = 600s, 25 secondes =... 25s, le résultat est logique car $3600 + 600 + 25 = 4225$. 😊
Bref, tout va bien.

Maintenant supposons que l'on veuille rendre certains paramètres facultatifs, par exemple parce qu'on utilise en pratique plus souvent les heures que le reste.
On va devoir modifier le prototype de la fonction (et non sa définition, attention).

Indiquez la valeur par défaut que vous voulez donner aux paramètres si on ne les a pas renseigné lors de l'appel de la fonction :

Code : C++ - [Sélectionner](#)

```
int nombreDeSecondes(int heures, int minutes = 0, int secondes = 0);
```

Dans cet exemple, seul le paramètre heures sera obligatoire, les deux autres étant désormais facultatifs. Si on ne renseigne pas les minutes et les secondes, les variables vaudront alors 0 dans la fonction.

Voici le code complet que vous devriez avoir sous les yeux :

Code : C++ - [Sélectionner](#)

```

#include <iostream>

using namespace std;

// Prototype avec les valeurs par défaut
int nombreDeSecondes(int heures, int minutes = 0, int secondes = 0);

// Main
int main()
{
    cout << nombreDeSecondes(1, 10, 25) << endl;

    return 0;
}

// Définition de la fonction, SANS les valeurs par défaut
int nombreDeSecondes(int heures, int minutes, int secondes)
{
    int total = 0;

    total = heures * 60 * 60;
    total += minutes * 60;
    total += secondes;

    return total;
}

```

Si vous avez lu attentivement ce code, vous avez dû vous rendre compte de quelque chose : les valeurs par défaut sont spécifiées uniquement dans le prototype, PAS dans la définition de la fonction ! On se fait souvent avoir, je vous préviens... 😠
Si vous vous trompez, le compilateur vous indiquera une erreur à la ligne de la définition de la fonction.

Bon, ce code ne change pas beaucoup du précédent. A part les valeurs par défaut dans le prototype, rien n'a été modifié (et le résultat à l'écran sera toujours le même). La nouveauté maintenant, c'est qu'on peut supprimer des paramètres lors de l'appel de la fonction (ici dans le main). On peut par exemple écrire :

Code : C++ - [Sélectionner](#)

```
cout << nombreDeSecondes(1) << endl;
```

Le compilateur lit les paramètres de gauche à droite. Comme il n'y en a qu'un et que seules les heures sont obligatoires, il devine que la valeur "1" correspond à un nombre d'heures.

Le résultat à l'écran sera le suivant :

Code : Console - [Sélectionner](#)

```
3600
```

Mieux encore, vous pouvez indiquer juste les heures et les minutes si vous le désirez :

Code : C++ - [Sélectionner](#)

```
cout << nombreDeSecondes(1, 10) << endl;
```

Code : Console - [Sélectionner](#)

```
4200
```

Du temps que vous indiquez au moins les paramètres obligatoires, il n'y a pas de problème. 😊

Cas particuliers, attention danger

Bon, mine de rien il y a quand même quelques pièges, ce n'est pas si simple que ça ! 🤔

On va voir ces pièges sous la forme de questions / réponses :

Et si je veux envoyer à la fonction juste les heures et les secondes, mais pas les minutes ?

Tel quel, c'est impossible. En effet, je vous l'ai dit plus haut, le compilateur va analyser les paramètres de gauche à droite. Le premier correspondra forcément aux heures, le second aux minutes et le troisième aux secondes.

Vous ne pouvez PAS écrire :

Code : C++ - [Sélectionner](#)

```
cout << nombreDeSecondes(1,,25) << endl;
```

C'est interdit. Si vous le faites, le compilateur vous fera comprendre qu'il n'apprécie guère vos manoeuvres. C'est comme ça : en C++, on ne peut pas "sauter" des paramètres, même s'ils sont facultatifs. Si vous voulez indiquer le premier et le dernier paramètre, il vous faudra obligatoirement spécifier ceux du milieu. On devra donc écrire :

Code : C++ - [Sélectionner](#)

```
cout << nombreDeSecondes(1, 0, 25) << endl;
```

Est-ce que je peux rendre juste les heures facultatives, et rendre les minutes et secondes obligatoires ?

Si le prototype est défini dans le même ordre que tout à l'heure : non.

Les paramètres facultatifs doivent obligatoirement se trouver à la fin (à droite).

Ce code ne compilera donc pas :

Code : C++ - [Sélectionner](#)

```
int nombreDeSecondes(int heures = 0, int minutes, int secondes);  
// Erreur : les paramètres par défaut doivent être à droite
```

3

5
6

4

La solution, pour régler ce problème, consiste à placer le paramètre heures à la fin :

Code : C++ - [Sélectionner](#)

```
int nombreDeSecondes(int secondes, int minutes, int heures = 0);  
// OK
```

Est-ce que je peux rendre tous mes paramètres facultatifs ?

Oui, ça ne pose pas de problème :

Code : C++ - [Sélectionner](#)

```
int nombreDeSecondes(int heures = 0, int minutes = 0, int secondes = 0);
```

Dans ce cas, l'appel de la fonction pourra être fait comme ceci :

Code : C++ - [Sélectionner](#)

```
cout << nombreDeSecondes() << endl;
```

Le résultat retourné sera bien entendu 0 dans notre cas. 😊

Règles à retenir

En résumé, il y a 2 règles que vous devez retenir pour les valeurs par défaut :

- Seul le prototype doit contenir les valeurs par défaut (pas la définition de la fonction).
- Les valeurs par défaut doivent se trouver à la fin de la liste des paramètres ("à droite").

La surcharge des fonctions

Ca, c'est probablement la nouveauté la plus importante des fonctions ! Cela nous aidera énormément lorsque nous ferons de la POO un peu plus loin 😊

De quoi s'agit-il ? D'un nouveau système en C++ qui permet de surcharger des fonctions.

En gros, et pour faire simple, c'est une technique qui nous permet de créer plusieurs fonctions ayant le même nom... sans que le compilateur crie au loup 😊

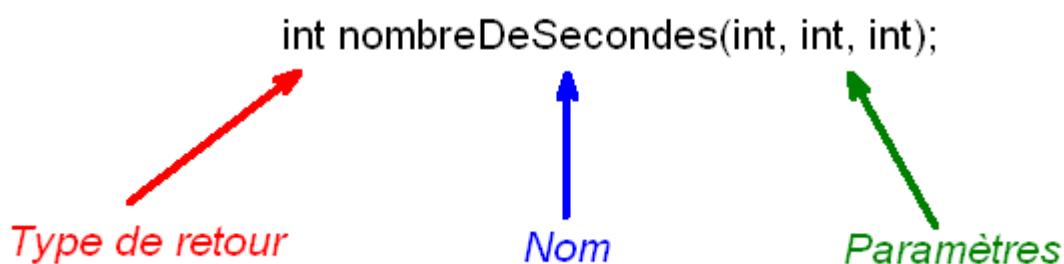
La signature d'une fonction

Avant toute chose, il faut que je vous parle de ce qu'on appelle la signature d'une fonction. C'est un peu sa carte d'identité, ce qui permet au compilateur de différencier les fonctions entre elles.

Chaque fonction est constituée de 3 éléments, ni plus ni moins :

- Un type de retour
- Un nom
- Une liste de paramètres

On va représenter ça sur un schéma pour être sûr qu'on voie bien la même chose 😊



Le compilateur se moque complètement des noms des variables passées en paramètre. Ce qui compte pour lui, c'est juste le type de ces paramètres. Je vous l'avais d'ailleurs dit dans le [chapitre sur la compilation modulaire](#) 😊

Voilà donc pourquoi j'ai marqué (int, int, int) pour les paramètres. C'est ce que le compilateur "voit", le nom des variables est donc ignoré pour l'identification de la fonction.

Bon, qu'est-ce qui permet d'identifier une fonction d'après vous ? Comment le compilateur fait-il pour vérifier si une fonction est bien différente d'une autre ?

En C, le compilateur se basait sur le nom, et uniquement sur le nom. Si 2 fonctions avaient le même nom, la compilation plantait. L'identification était donc faite sur le nom.

En C++, le compilateur se base sur le nom et les paramètres ! On peut avoir du coup 2 fonctions avec le même nom, à condition que celles-ci reçoivent des paramètres différents.

Le nom et les paramètres de la fonction constituent ce qu'on appelle la signature de la fonction. C'est ce qui permet au compilateur de l'identifier en C++.

int nombreDeSecondes(int, int, int);



Signature de la fonction

Le type de retour n'est donc pas pris en compte pour identifier la fonction.

La surcharge d'une fonction

La surcharge consiste à créer des fonctions qui ont le même nom, mais qui ont des paramètres différents (donc une signature différente).

Voici ce qui peut varier :

- Le nombre de paramètres
- Le type de chacun de ces paramètres

Encore une fois, je le rappelle, le nom que l'on donne à chacun des paramètres, le compilo il s'en fout complètement 🍷

Prenons un exemple pour bien comprendre ce que ça va nous permettre de faire. Imaginez une fonction addition. On peut additionner des entiers (int), mais aussi des décimaux (double).

En C, il aurait fallu nommer les deux fonctions différemment (par exemple sommeEntiers et sommeDecimaux). En C++, on peut leur donner le même nom et ça va grandement simplifier leur utilisation, vous allez voir.

Code : C++ - Sélectionner

```
int somme(int nb1, int nb2)
{
    return nb1 + nb2;
}

double somme(double nb1, double nb2)
{
    return nb1 + nb2;
}
```

Les prototypes de ces fonctions sont donc :

Code : C++ - Sélectionner

```
int somme(int nb1, int nb2);
double somme(double nb1, double nb2);
```

Leurs signatures sont :

Code : C++ - Sélectionner

```
somme(int, int)
somme(double, double)
```

Ces fonctions ont des signatures différentes et portent le même nom. Ce sont des fonctions surchargées 😊

Maintenant, dans le main on peut faire appel à la fonction somme pour additionner indifféremment des entiers ou des décimaux. C'est le compilateur qui décide quelle fonction il appelle en fonction du nombre et du type des paramètres.

Voici un code complet que vous pouvez tester :

Code : C++ - Sélectionner

```
#include <iostream>

using namespace std;

int somme(int nb1, int nb2);
double somme(double nb1, double nb2);

int main()
{
    cout << somme(10, 15) << endl << somme(2.5, 0.3) << endl;

    return 0;
}

int somme(int nb1, int nb2)
{
    return nb1 + nb2;
}

double somme(double nb1, double nb2)
{
    return nb1 + nb2;
}
```

Résultat :

Code : Console - Sélectionner

25

2.8

On a appelé 2 fois la fonction "somme", mais c'est en fait une fonction différente qui a été appelée à chaque fois 😊

Vous pouvez surcharger la fonction autant de fois que vous le désirez. On pourrait donc aussi rajouter par exemple la fonction qui fait la somme d'un entier et d'un décimal :

Code : C++ - Sélectionner

```
double somme(int nb1, double nb2)
{
    return nb1 + nb2;
}
```

... ou encore celle qui fait la somme de 3 entiers :

Code : C++ - [Sélectionner](#)

```
int somme(int nb1, int nb2, int nb3)
{
    return nb1 + nb2 + nb3;
}
```

Les possibilités sont infinies 😊

Bien entendu, on fait de la surcharge de fonction pour des choses plus "intéressantes" que des sommes, mais ça on le découvrira petit à petit en fonction de nos besoins.

Les fonctions inline

Ce que nous allons voir ici ressemble beaucoup aux macros (relisez le [chapitre sur le préprocesseur](#) si vous avez oublié ce que c'est 😊).

Les macros sont un bon moyen, utilisées intelligemment, d'accélérer la vitesse d'exécution du programme si certains bouts de code sont souvent réutilisés.

Toutefois, les macros sont assez délicates à manipuler et impliquent l'utilisation du préprocesseur.

En C++, on a inventé le mot-clé `inline` qui permet de faire, grossièrement, la même chose que les macros sans cette fois passer par le préprocesseur. C'est donc le compilateur qui se charge de faire le "remplacement de code" au moment de la compilation. L'avantage, c'est qu'on peut faire plus de vérifications (notamment sur les types des paramètres).

Exemple d'utilisation d'une fonction inline

Prenons l'exemple suivant (on le discutera ensuite) :

Code : C++ - [Sélectionner](#)

```
inline int carre(int nombre);

int main()
{
    cout << carre(10) << endl;

    return 0;
}

inline int carre(int nombre)
{
    return nombre * nombre;
}
```

Vous voyez que j'ai ajouté le mot-clé `inline` au début du prototype ET au début de la définition de la fonction. Cela signifie pour le compilateur "A chaque fois qu'on fera appel à la fonction `carre`, je placerais directement le code de cette fonction à l'endroit de l'appel".

En clair, après compilation voici ce qu'il restera dans votre exécutable :

Code : C++ - [Sélectionner](#)

```
int main()
{
    cout << 10 * 10 << endl;

    return 0;
}
```

La fonction inline disparaît complètement après compilation. Tout son code se trouve placé à l'endroit de l'appel (la ligne du cout dans notre cas).

L'avantage est que l'exécution du programme sera plus rapide, surtout si la fonction est appelée plusieurs fois. En effet, lors d'un appel "classique" de fonction, le processeur va sauter à l'adresse de la fonction en mémoire, retenir l'adresse où il en était pour revenir à la fonction appelante une fois l'autre fonction terminée... Bref, c'est très rapide, mais si la fonction est amenée à être appelée très souvent, il est préférable d'en faire une inline (on dit l'inliner ) pour éviter de répéter tout ce processus.

Le défaut, c'est que le programme risque de grossir un peu une fois compilé (le même code étant répété dans l'exécutable). Mais bon, en général cette différence est quand même négligeable 

En règle générale, les fonctions inline sont donc des fonctions très courtes que l'on est susceptible de réutiliser souvent, comme c'est le cas de la fonction carre ici.

En pratique, on utilise quand même assez peu les fonctions inline, sachez-le (c'est comme les macros, je ne pense pas que vous vous en soyez beaucoup servis jusqu'ici ). Ca reste cependant une des nouveautés du C++ relatives aux fonctions que je devais vous présenter 

pssst, puisqu'on y est, serez-vous capables de surcharger la fonction inlinée carre pour qu'elle calcule le carré d'un nombre décimal ? 

Bon, vous êtes capables de surcharger des fonctions inlinées avec des paramètres par défaut, qu'est-ce qui pourrait bien vous faire peur maintenant ? 

Oh, mais ne faites pas les malins, tout ceci n'était qu'une misérable mise en bouche comparé à ce qui vous attend 

En effet, dans le prochain chapitre on va rentrer en plein dans le coeur du C++ : on va découvrir la programmation orientée objet. Bien sûr, on va y aller pas à pas, en douceur, sinon ça risque d'être un peu... violent 

Quand vous êtes prêts, rendez-vous au proch... bon, je suis déjà dans le chapitre suivant moi, qu'est-ce que vous attendez ? 

La magie de la POO par l'exemple : string

Nous attaquons maintenant la 2ème moitié de la première partie du cours de C++. Et comme dans la vie rien n'est jamais simple, cette "deuxième moitié" sera la plus grosse et... la plus délicate aussi 

Nous allons maintenant, et dans les chapitres suivants, découvrir la notion de programmation orientée objet (POO). Comme je vous l'ai dit plus tôt, c'est une nouvelle façon de programmer. Ca ne va pas révolutionner immédiatement vos programmes, ça va vous paraître un peu inutile au début (comme lorsque vous aviez découvert les pointeurs ), mais faites-moi confiance : faites l'effort de faire ce que je dis à la lettre, et bientôt vous serez bien plus efficaces lorsque vous programmerez.

Ce chapitre va vous parler des 2 facettes de la POO, le côté utilisateur et le côté créateur.

Puis, je vais faire carrément l'inverse de ce que tous les cours de programmation font (je sais je suis fou ) : au lieu de commencer par vous apprendre à créer des objets, je vais d'abord vous montrer comment les utiliser avec pour exemple le type string fourni par le langage C++.

Des objets... pour quoi faire ?

Ils sont beaux, ils sont frais mes objets

S'il y a bien un mot qui doit vous frustrer depuis que vous en entendez parler, c'est celui-ci : objet.

Encore un concept mystique ? Un délire de programmeurs après une soirée trop arrosée ?

Non parce que franchement, un objet c'est quoi ? Mon écran est un objet, ma voiture est un objet, mon téléphone portable... ce sont tous des objets !

Bien vu, c'est un premier point 😊

En effet, nous sommes entourés d'objets. En fait, tout ce que nous connaissons (ou presque) peut être considéré comme un objet. L'idée de la programmation orientée objet, c'est de manipuler des éléments que l'on appelle des "objets" dans son code source.

Mais concrètement, c'est quoi ? Une variable ? Une fonction ?

Ni l'un, ni l'autre. C'est un nouvel élément en programmation.

Pour être plus précis, un objet c'est... un mélange de plusieurs variables et fonctions 😊

Ne faites pas cette tête-là, vous allez découvrir tout cela par la suite 😊

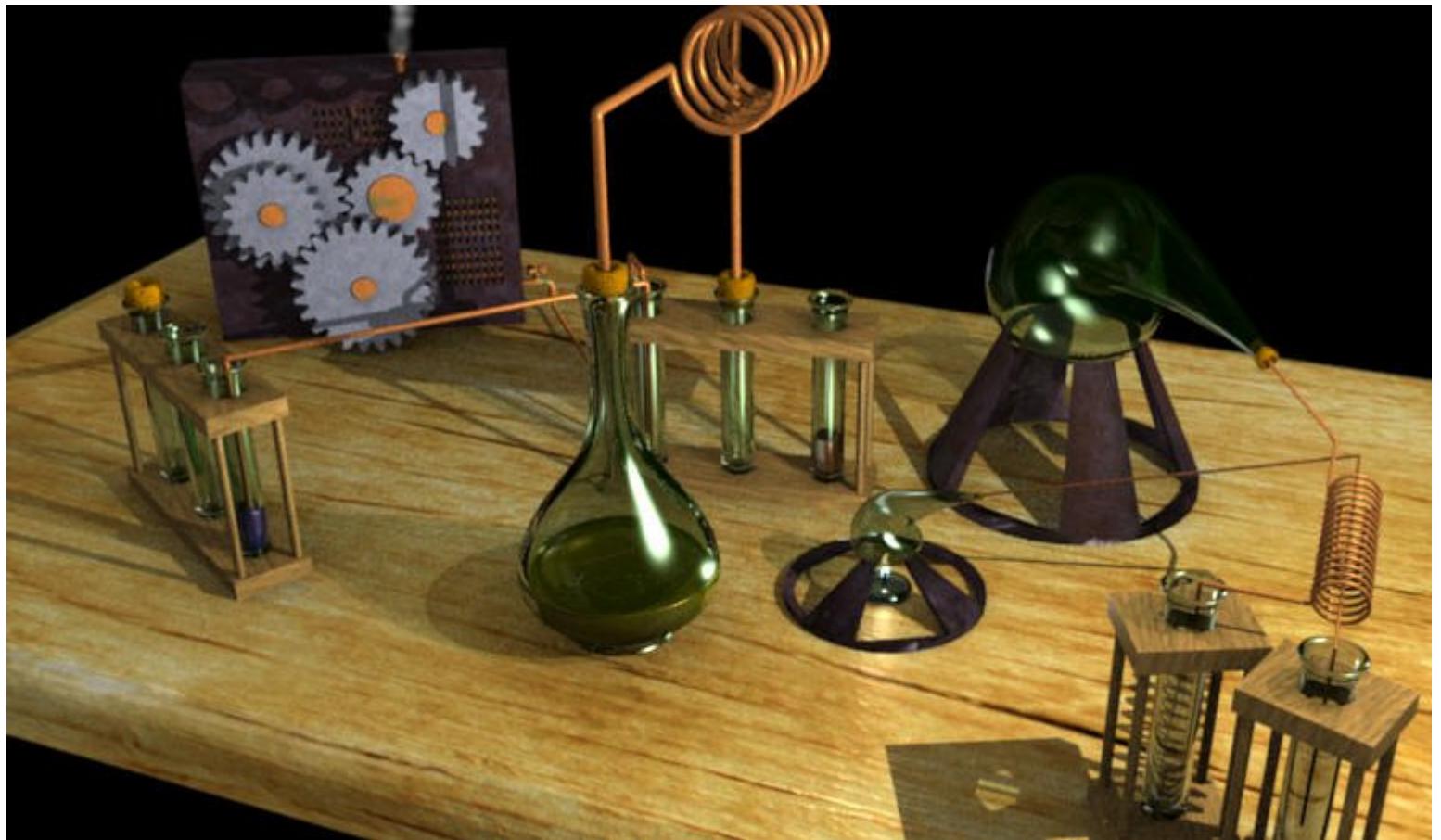
Imaginez... un objet

Pour éviter que ce que je vous raconte ressemble à un traité d'art moderne conceptuel, on va imaginer ensemble ce qu'est un objet à l'aide de plusieurs schémas concrets.

Les schémas 3D que vous allez voir par la suite ont été réalisés pour moi par l'ami Nab, que je remercie d'ailleurs vivement au passage.

Imaginez qu'un programmeur décide un jour de créer un programme qui permet d'afficher une fenêtre à l'écran, de la redimensionner, de la déplacer, de la supprimer... Le code est complexe : il va avoir besoin de plusieurs fonctions qui s'appellent entre elles, et de variables pour mémoriser la position, la taille de la fenêtre, etc.

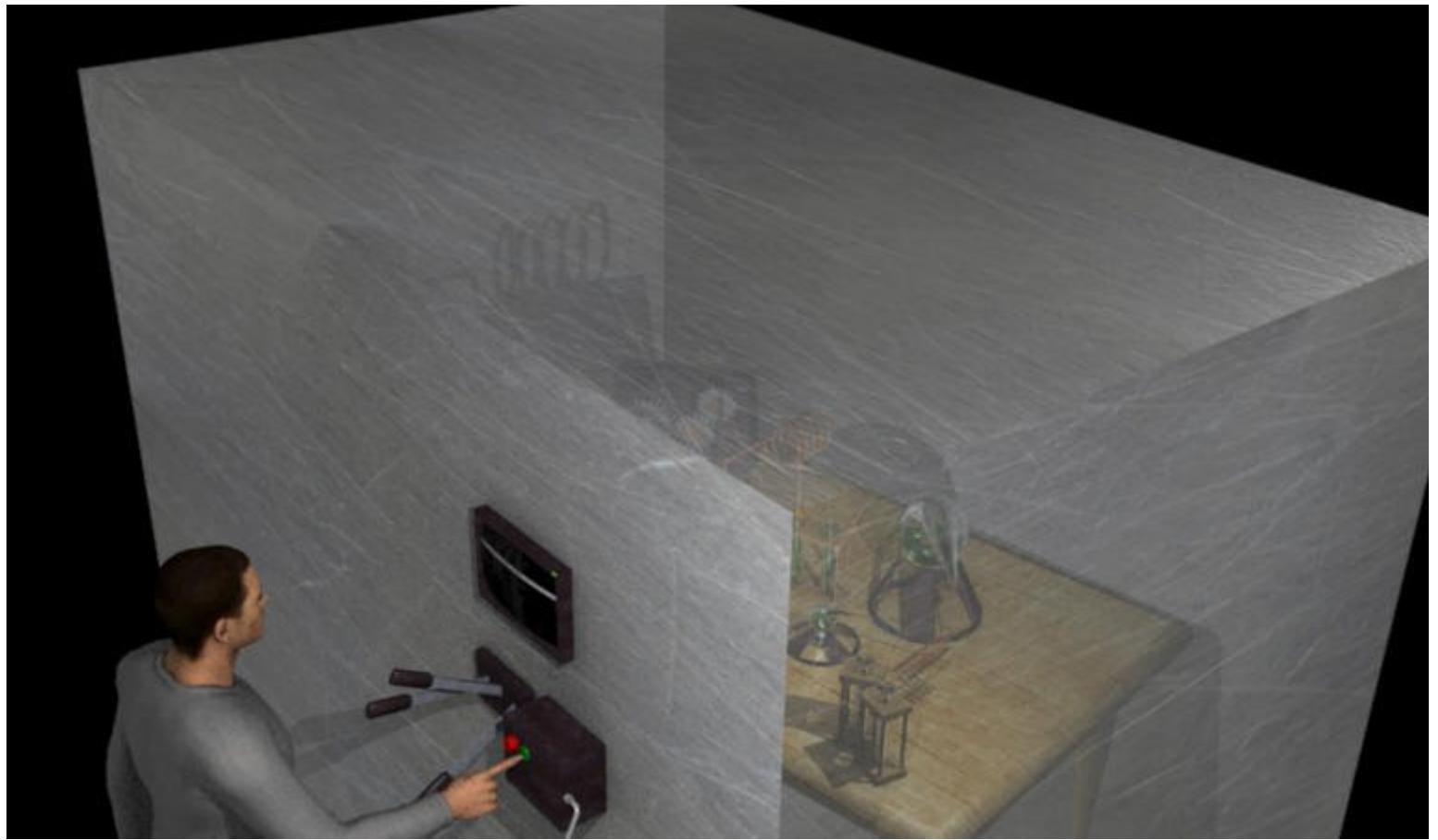
Il met du temps à écrire ce code, c'est un peu compliqué, mais il y arrive. Au final, le code qu'il a écrit est composé de plusieurs fonctions et variables. Quand on regarde ça pour la première fois, ça ressemble à une expérience de savant fou à laquelle on ne comprend rien :



Ce programmeur est content de son code et veut le distribuer sur internet pour que tout le monde puisse créer des fenêtres sans passer du temps à tout réécrire. Seulement voilà, à moins d'être un expert en chimie certifié, vous allez mettre pas mal de temps avant de comprendre comment tout ce bazar fonctionne.

Quelle fonction appeler en premier ? Quelles valeurs envoyer à quelle fonction pour redimensionner la fenêtre ?

C'est là que notre ami programmeur pense à nous. Il conçoit son code de manière orientée objet. Cela signifie qu'il place tout son bazar chimique à l'intérieur d'un simple cube. Ce cube est ce qu'on appelle un objet :



Ici, une partie du cube a été volontairement mise en transparence pour vous montrer que nos fioles chimiques sont bien situées à l'intérieur du cube. Mais en réalité, le cube est complètement opaque, on ne voit rien de ce qu'il y a à l'intérieur :



Ce cube contient toutes les fonctions et les variables (nos fioles de chimie), mais il les masque à l'utilisateur.

Au lieu d'avoir des tonnes de tubes et fioles chimiques dont il faut comprendre le fonctionnement, on nous propose juste quelques boutons sur la face avant du cube : un bouton "ouvrir fenêtre", un bouton "redimensionner", etc. L'utilisateur n'a plus qu'à se servir des boutons du cube et n'a plus besoin de se soucier de tout ce qui se passe à l'intérieur. Pour l'utilisateur, c'est donc complètement simplifié.

En clair : programmer de manière orientée objet, c'est **créer** du code source (peut-être complexe), mais que l'on masque en le plaçant à l'intérieur d'un cube (un objet) à travers lequel on ne voit rien. Pour le programmeur qui va **l'utiliser**, travailler avec un objet est donc beaucoup plus simple qu'avant : il a juste à appuyer sur des boutons et n'a pas besoin d'être diplômé en chimie pour s'en servir.

Bien sûr, c'est une image, mais c'est ce qu'il faut comprendre et retenir pour le moment 😊

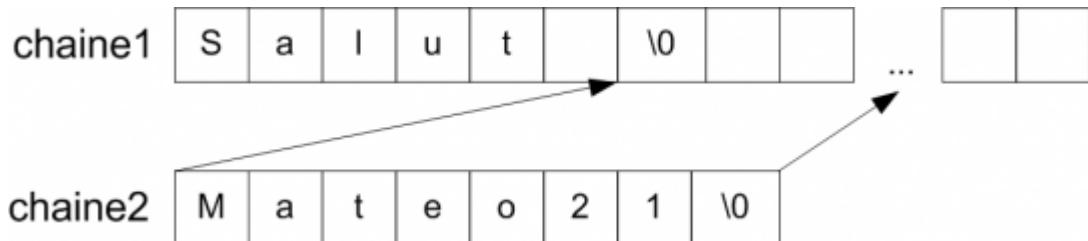
Nous n'allons pas voir tout de suite comment faire pour **créer** des objets. En revanche, nous allons apprendre à en **utiliser** un. Nous allons créer des objets de type string. Le type string est fourni par la librairie standard du C++.

Ce qui va suivre devrait vous convaincre une fois pour toutes que travailler avec des objets, bon sang que c'est simple 😊 (bon les créer sera une autre paire de manches, mais ne gâchons pas la fête qui va suivre 🍻).

Lire et écrire dans une chaîne via string

Vous vous souvenez des chaînes de caractères ? Vous vous souvenez de ce chapitre un peu compliqué où je vous avais dit qu'une chaîne était un tableau de char, que toute chaîne devait se terminer par un \0, qu'il fallait bien calculer la longueur de son tableau lorsqu'on le déclarait pour ne pas oublier la place pour l'\0 ?

Ça avait donné lieu à des schémas comme celui-ci, pour expliquer par exemple la concaténation de 2 chaînes :



Bon, soyons clairs, en C++ votre ordinateur ne fonctionne pas différemment 🤪

C'est toujours aussi complexe à gérer (et parfois dangereux si vous omettez l'\0)... Mais si on gérait les chaînes comme des objets ? Si au lieu d'avoir tout ce bazar d'\0 et de longueurs de tableaux à gérer, on plaçait tout ça dans un gros cube (un objet) ?

Notre cube proposerait en façade plusieurs boutons pour faire toutes les opérations possibles et imaginables avec des chaînes de caractères, tout en nous évitant d'avoir à savoir comment ça fonctionne à l'intérieur.

Ce type d'objet existe, il est déjà créé et il est livré dans la librairie standard du C++. Il s'appelle string ("chaîne" en anglais).

Votre premier objet

Allez n'attendons plus, et voyons comment on crée un objet de type string dans un code source 😊

Code : C++ - [Sélectionner](#)

```

#include <iostream>
#include <string> // Obligatoire pour pouvoir utiliser les objets string

using namespace std;

int main()
{
    string maChaine; // Création d'un objet "maChaine" de type string

    return 0;
}

```

Vous remarquerez pour commencer qu'il est nécessaire d'inclure le header de la librairie string pour pouvoir utiliser des objets de type string dans le code 😊 C'est ce que j'ai fait à la 2ème ligne.

Intéressons-nous maintenant à la ligne où je crée un objet de type string...

Mais... on crée un objet de la même manière qu'on crée une variable ?

Il y a plusieurs façons de créer un objet, celle que vous venez de voir est la plus simple. Et, oui, c'est exactement comme si on avait créé une variable !

Mais mais... comment on fait pour différencier les objets des variables ?

C'est bien tout le problème. Pour éviter la confusion, il y a des conventions (qu'on n'est pas obligé de suivre). La plus célèbre d'entre elles est la suivante :

- Si c'est une variable, la première lettre du type doit être en minuscule (ex : int)
- Si c'est un objet, la première lettre du type doit être en majuscule (ex : Voiture)

Je sais ce que vous allez me dire : "string ne commence pas par une majuscule !". Il faut croire que ceux qui ont créé string ne respectaient pas cette convention. Mais rassurez-vous, maintenant la plupart des gens mettent une majuscule au début de leurs objets (dont moi), ça sera pas la foire dans la suite de ce cours 😊

Affecter une valeur à la chaîne lors de la déclaration

Pour affecter une valeur à notre objet au moment de la déclaration, il y a plusieurs possibilités. La plus courante consiste à ouvrir des parenthèses comme si on appelait une fonction :

Code : C++ - Sélectionner

```

int main()
{
    string maChaine("Bonjour !"); // Création d'un objet "maChaine" de type string et affe
    return 0;

```

(vous commencez à comprendre ce que je vous racontais tout à l'heure quand je disais que les objets étaient une sorte de mélange de variables et fonctions 😊)

Dans la plupart des cas donc, on ouvrira des parenthèses pour affecter une valeur à l'objet lors de sa création. Pour le type string cependant, il est possible de faire encore plus simple en utilisant le symbole égal :

Code : C++ - Sélectionner

```
int main()
{
    string maChaine = "Bonjour !"; // Création d'un objet "maChaine" de type string et af
    return 0;
}
```

Bref, voilà qui est fait. On a un objet maChaine qui contient la chaîne "Bonjour !".
On peut l'afficher comme n'importe quelle chaîne de caractères avec un cout :

Code : C++ - Sélectionner

```
int main()
{
    string maChaine = "Bonjour !";
    cout << maChaine << endl; // Affichage du string comme si c'était une chaîne de caract
    return 0;
}
```

Code : Console - Sélectionner

Bonjour !

Affecter une valeur à la chaîne après déclaration

Maintenant que notre objet est créé, ne nous arrêtons pas là. Changeons la chaîne qui se trouve à l'intérieur. Donnez-lui la chaîne que vous voulez, il la stockera :

Code : C++ - Sélectionner

```
int main()
{
    string maChaine = "Bonjour !";
    cout << maChaine << endl;

    maChaine = "Bien le bonjour !";
    cout << maChaine << endl;

    return 0;
}
```

Code : Console - Sélectionner

Bonjour !
Bien le bonjour !

Mais... on n'a pas précisé la longueur de la chaîne au début, et maintenant on stocke dedans une chaîne plus grande qu'avant !
Comment on sait s'il y aura la place de stocker une chaîne aussi longue ?

C'est là que la magie de la POO opère. Vous, l'utilisateur, vous avez appuyé sur un bouton pour dire "Je veux maintenant que la chaîne à l'intérieur change pour Bien le bonjour !". A l'intérieur de l'objet, des mécanismes (des fonctions) se sont activées lorsque vous avez fait ça. Ces fonctions ont vérifié entre autres s'il y avait de la place pour stocker la chaîne. Elles ont vu que non. Elles ont alors créé un nouveau tableau de char, suffisamment long cette fois, pour stocker la nouvelle chaîne. Et elles ont détruit l'ancien tableau qui ne servait plus à rien, tant qu'à faire.

Et permettez-moi de vous parler franchement : ce qui s'est passé à l'intérieur de l'objet, on s'en fout royalement 😊
C'est bien là tout l'intérêt de la POO : l'utilisateur n'a pas besoin de comprendre comment ça marche à l'intérieur. L'objet est en quelque sorte intelligent et gère tous les cas. Nous, on ne fait que l'utiliser ici.

Du coup, pour nous c'est simplifié comme vous avez pas idée 😊

Avant, on ne pouvait pas affecter une chaîne avec le signe égal (sauf au moment de la déclaration), maintenant on peut le faire à n'importe quel moment ! Et on n'a plus à se soucier de la taille du tableau, elle est automatiquement recalculée !

Et ça, c'est rien qu'un début 🤪

Concaténation de chaînes

Allez, je vais continuer à vous faire baver 😊

On va concaténer (assembler) 2 chaînes. Regardez comment on fait :

Code : C++ - [Sélectionner](#)

```
int main()
{
    string chaine1 = "Bonjour !";
    string chaine2 = "Comment allez-vous ?";
    string chaine3;

    chaine3 = chaine1 + chaine2; // 3... 2... 1... Concaténatioooooon
    cout << chaine3 << endl;

    return 0;
}
```

Code : Console - [Sélectionner](#)

```
Bonjour !Comment allez-vous ?
```

Ah, allez je reconnaiss, il manque un espace au milieu. On n'a qu'à changer la ligne de la concaténation :

Code : C++ - [Sélectionner](#)

```
chaine3 = chaine1 + " " + chaine2;
```

Résultat :

Code : Console - [Sélectionner](#)

```
Bonjour ! Comment allez-vous ?
```

Niveau de simplicité : effarante 😊

Avant, il aurait fallu appeler une fonction pour la concaténation, et faire attention à la longueur de la chaîne qui est modifiée pour être sûr qu'il y ait suffisamment de place dedans. Ici, rien de tout cela 😊

On assemble donc nos chaînes de caractères à l'aide du symbole +. Et avouez franchement, si vous avez un tant soit peu programmé avant, que ça va vous faire gagner un temps de fou 😊

Comparaison de chaînes

Vous en voulez encore ? Très bien !

Sachez que l'on peut comparer des chaînes entre elles à l'aide des symboles == ou != (que l'on peut donc utiliser dans un if !).

Code : C++ - Sélectionner

```
int main()
{
    string chaine1 = "Bonjour !";
    string chaine2 = "Comment allez-vous ?";

    if (chaine1 == chaine2) // Faux
    {
        cout << "Les chaines sont identiques" << endl;
    }
    else
    {
        cout << "Les chaines sont differentes" << endl;
    }

    return 0;
}
```

Code : Console - Sélectionner

```
Les chaines sont differentes
```

C'est tout bête 😊

Vous pouvez vérifier que ça marche aussi dans le cas contraire :

Code : C++ - Sélectionner

```
int main()
{
    string chaine1 = "Bonjour !";
    string chaine2 = "Bonjour !";

    if (chaine1 == chaine2) // Vrai
    {
        cout << "Les chaines sont identiques" << endl;
    }
    else
    {
        cout << "Les chaines sont differentes" << endl;
    }

    return 0;
}
```

Code : Console - Sélectionner

```
Les chaines sont identiques
```

Si ça vous amuse, testez aussi le symbole "différent de" (!=), vous verrez que 2 objets de type string sont capables de se comparer entre eux 😊

Conclusion : plus simple tu meurs 😊

Opérations sur les string

Le type string ne s'arrête pas à ce que nous venons de voir. Comme tout bon objet qui se respecte, il propose un nombre important d'autres fonctionnalités qui permettent de faire tout ce dont on a besoin.

Nous n'allons pas passer toutes les fonctionnalités des string en revue (elles sont pas toutes indispensables et ce serait un peu long). Nous allons voir les principales dont vous pourriez avoir besoin dans la suite du cours 😊

Attributs et méthodes

Je vous avais dit qu'un objet était constitué de variables et de fonctions. En fait, on en reparlera plus tard mais le vocabulaire est un peu différent avec les objets. Les variables contenues à l'intérieur des objets sont appelées attributs, et les fonctions sont appelées méthodes.

Imaginez que chaque méthode (fonction) que propose un objet correspond à un bouton différent sur la façade avant du cube 😊

On parle aussi de "variables membres" et de "fonctions membres", ce qui est peut-être un peu moins déroutant que "attributs" et "méthodes" qui sont des mots complètement nouveaux et auxquels on a un peu de mal à se faire au début 😞

Appeler une méthode d'un objet se fait de la même manière qu'avec les structures. On utilise le point pour séparer l'objet de sa méthode :

objet . méthode()

En théorie, on peut aussi accéder aux variables membres (les "attributs") de l'objet de la même manière qu'on le faisait avec les structures. Cependant, en POO, il y a une règle super importante qui dit que l'utilisateur ne doit pas pouvoir accéder aux variables membres, mais seulement aux fonctions membres. On en reparlera dans le prochain chapitre plus en détail.

Quelques méthodes utiles du type string

La méthode size()

La méthode size() permet de connaître la longueur de la chaîne actuellement stockée dans l'objet de type string. C'est un peu l'équivalent de strlen(), mais pour les string cette fois 😊

Cette méthode ne prend aucun paramètre et renvoie la longueur de la chaîne. Comme vous venez de le découvrir, il va falloir appeler la méthode de la manière suivante :

Code : C++ - [Sélectionner](#)

```
maChaine.size()
```

Essayons ça dans un code complet qui affiche la longueur de la chaîne :

Code : C++ - [Sélectionner](#)

```
int main()
{
    string maChaine = "Bonjour !";
    cout << "Longueur de la chaine : " << maChaine.size();

    return 0;
}
```

Code : Console - [Sélectionner](#)

```
Longueur de la chaine : 9
```

Bingo ! 😊

C'est là toute la subtilité. Avant on aurait dû faire :

Code : C - [Sélectionner](#)

```
strlen(maChaine)
```

La fonction strlen était valable pour n'importe quelle chaîne, mais il fallait préciser en paramètre à chaque fois quelle était la chaîne sur laquelle la fonction devait travailler.

Maintenant, l'ordre est un peu inversé. La fonction size() est contenue dans l'objet maChaine. Quand on l'appelle comme on vient de le faire, la fonction membre size() sait qu'elle doit calculer la longueur de la chaîne contenue dans l'objet où elle se trouve :

Code : C++ - [Sélectionner](#)

```
maChaine.size()
```

La fonction size() est donc propre à l'objet maChaine. Si vous créez un deuxième objet de type string, il y aura donc une autre fonction size() propre à cet autre objet.

Rassurez-vous, les compilateurs C++ sont suffisamment intelligents pour optimiser l'utilisation de la mémoire. Si vous créez 50 string, il n'y aura pas 50 fois la même fonction en mémoire (une seule suffit). Mais ce que je vous dis là, c'est ce qu'il faut "imaginer". Dans chaque objet (chaque boîte), il y a une fonction size() qui est propre à l'objet. C'est comme cela qu'il faut le voir.

La méthode erase()

Cette méthode très simple supprime tout le contenu de la chaîne :

Code : C++ - [Sélectionner](#)

```
int main()
{
    string chaine = "Bonjour !";
    chaine.erase();
    cout << "La chaine contient : " << chaine << endl;

    return 0;
}
```

Code : Console - [Sélectionner](#)

```
La chaine contient :
```

Comme on pouvait s'y attendre, la chaîne ne contient plus rien 😊

Notez que c'est équivalent à faire :

Code : C++ - [Sélectionner](#)

```
chaine = "";
```

La méthode substr()

Une autre méthode qui peut s'avérer utile : substr(). Elle permet de ne prendre qu'une partie de la chaîne stockée dans un string. substr signifie "substring", soit "sous-chaîne" en anglais.

Tenez, on va regarder son prototype, vous allez voir que c'est intéressant :

Code : C++ - [Sélectionner](#)

```
string substr( size_type index, size_type num = npos );
```

Cette méthode retourne donc un objet de type string. Ce sera la sous-chaîne après "découpage". Elle prend 2 paramètres, ou plus exactement : 1 paramètre obligatoire, 1 paramètre facultatif. En effet, num possède une valeur par défaut (npos) ce qui fait que le second paramètre ne doit pas obligatoirement être renseigné.

- index permet d'indiquer à partir de quel caractère on doit couper (ce doit être un numéro de caractère)
- num permet d'indiquer le nombre de caractères que l'on prend. Par défaut, la valeur est npos, ce qui correspond à prendre tous les caractères qui restent. Si vous indiquez 2, la méthode ne renverra que 2 caractères.

Allez, un exemple sera plus parlant je crois 😊

Code : C++ - [Sélectionner](#)

```
int main()
{
    string chaine = "Bonjour !";
    cout << chaine.substr(3) << endl;

    return 0;
}
```

Code : Console - [Sélectionner](#)

```
jour !
```

On a demandé à couper à partir du 3ème caractère (soit la lettre "j" vu que la première lettre correspond au caractère n°0). On a volontairement omis le second paramètre facultatif, ce qui fait que du coup substr() a renvoyé tous les caractères restants avant la fin de la chaîne. Essayons de renseigner le paramètre facultatif pour ne pas prendre le point d'exclamation par exemple :

Code : C++ - [Sélectionner](#)

```
int main()
{
    string chaine = "Bonjour !";
    cout << chaine.substr(3, 4) << endl;

    return 0;
}
```

Code : Console - [Sélectionner](#)

```
jour
```

Bingo ! 😊

On a demandé à prendre 4 caractères en partant du caractère n°3, ce qui fait qu'on a récupéré "jour" 😊

La méthode c_str()

Celle-là est un peu particulière, mais parfois fort utile. Son rôle ? Retourner un pointeur vers le tableau de char que contient l'objet de type string.

Quel intérêt me direz-vous ? En C++, à priori aucun intérêt.

Mais il peut (j'ai bien dit il "peut") arriver que vous deviez envoyer à une fonction un tableau de char classique, façon C. Dans ce cas, la méthode c_str() vous permet de récupérer un bon vieux tableau de char comme on faisait en C.

Code : C++ - [Sélectionner](#)

```
int main()
{
    string chaine = "Bonjour !";
    const char* chaineC = NULL;

    chaineC = chaine.c_str(); // On récupère le tableau de char dans chaineC
    cout << "La chaine contient : " << chaineC << endl; // On l'affiche pour vérifier que

    return 0;
}
```

3

5

6

4

Récupérer le tableau de char vous sera utile si vous devez envoyer une chaîne à une fonction à la base prévue pour le C qui ne reconnaît pas les string. C'est rare, mais ça arrive. Je préfère que vous sachiez qu'on a cette possibilité pour pas que vous soyez bêtement bloqué à un moment.

Autant que possible, utilisez des objets de type string plutôt que des tableaux de char : vous avez vu que c'était bien plus facile à utiliser 😊

Comme le disait si bien ma prof d'informatique "C'est plus confortable de travailler avec un string" (je vous jure que c'est vrai, j'étais là 😊)

Bon plus sérieusement 😊

Vous avez découvert le côté **utilisateur** de la POO et à quel point ces nouveaux mécanismes pouvaient vous simplifier la vie.

Le côté **utilisateur** est en fait le côté simple de la POO. Les choses se compliquent lorsqu'on passe du côté **créateur**. Nous allons justement apprendre à créer des objets dans le prochain chapitre et tous les suivants. Une longue route pleine de péripéties nous attend 😊

Les classes (Partie 1/2)

Dans le chapitre précédent, vous avez vu que la programmation orientée objet pouvait nous simplifier la vie en "masquant" en quelque sorte le code complexe. Ca c'est un des avantages de la POO, mais ce n'est pas le seul comme vous allez le découvrir petit à petit. Par exemple, un autre gros avantage des objets est qu'ils sont facilement réutilisables et modifiables.

A partir de maintenant, nous allons apprendre à **créer** des objets. Vous allez voir que c'est tout un art et que ça demande de la pratique. Il y a beaucoup de programmeurs qui prétendent faire de la POO et qui le font pourtant très mal (et je ne m'exclue pas forcément du lot 😊). En effet, on peut créer un objet de 100 façons différentes, et c'est à nous de choisir à chaque fois la meilleure, la plus adaptée. Pas évident. Il faudra donc bien réfléchir avant de se lancer dans le code comme des forcenés 🤪

Allez, on prend une **grande** inspiration, et on plonge ensemble dans l'océan de la POO ! 😊

Créer une classe

Commençons par la question qui doit vous brûler les lèvres 😱

Je croyais qu'on allait apprendre à créer des objets, pourquoi tu nous parles de créer une classe maintenant ?
Quel est le rapport ?

Eh bien justement, pour créer un objet, il faut d'abord créer une classe !

Je m'explique : pour construire une maison, vous avez besoin d'un plan d'architecte non ? Eh bien imaginez simplement que la classe c'est le plan, et que l'objet c'est la maison.

"Créer une classe", c'est donc dessiner les plans de l'objet.

Une fois que vous avez les plans, vous pouvez faire autant de maisons que vous voulez en vous basant sur les plans. Pour les objets c'est pareil : une fois que vous avez fait la classe (le plan), vous pourrez créer autant d'objets du même type que vous voulez 😊

Vocabulaire : on dit qu'un objet est une instance d'une classe. C'est un mot très courant que l'on rencontre souvent en POO. Cela signifie qu'un objet est la matérialisation concrète d'une classe (tout comme la maison est la matérialisation concrète du plan de la maison).

Oui je sais c'est très métaphysique la POO, mais vous allez voir on s'y fait 😊

Créer une classe, oui mais laquelle ?

Avant tout, il va falloir choisir la classe sur laquelle nous allons travailler.

Pour reprendre mon exemple sur l'architecture : allons-nous créer un appartement, une villa avec piscine, un spacieux loft ? En clair, quel type d'objet voulons-nous être capable de créer ?

Les choix ne manquent pas. Je sais que, quand on débute, on a du mal à imaginer ce qui peut être considéré comme un objet. La réponse est : presque tout !

Vous allez voir, vous allez petit à petit avoir le feeling qu'il faut avec la POO. Puisque vous débutez, c'est moi qui vais choisir (vous avez pas trop le choix de toute façon 😅).

Pour notre exemple, nous allons créer une classe Personnage qui va permettre de représenter un personnage de jeu de rôle (RPG).

Si vous n'avez pas l'habitude des jeux de rôle, rassurez-vous, moi non plus. Vous n'avez pas besoin de savoir jouer à des RPG pour suivre ce chapitre. J'ai choisi cet exemple car il me paraît didactique, amusant, et qu'il peut déboucher sur la création d'un jeu à la fin 😊

Bon, on la crée cette classe ?

C'est parti 😊

Pour commencer, je vous rappelle qu'une classe est constituée :

- De variables, ici appelées attributs (on parle aussi de variables membres)
- De fonctions, ici appelées méthodes (on parle aussi de fonctions membres)

(n'oubliez pas ce vocabulaire, il est fon-da-men-tal !)

Pour tout vous dire, les classes ressemblent beaucoup aux [structures](#) qu'on avait étudiées en C, sauf qu'elles contiennent en plus des méthodes (les fonctions).

Vous allez donc voir que cela ressemble pas mal aux structures, du moins au premier abord.

Voici le code minimal pour créer une classe :

Code : C++ - [Sélectionner](#)

```
class Personnage
{
}; // N'oubliez pas le point-virgule à la fin !
```

On utilise comme vous le voyez le mot-clé `class`.

Il est suivi du nom de la classe que l'on veut créer. Ici, c'est `Personnage`.

Souvenez-vous de cette règle très importante : il faut que le nom de vos classes commence toujours par une lettre majuscule ! Bien que ce ne soit pas obligatoire (le compilateur ne gueulera pas si vous commencez par une minuscule), cela vous sera très utile par la suite pour différencier les types de variable classiques (`int`, `double`, `bool`, ...) des classes (`Personnage`, ...).

C'est entre les accolades que nous allons écrire la définition de la classe. Tout ou presque se passera à l'intérieur de ces accolades. Et surtout, super important, le truc qu'on oublie au moins une fois dans sa vie : il y a un point-virgule après l'accolade fermante, tout comme avec les structures !

Ajout de méthodes et d'attributs

Bon c'est bien beau, mais notre classe `Personnage` est plutôt... vide.

Que va-t-on mettre dans la classe ? Vous le savez déjà voyons 😊

- Des attributs, c'est le nom que l'on donne aux variables contenues dans des classes
- Des méthodes, c'est le nom que l'on donne aux fonctions contenues dans des classes

Le but du jeu maintenant, c'est justement d'arriver à faire la liste de tout ce qu'on veut mettre dans notre `Personnage`. De quels attributs et de quelles méthodes a-t-il besoin ? Ca, c'est justement l'étape de réflexion, la plus importante. C'est pour ça que je vous ai dit au début de ce chapitre qu'il fallait surtout pas coder comme des barbares dès le début, mais prendre le temps de réfléchir.

Cette étape de réflexion avant le codage est essentielle quand on fait de la POO. Beaucoup de gens, dont moi, ont l'habitude de sortir une feuille de papier et un crayon pour arriver à établir la liste des attributs et méthodes dont ils vont avoir besoin. On en reparlera plus tard, mais sachez déjà qu'un langage spécial appelé UML a été spécialement conçu pour "dessiner" les classes avant de commencer à les coder.

Par quoi commencer : les attributs ou les méthodes ? Il n'y a pas d'ordre en fait, mais je trouve un peu plus logique de commencer par voir les attributs puis les méthodes.

Les attributs

C'est ce qui va caractériser votre classe, ici le personnage. Ce sont des variables, elles peuvent donc évoluer au fil du temps. Mais qu'est-ce qui caractérise un personnage de jeu de rôle ? Allons, un petit effort 😊

- Par exemple, tout personnage a un niveau de vie. Hop, ça fait un premier attribut : `vie` ! On dira que ce sera un `int`, et qu'il sera compris entre 0 et 100 (0 = mort, 100 = toute la vie).
- Dans un jeu de rôle (RPG), il y a le niveau de magie, aussi appelé mana. Là encore, on va dire que c'est un `int` compris entre 0 et 100. Si le personnage a 0 de mana, il ne peut plus lancer de sorts magiques et doit attendre que sa mana se recharge toute seule au fil du temps (ou boire une potion de mana !).
- On pourrait rajouter aussi le nom de l'arme que porte le joueur : `nomArme`. Puisque c'est une chaîne de caractères et qu'on fait du C++, on n'est pas fou, on va utiliser un `string` 😊
- Enfin, il me semble indispensable d'ajouter un attribut `degatsArme`, un `int` toujours, qui indiquerait cette fois le nombre de dégâts que fait notre arme à chaque coup

On peut donc déjà commencer à compléter notre classe avec ces premiers attributs :

Code : C++ - [Sélectionner](#)

```
class Personnage
{
    int m_vie;
    int m_mana;
    string m_nomArme;
    int m_degatsArme;
};
```

Deux trois petites choses à savoir sur ce code :

- Ce n'est pas une obligation, mais une grande partie des programmeurs (dont moi) a l'habitude de faire commencer tous les noms des attributs de classe par `m_` (le "m" signifiant "membre", pour indiquer que c'est une variable membre, c'est-à-dire un attribut). Cela permet de bien différencier les attributs des variables "classiques" (contenues dans des fonctions par exemple).
- Il est impossible d'initialiser les attributs ici. Cela doit être fait via ce qu'on appelle un constructeur, comme on le verra un peu plus loin.
- Comme on utilise un objet `string`, il faut bien penser à rajouter un `#include <string>` dans votre fichier.

Les méthodes

Les méthodes, elles, sont grossièrement les actions que le personnage peut faire ou qu'on peut lui faire faire. Les méthodes lisent et modifient les attributs.

Voici quelques actions qu'on peut faire avec notre personnage :

- `recevoirDegats` : le personnage prend un certain nombre de dégâts, donc perd de la vie.
- `attaquer` : le personnage attaque un autre personnage avec son arme. Il fait autant de dégâts que son arme lui permet d'en faire (c'est-à-dire `degatsArme`).
- `boirePotionDeVie` : le personnage boit une potion de vie et regagne un certain nombre de points de vie.
- `changerArme` : le personnage récupère une nouvelle arme plus puissante. On change le nom de l'arme et les dégâts qui vont avec.
- `estVivant` : renvoie vrai si le personnage est toujours vivant (+ que 0 points de vie), renvoie faux sinon.

Voilà c'est un bon début je trouve 😊

On va rajouter ça dans la classe avant les attributs (on préfère présenter les méthodes avant les attributs en POO, bien que ça ne soit pas obligatoire) :

Code : C++ - [Sélectionner](#)

```

class Personnage
{
    // Méthodes
    void recevoirDegats(int nbDegats)
    {

    }

    void attaquer(Personnage &cible)
    {

    }

    void boirePotionDeVie(int quantitePotion)
    {

    }

    void changerArme(string nomNouvelleArme, int degatsNouvelleArme)
    {

    }

    bool estVivant()
    {

    }

    // Attributs
    int m_vie;
    int m_mana;
    string m_nomArme;
    int m_degatsArme;
};

```

Je n'ai pas écrit le code des méthodes exprès, on le fera après 😊

Ceci dit, vous devriez déjà avoir une petite idée de ce que vous allez mettre dans ces méthodes.

Par exemple, recevoirDegats retranchera le nombre de dégâts indiqués en paramètre par nbDegats à la vie du personnage. Intéressante aussi : la méthode attaquer. Elle prend en paramètre... un autre personnage, plus exactement une référence vers le personnage cible que l'on doit attaquer ! Et que fera cette méthode à votre avis ? Eh oui, elle appellera la méthode recevoirDegats de la cible pour lui infliger des dégâts 😊

Vous commencez à comprendre un peu comment tout cela est lié et terriblement logique ? 😊
On met en général un peu de temps avant de "penser objet" correctement. Si vous vous dites que vous n'auriez pas pu inventer un truc comme ça tout seul, rassurez-vous, tous les débutants passent par là. A force de pratiquer, ça va venir 😊

Pour info, toutes les méthodes que l'on pourrait créer ne sont pas là : par exemple, on n'utilise pas de magie (mana) ici. Le personnage attaque seulement avec une arme (une épée par exemple) et n'utilise donc pas de sorts magiques. Je laisse exprès quelques fonctions manquantes pour vous inciter à compléter la classe avec vos idées 😊

En résumé : comme je vous l'avais dit, un objet est bel et bien un mix de "variables" (les attributs) et de "fonctions" (les méthodes). La plupart du temps, les méthodes lisent et modifient les attributs de l'objet pour le faire évoluer.
Un objet est au final un petit système intelligent et autonome qui est capable de surveiller son bon fonctionnement tout seul.

Droits d'accès et encapsulation

Nous allons maintenant nous intéresser au concept le plus fondamental de la POO : l'encapsulation. Ne vous laissez pas effrayer par

ce mot, vous allez vite comprendre ce que ça signifie.

Tout d'abord un petit rappel. En POO, il y a 2 parties bien distinctes :

- On **crée** des classes pour définir le fonctionnement des objets. C'est ce qu'on apprend à faire ici.
- On **utilise** des objets. C'est ce qu'on a appris à faire dans le chapitre précédent.

Il faut bien distinguer ces 2 parties, car ça devient ici très important.

Je mets un exemple création / utilisation côté à côté pour que vous puissiez bien les différencier :

Création de la classe

Code : C++ - Sélectionner

```
class Personnage
{
    void recevoirDegats(int nbDegats)
    {

    }

    void attaquer(Personnage &cible)
    {

    }

    void boirePotionDeVie(int quantitePotion)
    {

    }

    void changerArme(string nomNouvelleArme, int degatsNouvelleArme)
    {

    }

    bool estVivant()
    {

    }

    int m_vie;
    int m_mana;
    string m_nomArme;
    int m_degatsArme;
};
```

3 4 5 6

Code : C++ - Sélectionner

```
int main()
{
    Personnage goliath;
    Personnage david;

    goliath.atttaquer(david);
    david.boirePotionDeVie(10);
    goliath.atttaquer(david);
    david.atttaquer(goliath);
    goliath.changerArme("Epée", 15);

    return 0;
}
```

3 4 5 6

Tenez, pourquoi on n'essaierait pas ce code ?

Allez, on met tout dans un même fichier (en prenant soin de définir la classe avant le main), et zou !

Code : C++ - Sélectionner

```

#include <iostream>
#include <string>

using namespace std;

class Personnage
{
    // Méthodes
    void recevoirDegats(int nbDegats)
    {
    }

    void attaquer(Personnage &cible)
    {
    }

    void boirePotionDeVie(int quantitePotion)
    {
    }

    void changerArme(string nomNouvelleArme, int degatsNouvelleArme)
    {
    }

    bool estVivant()
    {
    }

    // Attributs
    int m_vie;
    int m_mana;
    string m_nomArme;
    int m_degatsArme;
};

int main()
{
    Personnage david, goliath; // Création de 2 objets de type Personnage : david et goliath

    goliath.atttaquer(david); // goliath attaque david
    david.boirePotionDeVie(20); // david boit une potion de vie qui lui rapporte 20 de vie
    goliath.atttaquer(david); // goliath réattaque david
    david.atttaquer(goliath); // david contre-attaque... c'est assez clair non ? ^^
    goliath.changerArme("Double hache tranchante vénéneuse de la mort", 40);
    goliath.atttaquer(david);

    return 0;
}

```

Compilez et admirez... la belle erreur de compilation ?! 😊

Error : void Personnage::attaquer(Personnage&) is private within this context

Une nouvelle insulte ?

Vous allez voir, le compilateur ne manque pas d'insultes en C++, vous allez sûrement en rencontrer pas mal 😊

On en arrive justement au problème qui nous intéresse : celui des droits d'accès (eh ouais j'ai fait exprès de provoquer cette erreur de compilation, vous aviez quand même pas cru que j'avais pas tout prévu ? ).

Ouvrez grand vos oreilles : chaque attribut et chaque méthode d'une classe peut posséder son propre droit d'accès. Il existe grossièrement 2 droits d'accès différents :

- public : l'attribut ou la méthode peut être appelé depuis l'extérieur de l'objet.
- private : l'attribut ou la méthode ne peut pas être appelé depuis l'extérieur de l'objet. Par défaut, tous les éléments d'un objet sont private.

Il existe d'autres droits d'accès mais ils sont un peu plus complexes. Nous les verrons plus tard.

Concrètement, qu'est-ce que ça signifie ? Qu'est-ce que "l'extérieur" de l'objet ?

Eh bien sur notre exemple, "l'extérieur" c'est le main. En effet, c'est là où on utilise l'objet. On fait appel à des méthodes, mais comme elles sont privées par défaut, on ne peut pas les appeler depuis le main !

Pour modifier les droits d'accès et mettre par exemple public, il faut taper `public` suivi du symbole : (deux points). Tout ce qui se trouvera à la suite sera public.

Voici ce que je vous propose de faire : on va mettre en public toutes les méthodes, et en privé tous les attributs.
Ca donne ça :

Code : C++ - Sélectionner

```
class Personnage
{
    // Tout ce qui suit est public (accessible depuis l'extérieur)
    public:

        void recevoirDegats(int nbDegats)
    {

    }

        void attaquer(Personnage &cible)
    {

    }

        void boirePotionDeVie(int quantitePotion)
    {

    }

        void changerArme(string nomNouvelleArme, int degatsNouvelleArme)
    {

    }

        bool estVivant()
    {

    }

    // Tout ce qui suit est privé (inaccessible depuis l'extérieur)
    private:

        int m_vie;
        int m_mana;
        string m_nomArme;
        int m_degatsArme;
};
```

Tout ce qui suit le **public** : est public. Donc toutes nos méthodes sont publiques.
Ensuite vient le mot-clé **private** : . Tout ce qui suit ce mot-clé est privé. Donc tous nos attributs sont privés.

Voilà, vous pouvez maintenant compiler ce code, et vous verrez qu'il n'y a pas de problème (même si le code ne fait rien pour l'instant 😐). On appelle des méthodes depuis le main, mais comme elles sont publiques, on a le droit de le faire.
... par contre, nos attributs sont privés, ce qui veut dire qu'on n'a pas le droit de les modifier depuis le main. En clair, on ne peut pas faire depuis le main :

Code : C++ - [Sélectionner](#)

```
goliath.m_vie = 90;
```

Essayez, vous verrez que le compilateur vous ressort la même erreur que tout à l'heure : "ton bidule est private... bla bla bla... pas le droit d'appeler un élément private depuis l'extérieur de la classe".

Mais alors... ça veut dire qu'on ne peut pas modifier la vie du personnage depuis le main ? Eh oui !
C'est nul ? Non au contraire, c'est très bien pensé, ça s'appelle l'encapsulation 😊

L'encapsulation

Moi j'ai une solution ! Si on mettait tout en public ? Les méthodes ET les attributs en public, comme ça on peut tout modifier depuis le main et plus aucun problème !
... quoi j'ai dit une connerie ? 😳

Oh, trois fois rien, vous venez juste de vous faire autant d'ennemis qu'il n'y a de programmeurs qui font de la POO dans le monde 😊

Il y a une règle d'or en POO, et tout découle de là. S'il vous plaît, imprimez ceci en gros sur une feuille, et placardez cette feuille sur un mur de votre chambre :

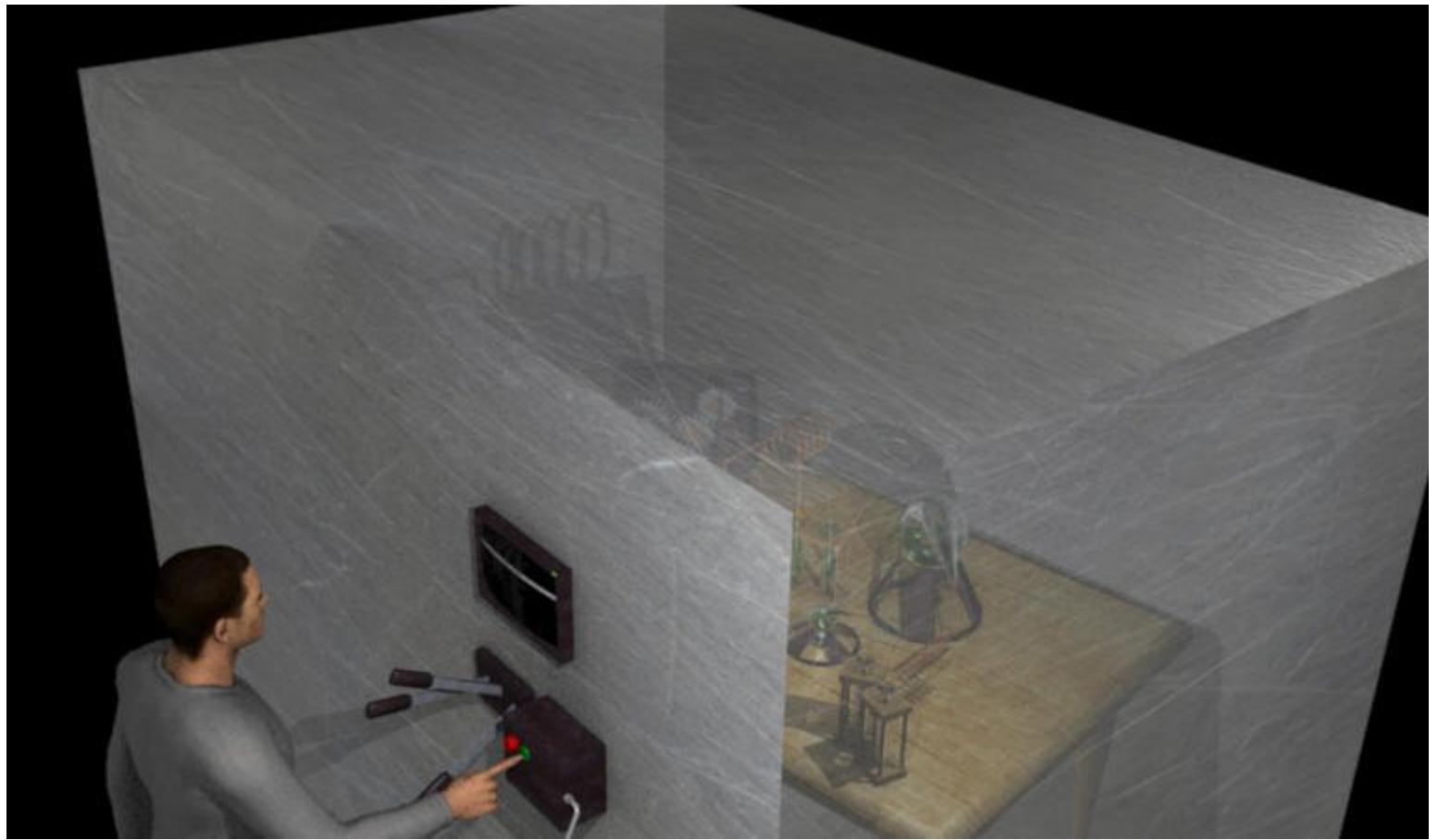
Encapsulation : tous les attributs d'une classe doivent toujours être privés

Ca a l'air bête, stupide, irréfléchi, et pourtant tout ce qui fait que la POO est un principe puissant vient de là.
En clair, si j'en vois un à partir de maintenant qui me met ne serait-ce qu'un seul attribut en public, je le brûle, je le torture, je l'écorche vif sur la place publique, compris ? 😠

Et vous, si vous voyez quelqu'un d'autre faire ça un jour, écorchez-le vif en pensant à moi, vous serez sympa 😊

Voilà qui explique pourquoi j'ai fait exprès dès le début de mettre les attributs en privé. Comme ça, on ne peut pas les modifier depuis l'extérieur de la classe, et ça respecte le principe d'encapsulation.

Vous vous souvenez de ce schéma du chapitre précédent ?



Les fioles chimiques, ce sont les attributs.

Les boutons sur la façade avant, ce sont les méthodes.

Et là, pif paf pouf, vous devriez avoir tout compris d'un coup. En effet, le but du modèle objet c'est justement de masquer les informations complexes à l'utilisateur (les attributs) pour éviter qu'il ne fasse des bêtises avec.

Imaginez par exemple que l'utilisateur puisse modifier la vie... qu'est-ce qui l'empêcherait de mettre 150 de vie alors que la limite maximale est 100 ? C'est pour ça qu'il faut toujours passer par des méthodes (des fonctions) qui vont d'abord vérifier qu'on fait les choses correctement avant de modifier les attributs.

Cela permet de faire en sorte que le contenu de l'objet reste une "boîte noire". On ne sait pas comment ça fonctionne à l'intérieur quand on l'utilise, et c'est très bien. C'est une sécurité, ça permet d'éviter de faire péter tout le bazar de fioles chimiques à l'intérieur 🍊

Séparer prototypes et définitions

Bon, on avance mais on n'a pas fini 😊

Voici ce que je voudrais qu'on fasse :

- Séparer les méthodes en prototypes et définitions dans 2 fichiers différents pour avoir un code plus modulaire
- Implémenter les méthodes de notre classe Personnage (c'est-à-dire écrire le code à l'intérieur parce que pour le moment y'a rien 🍊)

Pour le moment, on a mis notre classe dans le fichier main.cpp, juste au-dessus du main. Et les méthodes sont directement écrites dans la définition de la classe.

Ca fonctionne, mais c'est un peu bourrin. Tout comme on avait appris en C à faire du code modulaire, on va voir comment on procède en POO pour séparer tout ça proprement dans des fichiers différents.

Tout d'abord, il faut clairement séparer le main (qui se trouve dans main.cpp) des classes.

Pour chaque classe, on va créer :

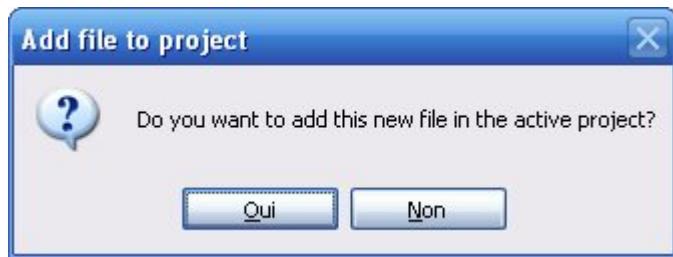
- Un header (*.h) qui contiendra les attributs et les prototypes de la classe
- Un fichier source (*.cpp) qui contiendra la définition des méthodes et leurs implémentations

Je vous propose d'ajouter à votre projet 2 fichiers nommés très exactement :

- Personnage.h
- Personnage.cpp

(vous noterez que je mets aussi une majuscule à la première lettre du nom de fichier, histoire d'être cohérent jusqu'au bout)

Vous devriez être capables de faire ça tous seuls avec votre IDE favori. Sous Code::Blocks, je fais File / New File, je rentre par exemple le nom "Personnage.h" avec l'extension, et je réponds "Oui" quand Code::Blocks me demande si je veux ajouter le nouveau fichier au projet en cours :



Personnage.h

Le fichier .h va donc contenir la déclaration de la classe avec les attributs et les prototypes des méthodes. Dans notre cas, pour la classe Personnage, ça va donner ça :

Code : C++ - Sélectionner

```
#ifndef DEF_PERSONNAGE
#define DEF_PERSONNAGE

class Personnage
{
    public:

        void recevoirDegats(int nbDegats);
        void attaquer(Personnage &cible);
        void boirePotionDeVie(int quantitePotion);
        void changerArme(std::string nomNouvelleArme, int degatsNouvelleArme);
        bool estVivant();

    private:

        int m_vie;
        int m_mana;
        std::string m_nomArme; // Pas de using namespace std, donc il faut mettre std:: devant
        int m_degatsArme;
};

#endif
```

Comme vous pouvez le constater, seuls les prototypes des méthodes sont présents dans le .h. C'est déjà beaucoup plus clair 😊

Dans les .h, il est recommandé de ne jamais mettre la directive `using namespace std;` car cela pourrait avoir des effets néfastes lorsque vous utiliserez la classe par la suite.

Par conséquent, il faut rajouter le préfixe "std::" devant chaque string du .h. Sinon, le compilateur vous sortira une erreur du type "string does not name a type".

Personnage.cpp

C'est là qu'on va écrire le code de nos méthodes (on dit qu'on implémente les méthodes).

Première chose à ne pas oublier, sinon ça va pas bien se passer, c'est d'inclure <string> et "Personnage.h".

On peut aussi rajouter ici un `using namespace std;`. On a le droit de le faire car on est dans le .cpp (par contre comme je vous l'ai expliqué plus tôt, il faut éviter de le mettre dans le .h).

Code : C++ - Sélectionner

```
#include <string>
#include "Personnage.h"

using namespace std;
```

Veillez à inclure <string> AVANT Personnage.h, sinon la déclaration de la classe contenue dans Personnage.h n'aura pas connu au préalable le type string... et donc la compilation plantera.

Maintenant, voilà comment ça se passe : pour chaque méthode, vous devez faire précéder le nom de la méthode par le nom de la classe suivi de deux fois deux points "::". Pour recevoirDegats ça donne ça :

Code : C++ - Sélectionner

```
void Personnage::recevoirDegats(int nbDegats)
{
}
```

Cela permet au compilateur de savoir que cette méthode se rapporte à la classe Personnage. En effet, comme la méthode est ici écrite en dehors de la définition de la classe, le compilateur n'aurait pas su à quelle classe appartenait cette méthode.

Personnage::recevoirDegats

Maintenant, c'est parti, implémentons la méthode recevoirDegats. Je vous avais expliqué un peu plus haut ce qu'il fallait faire. Vous allez voir, c'est très simple :

Code : C++ - Sélectionner

```
void Personnage::recevoirDegats(int nbDegats)
{
    m_vie -= nbDegats; // On enlève le nombre de dégâts reçus à la vie du personnage

    if (m_vie < 0) // Pour éviter d'avoir une vie négative
    {
        m_vie = 0; // On met la vie à 0 (ça veut dire mort)
    }
}
```

La méthode modifie donc la valeur de la vie. La méthode a le droit de modifier l'attribut, car elle fait partie de la classe. Ne soyez donc pas surpris, c'est justement l'endroit où on a le droit de toucher aux attributs 😊

La vie est diminuée du nombre de dégâts reçus. En théorie, on aurait pu se contenter de la première instruction, mais on fait une vérification supplémentaire. Si la vie est descendue en-dessous de 0 (parce qu'on a reçu 20 de dégâts alors qu'on n'avait que 10 de vie), on ramène la vie à 0 afin d'éviter une vie négative (ça fait pas très pro une vie négative 🤪). De toute façon, à 0 de vie, le personnage est considéré comme mort 😊

Et voilà pour la première méthode ! Allez on enchaîne hop hop hop !

Personnage::attaquer

Code : C++ - [Sélectionner](#)

```
void Personnage::attaquer(Personnage &cible)
{
    cible.recevoirDegats(m_degatsArme); // On inflige à la cible les dégâts que causent nos armes
}
```

Cette méthode est peut-être très courante, elle n'en est pas moins très intéressante !

On reçoit en paramètre une référence vers un objet de type Personnage. On aurait pu recevoir un pointeur aussi, mais comme les références sont plus faciles à manipuler (cf les chapitres précédents) on ne va pas s'en priver.

La référence concerne le personnage cible que l'on doit attaquer. Pour infliger des dégâts à la cible, on appelle sa méthode `recevoirDegats` en faisant : `cible.recevoirDegats`

Quelle quantité de dégâts envoyer à la cible ? Vous avez la réponse sous vos yeux : le nombre de points de dégâts indiqués par l'attribut `m_degatsArme` ! On envoie donc la valeur des `m_degatsArme` de notre personnage à la cible.

Personnage::boirePotionDeVie

Code : C++ - [Sélectionner](#)

```
void Personnage::boirePotionDeVie(int quantitePotion)
{
    m_vie += quantitePotion;

    if (m_vie > 100) // Interdiction de dépasser 100 de vie
    {
        m_vie = 100;
    }
}
```

Le personnage reprend autant de vie que ce que la potion qu'il boit lui permet d'en récupérer. On vérifie au passage qu'il ne dépasse pas les 100 de vie, car comme on l'a dit plus tôt, il est interdit d'avoir plus de 100 de vie.

Personnage::changerArme

Code : C++ - [Sélectionner](#)

```
void Personnage::changerArme(string nomNouvelleArme, int degatsNouvelleArme)
{
    m_nomArme = nomNouvelleArme;
    m_degatsArme = degatsNouvelleArme;
}
```

Pour changer d'arme, on stocke dans nos attributs le nom de la nouvelle arme ainsi que ses nouveaux dégâts. Les instructions sont très simples : on fait juste passer ce qu'on a reçu en paramètres dans nos attributs.

Grâce à l'objet `string` d'ailleurs, il suffit de faire un simple `"=` pour affecter la chaîne, et on n'a plus à se préoccuper de la taille du tableau car l'objet `string` se débrouille tout seul pour ça (à chaque fois que j'y pense je trouve ça génial 😊).

Personnage::estVivant

Code : C++ - [Sélectionner](#)

```
bool Personnage::estVivant()
{
    if (m_vie > 0) // Plus de 0 de vie ?
    {
        return true; // VRAI, il est vivant !
    }
    else
    {
        return false; // FAUX, il n'est plus vivant !
    }
}
```

Cette méthode permet de vérifier si le personnage est toujours vivant. Elle renvoie vrai (true) s'il a plus de 0 de vie, et faux (false) sinon.

[Code complet de Personnage.cpp](#)

En résumé, le code complet de notre Personnage.cpp est le suivant :

Code : C++ - [Sélectionner](#)

```

#include <string>
#include "Personnage.h"

using namespace std;

void Personnage::recevoirDegats(int nbDegats)
{
    m_vie -= nbDegats; // On enlève le nombre de dégâts reçus à la vie du personnage

    if (m_vie < 0) // Pour éviter d'avoir une vie négative
    {
        m_vie = 0; // On met la vie à 0 (ça veut dire mort)
    }
}

void Personnage::attaquer(Personnage &cible)
{
    cible.recevoirDegats(m_degatsArme); // On inflige à la cible les dégâts que causent i
}

void Personnage::boirePotionDeVie(int quantitePotion)
{
    m_vie += quantitePotion;

    if (m_vie > 100) // Interdiction de dépasser 100 de vie
    {
        m_vie = 100;
    }
}

void Personnage::changerArme(string nomNouvelleArme, int degatsNouvelleArme)
{
    m_nomArme = nomNouvelleArme;
    m_degatsArme = degatsNouvelleArme;
}

bool Personnage::estVivant()
{
    if (m_vie > 0) // Plus de 0 de vie ?
    {
        return true; // VRAI, il est vivant !
    }
    else
    {
        return false; // FAUX, il n'est plus vivant !
    }
}

```

3

4

5

6

main.cpp

Retour au main. Première chose à ne pas oublier : inclure Personnage.h pour pouvoir créer des objets de type Personnage.

Code : C++ - [Sélectionner](#)

```
#include "Personnage.h" // Ne pas oublier
```

Après, le main reste le même que tout à l'heure, on n'a pas besoin de le changer. Au final, le code du main est donc très court, et le fichier main.cpp ne fait qu'utiliser les objets :

Code : C++ - [Sélectionner](#)

```

#include <iostream>
#include <string>
#include "Personnage.h" // Ne pas oublier

using namespace std;

int main()
{
    Personnage david, goliath; // Création de 2 objets de type Personnage : david et goliath

    goliath.atttaquer(david); // goliath attaque david
    david.boirePotionDeVie(20); // david boit une potion de vie qui lui rapporte 20 de vie
    goliath.atttaquer(david); // goliath réattaque david
    david.atttaquer(goliath); // david contre-attaque... c'est assez clair non ? ^^
    goliath.changerArme("Double hache tranchante vénéneuse de la mort", 40);
    goliath.atttaquer(david);

    return 0;

```

3

6

N'exécutez pas le programme pour le moment. En effet, nous n'avons toujours pas vu comment faire pour initialiser les attributs, ce qui fait que notre programme n'est pas encore utilisable.

Nous verrons comment le rendre pleinement fonctionnel dans le chapitre suivant, et vous pourrez alors (enfin) l'exécuter 😊

Il faudra donc pour le moment vous contenter de votre imagination. Essayez d'imaginer que David et Goliath sont bien en train de combattre ! (et je veux pas faire mon gros spoiler, mais normalement c'est David qui gagne à la fin 😊).

Là, on peut dire qu'on est rentré en plein dans la POO 😊

Pourtant, ce n'est encore qu'un début ! De nombreuses nouvelles choses complètement dingues vous attendent dans les chapitres qui suivent (et elles vont vous rendre dingues ça c'est sûr 😊)

Un conseil si je puis me permettre : assurez-vous d'avoir bien compris qu'il y avait deux faces dans la POO, la création de la classe, et l'utilisation des objets. Il faut être à l'aise avec ce concept.

Mais tout n'est pas si simple. Comme vous le verrez, ce que font les objets la plupart du temps c'est... utiliser d'autres objets ! Et c'est en combinant plusieurs objets entre eux que l'on découvrira le vrai pouvoir de la POO 😊

Les classes (Partie 2/2)

Allez, hop hop hop, on enchaîne ! Pas question de s'endormir, on est en plein dans la POO là 😊

Dans le chapitre précédent, nous avons appris à créer une classe basique, à rendre le code modulaire en POO, et surtout nous avons découvert le principe d'encapsulation (super important l'encapsulation, c'est la base de tout je le rappelle).

Dans cette seconde partie du chapitre, nous allons découvrir comment initialiser nos attributs à l'aide d'un constructeur, un élément indispensable à toute classe qui se respecte. Puisqu'on parlera de constructeur, on parlera aussi de destructeur, ça va de paire vous verrez.

Nous compléterons notre classe Personnage et nous l'associerons avec une nouvelle classe Arme que nous allons créer. Nous découvrirons alors tout le pouvoir qu'il y a de combiner des classes entre elles, et vous devriez normalement commencer à imaginer pas mal de possibilités à partir de là 😊

Constructeur et destructeur

Reprenons. Nous avons maintenant 3 fichiers :

- main.cpp : il contient le main, dans lequel on a créé 2 objets de type Personnage : david et goliath.
- Personnage.h : c'est le header de la classe Personnage. On y liste les prototypes des méthodes et les attributs. On y définit la portée (public / private) de chacun des éléments. Pour respecter le principe d'encapsulation, tous nos attributs sont privés, c'est-à-dire non accessibles de l'extérieur.
- Personnage.cpp : c'est le fichier dans lequel on implémente nos méthodes, c'est-à-dire qu'on écrit le code source des méthodes.

Pour l'instant, nous avons défini et implémenté pas mal de méthodes. Je voudrais vous parler ici de 2 méthodes particulières que l'on retrouve dans la plupart des classes : le constructeur et le destructeur.

- Le constructeur : c'est une méthode qui est appelée automatiquement à chaque fois que l'on crée un objet basé sur cette classe.
- Le destructeur : c'est une méthode qui est automatiquement appelée lorsqu'un objet est détruit, par exemple à la fin de la fonction dans laquelle il a été déclaré ou lors d'un delete si l'objet a été alloué dynamiquement avec new.

Voyons voir plus en détail comment fonctionnent ces méthodes un peu particulières...

Le constructeur

Comme son nom l'indique, c'est une méthode qui sert à construire l'objet. Dès qu'on crée un objet, le constructeur est automatiquement appelé s'il existe.

Par exemple, lorsqu'on fait dans notre main :

Code : C++ - [Sélectionner](#)

```
Personnage david, goliath;
```

S'il existe, le constructeur de l'objet david est appelé, et de même pour le constructeur de l'objet goliath. Mais... comme nous n'avons pas encore défini de constructeur dans la classe Personnage, rien de particulier ne s'est passé. Le constructeur n'est pas obligatoire, mais on a presque toujours besoin d'en créer un, vous allez vite comprendre pourquoi.

Le rôle du constructeur

Si le constructeur est appelé lors de la création de l'objet, ce n'est pas pour faire joli. En fait, le rôle principal du constructeur est d'initialiser les attributs.

En effet, souvenez-vous : nos attributs sont déclarés dans Personnage.h, mais pas initialisés !

Revoici Personnage.h :

Code : C++ - [Sélectionner](#)

```

#include <string>

class Personnage
{
public:

    void recevoirDegats(int nbDegats);
    void attaquer(Personnage &cible);
    void boirePotionDeVie(int quantitePotion);
    void changerArme(std::string nomNouvelleArme, int degatsNouvelleArme);
    bool estVivant();

private:

    int m_vie;
    int m_mana;
    std::string m_nomArme;
    int m_degatsArme;
};


```

Nos attributs m_vie, m_mana, et m_degatsArmes ne sont pas initialisés ! Pourquoi ? Parce qu'on n'a pas le droit d'initialiser les attributs ici. C'est justement dans le constructeur qu'il faut le faire.

En fait, le constructeur est indispensable pour initialiser les attributs qui ne sont pas des objets (type classique : int, double, char...). En effet, ceux-ci ont une valeur inconnue en mémoire (ça peut être 0 comme -3451).

En revanche, les attributs qui sont des objets, comme c'est le cas de m_nomArme ici qui est un string, sont automatiquement initialisés par le langage C++ avec une valeur par défaut.

Créer un constructeur

Le constructeur est une méthode, mais une méthode un peu particulière.

En effet, pour créer un constructeur, il y a 2 règles à respecter :

- Il faut que la méthode ait le même nom que la classe. Dans notre cas, la méthode devra s'appeler "Personnage".
- La méthode ne doit RIEN renvoyer, pas même void ! C'est une méthode sans aucun type de retour.

Si on déclare son prototype dans Personnage.h, ça donne ça :

Code : C++ - [Sélectionner](#)

```

#include <string>

class Personnage
{
public:

    Personnage(); // Constructeur
    void recevoirDegats(int nbDegats);
    void attaquer(Personnage &cible);
    void boirePotionDeVie(int quantitePotion);
    void changerArme(std::string nomNouvelleArme, int degatsNouvelleArme);
    bool estVivant();

private:

    int m_vie;
    int m_mana;
    std::string m_nomArme;
    int m_degatsArme;
};


```

Le constructeur se voit du premier coup d'oeil : déjà parce qu'il n'a aucun type de retour (pas de void ni rien), et ensuite parce qu'il a le même nom que la classe 😊

Et si on en profitait pour implémenter ce constructeur dans Personnage.cpp maintenant ? 😊
Voici à quoi pourrait ressembler son implémentation :

Code : C++ - Sélectionner

```

Personnage::Personnage()
{
    m_vie = 100;
    m_mana = 100;
    m_nomArme = "Epée rouillée";
    m_degatsArme = 10;
}

```

Vous noterez une fois de plus qu'il n'y a pas de type de retour, pas même void (très important, c'est une erreur que l'on fait souvent 😞).

J'ai choisi de mettre la vie et la mana à 100, le maximum, ce qui est logique. J'ai mis par défaut une arme appelée "Epée rouillée" qui fait 10 de dégâts à chaque coup.

Et voilà ! Notre classe Personnage a un constructeur qui initialise les attributs, elle est désormais pleinement utilisable 😊
Maintenant, à chaque fois que l'on crée un objet de type Personnage, celui-ci est initialisé à 100 points de vie et de mana, avec l'arme "Epée rouillée". Nos deux compères david et goliath commencent donc à égalité lorsqu'ils sont créés dans le main :

Code : C++ - Sélectionner

```

Personnage david, goliath; // Les constructeurs de david et goliath sont appelés.

```

Autre façon d'initialiser avec un constructeur : la liste d'initialisation

Le C++ permet d'initialiser les attributs de la classe d'une autre manière (un peu déroutante) appelée liste d'initialisation.
Reprendons le constructeur qu'on vient de créer :

Code : C++ - Sélectionner

```

Personnage::Personnage()
{
    m_vie = 100;
    m_mana = 100;
    m_nomArme = "Epée rouillée";
    m_degatsArme = 10;
}

```

Le code que vous allez voir ci-dessous produit le même effet :

Code : C++ - Sélectionner

```

Personnage::Personnage() : m_vie(100), m_mana(100), m_nomArme("Epée rouillée"), m_degatsArme(5)
{
    // Rien à mettre dans le corps du constructeur, tout a déjà été fait !
}

```

La nouveauté, c'est qu'on rajoute un symbole deux-points (:) suivi de la liste des attributs que l'on veut initialiser avec la valeur entre parenthèses. Avec ce code, on initialise la vie à 100, la mana à 100, l'attribut m_nomArme à "Epée rouillée", etc.

Cette technique est un peu surprenante, surtout que du coup on n'a plus rien à mettre dans le corps du constructeur entre les accolades, vu que tout a déjà été fait avant ! Elle a toutefois l'avantage d'être "plus propre" et se révèlera pratique dans la suite du chapitre.

On va donc utiliser autant que possible les listes d'initialisation avec les constructeurs, c'est une bonne habitude à prendre.

Le prototype du constructeur (dans le .h) ne change pas. Toute la partie après les deux-points n'apparaît pas dans le prototype.

Surcharger le constructeur

Vous savez qu'en C++ on a le droit de surcharger les fonctions, donc de surcharger les méthodes. Et comme le constructeur est une méthode, on a le droit de le surcharger lui aussi.

Pourquoi je vous en parle ? Ce n'est pas par hasard : en fait, le constructeur est une méthode que l'on a tendance à beaucoup surcharger. Cela permet de créer un objet de plusieurs façons différentes.

Pour l'instant, on a créé un constructeur sans paramètres :

Code : C++ - Sélectionner

```
Personnage();
```

On appelle ça : le constructeur par défaut (il fallait bien lui donner un nom le pauvre 🤷).

Supposons que l'on souhaite créer un personnage qui ait dès le départ une meilleure arme... comment diable faire ? C'est là que la surcharge devient utile. On va créer un 2ème constructeur qui prendra en paramètre le nom de l'arme et ses dégâts.

Dans Personnage.h, on va donc rajouter ce prototype :

Code : C++ - Sélectionner

```
Personnage(std::string nomArme, int degatsArme);
```

Le préfixe std:: est obligatoire ici comme je vous l'ai dit plus tôt car on n'utilise pas la directive `using namespace std;` dans le .h (cf chapitre précédent).

L'implémentation dans Personnage.cpp sera la suivante :

Code : C++ - Sélectionner

```
Personnage::Personnage(string nomArme, int degatsArme) : m_vie(100), m_mana(100), m_nomArme(nomArme)
{
    ...
}
```

3

5
6
4

Vous noterez ici tout l'intérêt de mettre le préfixe **m_** au début des attributs : comme ça on peut faire la différence dans notre code entre **m_nomArme**, qui est un attribut, et **nomArme**, qui est le paramètre envoyé au constructeur.

Ce qu'on fait ici, c'est juste placer dans l'attribut de l'objet le nom de l'arme envoyé en paramètre. On recopie juste la valeur. C'est tout bête, mais il faut le faire, sinon l'objet ne se "souviendra pas" du nom de l'arme qu'il possède.

La vie et la mana, eux, sont toujours fixés à 100 (il faut bien les initialiser), mais l'arme, elle, peut maintenant être indiquée par l'utilisateur lorsqu'il crée l'objet.

Quel utilisateur ?

Souvenez-vous, l'utilisateur c'est celui qui crée et utilise les objets. Le concepteur c'est celui qui crée les classes. Dans notre cas, la création des objets est faite dans le main. Pour le moment, la création de nos objets ressemble à ça :

Code : C++ - Sélectionner

```
Personnage david, goliath;
```

Comme on n'a spécifié aucun paramètre, c'est le constructeur par défaut (celui sans paramètres) qui sera appelé.

Maintenant supposons que l'on veuille donner dès le départ une meilleure arme à Goliath (c'est lui le plus fort après tout). On va indiquer entre parenthèses le nom et la puissance de cette arme :

Code : C++ - Sélectionner

```
Personnage david, goliath("Epée aiguisée", 20);
```

Goliath est équipé de l'épée aiguisée dès sa création. David est équipé de l'arme par défaut, l'épée rouillée.

Comme on n'a spécifié aucun paramètre lors de la création de **david**, c'est le constructeur par défaut qui sera appelé pour lui. Pour **goliath**, comme on a spécifié des paramètres, c'est le constructeur correspondant à la signature (**string, int**) qui sera appelé.

Si vous avez oublié ce qu'est une signature de fonction (ou de méthode, c'est pareil), je vous invite très fortement à relire [ce passage du cours](#), que vous avez normalement dû lire quelques chapitres plus tôt

Exercice : on aurait aussi pu permettre à l'utilisateur de modifier la vie et la mana de départ, mais je ne l'ai pas fait ici. Ce n'est pas compliqué, vous pouvez le faire pour vous entraîner. Ca vous fera un troisième constructeur surchargé

Le destructeur

Le destructeur est une méthode appelée lorsque l'objet est supprimé de la mémoire. Son principal rôle est de désallouer la mémoire (via des **delete**) qui a été allouée dynamiquement.

Dans le cas de notre classe Personnage, on n'a fait aucune allocation dynamique (il n'y a aucun **new**). Le destructeur est donc inutile. Cependant, vous en aurez certainement besoin un jour où l'autre, car on est souvent amené à faire des allocations dynamiques. Tenez, l'objet **string** par exemple, vous croyez qu'il fonctionne comment ? Il a un destructeur qui lui permet, juste avant la destruction de l'objet, de supprimer le tableau de char qu'il a alloué dynamiquement en mémoire. Il fait donc un **delete** sur le tableau de char, ce qui permet de garder une mémoire propre et d'éviter les fameuses "fuites de mémoire"

Créer un destructeur

Bien que ce soit inutile dans notre cas (je n'ai pas mis d'allocations dynamiques pour ne pas trop compliquer de suite ) , je vais vous montrer comment on crée un destructeur. Voici les règles à suivre :

- Un destructeur est une méthode qui commence par un tilde ~ suivi du nom de la classe
- Un destructeur ne renvoie aucune valeur, pas même void (comme le constructeur)
- Et, nouveauté : le destructeur ne peut prendre aucun paramètre. Il y a donc toujours un seul destructeur, il ne peut pas être surchargé.

Dans Personnage.h, le prototype du destructeur sera donc :

Code : C++ - [Sélectionner](#)

```
~Personnage();
```

Dans Personnage.cpp, l'implémentation sera :

Code : C++ - [Sélectionner](#)

```
Personnage::~Personnage()
{
    /* Rien à mettre ici car on ne fait pas d'allocation dynamique
       dans la classe Personnage. Le destructeur est donc inutile mais
       je le mets pour montrer à quoi ça ressemble ^^
       En temps normal, un destructeur fait souvent des delete et quelques
       autres vérifications si nécessaire avant la destruction de l'objet */
}
```

Bon vous l'aurez compris, mon destructeur ne fait rien. C'était même pas la peine de le créer (il n'est pas obligatoire après tout). Cela vous montre néanmoins la procédure à suivre. Soyez rassurés, nous ferons des allocations dynamiques plus tôt que vous ne le pensez (je sais je suis diabolique ) , et nous aurons alors grand besoin du destructeur pour désallouer la mémoire !

Associer des classes entre elles

La programmation orientée objet devient vraiment intéressante et puissante lorsqu'on se met à combiner plusieurs objets entre eux. Pour l'instant, nous n'avons créé qu'une seule classe : Personnage.

Or en pratique, un programme objet est un programme constitué d'une multitude d'objets différents !

Il n'y a pas de secret, c'est en pratiquant que l'on apprend petit à petit à penser objet.

Ce que nous allons voir par la suite ne sera pas nouveau : vous allez réutiliser tout ce que vous savez déjà sur la création de classes, de manière à améliorer notre petit RPG et à vous entraîner encore plus à manipuler des objets 

La classe Arme

Ce que je vous propose dans un premier temps, c'est de créer une nouvelle classe Arme. Plutôt que de mettre les informations de l'arme (m_nomArme, m_degatsArme) directement dans le Personnage, nous allons l'équiper d'un objet de type Arme. Le découpage de notre programme sera alors un peu plus dans la logique d'un programme orienté objet.

Souvenez-vous ce que je vous ai dit au début : il y a 100 façons différentes de concevoir un même programme en POO. Tout est dans l'organisation des classes entre elles, comment elles communiquent, etc.

Ce que nous avons fait jusqu'ici était pas mal, mais je veux vous montrer ici qu'on peut faire autrement, un peu plus dans l'esprit objet, donc... mieux 

Qui dit nouvelle classe dit 2 nouveaux fichiers :

- Arme.h : contient la définition de la classe
- Arme.cpp : contient l'implémentation des méthodes de la classe

On n'est pas obligé de procéder ainsi. On pourrait tout mettre dans un seul fichier. On pourrait même mettre plusieurs classes par fichier, rien ne l'interdit en C++. Cependant, pour des raisons d'organisation, je vous recommande de faire comme moi.

Arme.h

Voici ce que je propose de mettre dans Arme.h :

Code : C++ - [Sélectionner](#)

```
#ifndef DEF_ARME
#define DEF_ARME

#include <string>

class Arme
{
public:

    Arme();
    Arme(std::string nom, int degats);
    void changer(std::string nom, int degats);
    void afficher();

private:

    std::string m_nom;
    int m_degats;
};

#endif
```

Mis à part les includes qu'il ne faut pas oublier, le reste de la classe est très simple.

On met le nom de l'arme et ses dégâts dans des attributs, et comme ce sont des attributs, on vérifie qu'ils soient bien privés (encapsulation). Vous remarquerez qu'au lieu de m_nomArme et m_degatsArme, j'ai choisi de nommer mes attributs m_nom et m_degats tout simplement. C'est plus logique en effet : vu qu'on est déjà dans l'Arme, ce n'est pas la peine de préciser dans les attributs qu'il s'agit de l'arme, on le sait déjà, on est dedans 😊

Ensuite, on ajoute un ou deux constructeurs, une méthode pour changer d'arme à tout moment, et une autre allez, soyons fous 🤪, pour afficher le contenu de l'arme.

Reste à implémenter toutes ces méthodes dans Arme.cpp. Pfeuh, fastoche ! 😊

Arme.cpp

Entraînez-vous à écrire Arme.cpp, c'est tout bête, les méthodes font maxi 2 lignes, bref c'est à la portée de tout le monde 😊

Voici mon Arme.cpp pour comparer :

Code : C++ - [Sélectionner](#)

```

#include <iostream>
#include "Arme.h"

using namespace std;

Arme::Arme() : m_nom("Epée rouillée"), m_degats(10)
{
}

Arme::Arme(string nom, int degats) : m_nom(nom), m_degats(degats)
{
}

void Arme::changer(string nom, int degats)
{
    m_nom = nom;
    m_degats = degats;
}

void Arme::afficher()
{
    cout << "Arme : " << m_nom << " (Dégâts : " << m_degats << ")" << endl;
}

```

Bon là je n'ai rien à ajouter vraiment, c'est beaucoup trop simple 😊

N'oubliez quand même pas d'inclure "Arme.h" si vous voulez que ça marche 😊

Et ensuite ?

Bon, notre classe Arme est créée, c'est bon pour ça. Mais maintenant, il va falloir adapter la classe Personnage pour qu'elle utilise non pas m_nomArme et m_degatsArme, mais un objet de type Arme.

Et là... c'est là que ça se complique 😬

Adapter la classe Personnage pour utiliser une Arme

La classe Personnage va subir quelques modifications pour utiliser la classe Arme. Restez attentifs, car utiliser un objet DANS un objet, c'est un peu particulier.

Personnage.h

Zou, direction le .h. On commence par virer nos 2 attributs m_nomArme et m_degatsArme qui ne servent plus à rien.

Les méthodes n'ont pas besoin d'être changées. En fait, il ne vaut mieux pas les changer. Pourquoi ? Parce que les méthodes sont déjà potentiellement utilisées par quelqu'un (par exemple dans notre main). Si on les renomme ou si on les supprime, notre programme ne fonctionnera plus.

Ce n'est peut-être pas grave pour un si petit programme, mais dans le cas d'un gros programme si on supprime une méthode, c'est la cata assurée dans le reste du programme. Et je vous parle même pas de ceux qui écrivent des librairies C++ : si d'une version à l'autre des méthodes disparaissent, tous les programmes qui utilisent la librairie ne fonctionneront plus ! 🚨

Je vais peut-être vous surprendre en vous disant ça, mais c'est là tout l'intérêt de la programmation orientée objet, et plus particulièrement de l'encapsulation. On peut changer nos attributs comme on veut, vu qu'ils ne sont pas accessibles de l'extérieur, on ne prend pas le risque que quelqu'un les utilise déjà dans le programme.

En revanche, pour les méthodes, faites plus attention. Vous pouvez ajouter de nouvelles méthodes, modifier l'implémentation des méthodes existantes, mais PAS en supprimer ou en renommer, sinon l'utilisateur risque d'avoir des problèmes.

Cette petite réflexion sur l'encapsulation étant faite (vous en comprendrez tout le sens avec la pratique 😊), il va falloir ajouter un objet de type Arme à notre Personnage.

Il faut penser à ajouter un include de "Arme.h" si on veut pouvoir utiliser un objet de type Arme.

Voici mon nouveau Personnage.h :

Code : C++ - Sélectionner

```
#ifndef DEF_PERSONNAGE
#define DEF_PERSONNAGE

#include "Arme.h" // Ne PAS oublier d'inclure Arme.h pour en avoir la définition

class Personnage
{
    public:

        Personnage();
        Personnage(std::string nomArme, int degatsArme);
        ~Personnage();
        void recevoirDegats(int nbDegats);
        void attaquer(Personnage &cible);
        void boirePotionDeVie(int quantitePotion);
        void changerArme(std::string nomNouvelleArme, int degatsNouvelleArme);
        bool estVivant();

    private:

        int m_vie;
        int m_mana;
        Arme m_arame; // Notre arme est "contenue" dans le Personnage
};

#endif
```

Personnage.cpp

Nous n'avons besoin de changer que les méthodes qui utilisent l'arme pour les adapter.

On commence par les constructeurs :

Code : C++ - Sélectionner

```
Personnage::Personnage() : m_vie(100), m_mana(100)
{
}

Personnage::Personnage(string nomArme, int degatsArme) : m_vie(100), m_mana(100), m_arame(
```

3

5

6

Notre objet m_arame est ici initialisé avec les valeurs reçues en paramètre par Personnage (nomArme, degatsArme). C'est là que la liste d'initialisation devient utile. En effet, on n'aurait pas pu initialiser m_arame sans une liste d'initialisation !

Peut-être ne voyez-vous pas bien pourquoi. Conseil perso : ne vous prenez pas la tête à essayer de comprendre le pourquoi du comment ici, et contentez-vous de toujours utiliser les listes d'initialisation avec vos constructeurs, ça vous évitera bien des problèmes.

Revenons au code.

Dans le premier constructeur, c'est le constructeur par défaut de la classe Arme qui est appelé, tandis que dans le second c'est celui ayant la signature (string, int) qui est appelé.

La méthode recevoirDegats n'a pas besoin de changer.

En revanche, la méthode attaquer est délicate. En effet, on ne peut pas faire :

Code : C++ - [Sélectionner](#)

```
void Personnage::attaquer(Personnage &cible)
{
    cible.recevoirDegats(m_arme.m_degats);
}
```

Pourquoi est-ce interdit ? Parce que m_degats est un attribut, et que comme tout bon attribut qui se respecte, il est privé ! Diantre... On est en train d'utiliser la classe Arme au sein de la classe Personnage, et comme on est utilisateurs, on ne peut pas accéder aux éléments privés 😕

(La POO, ça peut parfois donner mal à la tête j'avais oublié de vous prévenir 🤪)

Bon, comment résoudre le problème ? Il n'y a pas 36 solutions. Ca va peut-être vous surprendre, mais on doit créer une méthode pour récupérer la valeur de cet attribut. Cette méthode est appelée accesseur et commence généralement par le préfixe get (récupérer, en anglais). Dans notre cas, notre méthode s'appellerait getDegats.

On conseille généralement de rajouter le mot-clé const aux accesseurs.

Une méthode... constante ? Qu'est-ce que ça signifie ? 😕

Une méthode constante est une méthode qui ne peut pas modifier les attributs de la classe. Cela garantit que la méthode ne fait que "lire" les attributs et qu'elle ne modifie donc pas l'objet. C'est une bonne habitude de programmation de créer des accesseurs const, bien que là encore ça ne soit pas obligatoire.

Voici à quoi ressemble la méthode, avec le mot-clé const :

Code : C++ - [Sélectionner](#)

```
int Arme::getDegats() const
{
    return m_degats;
}
```

Oubliez pas de mettre à jour Arme.h avec le prototype aussi, qui sera le suivant :

Code : C++ - [Sélectionner](#)

```
int getDegats() const;
```

Voilà, c'est con comme bonjour, ça peut paraître lourd, et pourtant c'est une sécurité nécessaire. On est parfois obligé de créer une méthode qui fait juste un return pour accéder indirectement à un attribut.

De même, on crée parfois des accesseurs permettant de modifier des attributs. Ces accesseurs sont généralement précédés du préfixe set (mettre, en anglais).

Vous avez peut-être l'impression qu'on viole la règle d'encapsulation ? Eh bien non. Car la méthode nous permet de faire des tests pour vérifier qu'on ne met pas n'importe quoi dans l'attribut, donc ça reste une façon sécurisée de modifier un attribut.

Vous pouvez maintenant retourner dans Personnage.cpp et écrire :

Code : C++ - Sélectionner

```
void Personnage::attaquer(Personnage &cible)
{
    cible.recevoirDegats(m_arme.getDegats());
}
```

getDegats renvoie le nombre de dégâts, qu'on envoie à la méthode recevoirDegats de la cible. Pfiou ! 😊

Le reste des méthodes n'a pas besoin de changer, à part changerArme de la classe Personnage :

Code : C++ - Sélectionner

```
void Personnage::changerArme(string nomNouvelleArme, int degatsNouvelleArme)
{
    m_arme.changer(nomNouvelleArme, degatsNouvelleArme);
}
```

On appelle la méthode changer de m_arme.

Le Personnage répercute donc la demande de changement d'arme à la méthode changer de son objet m_arme.

Comme vous pouvez le voir, on peut faire communiquer des objets entre eux, à condition d'être bien organisé et de se demander à chaque instant "est-ce que j'ai le droit d'accéder à cet élément ou pas ?".

N'hésitez pas à créer des accesseurs si besoin est, même si ça peut paraître lourd c'est la bonne méthode. En aucun cas vous ne devez mettre un attribut public pour simplifier un problème. Vous perdriez tous les avantages et la sécurité de la POO (et vous n'auriez aucun intérêt à continuer le C++ dans ce cas 🤪).

Action !

Nos personnages combattent dans le main, mais... on ne voit rien de ce qui se passe. Il serait bien d'afficher l'état de chacun des personnages pour savoir où ils en sont.

Je vous propose de créer une méthode afficherEtat dans Personnage. Cette méthode sera chargée de faire des cout pour afficher dans la console la vie, la mana et l'arme du personnage.

Prototype et include

On va rajouter le prototype, tout bête, dans le .h :

Code : C++ - Sélectionner

```
void afficherEtat();
```

Implémentation

Implémentons ensuite la méthode. C'est simple, on a juste des cout à faire. Grâce aux attributs, on peut indiquer toutes les infos sur le personnage :

Code : C++ - Sélectionner

```

void Personnage::afficherEtat()
{
    cout << "Vie : " << m_vie << endl;
    cout << "Mana : " << m_mana << endl;
    m_arme.afficher();
}

```

Comme vous pouvez le voir, les informations sur l'arme sont demandées à l'objet m_arme via sa méthode afficher(). Encore une fois, les objets communiquent entre eux pour récupérer les informations dont ils ont besoin.

Appel de afficherEtat dans le main

Bien, tout ça c'est bien beau, mais tant qu'on n'appelle pas la méthode, elle ne sert à rien 😱
Je vous propose donc de compléter le main et de rajouter à la fin les appels de méthode :

Code : C++ - [Sélectionner](#)

```

int main()
{
    // Création des personnages
    Personnage david, goliath("Epée aiguisee", 20);

    // Au combat !
    goliath.atttaquer(david);
    david.boirePotionDeVie(20);
    goliath.atttaquer(david);
    david.atttaquer(goliath);
    goliath.changerArme("Double hache tranchante vénéneuse de la mort", 40);
    goliath.atttaquer(david);

    // Temps mort ! Voyons voir la vie de chacun...
    cout << "David" << endl;
    david.afficherEtat();
    cout << endl << "Goliath" << endl;
    goliath.afficherEtat();

    return 0;
}

```

On peut enfin exécuter le programme et voir quelque chose dans la console 😊

Code : Console - [Sélectionner](#)

```

David
Vie : 40
Mana : 100
Arme : Epée rouillée (Degats : 10)

Goliath
Vie : 90
Mana : 100
Arme : Double hache tranchante vénéneuse de la mort (Degats : 40)

```

Si vous êtes sous Windows, vous aurez probablement un bug avec les accents dans la console. Ignorez-le, ne vous en préoccupez pas, ce qui nous intéresse c'est le fonctionnement de la POO ici. Et puis de toute manière, dans la prochaine partie du cours on travaillera avec de vraies fenêtres, donc la console c'est temporaire pour nous 😊

Pour que vous puissiez vous faire une bonne idée du projet dans son ensemble, je vous propose de télécharger un fichier zip contenant :

- main.cpp
- Personnage.cpp
- Personnage.h
- Arme.cpp
- Arme.h

... bref, c'est-à-dire tout le projet tel qu'il est sur mon ordinateur à l'heure actuelle.

Télécharger le projet RPG (3 Ko)

Je vous invite à faire des tests pour vous entraîner. Par exemple :

- Continuez à faire combattre david et goliath dans le main en affichant leur état de temps en temps.
 - Introduisez un troisième personnage dans l'arène pour rendre le combat plus ~~brutal~~ intéressant 🍸
 - Rajoutez un attribut m_nom pour stocker le nom du personnage dans l'objet. Pour le moment, nos personnages ne savent même pas comment ils s'appellent, c'est un peu bête 😅
- Du coup, je pense qu'il faudrait modifier les constructeurs et obliger l'utilisateur à indiquer un nom pour le personnage lors de sa création... à moins que vous ne donnez un nom par défaut si rien n'est précisé ? A vous de choisir !
- Rajoutez des cout dans les autres méthodes de Personnage pour indiquer à chaque fois ce qui est en train de se passer ("machin boit une potion qui lui redonne 20 points de vie")
 - Rajoutez d'autres méthodes au gré de votre imagination... et pourquoi pas des attaques magiques qui utilisent de la mana ?
 - Enfin, pour l'instant le combat est tout écrit dans le main, mais vous pourriez laisser le joueur choisir les attaques dans la console. Vous savez le faire, allez allez !

Prenez cet exercice très au sérieux, ceci est peut-être la base de votre futur MMORPG révolutionnaire !

Précision utile : la phrase ci-dessus était une boutade 🍸

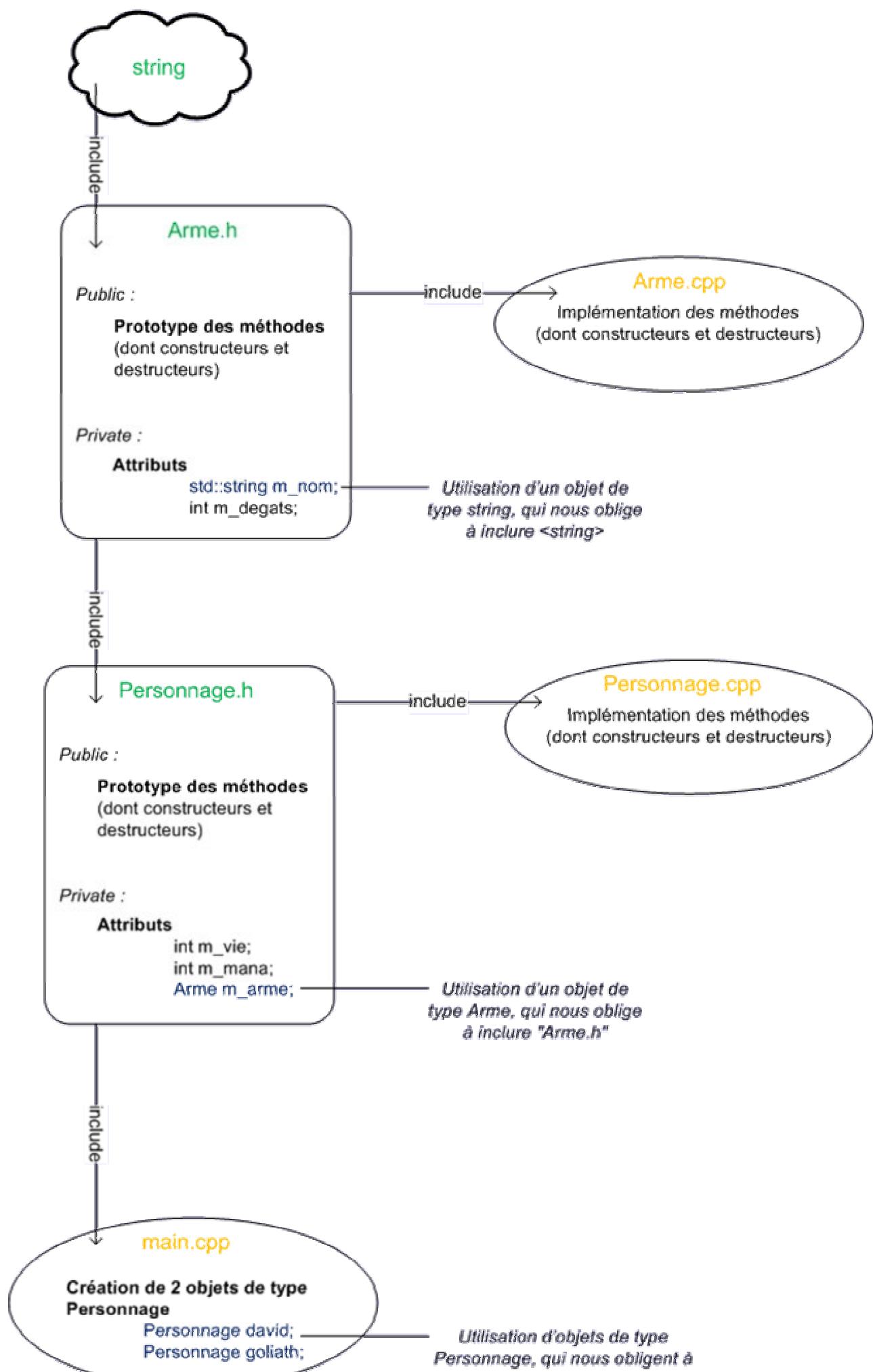
Ce cours ne vous apprendra pas à créer un MMORPG, vu le travail phénoménal que cela représente. Mieux vaut commencer par se concentrer sur de plus petits projets réalistes, et notre RPG en est un. Ce qui est intéressant ici, c'est de voir comment est conçu un jeu orienté objet (comme c'est le cas de la plupart des jeux aujourd'hui). Si vous avez bien compris le principe, vous devriez commencer à voir des objets dans tous les jeux que vous connaissez ! Par exemple, un bâtiment dans Age of Empires est un objet qui a un niveau de vie, un nom, il peut produire des unités (via une méthode), etc.

Si vous commencez à voir des objets partout, c'est bon signe ! C'est ce que l'on appelle "penser objet" 😊

Méga schéma résumé

Croyez-moi si vous le voulez, mais je vous demande même pas vraiment d'être capable de programmer tout ce qu'on vient de voir en C++. Je veux que vous reteniez le principe, le concept, comment tout cela est agencé.

Et pour retenir, rien de tel qu'un méga schéma bien mastoc, non ? Ouvrez grand vos yeux, je veux que vous soyez capable de le reproduire les yeux fermés la tête en bas avec du poil à gratter dans le dos !



Si vous avez dû retenir une bonne chose de ce second chapitre, c'est cet échange, cette communication constante entre les objets. Et encore ! On n'avait ici que 2 classes, Personnage et Arme. Je vous laisse imaginer dans un vrai projet ce que ça donne 😊 L'intérêt de la POO est là : une organisation précise, chaque objet fait ce qu'il a à faire et délègue certaines parties de son travail à d'autres objets (ici, Personnage déléguaient la gestion de l'arme à un objet de type Arme).

On ne peut pas dire "Je fais de la POO" du jour au lendemain, c'est clair. C'est un travail qui demande de l'organisation, de la méthode. Il faut toujours bien réfléchir avant de se lancer dans un projet, si simple soit-il.

Mais réfléchir un peu avant de programmer, est-ce un mal ? Je ne crois pas 😊

Concentrez-vous sur le fichier zip que je vous ai donné et essayez de vous familiariser avec, en faisant par exemple les améliorations proposées. Il ne faut surtout pas que vous soyez perdus.

Dans le chapitre suivant, nous allons aller un peu plus dans le détail des classes en introduisant... les pointeurs ! 😊 Les pointeurs en POO méritent en effet à eux seuls au moins un chapitre entier.

Classes et pointeurs

Dans les chapitres précédents, j'ai volontairement évité d'introduire les pointeurs avec les classes. En effet, les pointeurs en C++ sont un vaste sujet, et un sujet sensible. Comme vous l'avez probablement remarqué par le passé, bien gérer les pointeurs est essentiel car à la moindre erreur votre programme risque de :

- Consommer trop de mémoire parce que vous oubliez de libérer certains éléments
- Voir tout simplement de planter si votre pointeur pointe vers n'importe où dans la mémoire

Comment associe-t-on classes et pointeurs ? Quelles sont les règles à connaître, les bonnes habitudes à prendre ? Voilà un sujet qui méritait au moins un chapitre à lui tout seul 😊

Attention : c'est un chapitre que je classe entre "très difficile" et "très très difficile". Bref, vous m'avez compris, les pointeurs en C++ c'est pas de la tarte, alors quadruplez d'attention lorsque vous lirez ce chapitre. Le sujet est complexe et épique, je ne vous le dirai pas deux fois 😊

Pointeur d'une classe vers une autre classe

Reprenons notre classe Personnage 😊

Dans le dernier chapitre, nous lui avons ajouté une Arme que nous avons directement intégré à ses attributs :

Code : C++ - Sélectionner

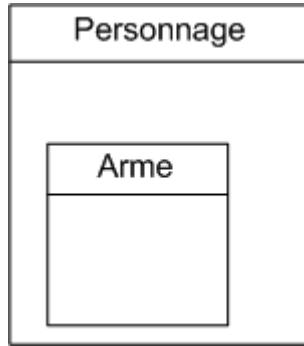
```
class Personnage
{
    public:

    // Quelques méthodes...

    private:
        Arme m_arme; // L'Arme est "contenue" dans le Personnage
        // ...
};
```

Il y a plusieurs façons différentes d'associer des classes entre elles. Celle-ci fonctionne bien dans notre cas, mais l'Arme est vraiment "liée" au Personnage. Elle ne peut pas en sortir.

Schématiquement, ça donnerait quelque chose de ce genre :



L'Arme est vraiment dans le Personnage.

Il y a une autre technique, plus souple, qui permet plus de possibilités, mais qui est plus complexe : ne pas intégrer l'Arme au Personnage et utiliser un pointeur à la place. Au niveau de la déclaration de la classe, le changement correspond à... une étoile en plus :

Code : C++ - [Sélectionner](#)

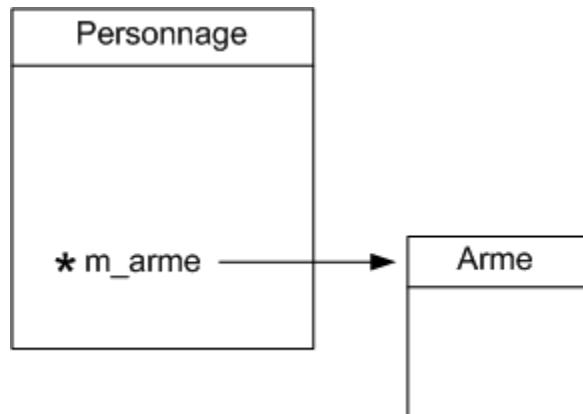
```

class Personnage
{
public:
    // Quelques méthodes...

private:
    Arme *m_arame; // L'Arme est un pointeur, l'objet n'est plus contenu dans le Personnage
    // ...
}

```

Notre Arme étant un pointeur, on ne peut plus dire qu'elle appartient au Personnage.
En schéma, ça donne ça :



On considère que l'arme est maintenant externe au personnage.

Les avantages de cette technique sont les suivants :

- Le Personnage peut changer d'Arme en faisant tout simplement pointer m_arame vers un autre objet. Par exemple, si le Personnage possède un inventaire (dans un sac à dos), il peut changer son Arme à tout moment en modifiant le pointeur.
- Le Personnage peut donner son Arme à un autre Personnage, il suffit de changer les pointeurs de chacun des personnages.
- Si le Personnage n'a plus d'Arme, il suffit de mettre le pointeur m_arame à NULL.

Mais des défauts, il y en a aussi. Gérer une classe qui contient des pointeurs, c'est pas de la tarte vous pouvez me croire, et

d'ailleurs vous allez le voir 😊

Alors, faut-il utiliser un pointeur ou pas pour l'arme ? Les 2 façons de faire sont valables, et ont chacune leurs avantages et défauts. Utiliser un pointeur est probablement ce qu'il y a de plus souple, mais c'est aussi plus difficile. Retenez donc qu'il n'y a pas de "meilleure" méthode adaptée à tous les cas, ce sera à vous de choisir en fonction de votre cas si vous intégrez directement un objet dans une classe ou si vous utilisez un pointeur.

Gestion de l'allocation dynamique

On va ici voir comment on travaille quand une classe contient des pointeurs vers des objets.

On travaille là encore sur notre classe Personnage et je suppose que vous avez mis l'attribut m_arme en pointeur comme je l'ai montré un peu plus haut :

Code : C++ - [Sélectionner](#)

```
class Personnage
{
    public:

        // Quelques méthodes...

    private:

        Arme *m_arme; // L'Arme est un pointeur, l'objet n'est plus contenu dans le Personnage
        // ...
    }.
```

(je ne réécris volontairement pas tout le code, juste l'essentiel pour qu'on puisse se concentrer dessus)

Notre arme étant un pointeur, il va falloir faire une allocation dynamique avec new pour créer l'objet. Sinon, l'objet ne se créera pas tout seul 😊

Allocation de mémoire pour l'objet

L'allocation de mémoire pour notre arme se fait où à votre avis ?

Il n'y a pas 36 endroits pour ça : c'est dans le constructeur. C'est en effet le rôle du constructeur que de faire en sorte que l'objet soit bien construit, donc notamment que tous les pointeurs pointent vers quelque chose 😊

Dans notre cas, on est obligé de faire une allocation dynamique, donc d'utiliser new. Voici ce que ça donne dans le constructeur par défaut :

Code : C++ - [Sélectionner](#)

```
Personnage::Personnage() : m_vie(100), m_mana(100)
{
    m_arme = new Arme();
}
```

Si vous vous souvenez bien, on avait aussi fait un second constructeur pour ceux qui veulent que le Personnage commence avec une arme plus puissante dès le départ. Il faut là aussi y faire une allocation dynamique :

Code : C++ - [Sélectionner](#)

```

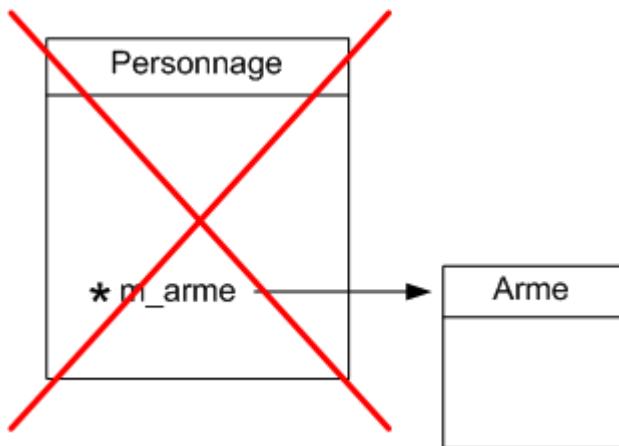
Personnage::Personnage(string nomArme, int degatsArme) : m_vie(100), m_mana(100)
{
    m_arame = new Arme(nomArme, degatsArme);
}

```

Explications : new Arme() appelle le constructeur par défaut de la classe Arme, tandis que new Arme(nomArme, degatsArme) appelle le constructeur surchargé. Le new renvoie l'adresse de l'objet créé, adresse qui est stockée dans notre pointeur m_arame. On ne peut pas faire de new dans la liste d'initialisation, ce qui explique pourquoi on le fait entre les accolades {}.

Désallocation de mémoire pour l'objet

Notre arme étant un pointeur, lorsque l'objet de type Personnage est supprimé l'arme ne disparaît pas toute seule ! Si on fait juste un new dans le constructeur, et rien dans le destructeur, il va se passer ceci lorsque l'objet de type Personnage sera détruit :



L'objet de type Personnage va bel et bien disparaître, mais l'objet de type Arme va subsister en mémoire et il n'y aura plus aucun pointeur pour se "rappeler" de son adresse. En clair, l'arme va traîner en mémoire et on ne pourra plus jamais la supprimer.

Pour résoudre ce problème, il faut faire un delete de l'arme dans le destructeur du personnage afin que l'arme soit supprimée avant le personnage. Le code est tout simple :

Code : C++ - Sélectionner

```

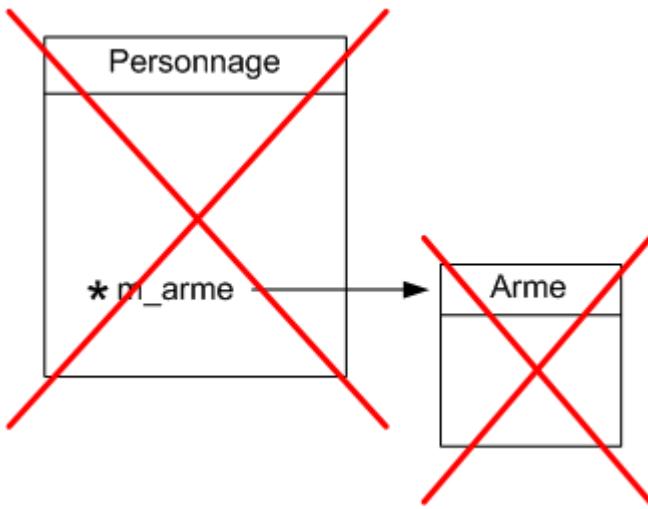
Personnage::~Personnage()
{
    delete m_arame;
}

```

Cette fois le destructeur est réellement indispensable. Maintenant, lorsque quelqu'un demandera à détruire le Personnage, il va se passer ceci :

1. Appel du destructeur... et donc dans notre cas suppression de l'Arme (avec le delete).
2. Puis enfin suppression du Personnage.

Au final, les 2 objets seront bel et bien supprimés et la mémoire sera propre :



N'oubliez pas que `m_arame` est maintenant un pointeur !

Cela implique de changer toutes les méthodes qui l'utilisent. Par exemple :

Code : C++ - Sélectionner

```
void Personnage::attaquer(Personnage &cible)
{
    cible.recevoirDegats(m_arame.getDegats());
}
```

... devient :

Code : C++ - Sélectionner

```
void Personnage::attaquer(Personnage &cible)
{
    cible.recevoirDegats(m_arame->getDegats());
}
```

Notez la différence : le point a été remplacé par la flèche, car `m_arame` est un pointeur. Cela ne devrait pas être nouveau pour vous si vous avez bien suivi le cours jusqu'ici, mais je préfère le rappeler sait-on jamais 😊

Le constructeur de copie

Le constructeur de copie, c'est une surcharge particulière du constructeur.

Le constructeur de copie devient généralement indispensable dans une classe qui contient des pointeurs, et ça tombe bien vu que c'est justement notre cas ici 😊

Le problème

Pour bien comprendre l'intérêt du constructeur de copie, voyons voir concrètement ce qui se passe lorsqu'on crée un objet en l'affectant par... un autre objet ! Par exemple :

Code : C++ - Sélectionner

```
int main()
{
    Personnage goliath("Epée aiguisée", 20);

    Personnage david = goliath; // On crée david à partir de goliath. David sera une "copie"

    return 0;
}
```

5

6

Lorsqu'on construit un objet en lui affectant directement un autre objet, comme on vient de le faire ici avec le signe "=", le compilateur appelle une méthode appelée constructeur de copie.

Le rôle du constructeur de copie est de copier la valeur de tous les attributs du premier objet dans le second. Donc david récupère la vie de goliath, la mana de goliath, etc.

Dans quels cas le constructeur de copie est-il appelé ?

On vient de le voir, le constructeur de copie est appelé lorsqu'on crée un nouvel objet en l'affectant par la valeur d'un autre :

Code : C++ - [Sélectionner](#)

```
Personnage david = goliath; // Appel du constructeur de copie (cas 1)
```

Ceci est strictement équivalent à écrire :

Code : C++ - [Sélectionner](#)

```
Personnage david(goliath); // Appel du constructeur de copie (cas 2)
```

Dans ce second cas le constructeur de copie est là aussi appelé.

Mais ce n'est pas tout ! Lorsque vous envoyez un objet à une fonction sans utiliser de pointeur ni de référence, l'objet est là aussi copié !

Imaginons la fonction :

Code : C++ - [Sélectionner](#)

```
void maFonction(Personnage unPersonnage)
{
}
```

Si vous appelez cette fonction qui n'utilise pas de pointeur ni de référence, alors l'objet sera copié en utilisant un constructeur de copie au moment de l'appel de la fonction :

Code : C++ - [Sélectionner](#)

```
maFonction(Goliath); // Appel du constructeur de copie (cas 3)
```

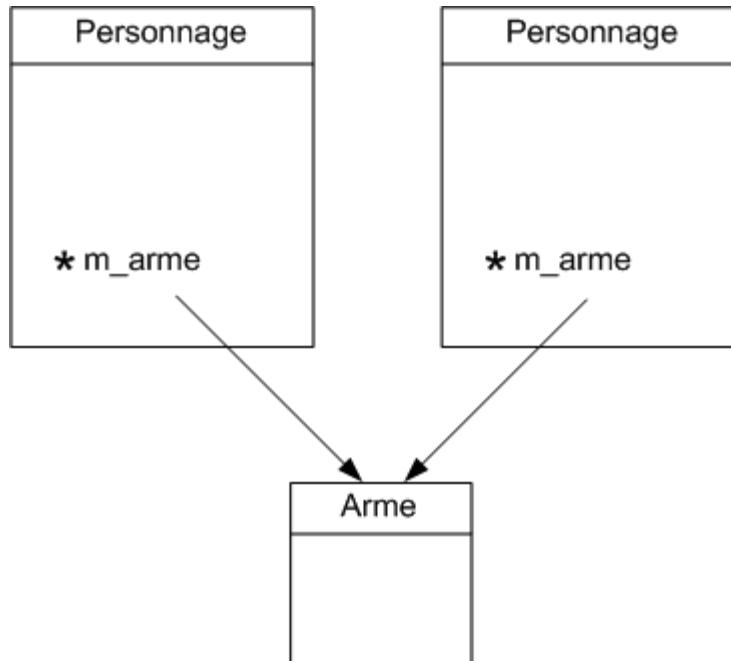
Bien entendu, il est préférable d'utiliser un pointeur ou une référence en général car l'objet n'a pas besoin d'être copié, donc ça va bien plus vite et ça prend moins de mémoire. Toutefois, il arrivera des cas où vous aurez besoin de créer une fonction comme ici qui fait une copie de l'objet.

Si vous n'écrivez pas vous-mêmes un constructeur de copie pour votre classe, il sera généré automatiquement pour vous par le compilateur. Ok, c'est sympa de sa part, mais le compilateur est... comment dire pour pas le froisser... bête 😊

En fait, le constructeur de copie généré se contente de copier la valeur de tous les attributs... même des pointeurs !

Le problème ? Eh bien justement, il se trouve qu'un des attributs est un pointeur dans notre classe Personnage ! Que fait l'ordinateur ? Il copie la valeur du pointeur, donc l'adresse de l'arme. Au final, les 2 objets ont un pointeur qui pointe vers le même objet de type Arme !

Ah les fourbes !



L'ordinateur a copié le pointeur, et donc les 2 pointeurs pointent vers la même arme !

Si on ne fait rien pour régler ça, imaginez ce qu'il va se passer lorsque les 2 personnages seront détruits... Le premier sera détruit, ainsi que son arme car le destructeur ordonnera la suppression de l'arme avec un delete. Et quand arrivera le tour du second personnage, le delete va planter (et votre programme avec 😱) parce que l'arme aura déjà été détruite !

Le constructeur de copie généré automatiquement par le compilateur n'est pas assez intelligent pour comprendre qu'il faut allouer de la mémoire pour une autre arme... Qu'à cela ne tienne, nous allons le lui expliquer 🍪

Création du constructeur de copie

Le constructeur de copie, comme je vous l'ai dit un peu plus haut, est une surcharge particulière du constructeur. C'est un constructeur qui prend pour paramètre... une référence constante vers un objet du même type !

Si vous trouvez pas ça clair, peut-être qu'un exemple vous aidera 😊

Code : C++ - [Sélectionner](#)

```

class Personnage
{
    public:

        Personnage();
        Personnage(const Personnage &personnageACopier); // Le prototype du constructeur de copie
        Personnage(std::string nomArme, int degatsArme);
        ~Personnage();

        /*
        ... plein d'autres méthodes qui ne nous intéressent pas ici
        */

    private:

        int m_vie;
        int m_mana;
        Arme *m_arame;
}

```

3

4

5

En résumé, le prototype d'un constructeur de copie est :

Code : C++ - Sélectionner

```
Objet(const Objet &objetACopier);
```

6

Le const indique juste qu'on n'a pas le droit de modifier les valeurs de l'objetACopier (c'est logique, on a juste besoin de "lire" ses valeurs pour le copier).

Ecrivons l'implémentation de ce constructeur. Il va falloir copier tous les attributs du personnageACopier dans le personnage actuel. Commençons par les attributs "simples", c'est-à-dire ceux qui ne sont pas des pointeurs :

Code : C++ - Sélectionner

```
Personnage::Personnage(const Personnage &personnageACopier)
{
    m_vie = personnageACopier.m_vie;
    m_mana = personnageACopier.m_mana;
}
```

Vous vous demandez peut-être comment cela se fait qu'on puisse accéder aux attributs m_vie et m_mana du personnageACopier ? Si vous vous l'êtes demandé, je vous félicite, ça veut dire que le principe d'encapsulation commence à rentrer dans votre tête 😊 Eh oui, en effet, m_vie et m_mana sont privés, donc on ne peut pas y accéder depuis l'extérieur de la classe... sauf qu'il y a une exception ici : on est dans une méthode de la classe Personnage, et on a le droit d'accéder à tous les éléments (même privés) d'un autre Personnage.

C'est un peu tordu je l'avoue, mais dans le cas présent ça nous simplifie grandement la vie 😊 Retenez donc qu'un objet de type X peut accéder à tous les éléments (même privés) d'un autre objet s'il est du même type X.

Il reste maintenant à "copier" m_arame. Si on écrit :

Code : C++ - Sélectionner

```
m_arame = personnageACopier.m_arame;
```

... on fait exactement la même erreur que le compilateur, c'est-à-dire qu'on ne copie que l'adresse de l'objet de type Arme, et pas l'objet en entier !

Pour résoudre le problème, il va falloir copier l'objet de type Arme en faisant une allocation dynamique, donc un new. Attention,

accrochez-vous parce que ce n'est pas simple 😱

Si on fait :

Code : C++ - [Sélectionner](#)

```
m_arame = new Arme();
```

... on va bien créer une nouvelle arme, mais on utilisera le constructeur par défaut, donc cela créera l'arme de base. Or, on veut avoir exactement la même arme que celle du personnageACopier (ben oui, c'est un constructeur de copie 😊).

La bonne nouvelle, comme je vous l'ai dit plus haut, c'est que le constructeur de copie est automatiquement généré par le compilateur. Tant que la classe n'utilise pas de pointeurs vers des attributs, il n'y a pas de danger. Et ça tombe bien, la classe Arme n'utilise pas de pointeurs, on va donc pouvoir se contenter du constructeur qui a été généré.

Il faut donc appeler le constructeur de copie, en envoyant en paramètre l'objet à copier. Vous pourriez penser qu'il faut faire ceci :

Code : C++ - [Sélectionner](#)

```
m_arame = new Arme(personnageACopier.m_arame);
```

Presque ! Sauf que m_arame est un pointeur, et le prototype du constructeur de copie est :

Code : C++ - [Sélectionner](#)

```
Arme(const Arme &arme);
```

... ce qui veut dire qu'il faut envoyer l'objet lui-même et pas son adresse. Vous vous souvenez comment on fait pour obtenir l'objet (ou la variable) à partir de son adresse ? On utilise l'étoile * !

Ce qui donne au final :

Code : C++ - [Sélectionner](#)

```
m_arame = new Arme(*(personnageACopier.m_arame));
```

Cette ligne alloue dynamiquement une nouvelle arme, en se basant sur l'arme du personnageACopier. Pas simple je le reconnais, mais relisez plusieurs fois les étapes de mon raisonnement et vous allez comprendre 😊
Pour bien suivre tout ce que j'ai dit, il faut vraiment que vous soyez au point sur tout : les pointeurs, les références, et les... constructeurs de copie 😊

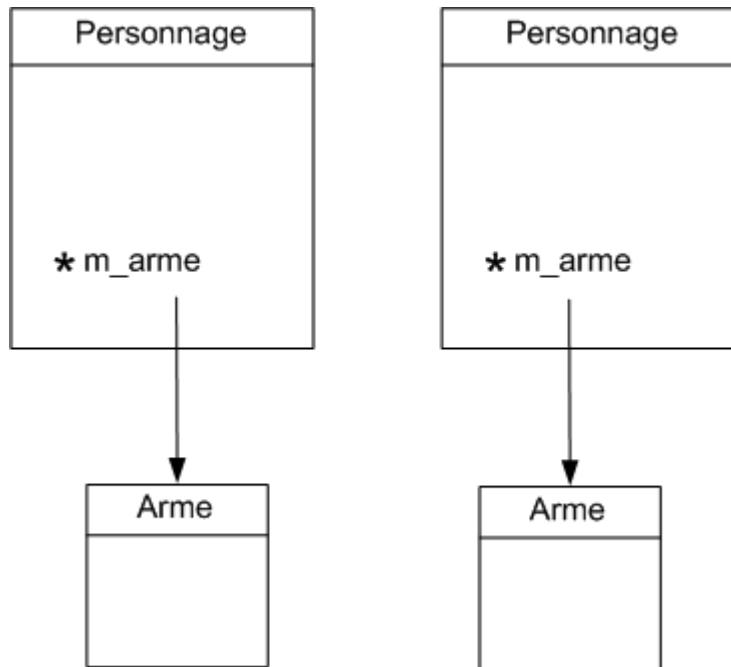
Le constructeur de copie une fois terminé

Le bon constructeur de copie ressemblera donc à ceci au final :

Code : C++ - [Sélectionner](#)

```
Personnage::Personnage(const Personnage &personnageACopier)
{
    m_vie = personnageACopier.m_vie;
    m_mana = personnageACopier.m_mana;
    m_arame = new Arme(*(personnageACopier.m_arame));
}
```

Ainsi, nos 2 personnages ont tous deux une arme identique, mais dupliquée afin d'éviter les problèmes que je vous ai expliqués plus haut :



Notez qu'on peut aussi utiliser la liste d'initialisation pour tous les attributs qui ne nécessitent pas d'allocation dynamique, à savoir m_vie et m_mana. On peut donc aussi écrire le constructeur de copie de cette manière :

Code : C++ - Sélectionner

```

Personnage::Personnage(const Personnage &personnageACopier) : m_vie(personnageACopier.m_vie),
{
    m_arame = new Arme(*(personnageACopier.m_arame));
}

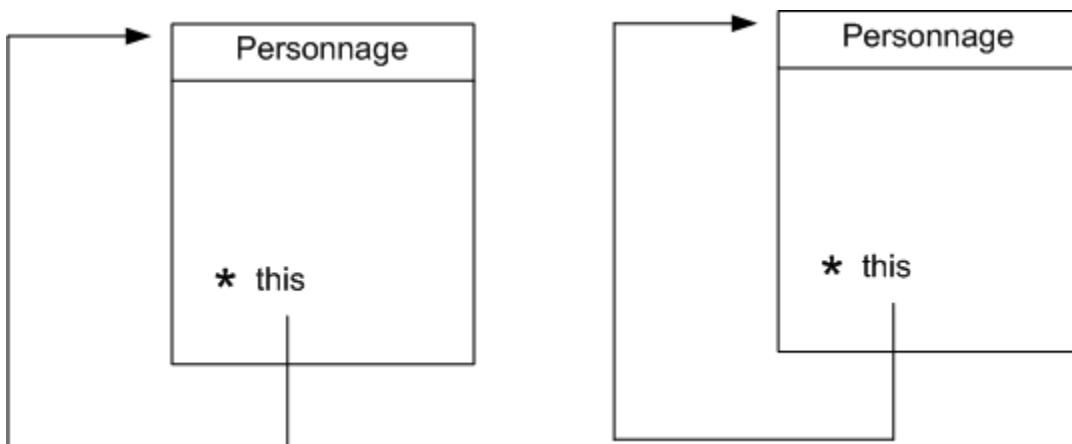
```

La POO n'est pas simple comme vous commencez à vous en rendre compte, surtout quand on commence à manipuler des objets. Heureusement, vous aurez l'occasion de pratiquer tout cela par la suite, et vous allez petit à petit prendre l'habitude d'éviter les pièges des pointeurs.

Le pointeur this

Pour terminer ce chapitre sur une note plus cool, puisqu'on parle de POO et de pointeurs, je me dois de vous parler du pointeur this. Pas de panique, c'est très simple, ça ira vite et vous ne sentirez aucune douleur 😊

Dans toutes les classes, on dispose d'un pointeur ayant pour nom this. Ce pointeur pointe vers l'objet actuel. Je reconnais que ce n'est pas simple à imaginer, mais je pense que ça passera mieux avec un schéma maison :



Chaque objet (ici de type Personnage) possède un pointeur this qui pointe vers... l'objet lui-même !

this étant utilisé par le langage C++ dans toutes les classes, vous ne pouvez donc pas créer de variable appelée this car cela créerait un conflit. De même, si vous commencez à essayer d'appeler vos variables class, new, delete, return, forcément ça risque de coincer un peu 😊

Ces mots-clés sont ce qu'on appelle des mots-clés réservés. Le langage C++ se les réserve pour son usage personnel, vous n'avez donc pas le droit de créer des variables (ou des fonctions) portant ces noms-là.

Mais... à quoi peut bien servir this ???

Répondre à cette question me sera délicat 😊

Je peux vous donner un exemple : vous êtes dans une méthode de votre classe, et cette méthode doit renvoyer un pointeur vers l'objet auquel elle appartient. Sans le this, on ne pourrait pas l'écrire. Voilà ce que ça pourrait donner :

Code : C++ - [Sélectionner](#)

```
Personnage* Personnage::getAdresse( )
{
    return this;
}
```

Dans l'immédiat, vous n'en avez certainement pas l'utilité, mais il arrivera un jour où, pour résoudre un problème particulier, vous aurez besoin d'un tel pointeur. Ce jour-là, souvenez-vous qu'un objet peut "retrouver" son adresse à l'aide du pointeur this.

Comme c'est l'endroit le plus adapté pour en parler dans ce cours, j'en profite. Ca ne va pas changer votre vie tout de suite, mais il se peut que bien plus tard, dans plusieurs chapitres je vous dise tel un vieillard sur sa canne "Souvenez-vous, souvenez-vous du pointeur this ! 🧑". Alors ne l'oubliez pas 😊

Si vous êtes en train de vous shooter à l'aspirine pour éviter que votre tête n'explose, je vous conseille de conserver encore des munitions 😊

En effet, on n'a pas fini d'en découdre avec la POO et il vous reste encore beaucoup de choses à apprendre. Heureusement, enfin si ça peut vous rassurer, ce chapitre était probablement l'un des plus difficiles de tout le cours (mais pas nécessairement LE plus difficile 😊).

Sachez quoiqu'il en soit que les pointeurs en C++ sont de véritables casse-têtes, même pour les programmeurs plus expérimentés. Il faut faire constamment attention, car une fuite de mémoire (oubli de libérer des objets) est très vite arrivée, et je ne vous parle pas des plantages de programme que ça peut occasionner. Une très très grande part des plantages des programmes que vous connaissez sont dûs à une mauvaise gestion de la mémoire, c'est vous dire !

Dans le prochain chapitre, nous allons jouer avec la surcharge des opérateurs, ce qui va nous permettre de faire des choses étonnantes avec nos objets.

Puis, chemin faisant, nous nous rapprocherons d'un des thèmes majeurs de la programmation orientée objet, quelque chose d'indispensable à quoi vous ne pouvez échapper et qui porte un bien funeste nom : l'héritage. Voilà un peu le genre de choses qui vous attend 😊

Qu'on ne s'y trompe pas : tout ceci est peut-être complexe et pas toujours très "amusant" à apprendre, mais vous en aurez vraiment besoin dans la partie II lorsque nous travaillerons avec la librairie Qt pour créer des fenêtres, travailler en réseau, etc. Donc on se motive, et on continue ! 😊

La surcharge d'opérateurs

On l'a vu, le langage C++ propose beaucoup de nouveautés qui peuvent se révéler très utiles, si on arrive à s'en servir correctement (je pense par exemple à la surcharge de fonctions).

Une des nouveautés les plus étonnantes est la surcharge des opérateurs, que nous allons étudier dans ce chapitre. C'est une technique qui permet de réaliser des opérations mathématiques intelligentes entre vos objets lorsque vous utilisez dans votre code des symboles comme +, -, *, etc.

Au final, votre code sera plus court et plus clair, et gagnera donc en lisibilité vous allez voir 😊

Petits préparatifs

Qu'est-ce que c'est ?

Le principe est très simple. Supposons que vous ayez créé une classe pour stocker une durée (ex. : 4h23m), et que vous avez 2 objets de type Duree. Vous voulez les additionner entre eux pour connaître la durée totale.

En temps normal, il faudrait créer une fonction "additionner" :

Code : C++ - [Sélectionner](#)

```
resultat = additionner(duree1, duree2);
```

La fonction additionner ferait ici la somme de duree1 et duree2 et stockerait ça dans resultat.

Ca fonctionne, mais ce n'est pas franchement lisible. Ce que je vous propose dans ce chapitre, c'est d'être capable d'écrire ça :

Code : C++ - [Sélectionner](#)

```
resultat = duree1 + duree2;
```

En clair, on fait ici comme si nos objets étaient de simples "nombres". Mais comme un objet c'est plus complexe qu'un nombre (vous avez eu l'occasion de vous en rendre compte 🤔), il va falloir expliquer à l'ordinateur comment effectuer l'opération.

La classe Duree pour nos exemples

Toutes les classes ne sont pas forcément adaptées à la surcharge d'opérateurs. Ainsi, ajouter des objets de type Personnage entre eux serait pour le moins un peu louche 😐

Nous allons donc changer d'exemple, ça sera l'occasion de vous aérer un peu l'esprit sinon vous allez finir par croire que le C++ ne sert qu'à créer des RPG 😅

Cette classe Duree sera capable de stocker des heures, des minutes et des secondes. Rassurez-vous, c'est une classe relativement facile à écrire (plus facile que Personnage en tout cas !), ça ne devrait vous poser aucun problème si vous avez compris les chapitres précédents.

[Duree.h](#)

Code : C++ - [Sélectionner](#)

```

#ifndef DEF_DUREE
#define DEF_DUREE

class Duree
{
public:

    Duree(int heures = 0, int minutes = 0, int secondes = 0);

private:

    int m_heures;
    int m_minutes;
    int m_secondes;
};

#endif

```

Chaque objet de type Duree stockera un certain nombre d'heures, minutes et secondes.

Vous noterez que j'ai utilisé des valeurs par défaut au cas où l'utilisateur aurait la flemme de les préciser 😊
On pourra donc créer un objet de plusieurs façons différentes :

Code : C++ - Sélectionner

```

Duree chrono; // Pour stocker 0 heures, 0 minutes et 0 secondes
Duree chrono(5); // Pour stocker 5 heures, 0 minutes et 0 secondes
Duree chrono(5, 30); // Pour stocker 5 heures, 30 minutes et 0 secondes
Duree chrono(0, 12, 55); // Pour stocker 0 heures, 12 minutes et 55 secondes

```

Duree.cpp

L'implémentation de notre constructeur est expédiée en 30 secondes montre en main 😊

Code : C++ - Sélectionner

```

#include "Duree.h"

Duree::Duree(int heures, int minutes, int secondes) : m_heures(heures), m_minutes(minutes)
{
}

```

3

5

6

4

Et dans main.cpp ?

Pour l'instant notre main.cpp ne va déclarer que 2 objets de type Duree, que j'initialise un peu au pif :

Code : C++ - Sélectionner

```

int main()
{
    Duree duree1(0, 10, 28), duree2(0, 15, 2);

    return 0;
}

```

Voilà, nous sommes prêts à affronter les surcharges d'opérateurs maintenant ! 😎

Les plus perspicaces d'entre vous auront remarqué que rien ne m'interdit de créer un objet avec 512 minutes et 1455 secondes. En

effet, on peut écrire `Duree chrono(0, 512, 1455);` sans être inquiété. Normalement, cela devrait être interdit, ou tout du moins notre constructeur devrait être assez intelligent pour "découper" les minutes et les convertir en heures/minutes, et de même pour les secondes, afin qu'elles ne dépassent pas 60.

Je ne le fais pas ici, mais je vous encourage à modifier votre constructeur pour faire cette conversion si nécessaire, ça vous entraînera ! Etant donné qu'il faut faire des if et quelques petites opérations mathématiques dans le constructeur, vous ne pourrez pas utiliser de liste d'initialisation.

Les opérateurs arithmétiques (+, -, *, /, %)

Nous allons commencer par voir les opérateurs mathématiques les plus classiques, à savoir l'addition, la soustraction, la multiplication, la division et le modulo.

Une fois que vous aurez appris à vous servir de l'un d'entre eux, vous verrez que vous saurez vous servir de tous les autres 😊

Pour être capable d'utiliser le symbole "+" entre 2 objets, vous devez créer une méthode ayant précisément pour nom `operator+` qui a pour prototype :

Code : C++ - [Sélectionner](#)

```
Objet operator+(const Objet &monObjet);
```

La méthode reçoit donc une référence sur l'objet (constant, donc on ne peut pas le modifier) à additionner. Dans notre classe Duree, on doit donc rajouter cette méthode (ici dans le .h) :

Code : C++ - [Sélectionner](#)

```
Duree operator+(const Duree &duree);
```

Mode d'utilisation

Comment ça marche ce truc ? 😊

Dès le moment où vous avez créé cette méthode `operator+`, vous pouvez additionner 2 objets de type Duree entre eux :

Code : C++ - [Sélectionner](#)

```
resultat = duree1 + duree2;
```

Ce n'est pas de la magie. En fait le compilateur "traduit" ça par :

Code : C++ - [Sélectionner](#)

```
resultat = duree1.operator+(duree2);
```

... ce qui est beaucoup plus classique et compréhensible pour lui 😊

Il appelle donc la méthode `operator+` de l'objet `duree1`, et envoie `duree2` en paramètre à la méthode. La méthode, elle, va retourner un résultat de type `Duree`.

Implémentation

L'implémentation n'est pas vraiment compliquée, mais il va quand même falloir réfléchir un peu. En effet, ajouter des secondes, minutes et heures ça va, mais il faut faire attention à la retenue si ça dépasse 60.

Je vous recommande d'essayer d'écrire la méthode vous-même, c'est un excellent exercice algorithmique, ça entretient le cerveau,

ça vous rend meilleur programmeur (je vous ai convaincus là ? 😊)

Voici ce que donne mon implémentation pour ceux qui ont besoin de la solution :

Code : C++ - [Sélectionner](#)

```
Duree Duree::operator+(const Duree &duree)
{
    int heures = m_heures;
    int minutes = m_minutes;
    int secondes = m_secondes;

    // 1 : ajout des secondes
    secondes += duree.m_secondes; // Exceptionnellement autorisé car même classe
    // Si le nombre de secondes dépasse 60, on rajoute des minutes et on met un nombre de
    minutes += secondes / 60;
    secondes %= 60;

    // 2 : ajout des minutes
    minutes += duree.m_minutes;
    // Si le nombre de minutes dépasse 60, on rajoute des heures et on met un nombre de
    heures += minutes / 60;
    minutes %= 60;

    // 3 : ajout des heures
    heures += duree.m_heures;

    // Création de l'objet résultat et retour
    Duree resultat(heures, minutes, secondes);
    return resultat;
}
```

3

4

5

6

Ce n'est pas un algorithme ultracomplexe, mais comme je vous avais dit il faut réfléchir un tout petit peu pour pouvoir l'écrire quand même 😊

On commence par créer et initialiser 3 variables locales (heures, minutes et secondes) qui correspondent au résultat. Ce résultat, on le mettra dans un objet de type Duree que l'on renverra à la fin.

On a initialisé ces 3 variables avec la valeur de l'objet sur lequel on travaille (duree1 si on se fie à l'exemple donné un peu plus haut).

On va y ajouter les heures, minutes et secondes de l'objet reçu en paramètre, à savoir duree2. Comme on l'avait vu dans le chapitre précédent, on a exceptionnellement le droit d'accéder directement aux attributs de cet objet car on se trouve dans une méthode de la même classe. C'est un peu tordu mais ça nous aide bien (sinon il aurait fallu créer des méthodes "accesseur" comme getHeures()).

Rajouter les secondes, c'est facile. Mais ensuite on doit rajouter un reste si on a dépassé 60 secondes (donc rajouter des minutes). Je ne vous explique pas comment ça fonctionne dans le détail, je vous laisse vous remuer les méninges un peu, ce n'est vraiment pas bien difficile (c'est du niveau des tous premiers chapitres du cours 😊). Vous noterez que c'est un cas où l'opérateur modulo (%), à savoir le reste de la division, est très utile.

Bref, on fait de même avec les minutes, et quant aux heures c'est encore plus facile vu qu'il n'y a pas de reste (on peut dépasser les 24 heures donc pas de problème).

Quelques tests

Pour mes tests, j'ai dû rajouter une méthode afficher() à la classe Duree (elle fait un cout de la durée tout bêtement).

Voilà mon bôôô main 😊 :

Code : C++ - [Sélectionner](#)

```

#include <iostream>
#include "Duree.h"

using namespace std;

int main()
{
    Duree duree1(0, 10, 28), duree2(0, 15, 2);
    Duree resultat;

    duree1.afficher();
    cout << "+" << endl;
    duree2.afficher();

    resultat = duree1 + duree2;

    cout << "=" << endl;
    resultat.afficher();

    return 0;
}

```

Et le tant attendu résultat à l'écran :

Code : Console - [Sélectionner](#)

```

0h10m28s
+
0h15m2s
=
0h25m30s

```

Cool, ça marche 😊

Bon mais ça c'était trop facile, il n'y avait pas de reste dans mon calcul. Corsons un peu les choses avec d'autres valeurs :

Code : Console - [Sélectionner](#)

```

1h45m50s
+
1h15m50s
=
3h1m40s

```

Yeahhh ! Ca marche ! (et du premier coup pour moi nananère 🥰)

J'ai bien entendu testé d'autres valeurs pour être bien sûr que ça fonctionnait, mais de toute évidence ça marche très bien et mon algo est donc bon 😊

Bon, on en viendrait presque à oublier l'essentiel dans tout ça. Tout ce qu'on a fait là, c'était pour pouvoir écrire cette ligne :

Code : C++ - [Sélectionner](#)

```
resultat = duree1 + duree2;
```

La surcharge de l'opérateur + nous a permis de rendre notre code clair, simple et lisible, alors qu'on aurait dû utiliser une méthode en temps normal.

Télécharger le projet

Pour ceux d'entre vous qui n'auraient pas bien suivi la procédure, ou qui sont tout simplement fainéants (XD), je vous propose de télécharger le projet contenant :

- main.cpp
- Duree.cpp
- Duree.h
- Ainsi que le fichier .cbp de Code::Blocks (si vous utilisez cet IDE comme moi)

[Télécharger le projet \(2 Ko\)](#)

Bonus track #1

Ce qui est vraiment sympa dans tout ça, c'est que tel que notre système est fait, on peut très bien additionner plusieurs durées en même temps sans aucun problème.

Par exemple, je rajoute juste une troisième durée dans mon main et je l'additionne avec les autres :

Code : C++ - [Sélectionner](#)

```
int main()
{
    Duree duree1(1, 45, 50), duree2(1, 15, 50), duree3 (0, 8, 20);
    Duree resultat;

    duree1.afficher();
    cout << "+" << endl;
    duree2.afficher();
    cout << "+" << endl;
    duree3.afficher();

    resultat = duree1 + duree2 + duree3;

    cout << "=" << endl;
    resultat.afficher();

    return 0;
}
```

Code : Console - [Sélectionner](#)

```
1h45m50s
+
1h15m50s
+
0h8m20s
=
3h10m0s
```

C'est cool non vous trouvez pas ?
En fait, la ligne-clé :

Code : C++ - Sélectionner

```
resultat = duree1 + duree2 + duree3;
```

... revient à écrire :

Code : C++ - Sélectionner

```
resultat = duree1.operator+(duree2.operator+(duree3));
```

Le tout s'imbrique dans une logique implacable et vient se placer finalement dans l'objet resultat 😊

Notez que le C++ ne vous permet pas de changer la priorité des opérateurs.

Vous voyez ici un exemple un peu plus mathématique (mais pas vraiment compliqué) de l'intérêt de la POO. En C, la même chose était faisable, mais on aurait mélangé les heures, minutes et secondes de chacun. Ici, tout est regroupé et chaque "Duree" est suffisamment intelligente pour être capable de s'additionner avec d'autres durées (et de faire bien d'autres choses encore !).

Bonus track #2

Et pour notre seconde bonus track, sachez qu'on n'est pas obligé d'additionner des Duree avec des Duree, du temps que ça reste logique et compatible.

Par exemple, on pourrait très bien additionner une Duree et un int. On considérerait dans ce cas que le nombre int est un nombre de secondes à ajouter.

Cela nous permettra d'écrire par exemple :

Code : C++ - Sélectionner

```
resultat = duree + 30;
```

Vive la surcharge des fonctions !

Code : C++ - Sélectionner

```
Duree operator+(const int secondes);
```

... mais vous croyiez tout de même pas que j'allais vous écrire l'implémentation. Allez hop hop hop au boulot ! 😊

Les autres opérateurs arithmétiques

Maintenant que vous avez vu assez en détail le cas d'un opérateur (celui d'addition pour ceux qui ont la mémoire courte 😊), vous allez voir que pour la plupart des autres opérateurs c'est très facile et qu'il n'y a pas de difficulté supplémentaire. Le tout est de s'en servir correctement pour la classe que l'on manipule.

Ces opérateurs sont du même "type" que l'addition. Vous les connaissez déjà :

- La soustraction (-)
- La multiplication (*)
- La division (/)

- Le modulo (%), c'est-à-dire le reste de la division

Pour surcharger ces opérateurs, rien de plus simple : créez une méthode dont le nom commence par operator suivi de l'opérateur en question. Cela donne donc :

- **operator-**
- **operator***
- **operator/**
- **operator%**

Pour notre classe Duree, il peut être intéressant de définir la soustraction (operator-).

Je vous laisse le soin de le faire, en vous basant sur l'addition ça ne devrait pas être trop compliqué 😊

En revanche, les autres opérateurs ne servent a priori à rien : en effet, on ne multiplie pas des durées entre elles, et on les divise encore moins. Comme quoi, tous les opérateurs ne sont pas utiles à toutes les classes : ne définissez donc que ceux qui vous seront vraiment utiles.

Si multiplier une Duree par une Duree n'a pas de sens, en revanche on peut imaginer que l'on multiplie une Duree par un nombre entier. Ainsi, l'opération 2h25m50s * 3 est envisageable. Attention à utiliser le bon prototype, en l'occurrence :

Code : C++ - [Sélectionner](#)

```
Duree operator*(int nombre);
```

Les opérateurs de comparaison (==, >, <, ...)

Ces opérateurs vont vous permettre de comparer des objets entre eux. Le plus utilisé d'entre eux est probablement l'opérateur d'égalité (==) qui permet de vérifier si 2 objets sont égaux. C'est à vous d'écrire le code de la méthode qui détermine si les objets sont identiques, l'ordinateur ne peut pas le deviner pour vous car il ne connaît pas la "logique" de vos objets 😊

Tous ces opérateurs de comparaison ont un point en commun particulier : ils renvoient un booléen (bool) et non un objet comme c'était le cas des autres opérateurs.

L'opérateur ==

On va écrire l'implémentation de l'opérateur d'égalité pour commencer, vous allez voir que c'est très simple :

Code : C++ - [Sélectionner](#)

```
bool Duree::operator==(const Duree &duree)
{
    if (m_heures == duree.m_heures && m_minutes == duree.m_minutes && m_secondes == duree.m_secondes)
        return true;
    else
        return false;
```

3

5

4

6

On compare à chaque fois un attribut de l'objet dans lequel on se trouve avec un attribut de l'objet auquel on se compare (les heures avec les heures, les minutes avec les minutes...). Si ces 3 valeurs sont identiques alors on peut considérer que les objets sont identiques et renvoyer true (vrai).

Dans le main, on peut faire un simple test de comparaison pour vérifier si on a fait les choses correctement :

Code : C++ - [Sélectionner](#)

```

int main()
{
    Duree duree1(0, 10, 28), duree2(0, 10, 28);

    if (duree1 == duree2)
        cout << "Les durees sont identiques";
    else
        cout << "Les durees sont differentes";

    return 0;
}

```

Résultat :

Code : Console - [Sélectionner](#)

Les durees sont identiques

L'opérateur <

Je vous préviens on va pas tous les faire sinon on y est encore demain 😊

Si l'opérateur == peut s'appliquer à la plupart des objets, il n'est pas certain que l'on puisse dire de tous nos objets qui est le plus grand. Tous n'ont pas forcément une notion de grandeur, prenez par exemple notre classe Personnage, il serait je pense assez stupide de vouloir vérifier si un Personnage est "inférieur" à un autre ou non (à moins que vous ne compariez les vies... à vous de voir).

En tout cas avec la classe Duree on a de la chance, il est facile et "logique" de vérifier si une Duree est inférieure à une autre.

Voici mon implémentation pour l'opérateur "est strictement inférieur à" (<) :

Code : C++ - [Sélectionner](#)

```

bool Duree::operator<(const Duree &duree)
{
    if (m_heures < duree.m_heures)
        return true;
    else if (m_heures == duree.m_heures && m_minutes < duree.m_minutes)
        return true;
    else if (m_heures == duree.m_heures && m_minutes == duree.m_minutes && m_secondes < duree.m_secondes)
        return true;
    else
        return false;
}

```

Avec un peu de réflexion on finit par trouver cet algorithme, il suffit d'activer un peu ses méninges 😊

Vous noterez que la méthode renvoie false si les durées sont identiques : c'est normal, car il s'agit de l'opérateur "strictement inférieur à" (<). En revanche, si ça avait été la méthode de l'opérateur "inférieur ou égal à" (<=), il aurait fallu renvoyer true.

Je vous laisse le soin de tester dans le main si ça fonctionne correctement 😊

Les autres opérateurs de comparaison

On ne va pas les implémenter ici, ça surchargerait inutilement. Par contre, je vous invite à essayer de les implémenter pour notre classe Duree, ça fera un bon exercice d'algorithmie. Il reste notamment :

- `operator>`
- `operator<=`

- **operator>=**
 - **operator!=**
-

L'opérateur d'affectation (=)

Un des principaux pièges de ce chapitre vient de l'opérateur "=" que l'on peut lui aussi surcharger. C'est l'opérateur d'affectation, qui permet donc de donner une valeur à un objet.

En fait, le "piège" vient du fait que vous risquez de le confondre avec le constructeur de copie. C'est pourquoi je vais insister plus précisément sur cet opérateur pour que vous voyiez bien la différence et ce à quoi il sert.

Rappel : le constructeur de copie

Quand le constructeur de copie est-il appelé ? Vous vous en souvenez ? Il y a en fait plusieurs cas, les 3 principaux étant :

1/ Lors de l'appel explicite au constructeur de copie

Lorsque vous déclarez un objet et que vous indiquez en paramètre un autre objet, le constructeur de copie est appelé :

Code : C++ - [Sélectionner](#)

```
Objet monObjet;
Objet copieObjet(monObjet); // Appel du constructeur de copie
```

2/ Lors d'une affectation au moment de la déclaration

C'est pareil, sauf qu'on utilise le signe "=" ce qui rend le code plus lisible :

Code : C++ - [Sélectionner](#)

```
Objet monObjet;
Objet copieObjet = monObjet; // Appel du constructeur de copie
```

3/ Lors d'un appel de fonction qui prend un objet en paramètre

Si la fonction prend un objet (et non pas un pointeur ni une référence) en paramètre, l'objet est "copié" spécialement pour la fonction. Il y a appel du constructeur de copie avant le début de la fonction.

Code : C++ - [Sélectionner](#)

```
void maFonction (Objet copieObjet) // Appel du constructeur de copie
{
}
```

Le rapport avec la surcharge de l'opérateur = ?

Si elle existe, la méthode **operator=** sera appelée dès qu'on essaie d'affecter une valeur à notre objet.

Par exemple, si à un moment dans le code on affecte à notre objet la valeur d'un autre objet :

Code : C++ - Sélectionner

```
monObjet = unAutreObjet;
```

Mais attends... C'est pas le constructeur de copie qui sera appelé là ?

Non justement, c'est là qu'est le piège. Dans le cas n°2 vu plus haut, c'est lors de la déclaration de l'objet qu'on fait une affectation. Dans ce cas, c'est le constructeur de copie qui est appelé.

En revanche, dans tout le reste du code, si on affecte une valeur à notre objet, c'est cette fois la méthode operator= qui sera appelée.

Donc en résumé :

- Lors de la déclaration (création) de notre objet, si vous utilisez le signe "=" pour lui affecter immédiatement la valeur d'un autre objet, c'est le constructeur de copie qui est appelé.

Code : C++ - Sélectionner

```
Objet copieObjet = monObjet; // Déclaration de l'objet
```

- Après, à n'importe quel autre moment, si vous décidez d'affecter la valeur d'un autre objet à votre objet, c'est la méthode surchargeant l'opérateur = (operator=) qui sera appelée.

Code : C++ - Sélectionner

```
copieObjet = monObjet; // Affectation APRES la déclaration
```

J'espère avoir été suffisamment clair 😊

Cela signifie donc qu'en général vous devrez écrire le code du constructeur de copie et de l'opérateur "=" en même temps si vous voulez qu'à n'importe quel moment dans votre code on puisse faire une affectation sur votre objet.

Implémentation de la méthode operator= pour la classe Duree

Puisqu'on y est, implémentons la méthode pour la classe Duree et on sera tranquille 😊

Code : C++ - Sélectionner

```
Duree Duree::operator=(const Duree &duree)
{
    m_heures = duree.m_heures;
    m_minutes = duree.m_minutes;
    m_secondes = duree.m_secondes;

    return *this;
}
```

Vous noterez que la méthode renvoie l'objet lui-même. En effet, souvenez-vous, this est un pointeur vers l'objet. Si on écrit *this, c'est donc l'objet lui-même que l'on renvoie. Cela permet de traiter le cas où on chaîne les affectations :

Code : C++ - Sélectionner

```
objet1 = objet2 = objet3;
```

Ce n'est pas très courant mais mieux vaut être prévoyant.

A part ça c'est tout bête. Et dans le même temps, si ce n'est pas fait, je vous conseille d'écrire le constructeur de copie pour que le signe "=" fonctionne dans tous les cas. Le code devrait être quasiment le même.

Le compilateur écrit un opérateur d'affectation par défaut automatiquement, mais c'est un opérateur "bête". Cet opérateur bête se contente de copier les valeurs des attributs un à un dans le nouvel objet.

Je sais ce que vous allez me dire : c'est exactement ce qu'on vient de faire ! En effet, dans notre cas réécrire l'opérateur d'affectation n'était donc pas nécessaire. En revanche, ça l'aurait été si on avait eu par exemple des pointeurs et qu'il avait fallu faire des allocations de mémoire, afin d'éviter les problèmes expliqués dans le chapitre précédent.

Les opérateurs de flux (<<, >>)

Parmi les nombreuses choses qui ont dû vous choquer quand vous avez commencé le C++, dans la catégorie "ouah c'est bizarre ça mais on verra plus tard", il y a les flux d'entrée-sortie. Derrière ce nom barbare se cachent ces petits symboles >> et <<.

Quand les utilise-t-on ? Allons allons, vous n'allez pas me faire croire que vous avez la mémoire si courte 😊

Code : C++ - [Sélectionner](#)

```
cout << "Coucou !";
cin >> variable;
```

Figurez-vous justement que << et >> sont des opérateurs. Le code ci-dessus revient donc à écrire :

Code : C++ - [Sélectionner](#)

```
cout.operator<<("Coucou !");
cin.operator>>(variable);
```

On a donc fait appel aux méthodes operator<< et operator>> des objets cout et cin ! 😊

Définir ses propres flux pour cout

Nous allons ici nous intéresser plus particulièrement à l'opérateur << utilisé avec cout.

Les opérateurs de flux sont définis par défaut pour les types de variables int, double, char*, ainsi que pour les objets comme string. C'est ainsi que l'on peut aussi bien écrire :

Code : C++ - [Sélectionner](#)

```
cout << "Coucou !";
```

... que :

Code : C++ - [Sélectionner](#)

```
cout << 15;
```

(et c'est là qu'on dit "merci la surcharge des méthodes !" 😊)

Bon, le problème c'est que cout ne connaît pas votre classe flambant neuve Duree, et donc qu'il ne possède pas de méthode surchargée pour les objets de ce type. On ne peut donc pas écrire :

Code : C++ - [Sélectionner](#)

```
Duree chrono(0, 2, 30);
cout << chrono; // Erreur : il n'existe pas de méthode cout.operator<<(Duree &duree)
```

Qu'à cela ne tienne, nous allons écrire cette méthode !

Quoi ?! Mais on ne peut pas modifier le code de cout non ?

Déjà si vous vous êtes posé la question, bravo, c'est que vous commencez à bien vous repérer. En effet, c'est une méthode de la classe ostream (dont l'objet cout est une instance) que l'on doit définir, et on n'a pas accès au code correspondant.

Lorsque vous incluez <iostream>, un objet cout est automatiquement déclaré comme ceci :

Code : C++ - [Sélectionner](#)

```
ostream cout;
```

ostream est la classe, cout est l'objet. C'est donc la classe ostream qu'il faudrait théoriquement retoucher pour pouvoir créer une nouvelle surcharge de l'opérateur <<... mais on n'a pas le droit car on n'a pas accès au code définissant la classe ostream !

Par contre, il est possible de créer de simples fonctions (en dehors des objets) pour surcharger des opérateurs. C'est un peu particulier je le reconnaît, mais on n'a pas le choix dans le cas présent.

Implémentation d'operator<< en tant que fonction

C'est donc une surcharge d'opérateur un peu particulière que nous allons faire : nous allons écrire une fonction, en dehors de toute classe donc, et non pas une méthode.

Comme c'est un cas assez particulier et que vous n'aurez pas à la reproduire tous les jours, je vous recommande de me suivre pas à pas.

Commencez par écrire cette fonction :

Code : C++ - [Sélectionner](#)

```
ostream &operator<<( ostream &out, Duree &duree )
{
    out << duree.m_heures << "h" << duree.m_minutes << "m" << duree.m_secondes << "s"; //
```

return out;

5

6

3

4

Vous devriez la placer avant le main (ou tout du moins son prototype), sinon le main ne la connaîtra pas.

Le premier paramètre (référence sur un objet de type ostream) qui vous sera automatiquement passé est en fait l'objet cout (que l'on appelle ici out dans la fonction pour éviter les conflits de nom). Le second paramètre est une référence vers l'objet de type Duree que vous tentez d'afficher en utilisant le flux <<.

La fonction doit récupérer les attributs qui l'intéressent dans l'objet et les envoyer à l'objet "out" (qui n'est autre que cout). Ensuite, elle retourne cet objet, ce qui permet de pouvoir faire une chaîne :

Code : C++ - [Sélectionner](#)

```
cout << duree1 << duree2;
```

Si je compile ça plante ! Ca me dit que je n'ai pas le droit d'accéder aux attributs de l'objet duree depuis la fonction !

Eh oui c'est parfaitement normal, car on est à l'extérieur de la classe, et les attributs m_heures, m_minutes et m_secondes sont privés. On ne peut donc pas les lire de cet endroit du code.

2 solutions :

- Ou bien vous créez des accesseurs comme on l'a vu (ces fameuses méthodes getHeures, getMinutes...), ça marche bien mais c'est un peu ennuyeux à écrire
- Ou bien vous utilisez la technique que je vais vous montrer 😊

On va opter ici pour la seconde solution 😊

Changez la 1ère ligne de la fonction comme ceci :

Code : C++ - [Sélectionner](#)

```
ostream &operator<<( ostream &out, Duree &duree )
{
    duree.afficher(out) ; // <- Changement ici
    return out;
}
```

Et rajoutez une méthode afficher dans la classe Duree.

Prototype à mettre dans Duree.h :

Code : C++ - [Sélectionner](#)

```
void afficher(std::ostream &out);
```

Implémentation de la méthode dans Duree.cpp :

Code : C++ - [Sélectionner](#)

```
void Duree::afficher(ostream &out)
{
    out << m_heures << "h" << m_minutes << "m" << m_secondes << "s" ;
}
```

On passe donc le relai à une méthode à l'intérieur de la classe, qui, elle, a le droit d'accéder aux attributs. La méthode prend en paramètre la référence vers l'objet out pour pouvoir lui envoyer les valeurs qui nous intéressent. Ce qu'on n'a pas pu faire dans la fonction operator<<, on le donne à faire à une méthode de la classe Duree.

Ouf ! Maintenant dans le main, que du bonheur !

Bon, c'était un peu gymnastique, mais maintenant c'est que du bonheur 😊

Vous allez pouvoir dans votre main afficher vos objets de type Duree très simplement :

Code : C++ - [Sélectionner](#)

```
int main()
{
    Duree duree1(2, 25, 28), duree2(0, 16, 33);

    cout << duree1 << " et " << duree2 << endl;

    return 0;
}
```

Résultat :

Code : Console - [Sélectionner](#)

Enfantin 😊

Comme quoi, on prend un peu de temps pour écrire la classe, mais ensuite quand on doit l'utiliser c'est extrêmement simple !

Si vous avez un peu du mal à vous repérer dans le code, ce que je peux comprendre, je mets à votre disposition le projet complet comme tout à l'heure dans ce zip :

Télécharger les sources (2 Ko)

Il y a énormément d'autres opérateurs surchargeables en C++, en fait presque tout peut être surchargé. Chaque opérateur étant particulier, il serait impossible de tout voir dans ce chapitre. Au moins avons-nous pu voir les principaux 😊

A titre d'information, sachez qu'il est aussi possible de surcharger :

- new et delete : l'allocation dynamique, s'il y a besoin de faire des vérifications spéciales lors d'une allocation de mémoire
- & et * : opérateurs d'indirection et de déréférencement pour manipuler les pointeurs
- (int) et compagnie : opérateurs de transtypage
- ++ et -- : opérateurs d'incrémentation et de décrémentation
- [] : pour parcourir l'objet comme un tableau. Le type string s'en sert d'ailleurs pour que l'on puisse écrire monString[3] et ainsi accéder au 4ème caractère comme si c'était un tableau, alors que c'est en fait un objet. Malin, il fallait y penser !
- etc.

Bref, vous l'aurez compris, la surcharge des opérateurs est un outil puissant, pour ne pas dire très puissant si on commence à s'en servir sur l'allocation dynamique, le transtypage ou encore les opérateurs d'indirection et de déréférencement.

Mon conseil serait : ne faites la surcharge que si elle vous sera vraiment utile. C'est certes un outil puissant, mais il n'est pas nécessaire de le mettre à toutes les sauces. Votre classe doit proposer des fonctionnalités utiles et non pas farfelues !

TP : La POO en pratique avec ZString

Vous avez dû vous en rendre compte au fil des chapitres : la programmation orientée objet n'est pas simple à comprendre. Il faut un temps avant d'arriver à imaginer que l'on manipule des "objets". Les objets sont des sortes de boîtes qui contiennent un ensemble de variables et de fonctions qui modifient ces variables.

On peut voir la POO de 2 côtés :

- **Le côté utilisateur** : cela correspond à utiliser les classes en créant des objets. C'est là que la POO se révèle simple et agréable.
- **Le côté créateur** : cela correspond à créer les classes. C'est le plus délicat car il faut bien réfléchir avant de se lancer à coder.

Nous avons déjà vu la POO côté utilisateur avec l'exemple de la classe string fournie avec la bibliothèque standard du C++. Ce que je vous propose dans ce TP, c'est de voir maintenant le côté créateur en pratique. Nous allons recréer la classe string.

Vous vous demandez peut-être : pourquoi refaire la classe string si elle existe déjà ? Tout simplement parce que c'est un très bon exercice et que ça va beaucoup vous faire progresser.

Comme je sais que la POO n'est pas simple à appréhender quand on débute, je ne vous laisserai pas vous débrouiller tous seuls dans ce TP. Au contraire, je vais vous aider tout au long de la création de notre classe.

Ce qui compte, c'est de lire, comprendre, et essayer de programmer. Si vous n'arrivez pas à programmer comme il faut du premier coup, ce n'est pas grave.

Si vous vous dites "Jamais je n'aurais pu deviner qu'il fallait faire comme ça", ce n'est pas grave non plus. C'est le métier qui rentre. Faites l'effort de comprendre comment j'ai fait, et ce sera déjà très bien 😊

Notre objectif

Notre objectif est de recréer la classe string de la bibliothèque standard du C++. C'est une classe qui gère les chaînes de caractères. Nous allons donc beaucoup manipuler les chaînes de caractères dans ce chapitre.

Il est important que vous soyez au point vis à vis des chaînes de caractères. Si vous ne vous souvenez plus qu'une chaîne de caractères se termine par un \0 de fin de chaîne par exemple, je vous invite à [relire le chapitre sur les chaînes de caractères](#) vu dans le cours de C.

C'est important, j'insiste. Prenez le temps de revoir ce chapitre si vous en avez besoin.

Vous avez déjà appris à [utiliser la classe "string"](#) dans un chapitre précédent de ce cours. Vous avez vu à quel point c'était simple : c'est la classe qui gère tout pour nous. Plus besoin de créer un tableau de la bonne taille, c'est la classe qui s'en occupe. Si la taille de la chaîne change, le tableau de caractères est automatiquement réalloué par la classe.

Code : C++ - [Sélectionner](#)

```
string maChaine = "Bonjour"; // Crée un tableau de caractères de 8 cases (\0 compris)
maChaine = "Bonjour Mateo"; // Change automatiquement la taille du tableau : 14 cases
```

Un objet string n'est au final rien d'autre qu'un objet qui contient un tableau de char (pour stocker la chaîne de caractères). La particularité c'est que c'est la classe qui gère la taille de ce tableau, l'utilisateur n'a pas à s'en soucier.

Quand vous modifiez le contenu de la chaîne, le tableau de char que l'objet maChaine contient est réalloué pour s'adapter à la nouvelle taille. Schématiquement il se passe donc ceci :



D'autre part, on bénéficie des outils puissants du C++ comme la surcharge des opérateurs. Cela nous permet d'écrire des choses intuitives comme :

Code : C++ - [Sélectionner](#)

```
string message = "Bonjour";
string maChaine = message + " Mateo"; // Vaudra "Bonjour Mateo"
```

C'est ce genre de choses que je veux que l'on arrive à refaire. On va y aller méthodiquement en commençant par écrire les constructeurs, le destructeur, puis on rajoutera des fonctionnalités à la classe en créant de nouvelles méthodes (comme une méthode pour connaître la longueur de la chaîne actuelle). On verra enfin la surcharge des opérateurs en dernier.

Quelques préparatifs

Bon assez bavardé, on a du pain sur la planche pour arriver à faire tout ça.

Choisir un nom

Il va falloir commencer par donner un nom à notre classe qui imite "string". On ne peut pas l'appeler "string" puisqu'il existe déjà une classe de ce nom dans la bibliothèque standard.

Je vous propose de l'appeler ZString, pour "Zéro String" 😊

Par convention, la plupart des programmeurs mettent au moins la première lettre du nom de leurs classes en majuscules. C'est ce que je fais ici. Bon j'ai mis aussi la seconde lettre pour faire joli, j'avoue.

La classe "string" de la bibliothèque standard est un mauvais exemple à ne pas suivre 🍷

Créer un nouveau projet

Pour faire ce TP, vous allez devoir créer un nouveau projet. Utilisez l'IDE que vous voulez, moi pour ma part vous savez que j'utilise Code::Blocks 😊

Demandez à créer un nouveau projet console C++.

Ce projet sera constitué de 3 fichiers que vous pouvez déjà créer :

- main.cpp : ce fichier contiendra uniquement la fonction main. Dans la fonction main, nous créerons des objets basés sur notre classe ZString pour tester son fonctionnement. **C'est le côté utilisateur**.
- ZString.h : ce fichier contiendra la définition de notre classe ZString avec la liste de ses attributs et les prototypes de ses méthodes. **C'est une partie du côté créateur**.
- ZString.cpp : ce fichier contiendra l'implémentation des méthodes de la classe ZString, c'est-à-dire le "code" à l'intérieur des méthodes. **C'est l'autre partie du côté créateur**.

Faites attention aux noms des fichiers et en particulier aux majuscules et minuscules. Les fichiers ZString.h et ZString.cpp commencent par 2 lettres majuscules, si vous écrivez "zstring" ou encore "Zstring" ça ne marchera pas et vous aurez des problèmes.

Le code de base de chaque fichier

Nous allons écrire un peu de code dans chacun de ces fichiers. Juste le strict minimum pour pouvoir commencer.

main.cpp

Ce fichier va contenir la fonction main, ainsi que les includes de iostream (pour faire des cout) et de ZString.h (pour pouvoir utiliser la classe ZString !).

Code : C++ - [Sélectionner](#)

```
#include <iostream>
#include "ZString.h"

using namespace std;

int main()
{
    ZString chaine; // Crée un objet de type ZString (appel du constructeur par défaut)
    return 0;
}
```

Comme vous pouvez le voir, le main se contentera dans un premier temps de créer un objet de type ZString appelé "chaine". Les objets commencent par une lettre minuscule par convention.

Ce code provoquera l'appel du constructeur par défaut de la classe ZString. Le constructeur est la méthode qui est appelée à chaque fois que l'on crée un nouvel objet, et là on parle de constructeur par défaut car on n'envoie aucun paramètre.

Le main est court mais on le complètera par la suite pour tester notre classe au fur et à mesure qu'on lui rajoutera des possibilités.

ZString.h

Ce fichier contiendra la définition de la classe ZString. Il fait aussi un include de iostream pour nos besoins futurs (nous aurons besoin de faire des cout dans la classe les premiers temps, ne serait-ce que pour debugger notre classe).

Code : C++ - [Sélectionner](#)

```
#ifndef DEF_ZSTRING
#define DEF_ZSTRING

#include <iostream>

class ZString
{
public:

private:

};

#endif
```

Vous noterez que je n'ai pas oublié de faire un #ifndef pour vérifier que le header n'a pas été inclus plusieurs fois. C'est une [technique de protection](#) que nous avons vue dans le cours de C et que je vous recommande d'utiliser dans chacun de vos headers.

La classe ZString est pour l'instant vide. Je l'ai séparée en deux : la partie publique et la partie privée.

La partie publique est accessible de l'extérieur de la classe (par l'utilisateur) et la partie privée n'est accessible qu'à l'intérieur de la classe elle-même.

Je vous rappelle que la règle d'or est que tous les attributs d'une classe doivent être privés. C'est le principe d'encapsulation. Les méthodes, elles, peuvent être soit publiques soit privées selon les cas (elles sont souvent publiques, mais il arrive qu'on ait besoin de créer des méthodes privées).

ZString.cpp

Ce fichier doit contenir l'implémentation des méthodes de la classe. Pour l'instant, nous n'avons écrit aucune méthode, mais nous allons au moins faire un include de ZString.h, c'est le strict minimum 😊

Code : C++ - [Sélectionner](#)

```
#include "ZString.h"
```

C'est tout !

De quels attributs notre classe a-t-elle besoin ?

Comme vous le savez, une classe est constituée d'attributs et de méthodes. Les attributs sont des variables. Les méthodes interagissent sur ces variables.

De quels attributs notre classe ZString doit-elle être constituée, vous en avez pas une petite idée hmm ?

Réfléchissez, le but de notre classe est de gérer de manière intelligente une chaîne de caractères. Or, vous savez qu'une chaîne de caractères se présente en mémoire sous la forme d'un tableau de char, terminé par un \0 qui signifie "fin de chaîne" (j'espère que vous savez tout ça, sinon il est grand temps d'aller relire le [chapitre sur les chaînes de caractères](#)!).

Nous aurons donc besoin au moins d'un tableau de char en attribut.

En plus de cela, il me semble nécessaire de mettre la taille de la chaîne de caractères (un int) en attribut aussi. Vous me direz : on peut toujours la recalculer (il suffit de compter le nombre de caractères jusqu'à l'\0), mais je pense que c'est une bonne idée de garder la taille de la chaîne en mémoire pour éviter d'avoir à la recalculer à chaque fois.

Nous allons donc modifier notre ZString.h pour y ajouter ces 2 attributs :

Code : C++ - Sélectionner

```
#ifndef DEF_ZSTRING
#define DEF_ZSTRING

#include <iostream>

class ZString
{
public:

private:
char *m_chaine; // Tableau de caractères (contientra la chaîne)
int m_longueur; // Longueur de la chaîne
};

#endif
```

Nos attributs commencent toujours par le préfixe "m_". C'est une bonne habitude de programmation que je vous ai enseignée dans les chapitres précédents 😊

Cela nous permettra par la suite de savoir si on est en train de manipuler un attribut de la classe ou une simple variable "locale" à une méthode.

Hé ! Tu avais dit qu'il fallait créer un tableau de char pour gérer la chaîne ! Or là je ne vois qu'un pointeur de char, pourquoi as-tu fait ça ?

J'attendais une question de ce genre 😊

Je vais vous répondre par une autre question : quelle taille vous donneriez à ce tableau de char vu que vous ne connaissez pas la taille de la chaîne à stocker ?

Vous pourriez certes me dire "Bah il suffit de créer un très grand tableau de char, par exemple m_chaine[10000]". Mais ce serait mauvais. Non, ce serait même carrément nul :

- Rien ne vous dit que personne ne dépassera jamais les 10 000 caractères.
- Ca fait beaucoup de mémoire inutilisée pour rien.
- Notre but est justement d'allouer un tableau en mémoire qui fasse pile la taille nécessaire.

Donc comme on ne sait pas la taille que fera le tableau dans la suite du programme, on crée juste un pointeur sur char. C'est nous qui allouerons la taille nécessaire par la suite, dans le constructeur (c'est son rôle, initialiser les attributs).

D'ailleurs en parlant de constructeur, je crois qu'il est temps de s'en occuper maintenant que nous nous sommes mis d'accord sur les attributs que la classe allait manipuler 😊

Constructeurs et destructeur

Nous allons commencer par écrire les méthodes les plus importantes d'une classe : les constructeurs et le destructeur.

J'ai bien dit LES constructeurs, car on peut surcharger le constructeur (en faire plusieurs versions), et LE destructeur, car celui-ci ne peut pas être surchargé.

Je vous propose de créer 3 constructeurs et le destructeur pour commencer :

- Le constructeur par défaut (celui qui ne prend pas de paramètre). Si l'utilisateur se sert de ce constructeur, la chaîne sera vide : "".
- Un autre constructeur (une surcharge) qui prendra en paramètre une chaîne de caractères pour initialiser la ZString avec une chaîne. La ZString contiendra donc dès le départ la chaîne qu'on lui aura envoyée.

Ce constructeur recevra en paramètre un tableau de char (un char *) correspondant à la chaîne envoyée par l'utilisateur pour

initialiser la ZString.

- Le constructeur de copie : quelle que soit la classe qu'on écrit, il est toujours conseillé d'écrire le constructeur de copie car il est souvent nécessaire. C'est un constructeur qui prend une référence vers un objet du même type (un const ZString &).
- Le destructeur pour supprimer le tableau de char m_chaine avant que l'objet ne soit lui-même supprimé. Cela permet d'éviter les fuites de mémoire.

On créera d'autres constructeurs par la suite, mais pour l'instant nous commençons simplement 😊

Commençons par ajouter les prototypes de nos méthodes dans ZString.h :

Code : C++ - [Sélectionner](#)

```
#ifndef DEF_ZSTRING
#define DEF_ZSTRING

#include <iostream>

class ZString
{
public:
    ZString(); // Constructeur par défaut (crée une chaîne vide "")
    ZString(const char *chaine); // Constructeur surchargé (crée la chaîne envoyée)
    ZString(const ZString &chaine); // Constructeur de copie
    ~ZString(); // Destructeur (détruit le tableau de char pour libérer la mémoire)

private:
    char *m_chaine;
    int m_longueur;
};

#endif
```

Bien, voilà qui est fait.

Il faut maintenant implémenter ces méthodes, rendez-vous dans le fichier ZString.cpp.

Le constructeur par défaut ZString()

On commence par implémenter le constructeur par défaut. Je vous rappelle que le but d'un constructeur est d'initialiser les attributs de la classe. La question est : quelle valeur on va leur mettre ? 🤔

Comme on travaille sur le constructeur par défaut, vous pouvez voir que celui-ci ne prend pas de paramètre. C'est le constructeur qui est appelé lorsqu'on crée un nouvel objet de type ZString sans préciser de paramètre.

C'est précisément ce que l'on a fait dans le main.cpp que je vous ai donné plus haut :

Code : C++ - [Sélectionner](#)

```
ZString chaine; // Appel du constructeur par défaut (aucun paramètre envoyé)
```

Que doit contenir la chaîne lorsqu'on n'envoie rien ?

Bah... rien 😐

Si l'utilisateur n'envoie aucun texte, nous n'allons rien mettre dans l'attribut m_chaine. Il est donc inutile d'allouer un tableau de char (y'a rien à stocker !).

Ce qu'on va faire en revanche, c'est mettre le pointeur m_chaine à NULL pour indiquer qu'il ne pointe sur rien pour le moment. Quant à la longueur de la chaîne m_longueur, bah elle vaudra 0 vu que pour l'instant notre objet ne contiendra aucune chaîne en mémoire 😐

On peut donc écrire dans ZString.cpp :

Code : C++ - [Sélectionner](#)

```
zString::zString()
{
    m_chaine = NULL;
    m_longueur = 0;
}
```

Notez que, comme je vous l'avais expliqué, on peut aussi initialiser les attributs avec une liste d'initialisation, comme ceci :

Code : C++ - [Sélectionner](#)

```
zString::zString() : m_chaine(NULL), m_longueur(0)
{
```

Ce code revient au même que celui que je vous ai donné plus haut. Le seul problème de la liste d'initialisation est qu'elle ne convient pas dans tous les cas, comme on le verra plus loin.

Le constructeur ZString(const char *)

Le constructeur par défaut était simple.

Les choses se corsent quand l'utilisateur envoie un paramètre lorsqu'il crée la chaîne dans main.cpp :

Code : C++ - [Sélectionner](#)

```
zString chaine("Bonjour");
```

... ou encore (ça revient au même) :

Code : C++ - [Sélectionner](#)

```
zString chaine = "Bonjour";
```

Lorsqu'un objet est créé de cette façon, cela appelle automatiquement le constructeur qui correspond à la signature ZString(const char *) car le fait d'écrire un texte entre guillemets dans le code source provoque la création d'un tableau de char par le compilateur.

Il va falloir écrire le code de ce constructeur dans ZString.cpp...

Mais là les choses se corsent, suivez-moi bien.

Notre but est d'initialiser nos attributs m_chaine et m_longueur correctement, on est bien d'accord ? C'est le but du constructeur d'initialiser des attributs.

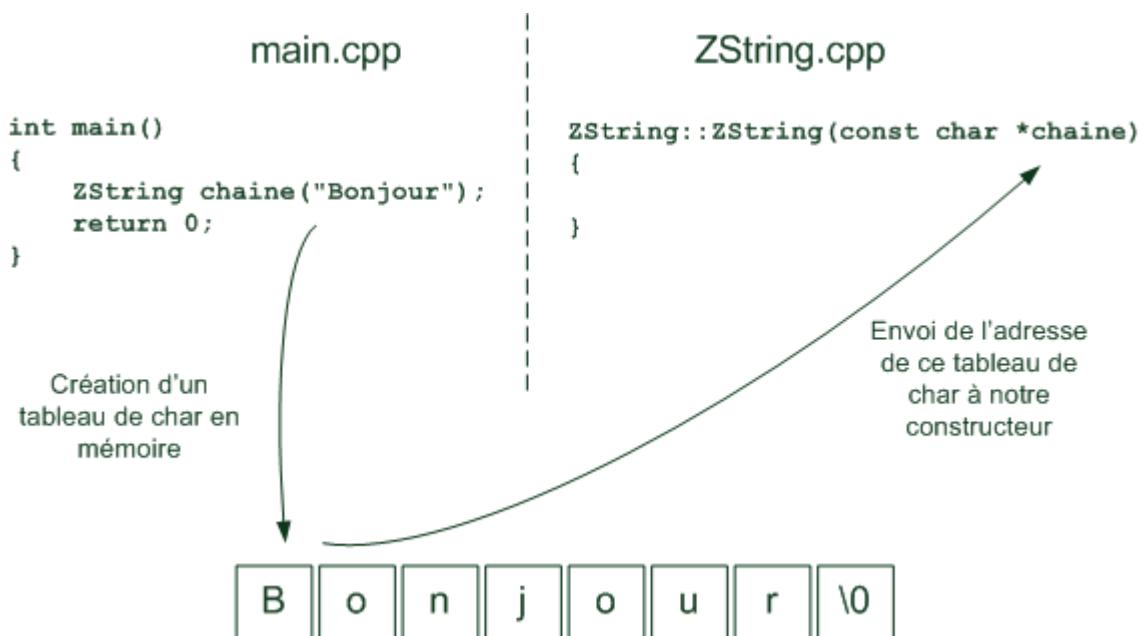
Le problème c'est que :

- Pour **m_longueur** : on ne connaît pas la taille de la chaîne qu'on nous envoie ! Impossible d'initialiser m_longueur si on ne connaît pas la taille de la chaîne. On pourrait utiliser la fonction strlen de la bibliothèque C, mais notre but est que notre classe ZString soit autonome et qu'elle n'ait pas besoin de la bibliothèque du C (on est en C++ que diable !). Solution : il va falloir réécrire la fonction strlen() pour pouvoir calculer la longueur de la chaîne.
- Pour **m_chaine** : on nous envoie un tableau de char (appelé chaine), mais il ne faut surtout pas écrire m_chaine = chaine ; ! Pourquoi ? Parce que en faisant cela, vous faites pointer notre attribut m_chaine vers un tableau qui nous a été envoyé par l'utilisateur. Qu'est-ce qui vous dit que l'utilisateur ne va pas supprimer ce tableau par la suite ? Dans un tel cas, votre pointeur m_chaine pointerait sur un tableau qui n'existe plus ! Solution : copier le tableau qu'on nous envoie et affecter m_chaine à ce tableau pour s'assurer que personne d'autre ne pourra supprimer ce tableau.

Pour le problème de l'initialisation de m_longueur je pense que vous avez compris : on ne connaît pas la longueur de la chaîne et il va nous falloir écrire une fonction qui la calcule manuellement en comptant le nombre de caractères.

Par contre, je pense que le problème de l'initialisation de m_chaine mérite plus d'explications (et même un schéma en fait).

Tout d'abord, il faut savoir que lorsqu'on envoie au constructeur une chaîne de caractères entre guillemets, un tableau de char est automatiquement créé en mémoire. Celui-ci est ensuite passé en paramètre au constructeur :



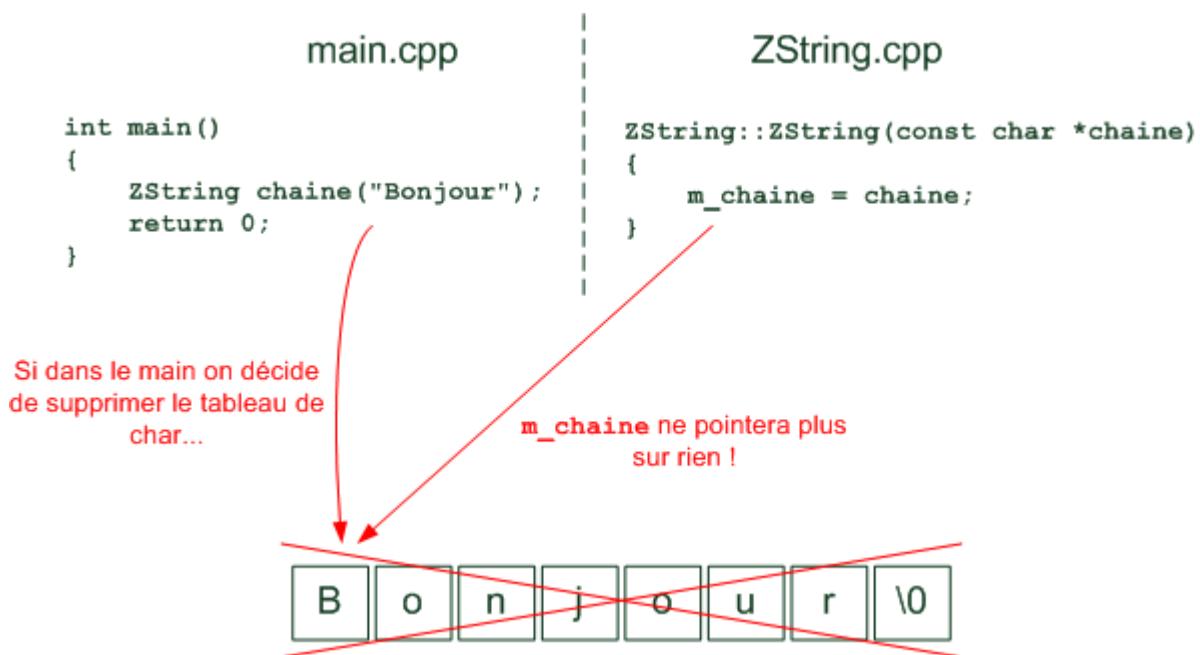
L'erreur qu'on serait tenté de faire, c'est d'assigner l'attribut m_chaine directement au tableau chaine qu'on nous envoie, avec un code comme ceci :

Code : C++ - [Sélectionner](#)

```
ZString::ZString(const char *chaine)
{
    m_chaine = chaine; // Très mauvaise idée !
}
```

Pourquoi ? Parce qu'en faisant pointer notre attribut m_chaine vers le tableau de char qu'on nous a envoyé, on prend le risque que ce tableau de char soit supprimé par le main !

Dans ce cas, si le tableau est supprimé par le main, notre attribut m_chaine ne pointera plus sur rien et on perdra la chaîne !



La solution ?

Comme on l'a vu dans un des chapitres précédents, il faut copier la chaîne (en appelant une fonction de copie que l'on écrira) et faire pointer m_chaine vers cette copie.

Code : C++ - Sélectionner

```
ZString::ZString(const char *chaine)
{
    m_chaine = copie(chaine); // Bonne idée : copier la chaîne pour en avoir une version propre
}
```

3

5

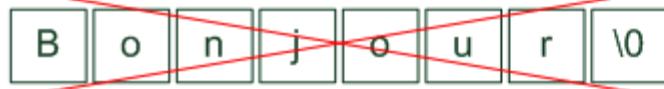
6

4

main.cpp

```
int main()
{
    ZString chaine("Bonjour");
    return 0;
}
```

Le main peut maintenant supprimer le tableau de char à tout moment...



ZString.cpp

```
ZString::ZString(const char *chaine)
{
    m_chaine = copie(chaine);
}
```

... notre classe s'en moque, elle a copié la chaîne pour en avoir une bien à elle !



Comme notre classe sera la seule à connaître la copie, elle sera sûre que personne d'autre ne la supprimera dans le programme !

Si j'insiste pour faire une copie du tableau, ce n'est pas pour rien. Il faut vraiment être sûr de travailler sur une version du tableau que nous sommes les seuls à connaître dans la classe, car sinon on prend le risque que quelqu'un d'autre la supprime sans notre autorisation.

Je vous propose d'écrire ce constructeur :

Code : C++ - Sélectionner

```
ZString::ZString(const char *chaine)
{
    m_chaine = copie(chaine);
    m_longueur = longueur(chaine);
}
```

Pour que ce constructeur marche, il nous faut écrire 2 fonctions :

- copie : qui copie un tableau de char et renvoie un pointeur vers la copie (équivalent de strcpy du C).
- longueur : qui calcule la longueur du tableau de char qu'on lui envoie (équivalent de strlen du C).

Ce sont des fonctions que vous avez déjà peut-être écrites si vous avez suivi mon cours de C. C'est un bon exercice que d'essayer de les réécrire.

Je vous donne la solution, sans l'expliquer, parce que ça ne devrait pas être nouveau pour vous (ou alors faut revoir votre cours de C sur les chaînes de caractères !) :

Code : C++ - Sélectionner

```
int ZString::longueur(const char *chaine)
{
    int i = 0;

    while (chaine[i] != '\0')
    {
        i++;
    }

    return i;
}

char *ZString::copie(const char *chaine)
{
    int taille = longueur(chaine);
    char *chaineCopie = new char[taille + 1]; // +1 pour stocker \0

    for (int i = 0 ; i < taille ; i++)
    {
        chaineCopie[i] = chaine[i];
    }
    chaineCopie[taille] = '\0';

    return chaineCopie;
}
```

Pensez à ajouter le prototype de ces méthodes dans ZString.h

Le constructeur de copie ZString(const ZString &)

Le constructeur de copie est un constructeur très utile qui est appelé dans plusieurs cas par le compilateur. Je ne reviens pas sur ces cas mais je vous invite en revanche à relire la [partie sur le constructeur de copie](#) dans les chapitres précédents.

Le constructeur de copie est un constructeur qui prend en paramètre une référence vers un autre objet du même type.
Voici le constructeur de copie de notre classe ZString :

Code : C++ - Sélectionner

```
ZString::ZString(const ZString &chaine)
{
    m_chaine = copie(chaine.m_chaine);
    m_longueur = chaine.m_longueur;
}
```

Ce constructeur est à peu de choses près identique au constructeur qu'on vient d'écrire il y a 2 minutes.

La seule différence est qu'il prend en entrée une ZString appelée chaine. Pour récupérer le tableau de char de la ZString, il suffit d'écrire chaine.m_chaine. Cela nous permet d'envoyer le tableau de char que les méthodes copie et longueur attendent.

Vous vous demandez peut-être pourquoi on n'a pas tout simplement écrit par exemple :

m_chaine = copie(chaine);

La réponse est simple. Dans ce constructeur :

- chaine est de type ZString (regardez le paramètre d'entrée)
- chaine.m_chaine est de type char *

Or nos méthodes copie et longueur attendent un char *, voilà pourquoi il faut dans ce cas envoyer chaine.m_chaine.

Comment peut-on avoir le droit d'écrire chaine.m_chaine ? Je croyais que m_chaine était un attribut privé, et donc qu'on ne pouvait pas y accéder ?

Il n'aurait pas fallu créer une méthode accesseur getChaine() plutôt à la place ?

En effet, on aurait très bien pu créer une méthode accesseur getChaine(). Faites-le si vous voulez d'ailleurs.

Normalement, on n'a pas le droit d'accéder aux membres privés d'une classe. Mais là nous sommes dans une exception, car nous travaillons dans la même classe (nous sommes dans la classe ZString et nous essayons d'accéder à un attribut privé d'un autre objet de type ZString, ce qui est autorisé).

Le destructeur ~ZString()

On arrive maintenant au destructeur. Son rôle est de détruire les attributs alloués dynamiquement en mémoire avant que l'objet ne soit supprimé (je vous rappelle que le destructeur est automatiquement appelé lorsqu'un objet va être supprimé).

Le seul attribut alloué dynamiquement (avec un new[]), c'est m_chaine. Il faut penser à le supprimer avec un delete[].

Notre destructeur sera tout simple :

Code : C++ - [Sélectionner](#)

```
zString::~ZString()
{
    delete[] m_chaine;
}
```

Si on ne fait pas ça, le tableau de char m_chaine persistera en mémoire après la suppression de l'objet. Du coup, des tableaux "perdus" risqueraient de se ballader en mémoire et on assisterait à ce qu'on appelle des "fuites de mémoire". Votre programme prendrait beaucoup de place en mémoire parce qu'il aurait oublié de supprimer la mémoire dont il n'a plus besoin !

Tester le code

Il est grand temps de compiler pour vérifier qu'on n'a pas fait d'erreur. Pour le moment, on va lancer le main que je vous ai donné au tout début, ce qui va provoquer l'appel du constructeur par défaut :

Code : C++ - [Sélectionner](#)

```
int main()
{
    ZString chaine;
    return 0;
}
```

Compilez, lancez. La console n'affichera rien (c'est normal, tout se passe dans la mémoire) mais si vous n'avez pas de plantage c'est que c'est bon signe déjà 😊

Testons le constructeur qui prend en paramètre un tableau de char pour initialiser la chaîne (celui qu'on a eu tant de mal à écrire, ne me dites pas que vous l'avez déjà oublié 😞) :

Code : C++ - [Sélectionner](#)

```
int main()
{
    ZString chaine( "Bonjour" );
    return 0;
}
```

Compilez, lancez. Toujours pas d'erreur ? C'est très bien, c'est qu'on est sur la bonne voie 😊

Hé ! J'ai essayé de faire un cout de ma chaîne et ça ne marche pas ! Pourquoi ?

Supposons que vous essayiez le code suivant :

Code : C++ - [Sélectionner](#)

```
int main()
{
    ZString chaine( "Bonjour" );
    cout << chaine;

    return 0;
}
```

Le compilateur vous répondra qu'il ne peut pas exécuter le cout car cout ne sait pas lire les objets de type ZString (pour lui c'est comme une boîte noire, il ne sait pas ce qu'il y a à l'intérieur). Il va falloir le lui apprendre en surchargeant l'opérateur << comme on l'a appris dans le chapitre sur la surcharge des opérateurs.

On verra ça un peu plus loin.

Ok, mais en attendant comment je fais pour afficher ce que contient ma chaîne de type ZString ?

Comme surcharger l'opérateur << est un peu délicat et compliqué, on ne le verra que plus loin.

En attendant par contre, vous pouvez écrire une méthode afficher() dans la classe ZString qui affichera la chaîne :

Code : C++ - [Sélectionner](#)

```
void ZString::afficher()
{
    cout << m_chaine << endl;
}
```

Tout ce que la méthode afficher() fait, c'est afficher la chaîne de caractères qu'elle stocke. Ca consiste à faire un cout de m_chaine. C'est tout bête, mais si vous ne le dites pas à l'ordinateur il ne pourra pas deviner 🤪

Dans le main, vous pouvez maintenant afficher votre chaîne !

Code : C++ - [Sélectionner](#)

```
int main()
{
    ZString chaine( "Bonjour" );
    chaine.afficher();

    return 0;
}
```

Résultat :

Code : Console - [Sélectionner](#)

Bonjour

La surcharge des opérateurs

Nous avons écrit des constructeurs, un destructeur et une méthode afficher().

Avec ça, nous pouvons créer des chaînes de type ZString et les afficher. Cool. Mais c'est pas encore bien passionnant.

Les choses vont commencer à devenir intéressantes à partir de maintenant. Nous allons faire quelques surcharges d'opérateurs pour profiter de toute la puissance du C++.

Nous allons surcharger les opérateurs suivants :

- L'opérateur = : c'est l'opérateur d'affectation qui permet d'affecter une nouvelle valeur à la chaîne après sa création.
- L'opérateur + : on va s'en servir pour combiner 2 chaînes de caractères (très pratique).
- L'opérateur << : nous allons surcharger l'opérateur << pour que cout soit capable d'afficher des ZString.

Au boulot !

Surcharger l'opérateur =

Si on souhaite changer la chaîne après la création de la ZString, il faut surcharger l'opérateur =.

Je vous propose de surcharger l'opérateur = 2 fois :

- Une fois pour prendre en paramètre un tableau de char (char *).

Code : C++ - [Sélectionner](#)

```
ZString chaine("Bonjour"); // Vaudra "Bonjour"  
chaine = "Salut"; // Vaudra "Salut" : appel de operator=(char *)
```

- Une autre fois pour prendre en paramètre une autre ZString.

Code : C++ - [Sélectionner](#)

```
ZString chaine("Bonjour"); // Vaudra "Bonjour"  
ZString autreChaine;  
  
autreChaine = chaine; // Vaudra "Bonjour" : appel de operator=(ZString &)
```

operator=(char *)

Commençons par le cas le plus simple : celui où on nous envoie un char * (un texte entre guillemets par exemple).

Comme vous l'avez appris dans le chapitre sur la surcharge des opérateurs (je n'y reviens pas), le prototype de la méthode devra être le suivant :

Code : C++ - [Sélectionner](#)

```
ZString operator=(const char *chaine);
```

A nous d'implémenter la méthode. Le but de l'opérateur = est de mettre dans l'objet la chaîne qu'on lui envoie en paramètre. Pour des raisons techniques que je ne détaillerai pas ici, il faudra que l'objet se renvoie lui-même à la fin de la méthode (via un return *this;).

Code : C++ - [Sélectionner](#)

```

ZString ZString::operator=(const char *chaine)
{
    delete[] m_chaine;
    m_chaine = copie(chaine);
    m_longueur = longueur(chaine);

    return *this;
}

```

On doit dans un premier temps supprimer le tableau de char m_chaine que contenait notre objet. En effet, nous allons affecter une nouvelle chaîne à notre objet qui va "écraser" l'ancienne. Il faut bien penser à supprimer l'ancienne d'abord, sinon l'ancienne chaîne va persister en mémoire et va consommer de la mémoire pour rien !

Vous noterez qu'à part ça et le return, c'est exactement le même code que le constructeur qu'on a écrit tout à l'heure. Et c'est logique, le but de cette méthode est le même (sauf qu'elle ne s'exécute pas au même moment) : il lui faut copier la chaîne qu'on lui envoie pour en avoir une propre à notre classe, et adapter l'attribut m_longueur pour qu'il indique la bonne longueur de chaîne.

Le return *this, je ne reviens pas dessus, c'est comme ça que tout operator= doit terminer, point barre 😊

On peut maintenant tester ce code dans le main et admirer comme c'est beau quand ça marche 😊

Code : C++ - [Sélectionner](#)

```

int main()
{
    ZString chaine("Bonjour");
    chaine.afficher();

    chaine = "Salut";
    chaine.afficher();

    return 0;
}

```

Code : Console - [Sélectionner](#)

```

Bonjour
Salut

```

[operator=\(const ZString &\)](#)

Nous avons réussi à surcharger l'opérateur = pour qu'il accepte les tableaux de char, maintenant nous allons faire en sorte qu'il accepte aussi les ZString (pour pouvoir affecter une ZString par une autre ZString).

Le code de cette méthode sera quasiment le même, il faut juste s'adapter au fait que l'on reçoit une référence vers une ZString au lieu d'un tableau de char :

Code : C++ - [Sélectionner](#)

```

ZString ZString::operator=(const ZString &chaine)
{
    delete[] m_chaine;
    m_chaine = copie(chaine.m_chaine);
    m_longueur = chaine.m_longueur;

    return *this;
}

```

Voici un main pour tester cet opérateur :

Code : C++ - Sélectionner

```
int main()
{
    ZString chaine( "Bonjour" );
    ZString autreChaine;

    autreChaine = chaine; // Vaudra "Bonjour" : appel de operator=(ZString &)

    // Vérifions que les chaînes soient les mêmes
    chaine.afficher();
    autreChaine.afficher();

    return 0;
}
```

Si tout va bien, les deux affichages devraient produire le même résultat :

Code : Console - Sélectionner

```
Bonjour
Bonjour
```

Parfait 😊

On peut maintenant affecter une ZString avec une autre ZString.

Surcharger l'opérateur +

Passons à la surcharge de l'opérateur +, qui va nous permettre d'assembler 2 chaînes de caractères. Là encore, je pense qu'il serait bien d'écrire 2 versions de cette méthode :

- Une fois pour prendre en paramètre un tableau de char (char *).

Code : C++ - Sélectionner

```
ZString chaine( "Bonjour" ); // Vaudra "Bonjour"
ZString resultat;

resultat = chaine + " Mateo"; // Vaudra "Bonjour Mateo" : appel de operator+(char *)
```

- Une autre fois pour prendre en paramètre une autre ZString.

Code : C++ - Sélectionner

```
ZString chaine( "Bonjour" ), nom( " Mateo" );
ZString resultat;

resultat = chaine + nom; // Vaudra "Bonjour Mateo" : appel de operator+(ZString &)
```

operator+(char *)

Un opérateur + ne doit pas modifier l'objet lui-même mais retourner un résultat correspondant à la somme des objets qu'on additionne (ouf ! 😊).

Cela veut dire qu'il ne faut pas trop se calquer sur l'opérateur = car ça fonctionne différemment.

L'écriture de cette méthode est assez délicate (il faut un peu réfléchir quoi :-°).

Voilà comment je vous propose d'additionner les 2 chaînes :

Code : C++ - [Sélectionner](#)

```
zString ZString::operator+(const char *chaine)
{
    int tailleTotale = m_longueur + longueur(chaine);
    char *sommeChaines = new char[tailleTotale + 1];

    for (int i = 0 ; i < m_longueur ; i++)
    {
        sommeChaines[i] = m_chaine[i];
    }

    for (int i = m_longueur ; i < tailleTotale ; i++)
    {
        sommeChaines[i] = chaine[i - m_longueur];
    }
    sommeChaines[tailleTotale] = '\0';

    zString resultat(sommeChaines);
    delete[] sommeChaines;
    return resultat;
}
```

Ce qu'il faut bien comprendre, c'est qu'on travaille sur 2 chaînes :

- m_chaine : correspondant à la chaîne de l'objet dans lequel on est (ici "Bonjour").
- chaine : qui est la chaîne qu'on ajoute (ici " Mateo").

Vous noterez qu'on se sert ici de l'attribut m_longueur de notre objet pour éviter d'avoir à recalculer la longueur de la chaîne contenue dans notre objet.

Après le reste, ben c'est un algorithme. On crée une chaîne sommeChaines de la taille correspondant à la somme des 2 chaînes, puis on fait une première boucle pour y ajouter m_chaine, et une seconde boucle pour y ajouter chaine.

Enfin, on crée un objet de type ZString (car il faut retourner une ZString impérativement) et on lui envoie la somme des chaînes pour que notre nouvelle ZString contienne "Bonjour Mateo".

Enfin, on n'oublie pas de supprimer le tableau de char sommeChaines qu'on avait alloué dynamiquement et qui ne nous sert plus à rien maintenant.

Pour information, avant d'arriver à faire marcher cette méthode j'y ai passé facilement une bonne heure. Mon programme plantait pour diverses raisons.

Tout ça pour vous dire que j'écris pas le bon algorithme du premier coup, qu'il m'arrive de faire des erreurs et de passer du temps à chercher pourquoi ça plante. Je suis un humain tout comme vous 

operator+(ZString &)

Cette surcharge fonctionne de la même manière mais prend en entrée une autre ZString. Il suffit d'adapter un peu le code, le plus dur ayant déjà été fait.

Et hop !

Code : C++ - [Sélectionner](#)

```

zString zString::operator+(const zString &chaine)
{
    int tailleTotale = m_longueur + chaine.m_longueur;
    char *sommeChaines = new char[tailleTotale + 1];

    for (int i = 0 ; i < m_longueur ; i++)
    {
        sommeChaines[i] = m_chaine[i];
    }

    for (int i = m_longueur ; i < tailleTotale ; i++)
    {
        sommeChaines[i] = chaine.m_chaine[i - m_longueur];
    }
    sommeChaines[tailleTotale] = '\0';

    zString resultat(sommeChaines);
    delete[] sommeChaines;
    return resultat;
}

```

Le principe est le même. On profite du fait que l'élément qu'on nous envoie est une ZString pour utiliser son attribut m_longueur (ce qui nous évite d'avoir à recalculer la longueur de sa chaîne).

Surcharger l'opérateur <<

Nous souhaitons maintenant pouvoir faire des cout sur des ZString.

C'est un peu délicat, car il faut en théorie modifier la classe qui est derrière l'objet cout.

En effet, faire :

Code : C++ - Sélectionner

```

zString chaine;

cout << chaine;

```

... revient à écrire comme vous le savez maintenant :

Code : C++ - Sélectionner

```

zString chaine;

cout.operator<<(chaine);

```

Il faudrait donc surcharger la méthode operator<< de la classe qui gère l'objet cout, à savoir la classe ostream.

Le problème, [comme je vous l'avais déjà expliqué](#), c'est qu'on ne peut pas modifier la classe ostream, on n'y a pas accès. En revanche, on peut tricher en créant une simple fonction (comme en C !) de cette forme-là :

Code : C++ - Sélectionner

```

ostream &operator<<( ostream &out, zString &chaine )
{
    out << chaine.getChaine();
    return out;
}

```

J'ai placé cette fonction dans ZString.cpp. Son prototype est dans ZString.h, mais attention, mettez-le en-dehors de la déclaration de la classe car ce n'est pas une méthode de la classe !

Pourquoi ne pas avoir écrit chaine.m_chaine cette fois ?

Jusqu'ici je pouvais le faire car j'étais à l'intérieur même de la classe, et donc j'avais accès à tous les attributs privés, même s'il s'agissait d'un autre objet de la même classe.

Là, on est dans une fonction qui n'a rien à voir avec la classe ZString. Elle n'a donc pas accès aux attributs. C'est pour cette raison que j'appelle la méthode getChaine() de ZString...

Je sais. On n'avait pas écrit de méthode getChaine() jusqu'ici. Il s'agit juste d'un accesseur : vous devriez être capable de l'écrire en 10s chrono, il fait juste un `return m_chaine;` 😊

On peut maintenant faire des cout d'objets de type ZString dans le main ! 😊

Code : C++ - [Sélectionner](#)

```
int main()
{
    ZString chaine( "Bonjour" );

    cout << chaine;

    return 0;
}
```

Code : Console - [Sélectionner](#)

Bonjour

Joie, bonheur et volupté : ça marche ! 😊

Récapitulatif

Je crois que vu tout ce qu'on a fait jusqu'ici, un petit récapitulatif s'impose.

Je vais vous donner le code source de chacun des 3 fichiers (main.cpp, ZString.cpp et ZString.h) puis je vous proposerai de télécharger le projet en l'état actuel.

Nous finirons ensuite ce TP par une liste de suggestions d'améliorations de la classe ZString. Elle marche, certes, mais on pourrait encore lui rajouter de nombreuses fonctionnalités !

main.cpp

Ce fichier contient le main qui fait quelques tests sur la classe ZString :

Code : C++ - [Sélectionner](#)

```

#include <iostream>
#include "ZString.h"

using namespace std;

int main()
{
    ZString chaine( "Bonjour" );
    ZString nom = "Mateo"; // Cette façon d'initialisation revient au même
    ZString resultat;

    resultat = chaine + " " + nom;

    cout << "Le resultat vaut maintenant : " << resultat << endl;

    return 0;
}

```

Le résultat qui doit s'afficher si tout va bien est :

Code : Console - [Sélectionner](#)

```
Le resultat vaut maintenant : Bonjour Mateo
```

Avec ce code on teste le constructeur, le destructeur, l'opérateur =, les opérateurs +, le cout...

ZString.h

Le cœur de notre classe est là. On y trouve la définition de ZString, ses attributs, ses méthodes. On trouve aussi le prototype de l'opérateur <<, en-dehors de la classe comme je vous l'ai dit car c'est l'opérateur de la classe ostream que l'on modifie là, pas celui de la classe ZString.

Code : C++ - [Sélectionner](#)

```

#ifndef DEF_ZSTRING
#define DEF_ZSTRING

#include <iostream>

class ZString
{
public:
    ZString();
    ZString(const char *chaine);
    ZString(const ZString &chaine);
    ~ZString();
    int longueur(const char *chaine);
    char *copie(const char *chaine);
    void afficher();
    ZString operator=(const char *chaine);
    ZString operator=(const ZString &chaine);
    ZString operator+(const char *chaine);
    ZString operator+(const ZString &chaine);
    char *getChaine();

private:
    char *m_chaine;
    int m_longueur;
};

std::ostream &operator<<( std::ostream &out, ZString &chaine );

#endif

```

ZString.cpp

C'est le plus gros fichier, celui qui nous aura donné le plus de fil à retordre aussi 😊
Il contient l'implémentation de toutes les méthodes de la classe ZString :

Code : C++ - [Sélectionner](#)

```

#include "ZString.h"

using namespace std;

ZString::ZString()
{
    m_chaine = NULL;
    m_longueur = 0;
}

ZString::ZString(const char *chaine)
{
    m_chaine = copie(chaine);
    m_longueur = longueur(chaine);
}

ZString::ZString(const ZString &chaine)
{
    m_chaine = copie(chaine.m_chaine);
    m_longueur = chaine.m_longueur;
}

ZString ZString::operator=(const char *chaine)
{
    delete[] m_chaine;
    m_chaine = copie(chaine);
    m_longueur = longueur(chaine);

    return *this;
}

ZString ZString::operator=(const ZString &chaine)
{
    delete[] m_chaine;
    m_chaine = copie(chaine.m_chaine);
    m_longueur = chaine.m_longueur;

    return *this;
}

ZString ZString::operator+(const char *chaine)
{
    int tailleTotale = m_longueur + longueur(chaine);
    char *sommeChaines = new char[tailleTotale + 1];

    for (int i = 0 ; i < m_longueur ; i++)
    {
        sommeChaines[i] = m_chaine[i];
    }

    for (int i = m_longueur ; i < tailleTotale ; i++)
    {
        sommeChaines[i] = chaine[i - m_longueur];
    }
    sommeChaines[tailleTotale] = '\0';

    ZString resultat(sommeChaines);
    delete[] sommeChaines;
    return resultat;
}

ZString ZString::operator+(const ZString &chaine)
{
    int tailleTotale = m_longueur + chaine.m_longueur;
    char *sommeChaines = new char[tailleTotale + 1];

    for (int i = 0 ; i < m_longueur ; i++)

```

Télécharger le projet

Vous pouvez télécharger le projet (réalisé sous Code::Blocks) en cliquant sur le lien ci-dessous :

Télécharger le projet (1 Ko)

Aller (encore) plus loin

Dans un premier temps, je vous conseille de bien potasser mon code source, d'essayer de le lire, le relire, le comprendre. Il y a peu de chances pour que vous ayez tout saisi du premier coup, mais si vous prenez le temps de bien analyser mon code et de relire mes explications, je suis sûr que vous allez progressivement vous sentir plus à l'aise là-dedans 😊

Maintenant, ce serait dommage de s'arrêter en si bon chemin vous ne trouvez pas ?

Je vous propose une série de modifications et ajouts que vous pouvez faire sur la classe ZString pour améliorer ses fonctionnalités :

- On vient de faire l'accesseur getChaine(), mais ça pourrait être bien aussi de faire l'accesseur getLongueur() pour que l'utilisateur puisse savoir à tout moment la longueur de sa chaîne.
- Une méthode vider() pourrait supprimer le contenu de la ZString. Il faudrait supprimer la chaîne mais aussi penser à remettre l'attribut m_longueur à 0.
- Une méthode recherche() pourrait faire une recherche dans la ZString. On pourrait même l'écrire en 3 versions :
 - Une qui prend en paramètre un char (recherche d'un caractère)
 - Une qui prend en paramètre un char * (recherche d'une chaîne)
 - Une autre qui prend en paramètre une ZString (recherche d'une chaîne).
- Dans le même style, on peut imaginer une méthode remplacer() qui prend au moins 2 paramètres : ce que vous recherchez, et par quoi vous voulez le remplacer.
- On n'a pas surchargé l'opérateur de comparaison == avec operator==() ! Si on veut pouvoir tester if (chaine1 == chaine2), il faut que l'on ait écrit cette méthode ! De même, vous devriez écrire operator!=() pour tester si 2 chaînes sont bien différentes, ça va de paire.
- Plus difficile : essayez de surcharger l'opérateur [] avec la méthode operator[](). Le but est de pouvoir écrire :

Code : C++ - [Sélectionner](#)

```
ZString chaine = "Mateo";
cout << chaine[2]; // Doit afficher "t"
```

Le paramètre passé à cette méthode est un nombre (int) correspondant au caractère de la chaîne que l'on veut extraire. En écrivant cette méthode, on peut alors récupérer n'importe quel caractère de la chaîne comme on le faisait avec les tableaux de char !

- Nous découvrirons dans un prochain chapitre ce que sont les méthodes statiques et constantes. Lorsque vous aurez lu ce chapitre, voyez si vous ne pouvez pas faire en sorte que certaines méthodes soient :
 - Constantes : ce sont les méthodes qui ne modifient pas les attributs de votre objet.
 - Statiques : ce sont les méthodes qui n'interagissent pas du tout avec les attributs de votre objet et qui pourraient être de simples fonctions. C'est le cas de copie() et longueur() par exemple.

Il vous faudra peut-être faire des recherches, voire demander de l'aide pour écrire certaines de ces méthodes.

Il y a du challenge, mais le jeu en vaut la chandelle !

Vous savez quoi ? Je crois que c'est un des premiers chapitres que j'écris où je suis soulagé d'arriver à la fin 😊

Il faut dire que ce TP n'était pas facile, vous comprenez pourquoi il était impensable de vous lâcher dans la nature tous seuls. J'ai tenu à vous expliquer pas à pas mon raisonnement et ma démarche pour écrire une classe en C++.

Je vous rassure : en temps normal on ne s'amuse pas à réécrire la classe string ! Toutefois, c'est vraiment un excellent exercice. Si vous prenez le temps de bien analyser ce qu'on a fait et de faire les améliorations proposées, vous allez vraiment progresser en C++. Vous voyez que ce n'est pas un langage simple, mais avec un peu de pratique on finit par acquérir certains automatismes qui limitent

nos erreurs. Et encore. Si vous saviez le nombre d'erreurs que j'ai faites avant d'arriver à faire marcher la classe ZString ! 

Tout ça pour vous dire qu'il ne faut pas paniquer devant cette apparente difficulté. Retrouvez vos manches, relisez, réfléchissez. Vous progresserez !

L'héritage

Nous allons maintenant découvrir une des notions les plus importantes de la POO : l'héritage.

Qu'on se rassure, il n'y aura pas de morts.

(voilà ça c'est fait)

L'héritage, c'est un concept très important qui fait à lui tout seul peut-être plus de la moitié de l'intérêt de la programmation orientée objet. Bref, ça rigole pas. C'est pas le moment de s'endormir au fond, j'veux ai à l'oeil 

Nous allons dans ce chapitre réutiliser notre exemple de la classe Personnage, mais on va beaucoup le simplifier pour se concentrer uniquement sur ce qui est important. En clair, on va juste garder le strict minimum, histoire d'avoir un exemple simple mais que vous connaissez déjà.

Allez, bon courage, cette notion n'est pas bien dure à comprendre, elle est juste très riche.

Exemple d'héritage simple

"Héritage", c'est un drôle de mot pour de la programmation hein 

Alors c'est quoi ? C'est une technique qui permet de créer une classe à partir d'une autre classe. Elle lui sert de modèle, de base de départ. Cela permet d'éviter à avoir à réécrire un même code source plusieurs fois.

Comment reconnaître un héritage ?

C'est LA question à se poser. Certains ont tellement été traumatisés par l'héritage qu'ils en voient partout, d'autres au contraire (surtout les débutants) se demandent à chaque fois s'il y a un héritage à faire ou pas. Pourtant, ce n'est pas "mystique", il est très facile de savoir s'il y a une relation d'héritage entre 2 classes.

Comment ? En suivant cette règle très simple :

Il y a héritage quand on peut dire :
"A est un B"

Pas de panique c'est pas des maths 

Prenez un exemple très simple. On peut dire "Un guerrier est un personnage", ou encore "Un magicien est un personnage". Donc on peut faire un héritage : "La classe Guerrier hérite de Personnage", "La classe Magicien hérite de Personnage".

Pour vous imprégner, voici quelques autres bons exemples où un héritage peut être fait :

- Une voiture est un véhicule (Voiture hérite de Véhicule)
- Un bus est un véhicule (Bus hérite de véhicule)
- Un moineau est un oiseau (Moineau hérite d'Oiseau)
- Un corbeau est un oiseau (Corbeau hérite d'Oiseau)
- Un chirurgien est un docteur (Chirurgien hérite de Docteur)
- Un diplodocus est un dinosaure (Diplodocus hérite de Dinosaure)
- etc.

En revanche, vous ne pouvez pas dire "Un dinosaure est un diplodocus", ou encore "Un bus est un oiseau". Donc on ne peut pas faire d'héritage dans ces cas-là, du moins ça n'aurait aucun sens 

Nous allons voir comment réaliser un héritage en C++, mais d'abord il faut que je pose l'exemple sur lequel on va travailler 😊

Notre exemple : la classe Personnage

Petit rappel : cette classe représente un personnage d'un jeu vidéo de type RPG (jeu de rôle). Il n'est pas nécessaire de savoir jouer ou d'avoir joué à un RPG pour suivre mon exemple. J'ai juste choisi celui-là car il est plus ludique que la plupart des exemples barbants que les profs d'informatique aiment utiliser (Voiture, Bibliothèque, Université, PompeAEssence...).

On va un peu simplifier notre classe Personnage. Voici ce sur quoi je vous propose de partir :

Code : C++ - [Sélectionner](#)

```
/*
Personnage.h
*/

#ifndef DEF_PERSONNAGE // Pour éviter les inclusions multiples
#define DEF_PERSONNAGE // (revoir au besoin cours C, partie II, ch. 5)

#include <iostream>
#include <string>

class Personnage
{
public:
    Personnage();
    void recevoirDegats(int degats);
    void coupDePoing(Personnage &cible);

private:
    int m_vie;
    std::string m_nom;
};

#endif
```

Notre Personnage a un nom et une quantité de vie.

On n'a mis qu'un seul constructeur, un constructeur par défaut. Il permet d'initialiser le Personnage avec une vie et un nom de base. Le Personnage peut recevoir des dégâts, via la méthode recevoirDegats et en distribuer, via la méthode coupDePoing.

A titre informatif, voici l'implémentation des méthodes dans Personnage.cpp :

Code : C++ - [Sélectionner](#)

```

/*
Personnage.cpp
*/

#include "Personnage.h"

using namespace std;

Personnage::Personnage() : m_vie(100), m_nom("Jack")
{
}

void Personnage::recevoirDegats(int degats)
{
    m_vie -= degats;
}

void Personnage::coupDePoing(Personnage &cible)
{
    cible.recevoirDegats(10);
}

```

Rien d'extraordinaire pour le moment.

La classe Guerrier hérite de la classe Personnage

Intéressons-nous maintenant à l'héritage. L'idée, c'est de créer une nouvelle classe qui est une sous-classe de Personnage. On dit que cette classe va hériter de Personnage.

Pour cet exemple, je vais créer une classe Guerrier qui hérite de Personnage. La définition de la classe, dans Guerrier.h, ressemble à ceci :

Code : C++ - Sélectionner

```

/*
Guerrier.h
*/
#ifndef DEF_GUERRIER
#define DEF_GUERRIER

#include <iostream>
#include <string>
#include "Personnage.h" // Ne pas oublier d'inclure Personnage.h pour pouvoir en hériter

class Guerrier : public Personnage // Signifie : créer une classe Guerrier qui hérite de
{
};

#endif

```

Grâce à ce qu'on vient de faire, la classe Guerrier contiendra de base tous les attributs et toutes les méthodes de la classe Personnage.

Dans un tel cas, la classe Personnage est appelée la classe "Mère", et la classe Guerrier la classe "Fille".

Mais quel intérêt de créer une nouvelle classe si c'est pour qu'elle contienne les mêmes attributs et les mêmes méthodes ?

Attendez, justement ! Le truc, c'est qu'on peut rajouter des attributs et des méthodes spéciales dans la classe Guerrier. Par exemple, on pourrait rajouter une méthode qui ne concerne que les guerriers, du genre frapperCommeUnSourdAvecUnMarteau (bon

ok c'est un nom de méthode un peu long j'avoue 😊).

Code : C++ - Sélectionner

```
/*
Guerrier.h
*/

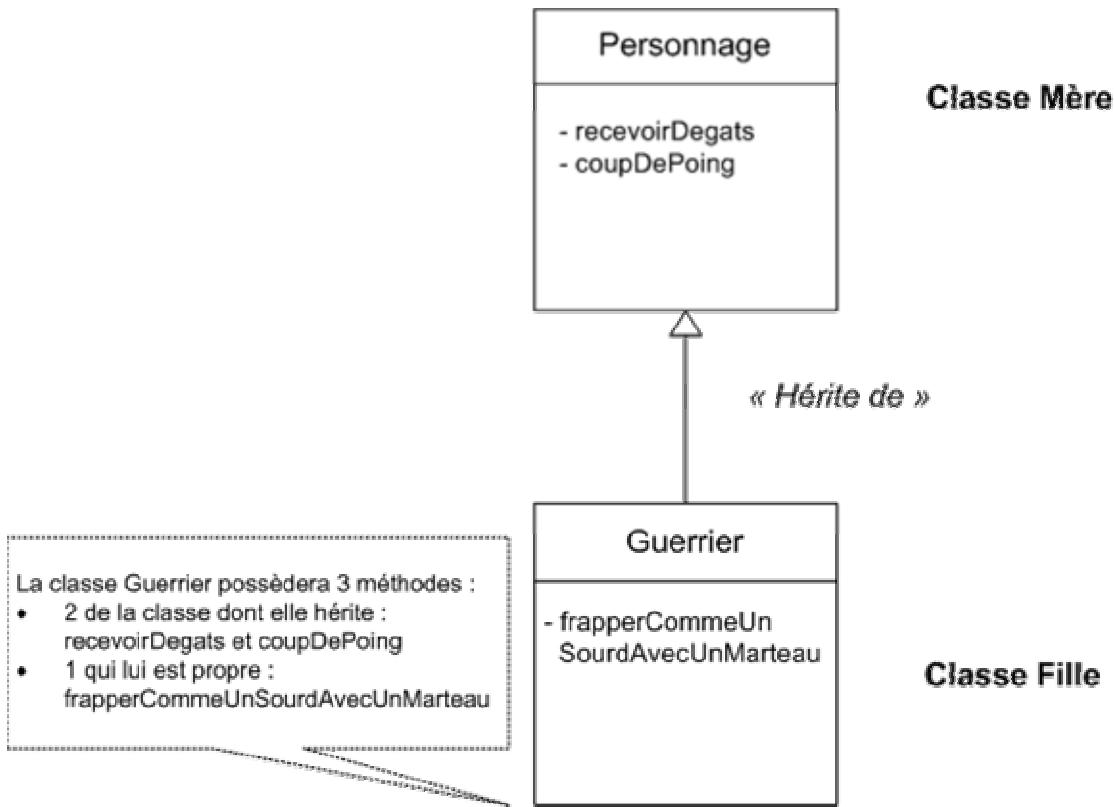
#ifndef DEF_GUERRIER
#define DEF_GUERRIER

#include <iostream>
#include <string>
#include "Personnage.h"

class Guerrier : public Personnage
{
public:
    void frapperCommeUnSourdAvecUnMarteau(); // Méthode qui ne concerne que les guerriers
};

// 3 4 5 6
```

Schématiquement, on représente la situation comme ça :



Le schéma se lit de bas en haut, c'est-à-dire "Guerrier hérite de Personnage".

Guerrier est la classe fille, Personnage est la classe mère. On dit que Guerrier est une "spécialisation" de la classe Personnage. Elle possède toutes les caractéristiques d'un Personnage (de la vie, un nom, elle peut recevoir des dégâts), mais possède en plus des caractéristiques propres au Guerrier comme `frapperCommeUnSourdAvecUnMarteau` 😊

Retenez bien que lorsqu'on fait un héritage, on hérite des méthodes et des attributs.

Je n'ai pas représenté les attributs sur le schéma ci-dessus pour ne pas surcharger, mais la vie et le nom du Personnage sont bel et bien hérités, ce qui fait qu'un Guerrier possède aussi de la vie et un nom !

Vous commencez à comprendre le principe ? En C++, on utilise tellement de classes que bien souvent ça ne sert à rien de recréer

une classe depuis le début, il vaut mieux hériter d'une classe plus "générale" pour éviter d'avoir à réécrire le même code 50 fois 😊

Ce concept a l'air de rien comme ça, mais croyez-moi ça fait la différence ! Vous n'allez pas tarder à voir tout ce que ça a de puissant lorsque vous pratiquerez plus loin dans le cours.

La classe Magicien hérite aussi de Personnage

Tant qu'il n'y a qu'un seul héritage, l'intérêt semble encore limité. Mais multiplions un peu les héritages et les spécialisations et nous allons vite voir tout l'intérêt de la chose.

Par exemple, si on créait une classe Magicien qui va elle aussi hériter de Personnage ? Après tout, un Magicien est un Personnage, donc il peut récupérer les mêmes propriétés de base : de la vie, un nom, donner un coup de poing, etc.

La différence, c'est que le Magicien peut aussi envoyer des sorts magiques, par exemple bouleDeFeu et bouleDeGlace. Pour utiliser sa magie, il a une réserve de magie qu'on appelle "Mana" (ça va faire un attribut à rajouter). Quand la Mana tombe à zéro, il ne peut plus lancer de sort.

Code : C++ - [Sélectionner](#)

```
/*
Magicien.h
*/

#ifndef DEF_MAGICIEN
#define DEF_MAGICIEN

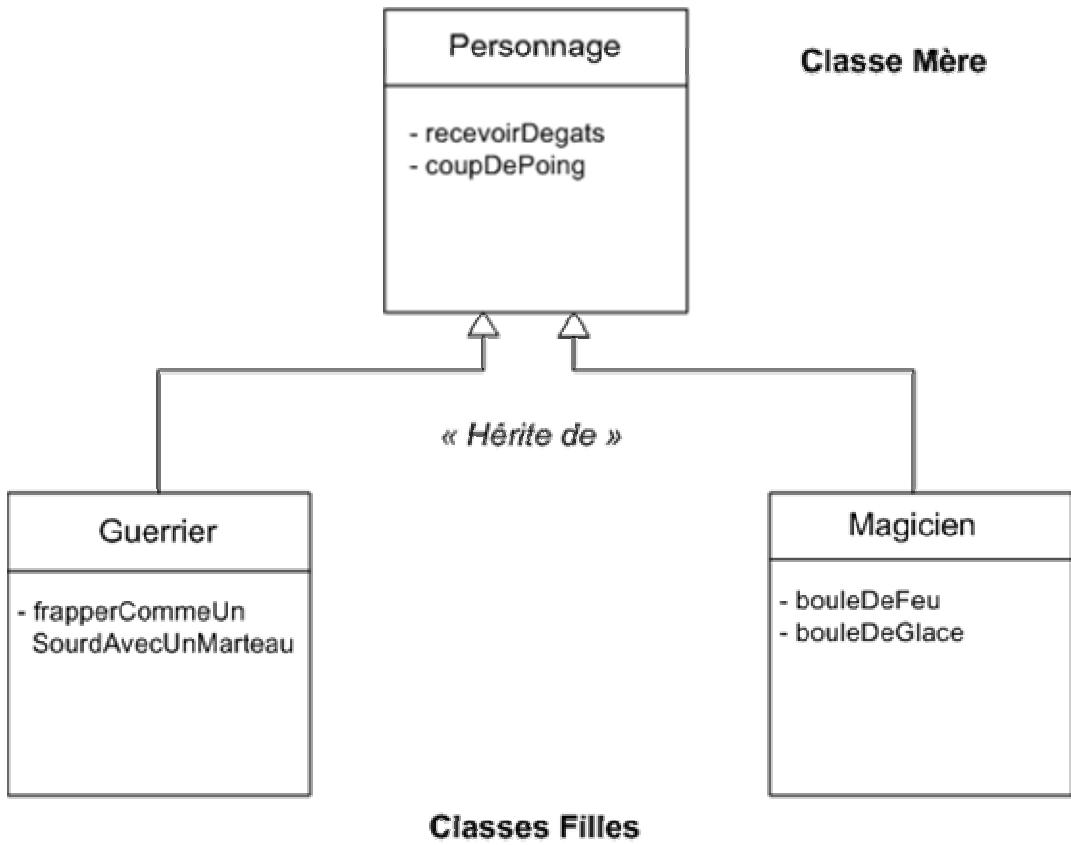
#include <iostream>
#include <string>
#include "Personnage.h"

class Magicien : public Personnage
{
public:
    void bouleDeFeu();
    void bouleDeGlace();

private:
    int mana;
};

#endif
```

Je ne vous donne pas l'implémentation des méthodes (le .cpp) ici, je veux juste que vous compreniez et reteniez le principe :

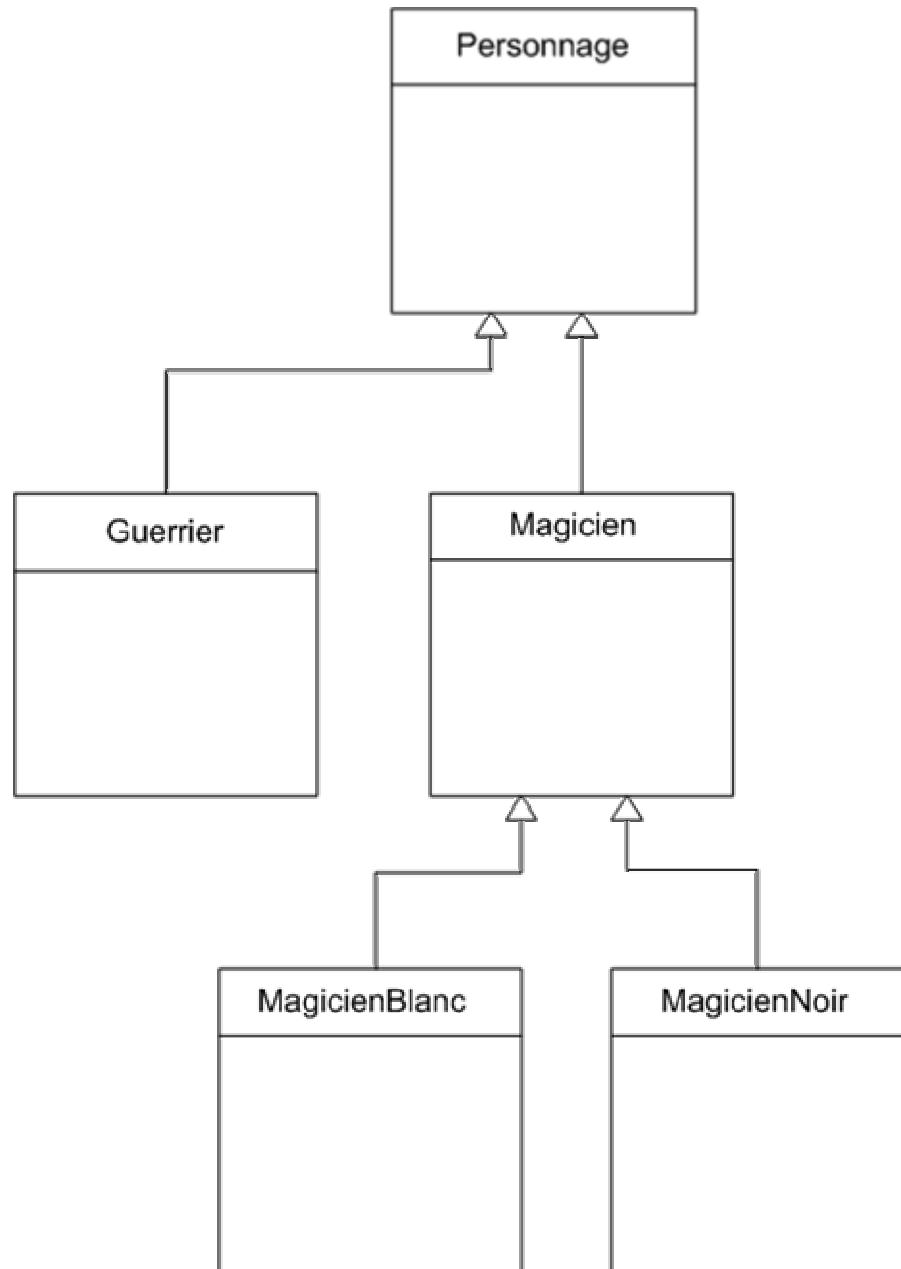


Notez que sur le schéma je n'ai représenté que les méthodes des classes, mais les attributs (vie, nom...) sont eux aussi hérités !

Et le plus beau, c'est qu'on peut faire une classe qui hérite d'une classe qui hérite d'une autre classe ! 😊

Imaginons qu'il y ait 2 types de magiciens : les magiciens blancs, qui sont des gentils qui envoient des sorts de guérison tout ça tout ça, et les magiciens noirs qui sont des méchants qui utilisent leurs sorts pour tuer des gens (super exemple, j'en suis fier).

Avada Kedavra !



Et ça pourrait continuer longtemps comme ça. Vous verrez dans la prochaine partie sur la librairie C++ Qt qu'il y a souvent 5 ou 6 héritages qui sont faits à la suite. C'est vous dire si c'est utilisé !

Je n'ai pas mis les noms des méthodes dans le schéma cette fois pour ne pas surcharger, c'est le principe qui compte hein 😊

La dérivation de type

Imaginons le code suivant :

Code : C++ - [Sélectionner](#)

```

Personnage monPersonnage;
Guerrier monGuerrier;

monPersonnage.coupDePoing(monGuerrier);
monGuerrier.coupDePoing(monPersonnage);
  
```

Compilez : ça marche. Mais si vous êtes attentif, vous devriez vous demander pourquoi ça a marché, parce que normalement ça

n'aurait pas dû !

... non, vous ne voyez pas ? 

Allez un effort, voici le prototype de coupDePoing (il est le même dans la classe Personnage et dans la classe Guerrier rappelez-vous) :

Code : C++ - [Sélectionner](#)

```
void coupDePoing(Personnage &cible);
```

Quand on fait `monGuerrier.coupDePoing(monPersonnage);`, on envoie bien un Personnage en paramètre.

Mais quand on fait `monPersonnage.coupDePoing(monGuerrier);`, ça marche aussi et le compilateur ne hurle pas à la mort alors que, selon toute logique, il devrait ! En effet, la méthode coupDePoing attend un Personnage et on lui envoie un Guerrier. Pourquoi diable cela fonctionne-t-il ? 

Eh bien... c'est justement une propriété très intéressante de l'héritage en C++ que vous venez de découvrir là. On peut substituer un objet fille à un pointeur ou une référence d'un objet mère. Ce qui veut dire, dans une autre langue que le chinois, qu'on peut faire ça :

Code : C++ - [Sélectionner](#)

```
Personnage *monPersonnage = NULL;
Guerrier *monGuerrier = new Guerrier();

monPersonnage = monGuerrier; // Mais... mais... Ca marche !?
```

Les 2 premières lignes n'ont rien d'extraordinaire : on crée un pointeur Personnage mis à NULL, et un pointeur Guerrier qu'on initialise avec l'adresse d'un nouvel objet de type Guerrier.

Par contre, la ligne n°4 est assez surprenante. Normalement, on ne devrait pas pouvoir donner à un pointeur de type Personnage un pointeur de type Guerrier. C'est comme mélanger des carottes et des patates, ça se fait pas.

Alors oui, en temps normal le compilateur n'accepte pas d'échanger des pointeurs (ou des références) de types différents. Or, Personnage et Guerrier ne sont pas n'importe quels types : Guerrier hérite de Personnage. Et la règle à connaître, c'est justement qu'on peut affecter un élément enfant à un élément parent !

L'inverse est faux par contre ! On ne peut PAS faire :

`monGuerrier = monPersonnage;`

Ceci plante et est strictement interdit. Attention au sens de l'affectation donc.

Cela nous permet donc de placer un élément dans un pointeur (ou une référence) de type plus général.
C'est très pratique dans notre cas lorsqu'on passe une cible en paramètre :

Code : C++ - [Sélectionner](#)

```
void coupDePoing(Personnage &cible);
```

Notre méthode coupDePoing est capable de faire mal à n'importe quel Personnage ! Qu'il soit Guerrier, Magicien, MagicienBlanc, MagicienNoir ou autre, c'est un Personnage après tout, donc on peut lui donner un coupDePoing 

C'est un peu choquant au début je le reconnaiss, mais on se rend compte au final qu'en fait c'est très bien fait. [Ca fonctionne, puisque la méthode coupDePoing ne fait qu'appeler des méthodes de la classe Personnage \(recevoirDegats\), et que ces méthodes se trouvent forcément dans toutes les classes filles \(Guerrier, Magicien\).](#)

Relisez-moi, essayez de comprendre, vous devriez saisir pourquoi ça marche 

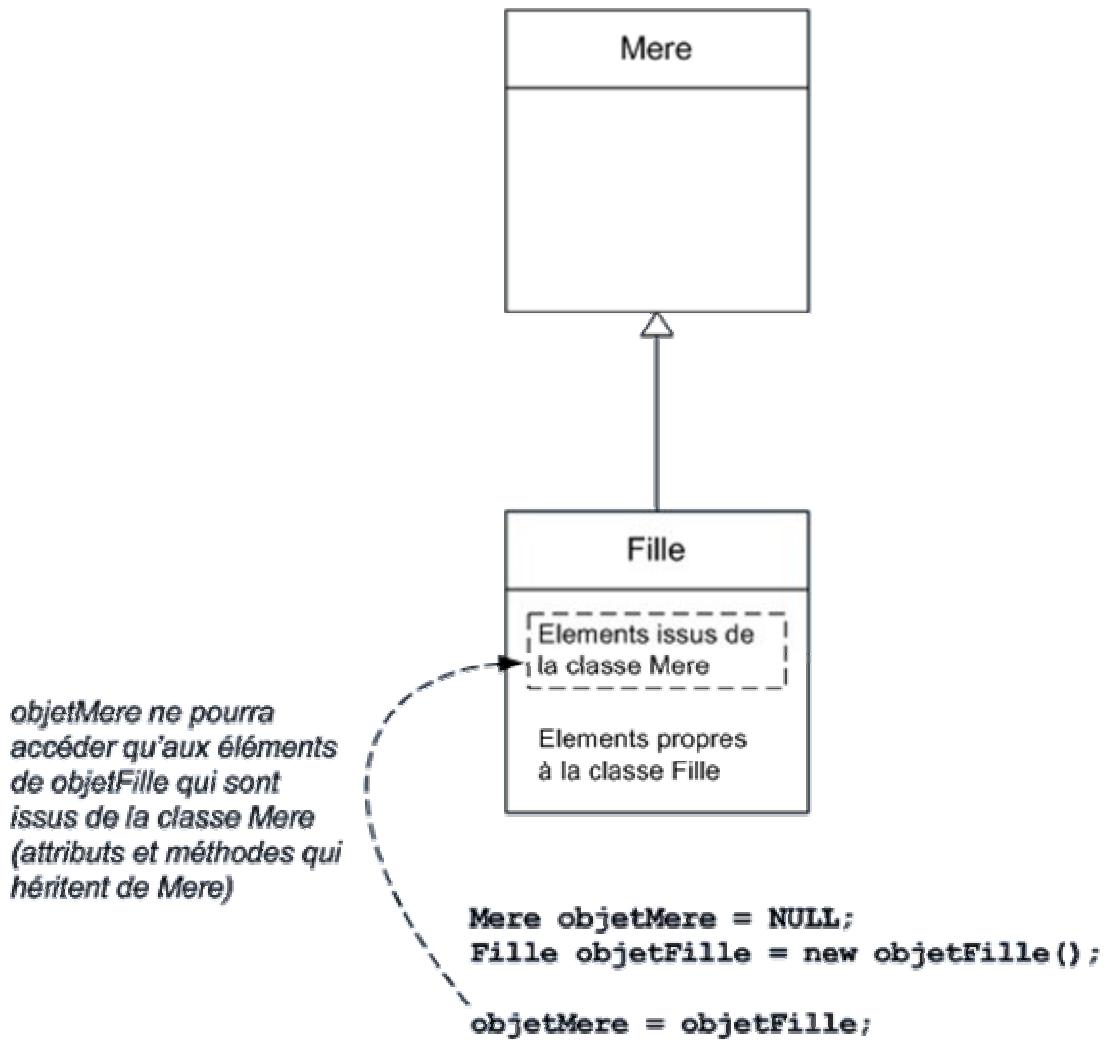
Eh ben non, moi je comprends PAS ! Je ne vois pas pourquoi ça marche si on fait :

`objetMere = objetFille;`

Là on affecte la fille à la mère, or la fille possède des attributs que la mère n'a pas. Ca devrait coincer ! L'inverse ne serait pas plus logique ?

Je vous rassure, personnellement j'ai mis des mois avant d'arriver à comprendre ce qui se passait vraiment (comment ça ça vous rassure pas ? 😊)

Votre erreur est de croire qu'on affecte la fille à la mère. Non on n'affecte pas la fille à la mère, on affecte un pointeur (ou une référence). Déjà c'est pas du tout pareil. Ensuite, ce n'est pas toute la fille qu'on affecte à la mère, mais seulement la partie qui hérite de la mère.



Voilà je peux difficilement pousser l'explication plus loin, j'espère que vous allez comprendre, sinon pas de panique j'ai survécu plusieurs mois de programmation en C++ sans bien comprendre ce qui se passait et j'en suis pas mort 😊 (mais c'est mieux si vous comprenez c'est clair !)

En tout cas sachez que c'est une technique très utilisée, on s'en sert vraiment souvent en C++ ! Vous découvrirez bien ça avec la pratique en utilisant Qt dans la prochaine partie.

Héritage et constructeurs

Vous avez peut-être remarqué que je n'ai pas encore parlé des constructeurs dans les classes filles (Guerrier, Magicien...). C'est le moment justement de s'y intéresser 😊

On sait que Personnage a un constructeur (un constructeur par défaut) défini comme ceci dans le .h :

Code : C++ - [Sélectionner](#)

```
Personnage();
```

... et son implémentation dans le .cpp :

Code : C++ - [Sélectionner](#)

```
Personnage::Personnage() : m_vie(100), m_nom("Jack")  
{  
}  
}
```

Comme vous le savez, lorsqu'on crée un objet de type Personnage, le constructeur est appelé avant toute chose.

Mais maintenant, que se passe-t-il lorsqu'on crée par exemple un Magicien qui hérite de Personnage ? Le Magicien a le droit d'avoir un constructeur lui aussi ! Est-ce que ça ne va pas interférer avec le constructeur de Personnage ? Il faut pourtant appeler le constructeur de Personnage si on veut que la vie et le nom soient initialisés !

En fait, les choses se dérouleront dans l'ordre suivant :

1. Vous demandez à créer un objet de type Magicien
2. Le compilateur appelle d'abord le constructeur de la classe mère (Personnage)
3. Puis, le compilateur appelle le constructeur de la classe fille (Magicien)

En clair, c'est d'abord le constructeur du "parent" qui est appelé, puis celui du fils, et éventuellement du petit fils (s'il y a un héritage d'héritage, comme c'est le cas avec MagicienBlanc).

Appeler le constructeur de la classe mère

Pour appeler le constructeur de Personnage en premier, il faut y faire appel depuis le constructeur de Magicien. C'est dans un cas comme ça qu'il est ~~bon~~ indispensable de se servir de la liste d'initialisation (vous savez, tout ce qui suit le symbole deux-points dans l'implémentation).

Code : C++ - [Sélectionner](#)

```
Magicien::Magicien() : Personnage(), mana(100)  
{  
}  
}
```

Le premier élément de la liste d'initialisation dit de faire d'abord appel au constructeur de la classe parente Personnage. Puis, les initialisations propres au Magicien sont faites (comme l'initialisation de la mana à 100).

Lorsqu'on crée un objet de type Magicien, le compilateur appelle le constructeur par défaut de la classe mère (celui qui ne prend pas de paramètre).

Transmission de paramètres

Le gros avantage de cette technique est que l'on peut "transmettre" les paramètres du constructeur de Magicien au constructeur de Personnage. Par exemple, si le constructeur de Personnage prenait un nom en paramètre, il faudrait que le Magicien accepte lui aussi ce paramètre et le fasse passer au constructeur de Personnage :

Code : C++ - [Sélectionner](#)

```
Magicien::Magicien(string nom) : Personnage(nom), mana(100)  
{  
}  
}
```

Bien entendu, si on veut que ça marche il faudra aussi surcharger le constructeur de Personnage pour qu'il accepte un paramètre string !

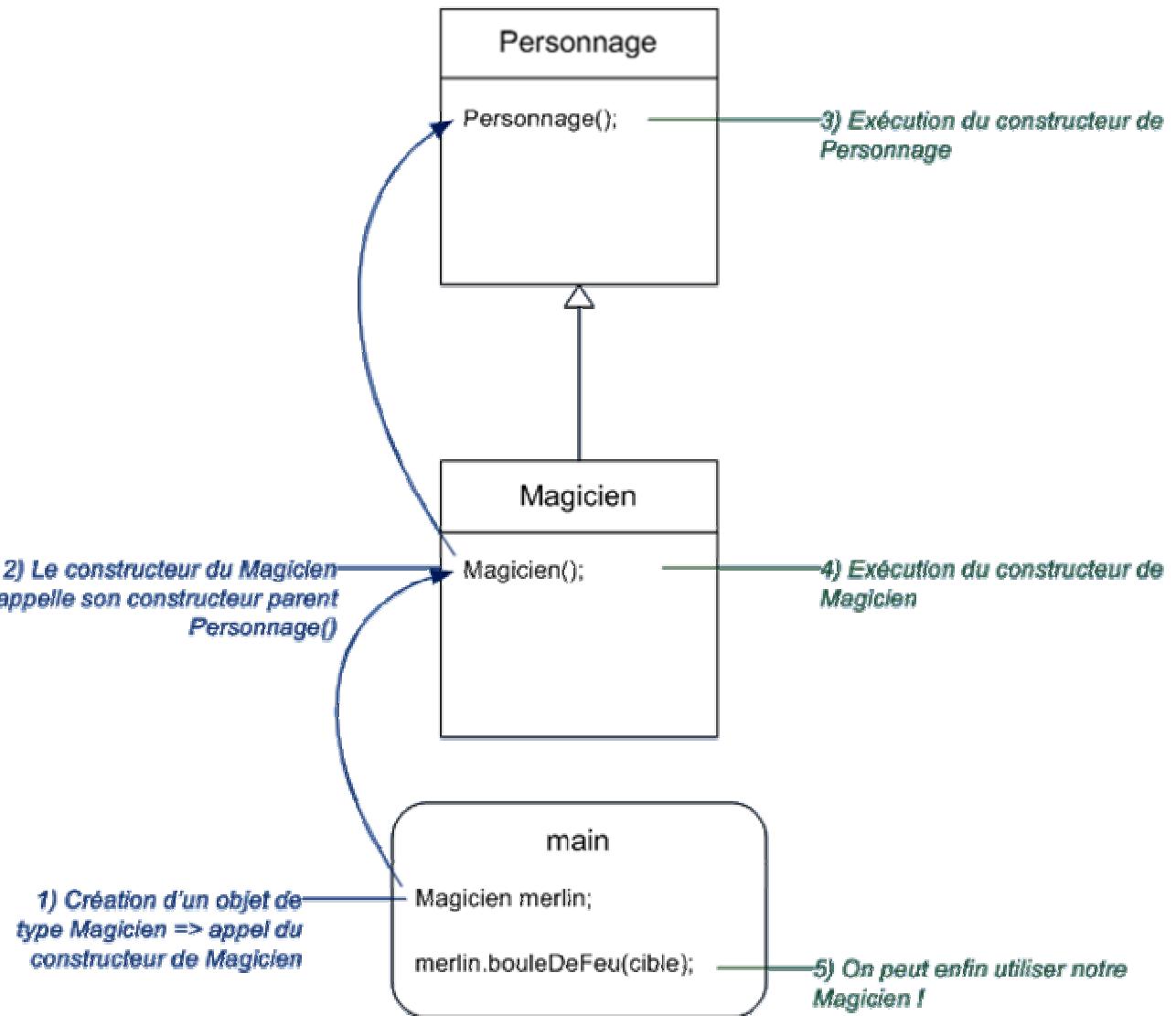
Code : C++ - Sélectionner

```
Personnage::Personnage(string nom) : m_vie(100), m_nom(nom)
{
}
```

Et voilà comment on fait "remonter" des paramètres d'un constructeur à un autre pour s'assurer que l'objet se crée correctement 😊

Schéma résumé

Pour bien mémoriser ce qui se passe, rien de tel qu'un schéma résumé n'est-ce pas ? 😊



Il faut bien entendu le lire dans l'ordre pour en comprendre le fonctionnement. On commence par demander à créer un Magicien. "Oh mais c'est un objet" se dit le compilateur, "il faut que j'appelle son constructeur".

Or, le constructeur du Magicien indique qu'il faut d'abord appeler le constructeur de la classe parente Personnage. Le compilateur va donc voir la classe parente, puis exécute son code. Il retourne ensuite au constructeur du Magicien et exécute son code.

La portée protected

Il me serait vraiment impossible de vous parler d'héritage sans vous parler de la portée protected.

Rappel : les portées (ou droits d'accès) que vous connaissez déjà sont :

- public : les éléments qui suivent seront accessibles depuis l'extérieur de la classe.
- private : les éléments qui suivent ne seront pas accessibles depuis l'extérieur de la classe.

Je vous ai en particulier donné la règle fondamentale du C++, l'encapsulation, qui veut que l'on empêche systématiquement au monde extérieur d'accéder aux attributs de nos classes.

La portée protected est un autre type de droit d'accès que je classerais entre public (le plus permissif) et private (le plus restrictif). Il n'a de sens que pour les classes qui se font hériter (les classes mères) mais on peut les utiliser sur toutes les classes même quand il n'y a pas d'héritage.

Sa signification est la suivante :

protected : les éléments qui suivent ne seront pas accessibles depuis l'extérieur de la classe, sauf si c'est une classe fille.

Cela veut dire par exemple que si l'on met des éléments en protected dans la classe Personnage, on y aura accès dans les classes filles Guerrier et Magicien. Avec la portée private, on n'aurait pas pu y accéder !

En pratique, personnellement je donne toujours la portée protected aux attributs de mes classes. C'est comme private (donc ça respecte l'encapsulation) sauf que comme ça, au cas où j'hérite un jour de cette classe, j'aurai aussi directement accès aux attributs.

Cela est souvent nécessaire voire indispensable sinon on doit utiliser des tonnes d'accesseurs (méthodes getTruc) et ça rend le code bien plus lourd.

Code : C++ - [Sélectionner](#)

```
class Personnage
{
public:
    Personnage();
    Personnage(std::string nom);
    void recevoirDegats(int degats);
    void coupDePoing(Personnage &cible);

protected: // Privé, mais accessible aux éléments enfants (Guerrier...)
    int m_vie;
    std::string m_nom;
};
```

On peut alors directement manipuler la vie et le nom dans tous les éléments enfants de Personnage, comme Guerrier et Magicien !

Ce chapitre en impose peut-être un peu par sa taille, mais ne vous y fiez pas ce sont surtout les schémas qui prennent de la place 😊

D'ailleurs, j'ai volontairement évité de trop montrer de codes sources complets différents et j'ai préféré que vous vous focalisiez sur ces schémas. C'est ce qu'on retient le mieux en général, et ça permet de bien se repérer. La pratique viendra dans la partie sur la librairie Qt.

Ceci étant, peut-être que vous aimerez avoir le code source complet de mes exemples (Personnage, Guerrier, Magicien...). Ce code n'est pas complet, certaines méthodes ne sont pas écrites, il ne fait rien d'extraordinaire. Mais il compile, et ça vous permettra peut-être de finir de mettre de l'ordre dans vos idées.

Voici donc le code source :

[Télécharger le code source complet \(3 Ko\)](#)

Eléments statiques et constants

Vous tenez le coup ? 😊

Courage, vos efforts seront bientôt largement récompensés.

Ce chapitre va d'ailleurs vous permettre de souffler un peu. Vous allez découvrir quelques notions spécifiques aux classes en C++ : les attributs et méthodes statiques et constants. Ce sont ce que j'appellerais des "points particuliers" du C++. Ce ne sont pas des détails pour autant, ce sont des choses à connaître.

Car oui, tout ce que je vous apprends là, vous allez en avoir besoin et vous allez largement le réutiliser. Je suis sûr aussi que vous en comprendrez mieux l'intérêt lorsque vous pratiquerez pour de bon.

N'allez pas croire que les programmeurs ont inventé des trucs un peu complexes comme ça juste pour le plaisir de programmer de façon tordue 🤪

Les méthodes constantes

On en a rapidement parlé lorsqu'on a introduit les accesseurs (méthodes get/set pour accéder aux attributs), mais je pense que ça vaut le coup de faire le point complètement sur cette notion ici. Ca sera court, mais au moins vous le retiendrez bien et vous ne serez pas surpris si vous voyez des gens en faire.

Euh de quoi je parle ? Des méthodes constantes ! 😊

Ce sont des méthodes qui possèdent le mot-clé const à la fin de leur prototype et de leur déclaration.

Quand vous dites "ma méthode est constante", vous indiquez au compilateur que votre méthode ne modifie pas l'objet, c'est-à-dire qu'elle ne modifie la valeur d'aucun de ses attributs. Par exemple, une méthode qui se contente d'afficher des informations à l'écran sur l'objet est une méthode constante : elle ne fait que lire les attributs. En revanche, une méthode qui met à jour le niveau de vie d'un personnage ne peut pas être constante 😊

Ca s'utilise comme ceci :

Code : C++ - Sélectionner

```
// Prototype de la méthode (dans le .h) :
void maMethode(int parametre) const;

// Déclaration de la méthode (dans le .cpp) :
void maMethode(int parametre) const
{
}
```

On utilisera souvent le mot-clé const sur les méthodes accesseur (getAttribut), ces méthodes qui se contentent de renvoyer la valeur d'un attribut pour respecter le principe d'encapsulation qui dit que l'attribut doit être privé.

Code : C++ - Sélectionner

```
int Personnage::getVie() const
{
    return m_vie;
}
```

Concrètement, ça sert à quoi de créer des méthodes constantes ?

Ca sert à 2 choses principalement :

- Pour vous : vous savez que votre méthode ne fait que lire les attributs, et vous vous interdisez dès le début de les modifier. Si par erreur vous en modifiez, le compilateur plantera en vous disant que vous ne respectez pas la règle que vous vous êtes fixée. Et ça c'est bien.
- Pour les utilisateurs de votre classe : c'est très important aussi pour eux, ça leur indique que la méthode ne fait que renvoyer un résultat mais qu'elle ne modifie pas l'objet. Dans une documentation, le mot-clé const apparaît dans le prototype de la méthode et est un excellent indicateur de ce qu'elle fait, ou plutôt de ce qu'elle ne peut pas faire (ça pourrait se traduire par : "cette méthode ne modifiera pas votre objet").

Les méthodes statiques

Ah les méthodes statiques... Alors ça, c'est un peu spécial 😊

Ce sont des méthodes qui appartiennent à la classe mais pas aux objets instanciés à partir de la classe... En fait, ce sont de bêtes "fonctions" rangées dans des classes qui n'ont pas accès aux attributs de la classe. Ca s'utilise d'une manière un peu particulière.

Le mieux est encore un exemple je pense !

Créer une méthode statique

Dans le .h, le prototype d'une méthode statique ressemble à ceci :

Code : C++ - Sélectionner

```
class MaClasse
{
public:
    MaClasse();
    static void maMethode();
};
```

Son implémentation dans le .cpp ne possède pas en revanche de mot-clé static :

Code : C++ - Sélectionner

```
void Personnage::maMethode() // Ne pas remettre "static" dans l'implémentation
{
    cout << "Bonjour !" << endl;
}
```

Ensuite, dans le main, la méthode statique s'appelle comme ceci :

Code : C++ - Sélectionner

```
int main()
{
    Personnage::maMethode();

    return 0;
}
```

Mais... on n'a pas créé d'objet de type Personnage et on appelle la méthode quand même ? C'est quoi ce bazar ?

C'est justement ça la particularité des méthodes statiques. Pour les utiliser, pas besoin de créer un objet. Il suffit juste de faire précéder le nom de la méthode par le nom de la classe suivi de deux deux-points.

D'où le : Personnage ::maMethode();

Cette méthode, comme je vous le disais, ne peut pas accéder aux attributs de la classe. C'est vraiment une bête fonction, mais rangée dans une classe. Ca permet de regrouper les fonctions dans des classes, par thème, et aussi d'éviter des conflits de nom.

Quelques exemples de l'utilité des méthodes statiques

Les méthodes statiques peuvent vous paraître un tantinet stupides. En effet, à quoi bon avoir inventé le modèle objet si c'est pour autoriser les gens à créer de bêtes "fonctions" regroupées dans des classes ?

La réponse, c'est qu'on a toujours besoin d'utiliser de "bêtes" fonctions même en modèle objet, mais pour être un peu cohérent on les regroupe dans des classes en précisant qu'elles sont statiques.

Il y a en effet des fonctions qui ne nécessitent pas de créer un objet, pour lesquelles ça n'aurait pas de sens.
Des exemples ?

- Il existe dans la librairie Qt une classe QDate qui permet de manipuler des dates. On peut comparer des dates entre elles (surcharge d'opérateur) etc etc. Cette classe propose aussi un certain nombre de méthodes statiques, comme currentDate qui renvoie la date actuelle. Pas besoin de créer un objet pour avoir cette information ! Il suffit donc de taper QDate::currentDate() pour récupérer la date actuelle 😊
- Toujours avec Qt, la classe QDir, qui permet de manipuler les dossiers du disque dur, propose quelques méthodes statiques. Par exemple, on trouve QDir::drives() qui renvoie la liste des disques présents sur l'ordinateur (par exemple "C:\\", "D:\\" etc). Là encore, ça n'aurait pas eu d'intérêt d'instancier un objet à partir de la classe car ce sont des informations générales.
- etc etc.

Mmmh mais c'est que ça donne envie de travailler avec Qt tout ça 😊

Les attributs statiques

Il existe aussi ce qu'on appelle des attributs statiques.

Tout comme les méthodes statiques, les attributs statiques appartiennent à la classe et non aux objets créés à partir de la classe.

Créer un attribut statique dans une classe

C'est assez simple en fait : il suffit de rajouter le mot-clé static au début de la ligne.

Un attribut static, bien qu'il soit accessible de l'extérieur, peut très bien être déclaré private ou protected. Appelez ça une exception, car c'en est bien une 😊

Exemple :

Code : C++ - Sélectionner

```
class MaClasse
{
    public:
        MaClasse();

    private:
        static int monAttribut;

};
```

Sauf qu'on ne peut pas initialiser l'attribut statique ici. Il faut le faire dans l'espace global, c'est-à-dire en dehors de toute classe ou fonction, en dehors du main notamment.

Code : C++ - [Sélectionner](#)

```
// Initialiser l'attribut en dehors de toute fonction ou classe (espace global)
int MaClasse::monAttribut = 5;
```

Un attribut déclaré comme statique se comporte comme une variable globale, c'est-à-dire une variable accessible partout dans le code.

Ouaaaaah ! Stop !

Tu nous avais pas dit à un moment que les variables globales c'était le mal absolu et que même si ça existait il fallait préférer se pendre plutôt que de les utiliser ? 

En effet. Bien qu'il y ait toujours des cas où ça se révèle utile et indispensable, c'est très rarement le cas. De manière générale, fuyez ces variables globales comme la peste.

Créer un attribut statique dans une méthode d'une classe

Il y a un cas particulier : on peut aussi créer une variable statique à l'intérieur d'une méthode d'une classe. Cette fois c'est un peu moins bourrin : la variable ne sera accessible que depuis la méthode où elle se trouve.

Le truc, c'est que la variable ne sera pas supprimée de la mémoire à la fin de la méthode. Elle reste en mémoire et sera réutilisée la prochaine fois que la méthode sera appelée, et ce quel que soit l'objet qui y fait appel.

Pour ceux qui s'en souviennent, c'est exactement le même principe que les variables statiques qu'on avait vues dans le cours de C.

Un exemple sera plus parlant 

Code : C++ - MaClasse.h - [Sélectionner](#)

```
class MaClasse
{
public:
    void methode();
};
```

Code : C++ - MaClasse.cpp - [Sélectionner](#)

```
void MaClasse::methode()
{
    static int compteur = 0;
    compteur++;

    cout << compteur << endl;
}
```

Code : C++ - main.cpp - [Sélectionner](#)

```

int main()
{
    MaClasse objet1, objet2;

    objet1.methode();
    objet2.methode();

    return 0;
}

```

Résultat à l'écran :

Code : Console - [Sélectionner](#)

```

1
2

```

Lors de l'appel de la méthode du premier objet, la variable statique compteur est créée et le nombre 1 est affiché.
Lors de l'appel de la méthode du second objet, la variable statique compteur existe déjà en mémoire donc elle n'est pas recréée.
C'est celle créée pour le premier objet qui est réutilisée ici. La preuve : le compteur est incrémenté à nouveau et l'écran affiche 2, ce qui signifie que la variable compteur est la même dans les 2 objets.

Les variables statiques au sein d'une méthode ont une portée limitée à leur méthode. Ce ne sont pas des variables globales et c'est donc déjà bien moins crade 😊. Elles peuvent avoir une utilité, comme par exemple compter le nombre d'objets créés à partir d'une classe (il faudrait dans ce cas mettre la variable statique compteur dans le constructeur).

Ces points un peu particuliers (mais pas bien compliqués) étant vus, je crois que vous avez suffisamment de bagage théorique pour commencer à pratiquer vraiment le C++.

C'est justement l'objectif de la partie suivante, qui va porter sur la librairie Qt dont je vous parle depuis un petit moment maintenant 😊.

Cette librairie est vraiment immense et va vous permettre entre autres choses de créer des fenêtres afin de rendre vos applications bien plus sympathiques d'utilisation.

Vous en avez bavé pendant cette partie, vous avez dû emmagasiner pas mal de nouvelles connaissances, aussi vous pouvez considérer que la partie qui va suivre est la... récompense 😊.

Tout ce que vous avez appris jusqu'ici va vous resservir, donc n'hésitez pas à relire les chapitres de cette partie que vous n'auriez pas trop bien compris. Parfois ça se débloque au bout de quelques lectures ! Et si ça débloque pas, tant pis, passez à la pratique quand même, je suis sûr que vous comprendrez mieux tous ces concepts du C++ en travaillant sur du concret !

Partie 2 : [Pratique] Créez vos propres fenêtres avec Qt

Vous l'avez compris en lisant la partie I : la POO, ce n'est pas évident à maîtriser au début, mais ça apporte un nombre important d'avantages : le code est plus facile à réutiliser, à améliorer, et... quand on utilise une bibliothèque là c'est carrément le pied 😊.

Le but de la partie II est entièrement de pratiquer, pratiquer, pratiquer. Vous n'apprendrez pas de nouvelles notions théoriques ici, mais par contre vous allez apprendre à maîtriser le C++ par la pratique, et ça c'est important.



Qt est une bibliothèque C++ très complète qui vous permet notamment de créer vos propres fenêtres, que vous soyez sous Windows, Linux ou Mac OS. Tout ce que nous allons faire sera très concret : ouverture de fenêtres, ajout de boutons, création de menus, de listes déroulantes... bref que des choses motivantes ! 😊

Introduction à Qt

Les amis, le temps n'est plus aux bavardages mais au concret !

Vous trouverez difficilement plus concret que cette partie II du cours 😊

Pour bien pouvoir comprendre cette partie, il est vital que vous ayez lu et compris la plupart de la partie I.

Si certaines zones de la première partie vous sont encore un peu obscures, n'hésitez pas à y faire un tour à nouveau. Au pire des cas, si vraiment ça ne rentre pas, vous pouvez quand même lire cette partie, vous aurez peut-être un déclic en pratiquant 😊

Nous commencerons dans un premier temps par découvrir ce qu'est Qt concrètement, ce que cette bibliothèque permet de faire, et quelles sont aussi les alternatives qui existent (car il n'y a pas qu'avec Qt qu'on peut créer des fenêtres !).

Nous verrons ensuite comment installer et configurer Qt.

Préparez-vous bien, parce que dès le chapitre suivant on attaque dare-dare !

Dis papa, comment on fait des fenêtres ?

Voilà une question que vous vous êtes tous déjà posés, j'en suis sûr ! J'en mettrai même ma main à couper (et j'y tiens à ma main, c'est vous dire 😱).

Alors alors, c'est comment qu'on programme des fenêtres ? 😊

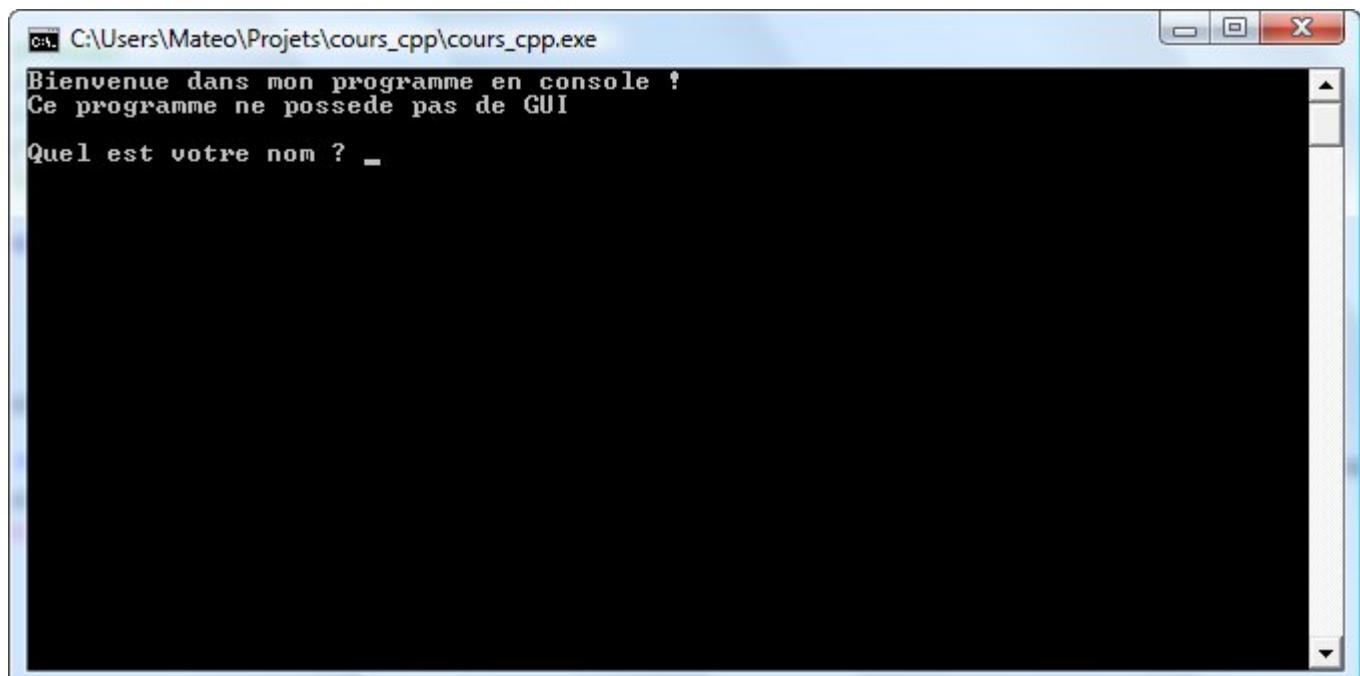
Doucement, pas d'impatience. Si vous allez trop vite vous risquez de brûler des étapes et de vous retrouver bloqué après, alors allez-y progressivement et dans l'ordre en écoutant bien tout ce que j'ai à vous dire.

Un mot de vocabulaire à connaître : GUI

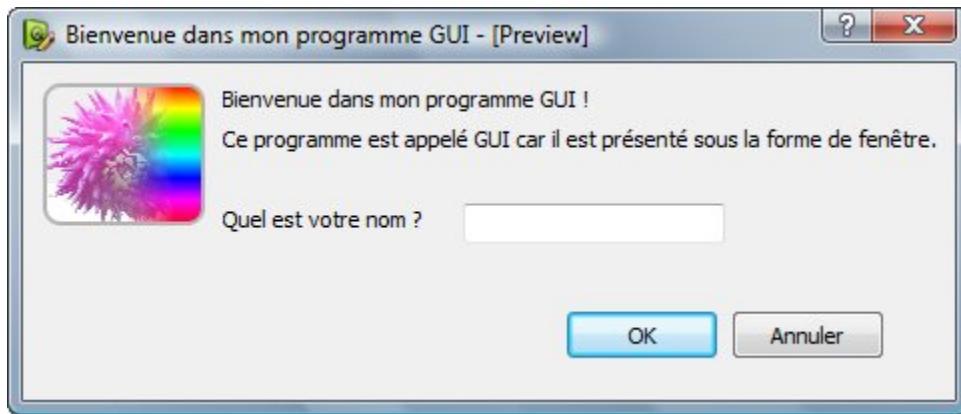
Avant d'aller plus loin, je voudrais vous faire apprendre ce petit mot de vocabulaire car je vais le réutiliser tout au long de cette partie GUI (prononcez "Goui").

C'est l'abréviation de Graphical User Interface, soit "Interface utilisateur graphique". Ca désigne tout ce qu'on appelle grossièrement "Programme avec des fenêtres".

Pour bien que vous puissiez comparer, voici un programme sans GUI (en console) et un programme GUI :



Programme sans GUI (console)



Programme GUI, ici sous Windows Vista

Les différents moyens de créer des GUI

Chaque système d'exploitation (Windows, Mac OS, Linux...) propose au moins un moyen de créer des fenêtres... le problème, c'est justement que ce moyen n'est en général pas portable, c'est-à-dire que votre programme créé uniquement pour Windows ne pourra marcher que sous Windows et pas ailleurs.

On a grossièrement 2 types de choix :

- Soit on écrit son application spécialement pour l'OS qu'on veut, mais le programme ne sera pas portable.
- Soit on utilise une bibliothèque qui s'adapte à tous les OS, c'est-à-dire une bibliothèque multi-plateforme.

La deuxième solution est en général la meilleure car c'est la plus souple. C'est d'ailleurs celle que nous allons choisir pour que personne ne se sente abandonné.

Histoire d'être suffisamment complet quand même, je vais dans un premier temps vous parler des bibliothèques propres aux principaux OS pour que vous connaissiez au moins leurs noms.

Ensuite, nous verrons quelles sont les principales bibliothèques multi-plateforme.

Les bibliothèques propres aux OS

Chaque OS propose au moins une bibliothèque qui permet de créer des fenêtres. Le défaut de cette méthode est qu'en général cette bibliothèque ne marche que pour l'OS pour lequel elle a été créée. Ainsi, si vous utilisez la bibliothèque de Windows, votre programme ne marchera que sous Windows.

- Sous Windows : on dispose de ce qu'on appelle l'API Win32. C'est une bibliothèque utilisable dans tous les langages (C, C++, Java, Python...) qui vous permet de créer des fenêtres sous Windows. Elle est toutefois assez complexe et il faut beaucoup de lignes de code pour arriver à ouvrir une fenêtre 😞.
L'API Win32 est un ensemble de fonctions. Ce n'est pas une bibliothèque qui utilise la POO. Pour palier ce problème, Microsoft a créé une autre bibliothèque appelée MFC. La MFC est une bibliothèque orientée objet qui se contente en fait d'appeler les fonctions de l'API Win32 (on dit que c'est une surcouche).
Ces bibliothèques tendent aujourd'hui à disparaître sous Windows, progressivement remplacées par la bibliothèque .NET qui est multi-plateforme et dont on reparlera donc un peu plus loin.
[> Tutoriel API Win32 réalisé par des membres du Site du Zéro](#)
- Sous Mac OS X : la bibliothèque de prédilection s'appelle Cocoa. On l'utilise en général en langage "Objective C". C'est une bibliothèque orientée objet.
- Sous Linux : tous les environnements de bureaux (appelés WM, Windows Managers) reposent sur X, la base des interfaces graphiques de Linux. X propose une bibliothèque appelée Xlib, mais on programme rarement en Xlib sous Linux. On préfère utiliser une bibliothèque plus simple d'utilisation et multi-plateforme comme GTK+ ou Qt.

Comme vous le voyez, il y a en gros une bibliothèque "de base" pour chaque OS.

L'API Win32 et la Xlib proposent des fonctions de bas niveau. Il faut en général beaucoup de lignes de code avant d'avoir un rendu

correct.

Quant à Cocoa, c'est une bibliothèque orientée objet qu'on ne peut utiliser que dans un langage orienté objet (traditionnellement Objective C, mais aussi C++, Python, Ruby...).

Ces bibliothèques ont le gros défaut de ne marcher que sur le système pour lequel elles ont été conçues et d'être relativement complexes, notamment l'API Win32 et la Xlib. Heureusement, il existe un grand nombre de bibliothèques multi-plateforme qui s'adaptent à tous les OS.

Les bibliothèques multi-plateforme

Les avantages d'utiliser une bibliothèque multi-plateforme sont nombreux. Même si vous voulez créer des programmes pour Windows et que vous n'en avez rien à faire de Linux et Mac OS, oui oui 😊

- Tout d'abord, elles simplifient grandement la création d'une fenêtre. Il faut beaucoup moins de lignes de code pour ouvrir une "simple" fenêtre.
- Ensuite, elles uniformisent le tout, elles forment un ensemble cohérent qui fait qu'il est facile de s'y retrouver. Les noms des fonctions et des classes sont choisis de manière logique de manière à vous aider autant que possible.
- Enfin, elles font abstraction du système d'exploitation mais aussi de la version du système. Cela veut dire que si demain l'API Win32 cesse d'être utilisable sous Windows, votre application continuera à fonctionner car la bibliothèque multi-plateforme s'adaptera aux changements.

Bref, choisir une bibliothèque multi-plateforme, ce n'est pas seulement pour que le programme marche partout, mais aussi pour être sûr qu'il marchera tout le temps et pour avoir un certain confort en programmant.

Voici quelques-unes des principales bibliothèques multi-plateforme à connaître, au moins de nom :

- .NET (prononcez "Dot Net") : c'est en quelque sorte le successeur de l'API Win32. On l'utilise souvent en langage C#, un langage créé par Microsoft qui ressemble à Java (il ressemble plus à Java qu'au C++ d'ailleurs 🎉). On peut néanmoins utiliser .NET dans une multitude d'autres langages dont le C++. .NET est portable car Microsoft a expliqué son fonctionnement. Ainsi, on peut utiliser un programme écrit en .NET sous Linux avec [Mono](#). Pour le moment néanmoins, .NET est principalement utilisé sous Windows.
- GTK+ : une des plus importantes bibliothèques utilisées sous Linux. Elle est portable, c'est-à-dire utilisable sous Linux, Mac OS et Windows. GTK+ est utilisable en C. Néanmoins, il existe une version C++ appelée GTKmm (on parle de wrapper, ou encore de surcouche). GTK+ est la bibliothèque de prédilection pour ceux qui écrivent des applications pour Gnome sous Linux, mais elle fonctionne aussi sous KDE. C'est la bibliothèque utilisée par Firefox par exemple, pour ne citer que lui.
[> Tutoriel GTK+ réalisé par des membres du Site du Zéro](#)
- Qt : bon je ne vous la présente pas trop longuement ici car tout ce chapitre est là pour ça 🎉 Sachez néanmoins que Qt est très utilisée sous Linux aussi, en particulier sous l'environnement de bureau KDE.
- wxWidgets : une bibliothèque objet très complète elle aussi, comparable en gros à Qt. Sa licence est très semblable à celle de Qt (elle vous autorise à créer des programmes propriétaires). Néanmoins, j'ai choisi quand même de vous montrer Qt car cette bibliothèque est plus facile à prendre en main au début. Sachez qu'une fois qu'on l'a prise en main, wxWidgets n'est pas beaucoup plus compliquée que Qt. wxWidgets est la bibliothèque utilisée pour réaliser le GUI de l'IDE Code::Blocks.
- FLTK : contrairement à toutes les bibliothèques "poids lourd" précédentes, FLTK se veut légère. C'est une petite bibliothèque dédiée uniquement à la création d'interfaces graphiques multi-plateforme.

Comme vous le voyez, j'ai dû faire un choix parmi tout ça 😊

Je sais que certains vont me reprocher le choix de Qt par rapport à wxWidgets. Oui j'ai hésité un temps entre les 2, car ce sont 2 très bonnes bibliothèques, mais Qt a finalement gagné car elle est facile à prendre en main. C'est donc une bibliothèque plus "pédagogique" en quelque sorte 🎉

Vous l'avez compris, Qt est une bibliothèque multi-plateforme pour créer des GUI (programme sous forme de fenêtre). Qt est écrite en C++ et est faite pour être utilisée à la base en C++, mais il est aujourd'hui possible de l'utiliser dans d'autres langages comme Java, Python, etc.

Plus fort qu'une bibliothèque : un framework

Qt est en fait... bien plus qu'une bibliothèque. C'est un ensemble de bibliothèques. Le tout est tellement énorme qu'on parle d'ailleurs plutôt de framework : cela signifie que vous avez à votre disposition un ensemble d'outils pour développer vos programmes plus efficacement.

Qu'on ne s'y trompe pas : Qt est à la base faite pour créer des fenêtres, c'est en quelque sorte sa fonction centrale. Mais ce serait dommage de limiter Qt à ça.



Qt est donc constituée d'un ensemble de bibliothèques, appelées "modules". On peut y trouver entre autres ces fonctionnalités :

- Module GUI : c'est toute la partie création de fenêtres. Nous nous concentrerons surtout sur le module GUI dans ce cours.
- Module OpenGL : Qt peut ouvrir une fenêtre contenant de la 3D gérée par OpenGL.
- Module de dessin : pour tous ceux qui voudraient dessiner dans leur fenêtre (en 2D), le module de dessin est très complet !
- Module réseau : Qt fournit une batterie d'outils pour accéder au réseau, que ce soit pour créer un logiciel de Chat, un client FTP, un client BitTorrent, un lecteur de flux RSS...
- Module SVG : possibilité de créer des images et animations vectorielles, à la manière de Flash.
- Module de script : Qt supporte le Javascript (ou ECMAScript), que vous pouvez réutiliser dans vos applications pour ajouter des fonctionnalités, sous forme de plugins par exemple.
- Module XML : pour ceux qui connaissent le XML, c'est un moyen très pratique d'échanger des données avec des fichiers formés à l'aide de balises, un peu comme le XHTML.
- Module SQL : permet un accès aux bases de données (MySQL, Oracle, PostgreSQL...).

Que les choses soient claires : Qt n'est pas gros, Qt est **énorme**, et il ne faut pas compter sur un tutoriel pour vous expliquer tout ce qu'il y a à savoir sur Qt. Je vais vous montrer beaucoup de ses possibilités mais on ne pourra jamais tout voir. On se concentrera surtout sur la partie GUI.

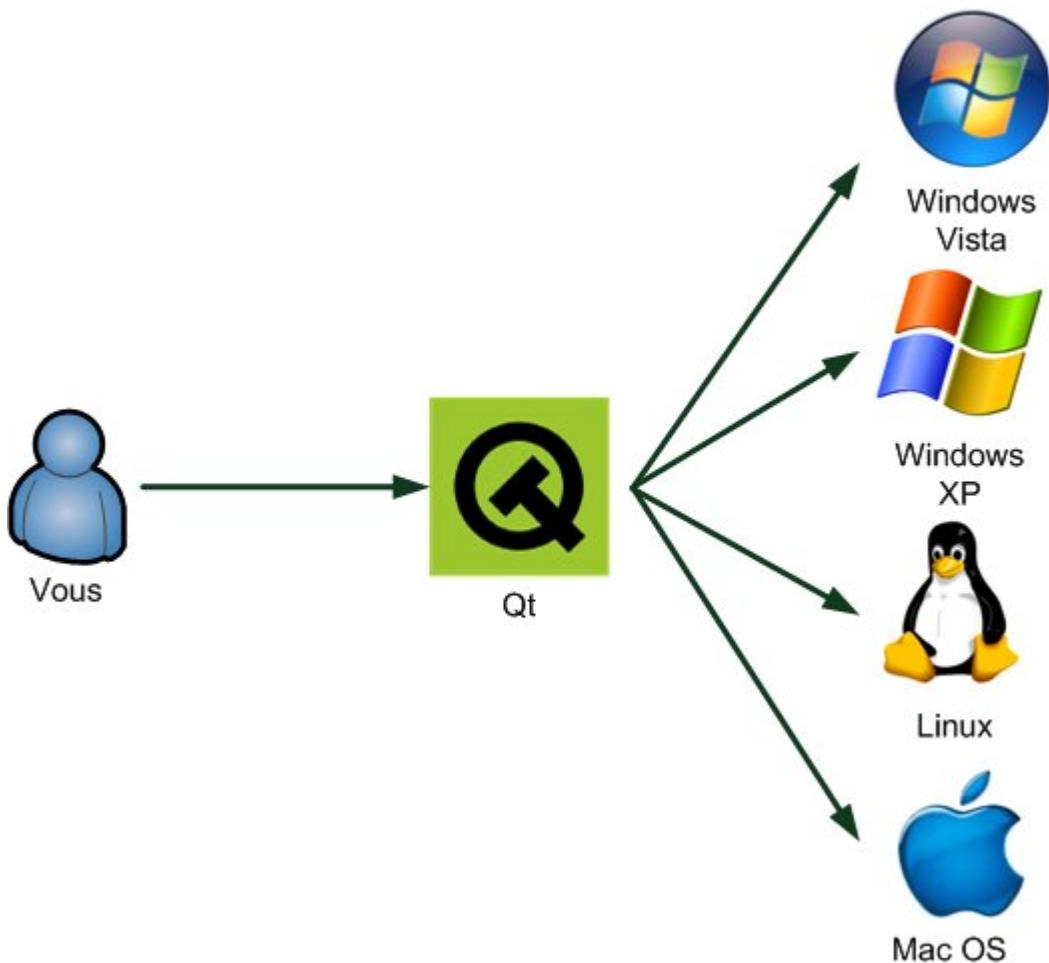
Pour ceux qui veulent aller plus loin, il faudra lire la [documentation officielle](#) (uniquement en anglais, comme toutes les documentations pour les programmeurs de toute façon). Cette documentation est très bien faite, elle détaille toutes les fonctionnalités de Qt, même les plus récentes.

Sachez d'ailleurs que j'ai choisi Qt en grande partie parce que sa documentation est très bien faite et facile à utiliser. Vous aurez donc intérêt à vous en servir 😊

Si vous êtes perdu ne vous en faites pas, je vous expliquerai dans un prochain chapitre comment on fait pour "lire" et naviguer dans une telle documentation.

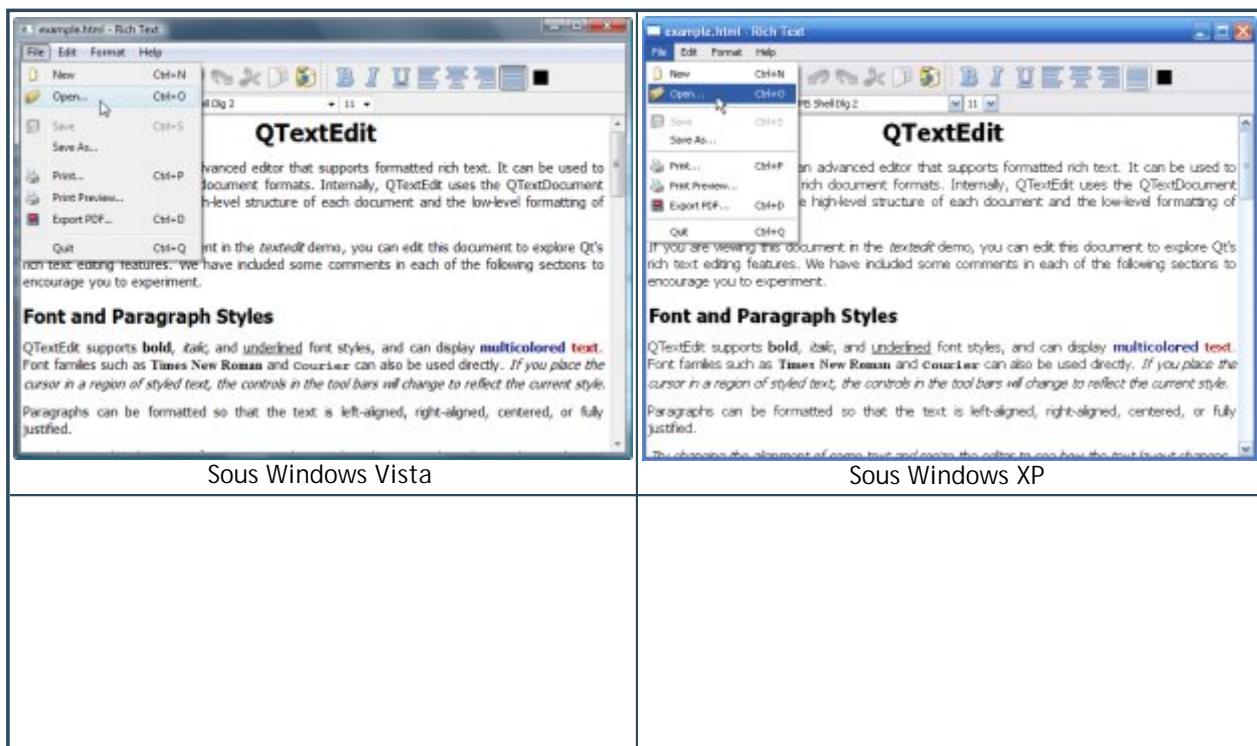
Qt est multiplateforme

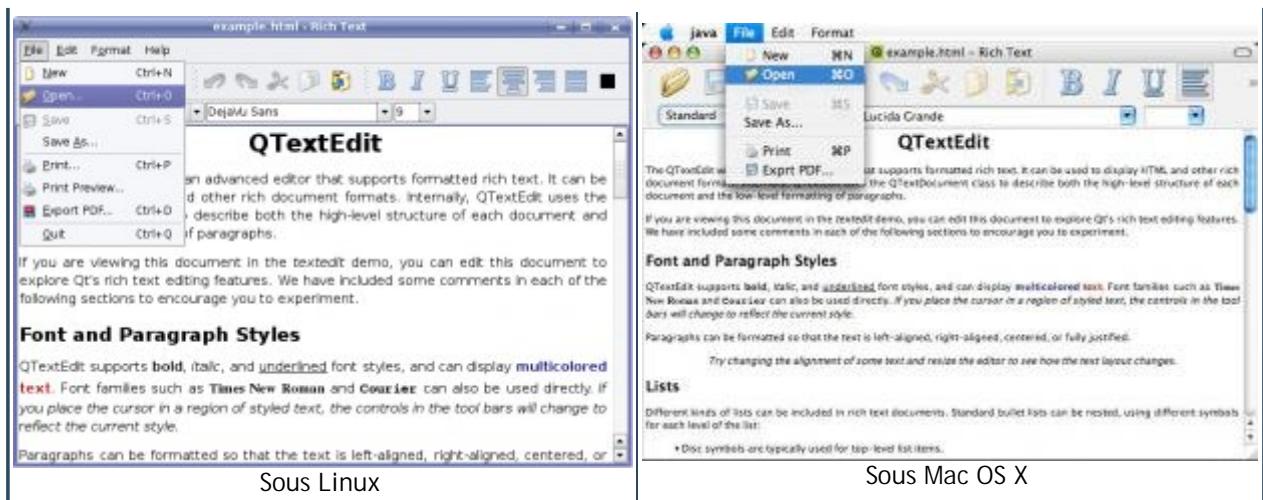
Qt est un framework multiplateforme. Je le sais je me répète, mais c'est important de l'avoir bien compris. Tenez, d'ailleurs voilà un schéma qui illustre le fonctionnement de Qt :



Grâce à cette technique, les fenêtres que vous codez ont un "look" adapté à chaque OS. Vous codez pour Qt, et Qt traduit les instructions pour l'OS. Les utilisateurs de vos programmes n'y verront que du feu et ne sauront pas que vous utilisez Qt (de toute manière ils s'en moquent 😊).

Voici une démonstration de ce que je viens de vous dire. Vous avez ci-dessous le même programme, donc la même fenêtre créée avec Qt, mais sous différents OS. Vous allez voir que Qt s'adapte à chaque fois :





Tout ce que vous avez à faire pour produire le même résultat, c'est recompiler votre programme sous chacun de ces OS. Par exemple, vous avez développé votre programme sous Windows, très bien, mais les .exe n'existent pas sous Linux. Il vous suffit simplement de recompiler votre programme sous Linux et c'est bon, vous avez une version Linux !

On est obligé de recompiler pour chacun des OS ?

Oui, ça vous permet de créer des programmes binaires adaptés à chaque OS qui tournent à pleine vitesse.

On ne va toutefois pas se préoccuper de compiler sous chacun des OS maintenant, on va déjà le faire pour votre OS ça sera bien 😊

Pour information, d'autres langages de programmation comme Java et Python ne nécessitent pas de recompilation car le terme "compilation" n'existe pas vraiment sous ces langages. Cela fait que les programmes sont un peu plus lents, mais ils s'adaptent automatiquement partout.

L'avantage du C++ par rapport à ces langages est donc sa rapidité (bien que la différence se sente de moins en moins, sauf pour les jeux vidéo qui ont besoin de rapidité et qui sont donc majoritairement codés en C++) .

L'histoire de Qt

Bon, ne comptez pas sur moi pour vous faire un historique long et chiant sur Qt, mais je pense qu'un tout petit peu de culture générale ne peut pas vous faire de mal et vous permettra de savoir de quoi vous parlez 😊

Qt est un framework développé initialement par la société Trolltech, qui fut racheté par Nokia par la suite.

Le développement de Qt a commencé en 1991 (ça remonte pas mal donc) et il a été dès le début utilisé par KDE, un des principaux environnements de bureau de Linux.

Qt s'écrit "Qt" et non "QT", donc avec un "t" minuscule (si vous faites l'erreur un fanatique de Qt vous égorgera probablement pour vous le rappeler 😱)

Qt signifie "Cute" (prononcez "Quioute"), ce qui signifie "Mignonne", parce que les développeurs trouvaient que la lettre Q était jolie dans leur éditeur de texte. Oui je sais, ils sont fous ces programmeurs.

La licence de Qt

A l'origine, Qt possédait une licence propriétaire, son code source était fermé.

Heureusement, maintenant Qt est sous licence LGPL. Cela signifie, en gros, que vous pouvez l'utiliser pour faire des programmes libres ou propriétaires, selon ce que vous souhaitez.

Pendant longtemps, Qt était sous licence GPL. Cela vous obligeait à faire des programmes libres.

Toutefois, depuis son rachat par Nokia, la licence est passée à la LGPL qui est plus souple et qui vous permet de faire des programmes propriétaires.

Bref, c'est vraiment l'idéal pour nous. On peut l'utiliser gratuitement et en faire usage dans des programmes libres comme dans des

programmes propriétaires. 😊

Qui utilise Qt ?

Une bibliothèque comme Qt a besoin de références, c'est-à-dire d'entreprises célèbres qui l'utilisent, pour montrer son sérieux. De ce point de vue là, pas de problème. Qt est utilisée par de nombreuses entreprises que vous connaissez sûrement :

-  **Adobe**
- **ARCHOS**
-  **BOEING**
-  **Google**
-  **NASA**
-  **skype**

Qt est utilisée pour réaliser de nombreux GUI, comme celui d'Adobe Photoshop Elements, de Google Earth ou encore de Skype !

Installation de Qt

Vous êtes prêts à installer Qt ?

On est parti !

Télécharger Qt

Commencez par télécharger Qt sur le [site de Qt](#).

Je vous conseille de prendre le Qt SDK qui contient plus de programmes pour vous aider à développer (notamment un IDE appelé Qt Creator).

Choisissez soit "Qt pour Windows: C++", "Qt pour Linux/X11: C++" ou "Qt pour Mac: C++" en fonction de votre système d'exploitation.

Il y a 3 téléchargements possibles, en fonction de votre OS :

- Windows : si vous avez Windows, passez par là ! Dans la page qui s'affiche, choisissez de préférence le ".exe" contenant mingw (plutôt que le zip), c'est un installateur prêt à l'emploi qui contient tout ce qu'il vous faut.
- Mac : pour ceux qui sont sous Mac OS ! Je vous conseille de prendre le fichier à l'extension .dmg (ex : qt-mac-opensource-4.4.0.dmg). Ce fichier contient un installateur .mpkg qui se charge d'installer tout le framework Qt de la même manière que Windows.
- Linux/X11 : si vous êtes sous Linux (vous utilisez donc X pour l'interface graphique), c'est pour vous ! Le plus simple sous Debian et Ubuntu est quand même de passer par la console (plutôt que par le site de Qt) et de taper `sudo apt-get install libqt4-dev`, ça fait tout pour vous 😊

Il y a une version 32 bits et une version 64 bits, à choisir en fonction de votre installation.

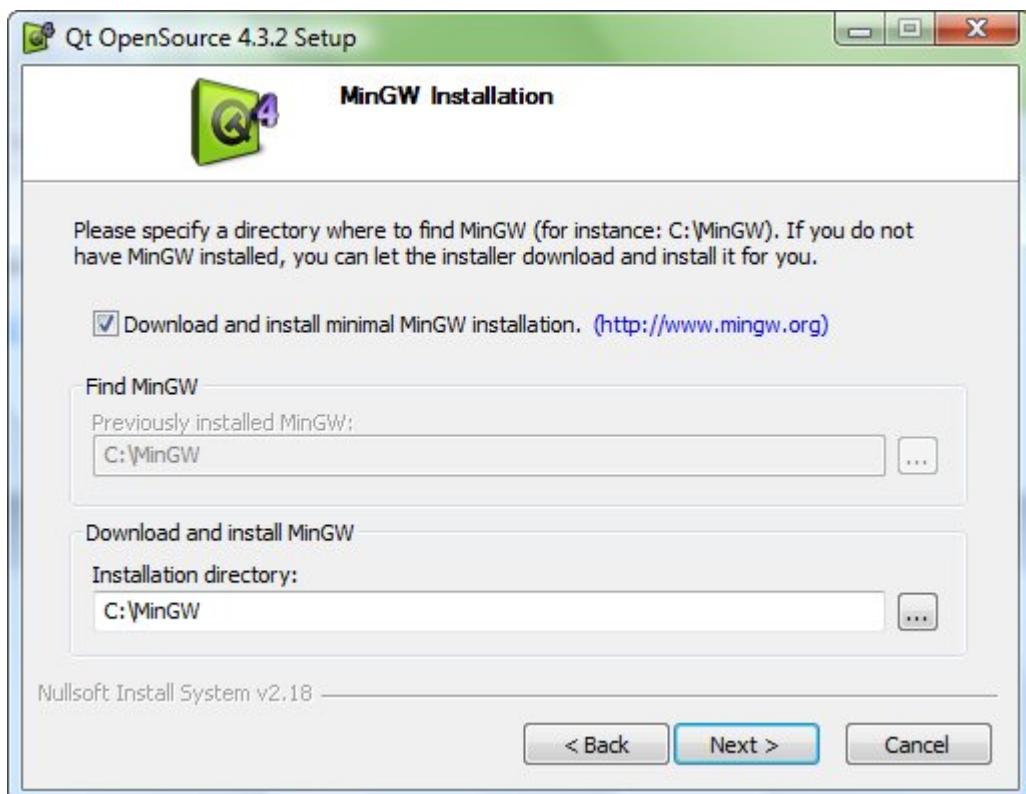
Installation sous Windows

L'installation sous Windows se présente sous la forme d'un assistant d'installation classique.
Je vais vous montrer comment ça se passe pas à pas, ce n'est pas bien compliqué.

La première fenêtre est la suivante :



Rien de particulier à signaler. Cliquez sur Next autant de fois que nécessaire en laissant les options par défaut, jusqu'à arriver à la fenêtre suivante :

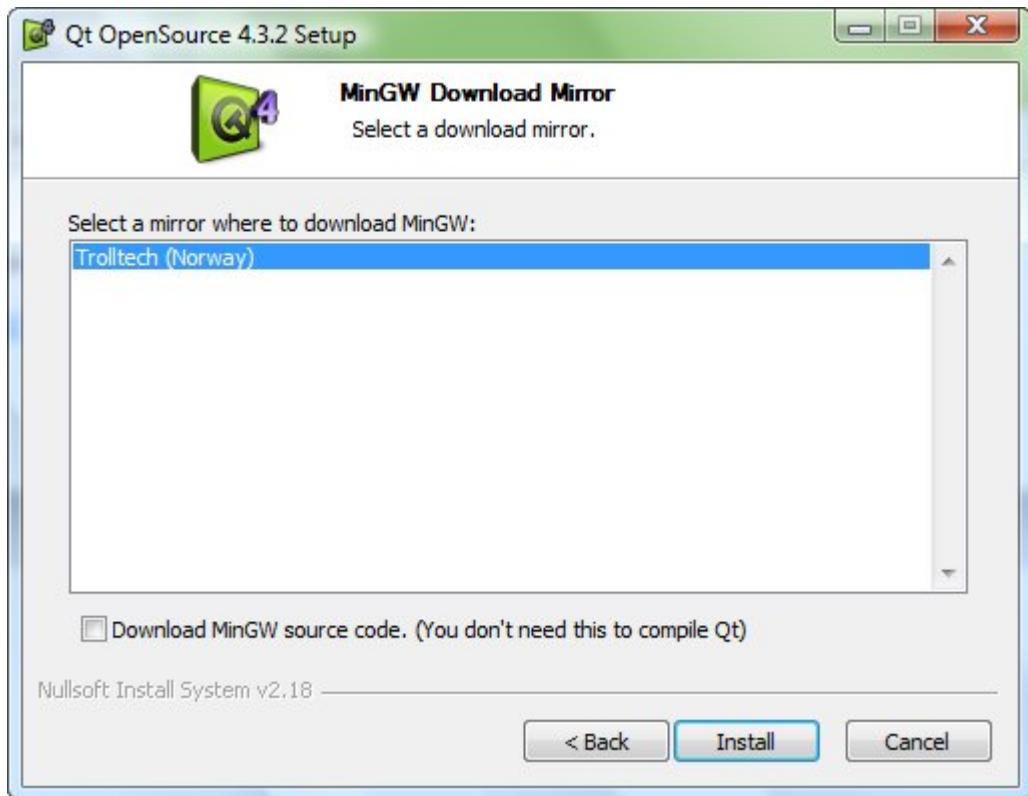


Cette fenêtre vous demande si vous voulez installer MinGW, le compilateur. Normalement, vous avez déjà installé MinGW en même temps que votre IDE, donc il est déjà sur votre disque.

Toutefois, il faut que vous ayez la bonne version de MinGW avec l'API Win32 pour que Qt puisse faire la traduction correctement. Je vous recommande donc fortement de le réinstaller (ça ne pose aucun problème) dans le répertoire par défaut proposé par l'installateur, ici C:\MinGW.

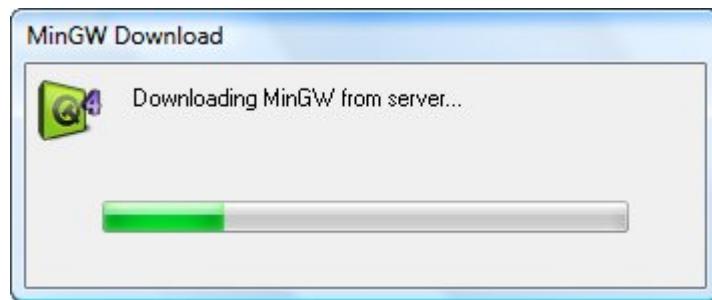
Lorsque les options sont comme chez moi, cliquez sur Next.

On vous demande alors où télécharger MinGW (sur quel miroir) :

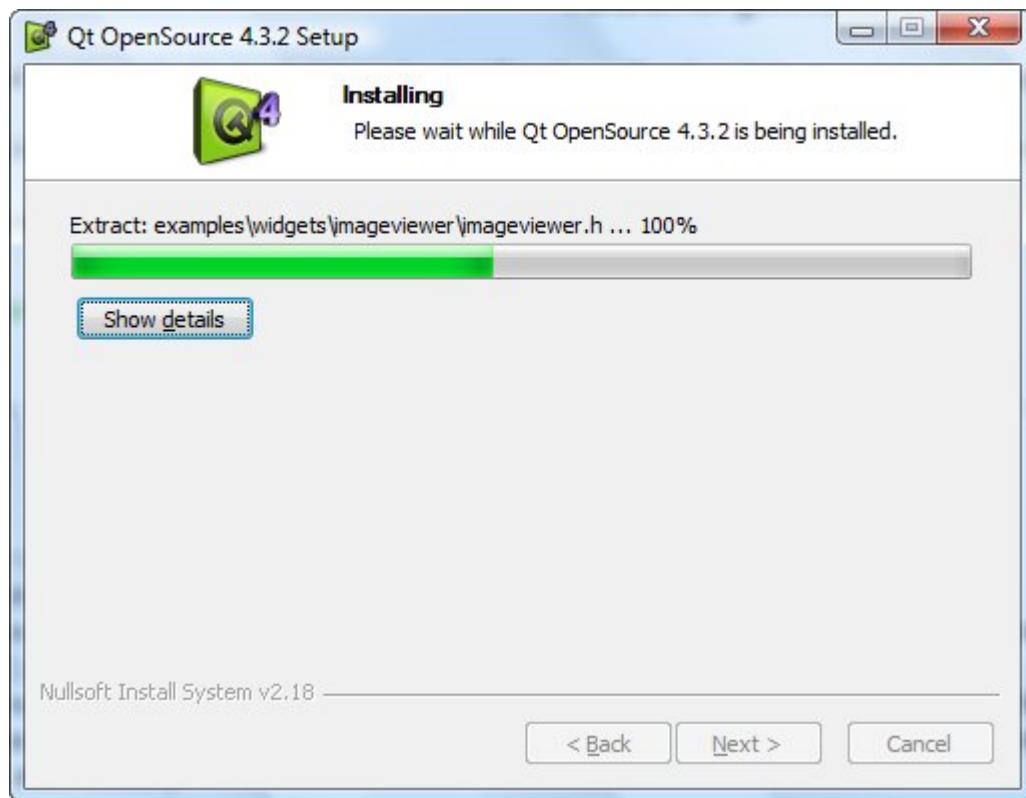


Vous n'avez pas trop le choix, vous ne pouvez télécharger MinGW que chez Qt sur leurs serveurs en Norvège 🇳🇴. Sélectionnez donc "Trolltech (Norway)", ne cochez pas "Download MinGW source code" (on n'en a pas besoin) et cliquez sur Install.

MinGW se télécharge et s'installe tout seul :



Puis Qt s'installe enfin (il y a beaucoup de fichiers ça peut prendre un peu de temps) :



Vous êtes à la fin ? Ouf !

On vous propose d'ouvrir 2 programmes installés par Qt, ouvrez-les si vous voulez.



Les programmes installés par Qt

En plus de tous les fichiers nécessaires au développement de GUI, Qt installe 4 programmes qui pourront vous être utiles par la suite.

Je vais rapidement vous les présenter mais on ne va pas rentrer dans le détail de chacun d'eux, il est trop tôt.

Qt Examples and Demos

The screenshot shows the "Text Edit" example within the Qt Examples and Demos application. On the left, a sidebar lists various examples: Affine Transformations, Arthur Plugin, Composition Modes, Gradients, Path Stroking, Vector Deformation, Books, Main Window, Spreadsheet, SQL Browser, Text Edit (which is selected), 40000 Chips, and Interview. The main area displays a window titled "example.html - Rich Text" with a menu bar (File, Edit, Format, Help) and a toolbar with icons for file operations and rich text formatting (Bold, Italic, Underline). The text area contains the heading "Tables" and the following text: "QTextEdit can arrange and format tables, supporting features such as row formatting within cells, and size constraints for columns." Below this is a table:

	Development Tools	Programming Techniques	Graphs
9:00 - 11:00		Introduction to Qt	
11:00 - 13:00	Using qmake	Object-oriented Programming	Layouts

At the bottom, there are links for "Main menu", "Launch", and "Documentation". The Trolltech logo is visible at the bottom center.

Ca c'est juste une démonstration des possibilités de Qt. Vous pouvez explorer ce programme autant que vous voulez, il est juste là pour présenter Qt.

Qt Assistant

Qt Assistant by Trolltech - Qt Reference Documentation (Open Source Edition)

File Edit View Go Bookmarks Help

Sidebar

Contents Index Bookmarks Search

Qt Assistant Manual
Qt Designer Manual
Qt Linguist Manual
Qt Reference Documentation
qmake Manual

Qt Reference Documentation (Open Source Edition)

Home · All Classes · Main Classes · Grouped Classes · Modules · Functions

TR^{OL}LTECH

Qt Reference Documentation (Open Source Edition)

Note: This edition is for the development of Free and Open Source software only; see Qt Commercial Editions.

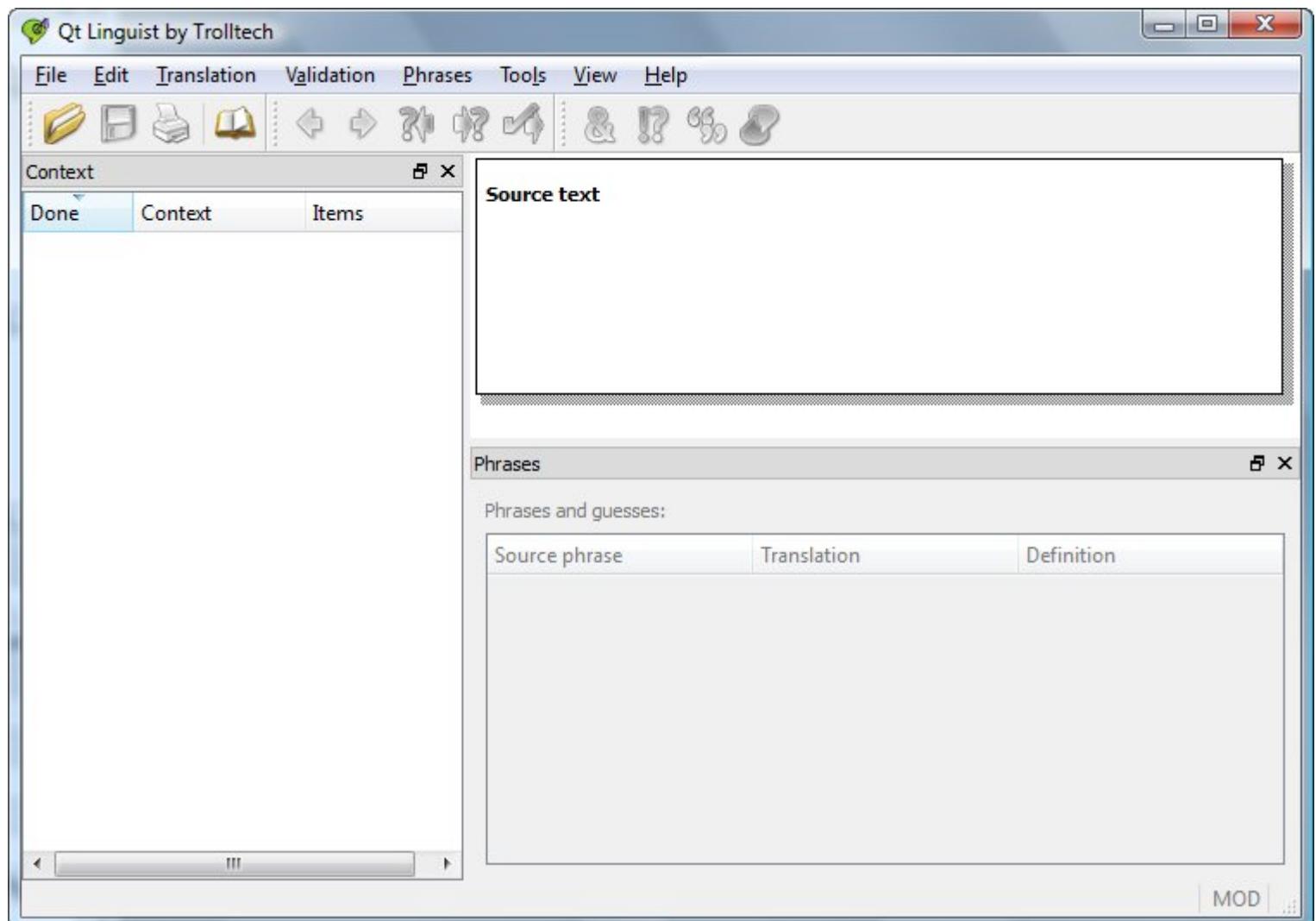
Getting Started	General	Developer Resources
<ul style="list-style-type: none">• What's New in Qt 4.3• How to Learn Qt• Installation• Tutorial and Examples• Porting from Qt 3 to Qt 4	<ul style="list-style-type: none">• About Qt• About Trolltech• Commercial Edition• Open Source Edition• Frequently Asked Questions	<ul style="list-style-type: none">• Mailing Lists• Qt Community Web Sites• Qt Quarterly• How to Report a Bug• Other Online Resources

API Reference	Core Features	Key Technologies
<ul style="list-style-type: none">• All Classes• Main Classes• Grouped Classes• Annotated Classes• Qt Classes by Module	<ul style="list-style-type: none">• Signals and Slots• Object Model• Layout Management• Paint System• Accessibility	<ul style="list-style-type: none">• Multithreaded Programming• Main Window Architecture• Rich Text Processing

Qt Assistant est la documentation de Qt. Dedans, il y a tout. Tout ce que vous avez besoin de savoir, toutes les fonctionnalités de Qt, toutes les fonctions, toutes les classes que vous pouvez utiliser.
C'est ce que vous trouverez de plus complet.

Certes, la documentation est en anglais, comme la plupart des documentations pour développeurs. Elle est néanmoins très bien faite, et savoir s'en servir est indispensable si on veut essayer d'autres choses que ce que j'expliquerai dans le tuto. Je vous apprendrai donc à la lire dans un prochain chapitre 😊

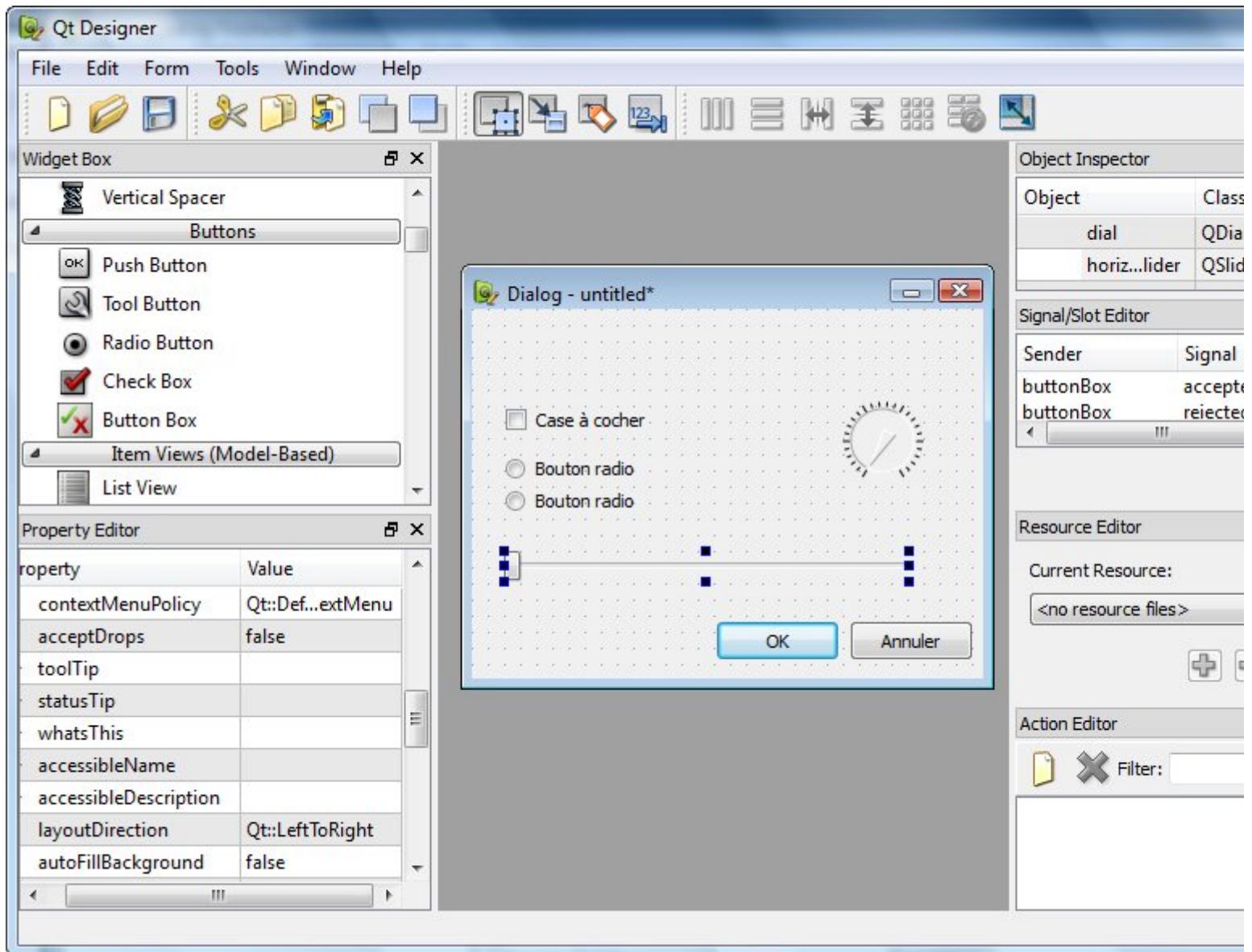
[Qt Linguist](#)



Qt Linguist est une application à destination des traducteurs. Si votre programme doit être décliné en plusieurs langues, Qt vous offre donc tous les outils dont vous pourriez avoir besoin pour les traduire.

Le gros avantage est qu'il n'y a pas besoin d'être programmeur pour traduire l'application. Il vous suffira de donner un fichier qui contient tout le texte de votre application (généré par Qt), de le donner à traduire à un traducteur avec Qt Linguist, et hop, votre application est multilingue ! 😊

Qt Designer



Qt Designer vous permet, vous l'aurez deviné, de créer les fenêtres de vos applications à la souris.

Normalement, une fenêtre se code (on peut créer une fenêtre rien qu'avec du code sans passer par Qt Designer). Qt Designer vous fera gagner du temps... lorsque vous saurez vous servir de Qt. Avant d'utiliser Qt Designer, il faut savoir coder la fenêtre à la main !

Attention Qt Designer est un piège pour les débutants ! Il est très attrayant, on pense que ça va être super simple de créer des fenêtres, mais en fait c'est bien plus complexe que cela. Vous ne DEVEZ PAS vous ruer dessus : vous ne pourriez pas vous en servir et l'exploiter correctement.

Je vous expliquerai comment fonctionne Qt Designer, mais ce sera plus tard dans le cours. Il est impératif que vous sachiez d'abord coder une fenêtre à la main, sinon vous ne pourrez pas l'utiliser correctement.

Sous ses apparences simples, Qt Designer est en fait une application complexe qu'on ne peut maîtriser que lorsqu'on a déjà de l'expérience avec Qt. Et cette expérience, on va l'acquérir au fil des chapitres qui suivent 😊

Notion de GUI... OK
 Présentation des bibliothèques GUI... OK
 Présentation des modules de Qt... OK
 Notion de framework multi-plateforme... OK
 Culture générale sur Qt... OK
 Téléchargement de Qt... OK
 Installation de Qt... OK
 Présentation des programmes livrés avec Qt... OK

- C'est bon mon commandant, ils sont parés au lancement 😊

- Ouvrez le sas et accrochez-vous lieutenant, ça risque de bouger un peu 

Compiler votre première fenêtre Qt

Bonne nouvelle, votre patience et votre persévérance vont maintenant payer 😊

Dans ce chapitre, nous réaliserons notre premier programme utilisant Qt, et nous verrons comment ouvrir notre première fenêtre !

La compilation sous Qt est un peu particulière car elle comporte plusieurs étapes. Je vais vous expliquer pourquoi la compilation avec Qt est différente et comment vous devez procéder pour compiler votre programme.

Let's go !

Codons notre première fenêtre !

Ok on est parti !

Voilà comment on va procéder : ouvrez votre IDE favori, par exemple Code::Blocks, et créez un nouveau projet console C++ comme vous le faisiez jusqu'ici. Appelez ce projet comme vous voulez, par exemple "Test". Placez votre projet dans un dossier qui ne contient pas d'espace dans le nom, c'est important pour la suite.

Sous Code::Blocks, il y a un assistant de création de nouveau projet Qt. Faites un projet console. J'insiste bien là-dessus, ne vous trompez pas 😊

Le code minimal d'un projet Qt

Votre projet est constitué normalement au départ d'un seul fichier : main.cpp.

Supprimez le code qui a pu être généré par votre IDE, et remplacez-le par celui-ci :

Code : C++ - Sélectionner

```
#include < QApplication>

int main( int argc, char *argv[] )
{
    QApplication app(argc, argv);

    return app.exec();
}
```

C'est le code minimal d'une application utilisant Qt !

Comme vous pouvez le constater, ce qui est génial c'est que c'est vraiment très court 😊

D'autres bibliothèques vous demandent beaucoup plus de lignes de code avant de pouvoir commencer à programmer, tandis qu'avec Qt c'est vraiment très simple et rapide 😊

Analysons ce code pas à pas !

Includes un jour, includes toujours

Code : C++ - Sélectionner

```
#include < QApplication>
```

C'est le seul include que vous avez besoin de faire au départ. Vous pouvez oublier iostream et compagnie, avec Qt on ne s'en sert

plus.

Vous noterez qu'on ne met pas l'extension ".h", c'est voulu. Faites exactement comme moi.

Cet include vous permet d'accéder à la classe QApplication, qui est la classe de base de tout programme Qt.

QApplication, la classe de base

Code : C++ - [Sélectionner](#)

```
QApplication app(argc, argv);
```

La première ligne du main crée un nouvel objet de type QApplication. On a fait ça tout le long des derniers chapitres, vous ne devriez pas être surpris 😊

Cet objet est appelé app (mais vous pouvez l'appeler comme vous voulez). Le constructeur de QApplication exige que vous lui passiez les arguments du programme, c'est-à-dire les paramètres argc et argv que reçoit la fonction main. Cela permet de démarrer le programme avec certaines options précises, mais on ne s'en servira pas ici.

Lancement de l'application

Code : C++ - [Sélectionner](#)

```
return app.exec();
```

Cette ligne fait 2 choses :

1. Elle appelle la méthode exec de notre objet app. Cette méthode démarre notre programme. Si vous ne le faites pas il ne se passera rien.
2. Elle retourne le résultat de app.exec() pour dire si le programme s'est bien déroulé ou pas. Le return provoque la fin de la fonction main, donc du programme.

C'est un peu du condensé en fait 🤪

Ce que vous devez vous dire, c'est qu'en gros tout notre programme s'exécute à partir de ce moment-là. La méthode exec est gérée par Qt : tant qu'elle s'exécute, notre programme est ouvert. Dès que la méthode exec est terminée, notre programme s'arrête.

Affichage d'un widget

Dans la plupart des bibliothèques GUI, dont Qt fait partie, tous les éléments d'une fenêtre sont appelés des widgets. Les boutons, les cases à cocher, les images... tout ça ce sont des widgets. La fenêtre elle-même est considérée comme un widget.

Pour provoquer l'affichage d'une fenêtre, il suffit de demander à afficher n'importe quel widget. Ici par exemple, nous allons afficher un bouton.

Voici le code complet que j'aimerais que vous utilisiez. Il utilise le code de base de tout à l'heure mais y ajoute quelques lignes :

Code : C++ - [Sélectionner](#)

```

#include <QApplication>
#include <QPushButton>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QPushButton bouton("Salut les Zéros, la forme ?");
    bouton.show();

    return app.exec();
}

```

Les lignes ajoutées ont été surlignées pour bien que vous puissiez les repérer.
On voit entre autres :

Code : C++ - Sélectionner

```
#include <QPushButton>
```

Cette ligne va vous permettre de créer des objets de type QPushButton, c'est-à-dire des boutons (vous noterez que sous Qt toutes les classes commencent par un "Q" d'ailleurs !).

Code : C++ - Sélectionner

```
QPushButton bouton("Salut les Zéros, la forme ?");
```

Cela crée un nouvel objet de type QPushButton que nous appelons tout simplement bouton, mais on aurait très bien pu l'appeler autrement. Le constructeur attend un paramètre : le texte qui sera affiché sur le bouton.

Malheureusement, le fait de créer un bouton ne suffit pas pour qu'il soit affiché. Il faut appeler sa méthode show :

Code : C++ - Sélectionner

```
bouton.show();
```

Et voilà !

Cette ligne commande l'affichage d'un bouton. Comme un bouton ne peut pas "flotter" comme ça sur votre écran, Qt l'insère automatiquement dans une fenêtre. On a en quelque sorte créé une "fenêtre-bouton" 😊

Bien entendu, dans un vrai programme plus complexe, on crée d'abord une fenêtre et on y insère ensuite plusieurs widgets, mais là on commence simplement 😊

Notre code est prêt, il ne reste plus qu'à compiler et exécuter le programme !



... mais, pour compiler avec Qt c'est un peu particulier. On ne pourra pas se contenter de cliquer sur le bouton "Compiler" de l'IDE. Je vais maintenant vous expliquer comment procéder pour compiler avec Qt, et je vous rassure ça n'a rien de compliqué, c'est juste un peu différent !

Compiler un projet Qt : la théorie

La bibliothèque Qt est tellement importante qu'elle apporte quelques ajouts au langage C++, en particulier le mécanisme des signaux et slots dont on reparlera un peu plus loin.

Pour ajouter ces fonctionnalités au langage C++, la compilation sort du schéma classique. On va en profiter ici pour revoir (ou voir

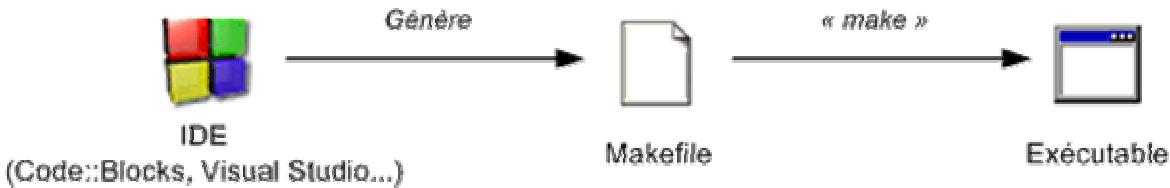
pour certains 😊) comment la compilation fonctionne d'habitude dans les grandes lignes, puis nous découvrirons comment il faut compiler un programme utilisant Qt.

La compilation "normale", sans Qt

Savez-vous vraiment ce qui se passe lorsque vous cliquez sur "Compiler" dans votre IDE favori (Code::Blocks, Visual Studio...) ? Il y a en fait 2 grosses étapes :

1. L'IDE regarde la liste des fichiers de votre projet (.cpp et .h) et génère un fichier appelé Makefile qui contient la liste des fichiers à compiler pour le compilateur.
2. Ensuite, l'IDE appelle le compilateur (via le programme make). L'utilitaire make recherche un fichier Makefile et l'utilise pour savoir quoi compiler et avec quelles options.

Schématiquement ça donne ça :



En fait, le gros avantage de l'IDE c'est qu'il écrit le fichier Makefile pour vous. On peut écrire le Makefile à la main, mais c'est honnêtement pas très pratique ni toujours très simple, surtout pour de gros projets. En effet, le fichier Makefile est parfois très gros.

Voici un aperçu (raccourci) d'un fichier Makefile pour vous donner une idée :

Code : Autre - [Sélectionner](#)

```
all: $(PROG)

$(PROG): $(OBJS)
    $(CC) $(CFLAGS) $(LDFLAGS) $(LDLIBS) -o $(PROG) $(OBJS)

.c.o:
    $(CC) $(CFLAGS) -c $*.c
```

Qu'un IDE génère le Makefile pour nous n'est donc pas du luxe 😊

La compilation particulière avec Qt

Le problème survient quand vous utilisez des bibliothèques importantes comme Qt. Il faut configurer votre IDE pour qu'il puisse écrire le Makefile correctement, c'est-à-dire indiquer l'emplacement des fichiers .a (ou .lib), des headers de la bibliothèque, etc.

On pourrait en théorie configurer votre IDE pour que ça marche en cliquant sur "Compiler"... mais ce serait un peu long et compliqué (il faudrait une explication par IDE, et parfois par version d'IDE). J'ai à la place choisi de vous montrer une technique universelle : [passer par la ligne de commande ! 😊](#)



Bah pourquoi tout le monde est parti ?

Je vous rassure, ce n'est pas aussi infaisable que ce que vous pouvez croire, ce sera même plus simple que de configurer notre IDE, c'est vous dire 😊

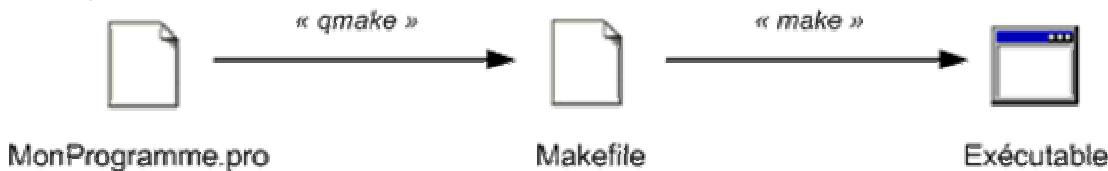
Qt est livré avec un petit programme en ligne de commande appelé "qmake". Ce programme est capable de générer un fichier

Makefile à partir d'un fichier spécifique à Qt : le .pro.

Le .pro (qui s'appelle en général nomDeVotreProjet.pro) est un fichier texte court et simple à écrire qui donne la liste de vos fichiers .cpp et .h, ainsi que les options à envoyer à Qt.

Sous Linux, la commande n'est pas qmake mais qmake-qt4.

Ca se passe donc comme ça avec Qt :



... Je suis obligé d'écrire moi-même le .pro ? Je ne sais pas faire ! Et puis faire la liste des fichiers du projet ça peut être long non ?



Rassurez-vous, Qt peut vous générer un .pro automatiquement ! Si on utilise d'abord qmake avec l'option -project dans le dossier de notre projet, qmake va analyser les fichiers du dossier et générer un fichier .pro basique (mais suffisant pour nous pour le moment).



En résumé, pour compiler avec Qt il y a 3 commandes très simples à taper en console. Dans l'ordre :

1. qmake -project
2. qmake
3. make (sous Linux) ou mingw32-make (sous Windows)

Attention ! Sous Windows, dans les dernières versions de Qt, il faut taper mingw32-make et non make si vous avez installé Mingw avec Qt !

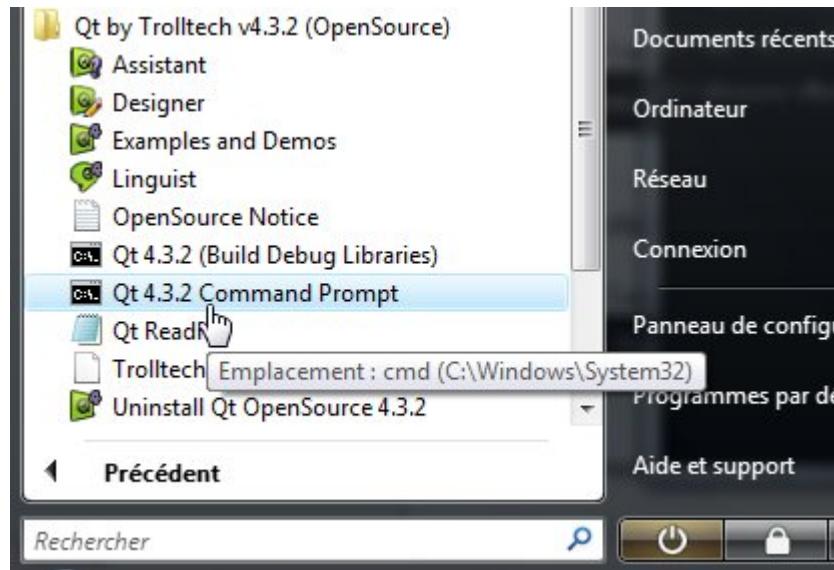
Normalement, il n'est nécessaire de taper les 2 premières commandes (qmake -project et qmake) que la première fois pour générer le Makefile. Ensuite, vous n'aurez plus besoin que de relancer make pour recompiler votre projet.

Il faudra en fait relancer les commandes qmake -project et qmake à chaque fois que votre projet évoluera, c'est-à-dire à chaque fois que de nouveaux fichiers .cpp et .h seront ajoutés ou supprimés. Tant que la liste des fichiers de votre projet ne change pas, il n'est pas nécessaire de retaper ces 2 premières commandes.

Compiler un projet Qt : la pratique

Bien, si on la compilait cette fenêtre ? 😊

Pour commencer, il faut lancer une console. Sous Windows, Qt vous a normalement fait un raccourci dans le menu démarrer appelé "Qt Command Prompt" :



Attention : vous savez peut-être lancer la console sous Windows en passant par un autre raccourci ou encore en faisant Démarrer / Exécuter / "cmd.exe". Mais ici, vous devez utiliser ce raccourci car il précharge certaines informations relatives à Qt.

Normalement, la console s'ouvre et affiche ces informations :

Code : Console - [Sélectionner](#)

```
Setting up a MinGW/Qt only environment...
-- QTDIR set to C:\Qt\4.3.2
-- PATH set to C:\Qt\4.3.2\bin
-- Adding C:\MinGW\bin to PATH
-- Adding C:\WINDOWS\System32 to PATH
-- QMAKESPEC set to win32-g++
```

```
C:\Qt\4.3.2>
```

Les premières lignes sont importantes. Elles signifient que dans cette console le compilateur connaîtra la position de la bibliothèque Qt ainsi que de ses headers. Ca nous enlève beaucoup beaucoup de maux de tête avec les paramétrages 😊

Pour l'instant, la console indique que nous sommes dans le dossier C:\Qt\4.3.2 (le chemin peut changer en fonction de la version de Qt mais ce n'est pas grave du tout).

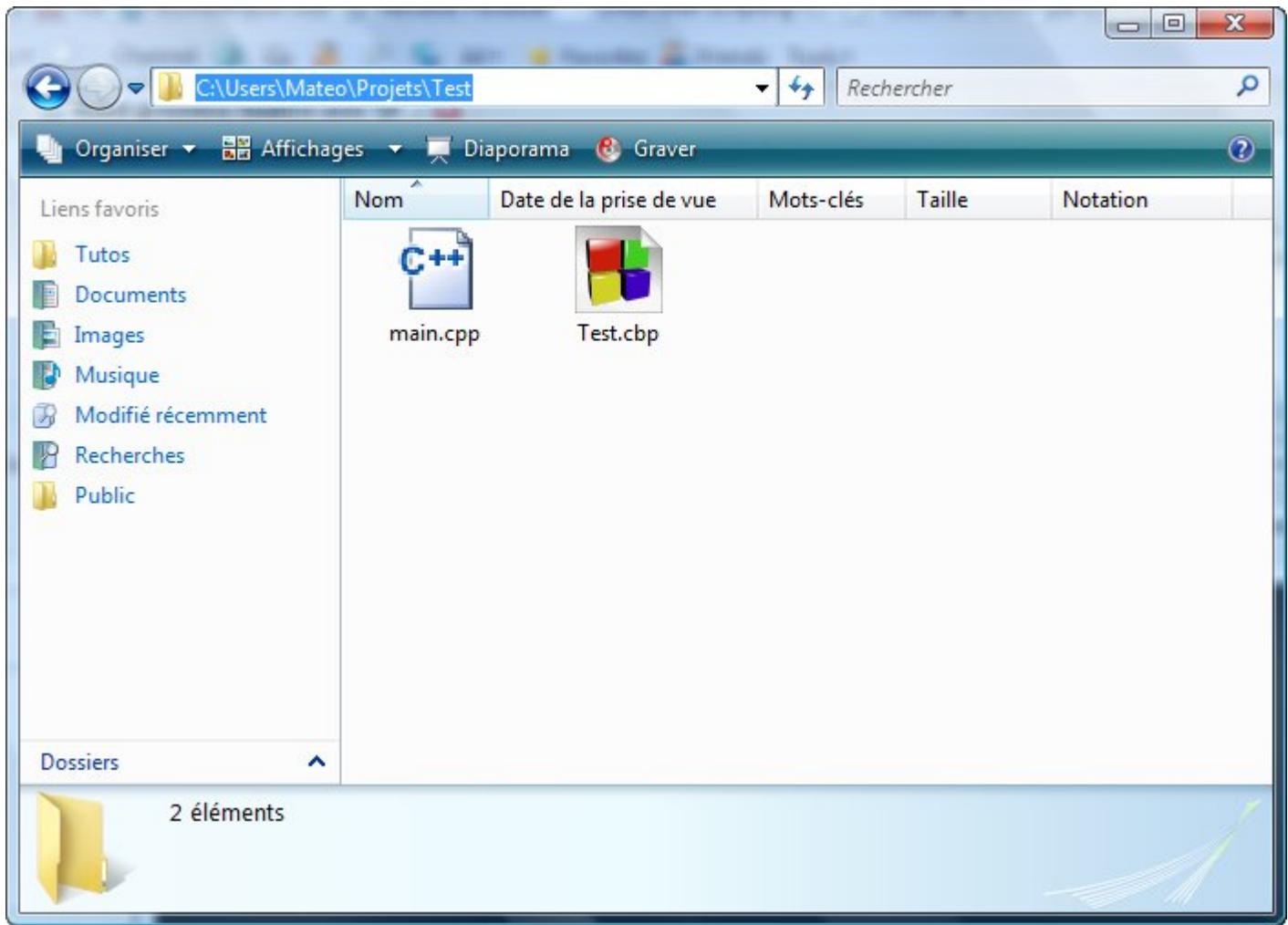
Je veux que vous vous rendiez dans le dossier où est enregistré votre projet. Pour cela, le plus simple est de taper dans la console :

Code : Console - [Sélectionner](#)

```
cd CheminDeVotreProjet
```

Attention : vous ne pourrez pas compiler avec make si le nom du dossier de votre projet comporte un espace. Veillez à placer votre projet dans un dossier qui ne contient pas d'espace.

Si vous copiez-collez le chemin depuis l'explorateur de Windows ça devrait aller très vite (il faudra faire un clic droit pour coller le texte dans la console).



Dans mon cas j'écrirai donc la commande suivante :

Code : Console - [Sélectionner](#)

```
cd C:\Users\Mateo\Projets\Test
```

Tapez Entrée. Vous devriez vous retrouver dans le dossier de votre projet.

Etape 1 : générer le .pro (qmake -project)

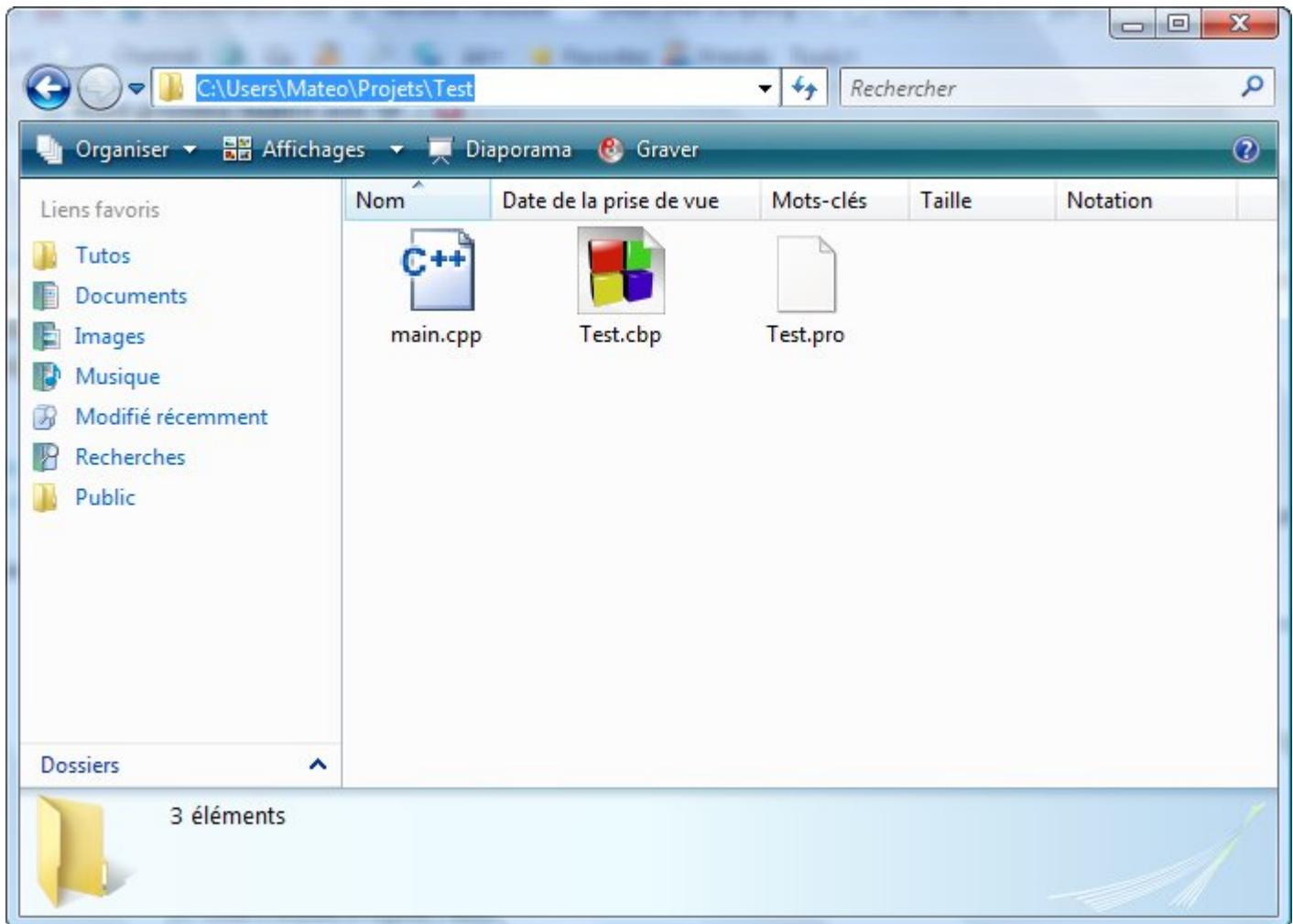
On va commencer par générer le fichier .pro automatiquement. Vous vous souvenez de la commande ? 😊

Code : Console - [Sélectionner](#)

```
C:\Users\Mateo\Projets\Test>qmake -project
```

```
C:\Users\Mateo\Projets\Test>
```

Si tout se passe bien, rien de spécial ne devrait s'afficher après avoir tapé la commande. Par contre, si vous regardez le dossier de votre projet, il contient maintenant un fichier .pro :



On ne va pas étudier le contenu du fichier .pro maintenant, mais sachez que vous pouvez l'ouvrir avec un éditeur de texte comme Bloc-Notes sans problème.

Passons à l'étape suivante !

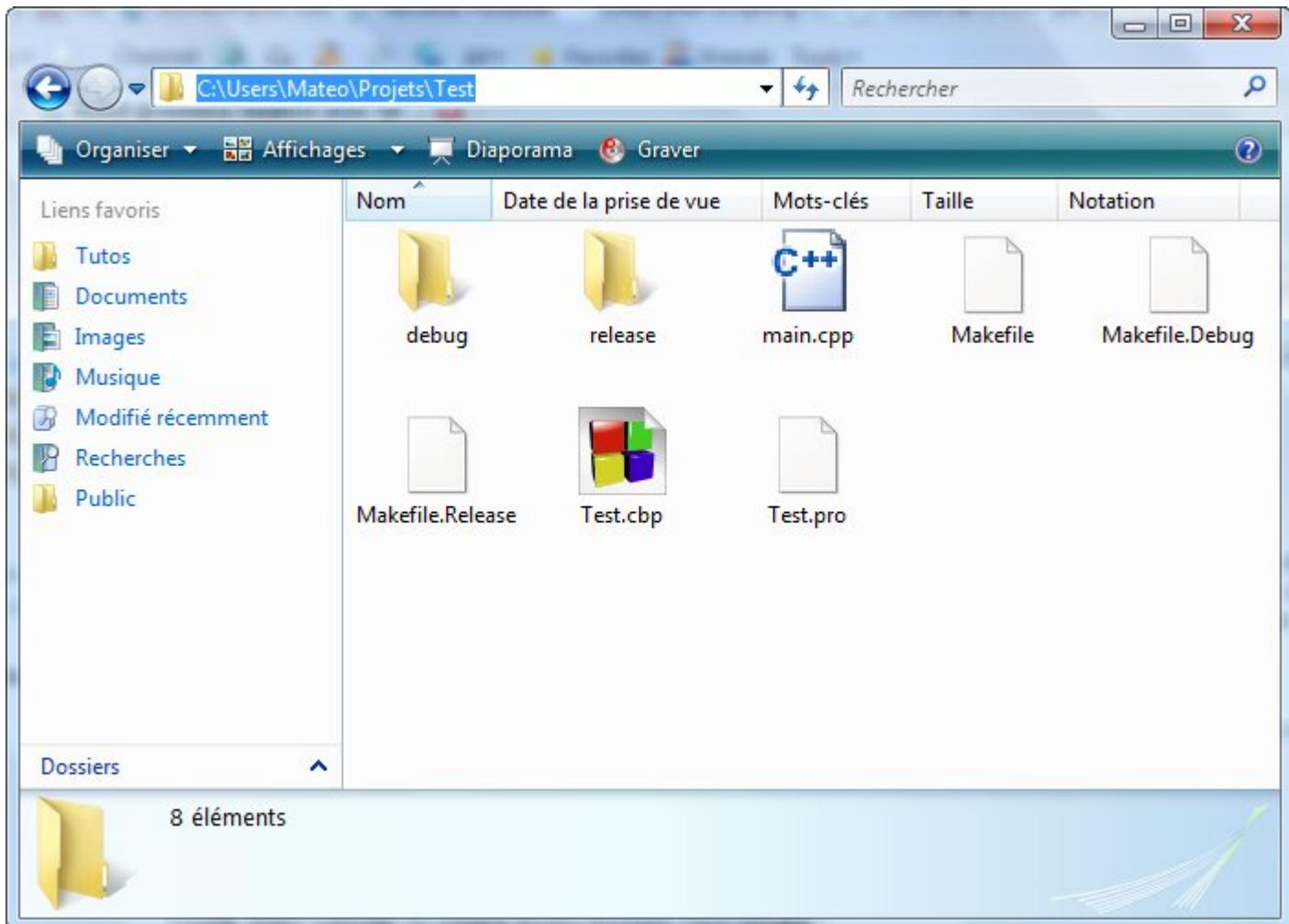
[Etape 2 : générer le Makefile \(qmake\)](#)

Retournez dans la console et tapez tout simplement `qmake` (sans le `-project` cette fois).

Code : Console - [Sélectionner](#)

```
C:\Users\Mateo\Projets\Test>qmake  
C:\Users\Mateo\Projets\Test>
```

Encore une fois, rien de particulier ne devrait s'afficher à l'écran, mais cette fois le fichier Makefile a été généré. Vous devriez en fait trouver plusieurs Makefiles et dossiers (selon si on compile en mode "Debug" pour le débogage, ou "Release" pour la compilation finale lorsque le programme est prêt à être distribué).



Vous pouvez ouvrir par exemple le fichier qui s'appelle "Makefile" (tout court, sans extension) avec un éditeur de texte comme Bloc-Notes. Vous constaterez que ce fichier, destiné à make, est beaucoup plus gros que le .pro. On ne modifiera jamais le Makefile, contrairement au .pro que l'on peut s'amuser à modifier facilement.

Si vous êtes sous Mac OS X, la commande qmake ne génère pas de Makefile par défaut. Il faut le lui dire avec l'option -spec. Tapez dans la console `qmake -spec macx-g++`. 😊

Etape 3 : compiler (make ou mingw32-make)

Il ne vous reste plus qu'à taper `make` (ou `mingw32-make` si vous êtes sous Windows) dans la console pour lancer le compilateur ! Celui-ci va rechercher automatiquement le fichier "Makefile" dans le dossier dans lequel vous vous trouvez.

Code : Console - [Sélectionner](#)

```
C:\Users\Mateo\Projets\Test>mingw32-make
mingw32-make -f Makefile.Release
mingw32-make[1]: Entering directory `C:/Users/Mateo/Projets/Test'
g++ -c -O2 -frtti -fexceptions -mthreads -Wall -DUNICODE -DQT_LARGEFILE_SUPPORT
-DQT_DLL -DQT_NO_DEBUG -DQT_GUI_LIB -DQT_CORE_LIB -DQT_THREAD_SUPPORT -DQT_NEEDS_QMAIN -I"../../Qt/4.3.2/include\QtCore" -I"../../Qt/4.3.2/include\Qt
Core" -I"../../Qt/4.3.2/include\QtGui" -I"../../Qt/4.3.2/include\Qt
Gui" -I"../../Qt/4.3.2/include" -I"." -I"c:\Qt\4.3.2\include\ActiveQt" -I"
release" -I"." -I"../../Qt/4.3.2\mkspecs\win32-g++" -o release\main.o main
.cpp
g++ -enablestdcall-fixup -Wl,-enable-auto-import -Wl,-enable-runtime-pseudo-rel
oc -Wl,-s -mthreads -Wl -Wl,-subsystem,windows -o "release\Test.exe" release\mai
n.o -L"c:\Qt\4.3.2\lib" -lmingw32 -lqtmain -lQtGui4 -lQtCore4
mingw32-make[1]: Leaving directory `C:/Users/Mateo/Projets/Test'

C:\Users\Mateo\Projets\Test>
```

Contrairement à qmake, la commande make est un poil plus bavarde 😊

Vous voyez toutes les options qui ont été envoyées au compilateur... Et y'en a un paquet !

Le compilateur se charge de compiler tous les fichiers qui ont été modifiés depuis la dernière compilation, puis il appelle le linker pour assembler tous les fichiers objet en un bel exécutable.

Si ce que je viens de vous dire à l'instant est du chinois pour vous, c'est que [vous n'avez sûrement pas lu ça](#).

Si par hasard il y a une erreur de compilation sur un des fichiers, l'erreur s'affichera dans la console. On vous indiquera dans quel fichier et à quelle ligne se trouve l'erreur, après il ne vous reste plus qu'à la corriger 😊

Pensez à configurer votre IDE dans les options pour qu'il affiche les numéros de ligne si ce n'est déjà fait, je ne veux pas vous voir "compter" les lignes à la main pour retrouver la ligne de l'erreur.

Résumé des commandes

En résumé, j'ai tapé les commandes suivantes :

Code : Console - [Sélectionner](#)

```
Setting up a MinGW/Qt only environment...
-- QTDIR set to C:\Qt\4.3.2
-- PATH set to C:\Qt\4.3.2\bin
-- Adding C:\MinGW\bin to PATH
-- Adding C:\Windows\System32 to PATH
-- QMAKESPEC set to win32-g++

C:\Qt\4.3.2>cd C:\Users\Mateo\Projets\Test

C:\Users\Mateo\Projets\Test>qmake -project

C:\Users\Mateo\Projets\Test>qmake

C:\Users\Mateo\Projets\Test>mingw32-make
mingw32-make -f Makefile.Release
mingw32-make[1]: Entering directory `C:/Users/Mateo/Projets/Test'
g++ -c -O2 -frtti -fexceptions -mthreads -Wall -DUNICODE -DQT_LARGEFILE_SUPPORT
-DQT_DLL -DQT_NO_DEBUG -DQT_GUI_LIB -DQT_CORE_LIB -DQT_THREAD_SUPPORT -DQT_NEEDS_QMAIN
-I"..\..\..\Qt\4.3.2\include\QtCore" -I"..\..\..\Qt\4.3.2\include\QtCore"
-I"..\..\..\Qt\4.3.2\include\QtGui" -I"..\..\..\Qt\4.3.2\include\QtGui"
-I"..\..\..\Qt\4.3.2\include" -I"." -I"c:\Qt\4.3.2\include\ActiveQt" -I"release"
-I"." -I"..\..\..\Qt\4.3.2\mkspecs\win32-g++" -o release\main.o main.cpp
g++ -enable-stdcall-fixup -Wl,-enable-auto-import -Wl,-enable-runtime-pseudo-reloc
-Wl,-s -mthreads -Wl,-Wl,-subsystem,windows -o "release\Test.exe" release\mai
n.o -L"c:\Qt\4.3.2\lib" -lmingw32 -lqtmain -lQtGui4 -lQtCore4
mingw32-make[1]: Leaving directory `C:/Users/Mateo/Projets/Test'

C:\Users\Mateo\Projets\Test>
```

Exécuter le programme

Pour tester le programme, vous avez 2 solutions :

- Soit vous le lancez depuis la console,
- Soit vous double-cliquez sur l'exécutable depuis l'explorateur.

Lancer le programme depuis la console

Normalement, l'exécutable a été placé dans le sous-dossier "release".

Il vous suffit de vous placer dans ce sous-dossier puis de taper le nom du programme pour qu'il s'exécute :

Code : Console - Sélectionner

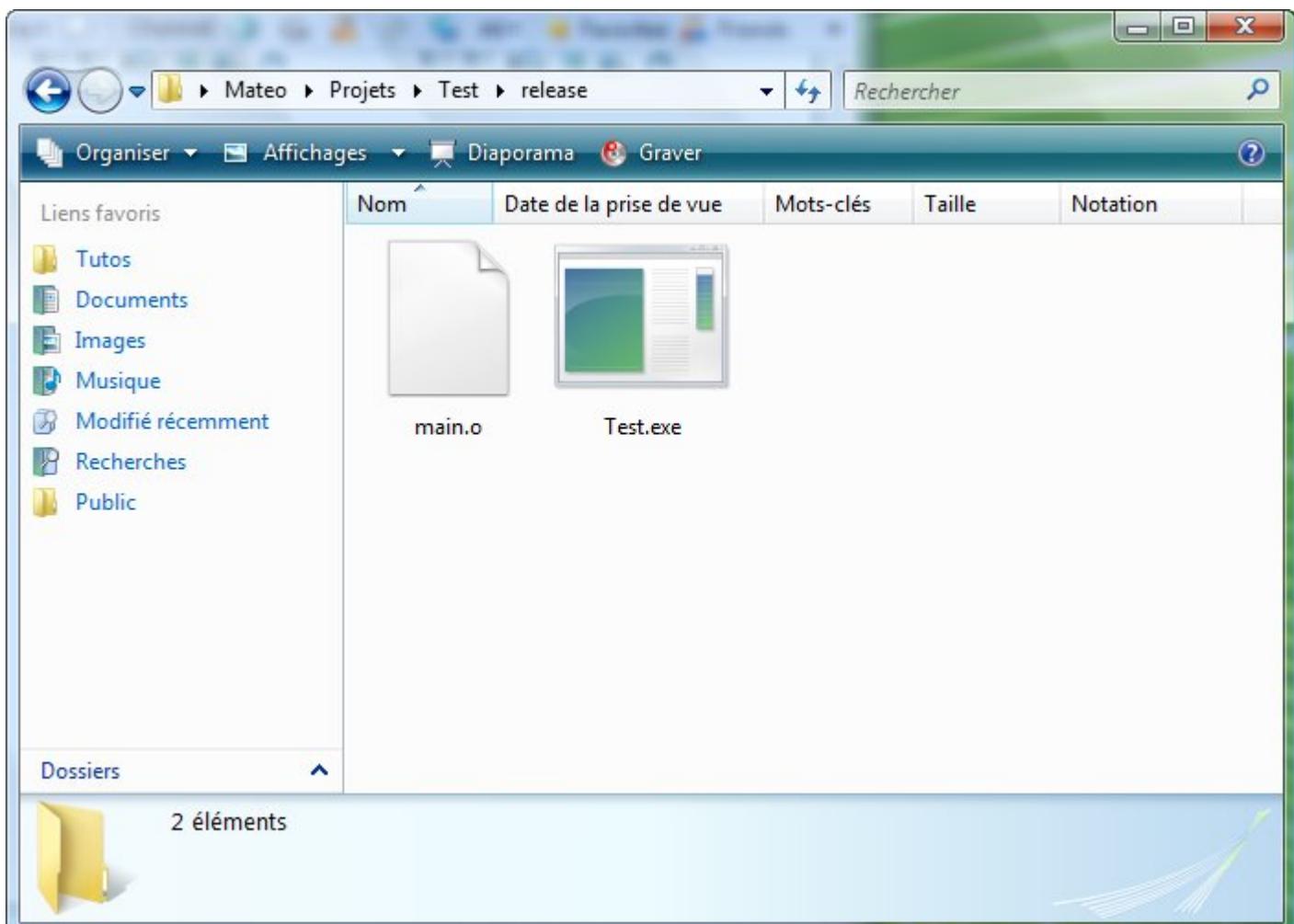
```
C:\Users\Mateo\Projets\Test>cd release
C:\Users\Mateo\Projets\Test\release>Test.exe
```

Lancer le programme depuis l'explorateur

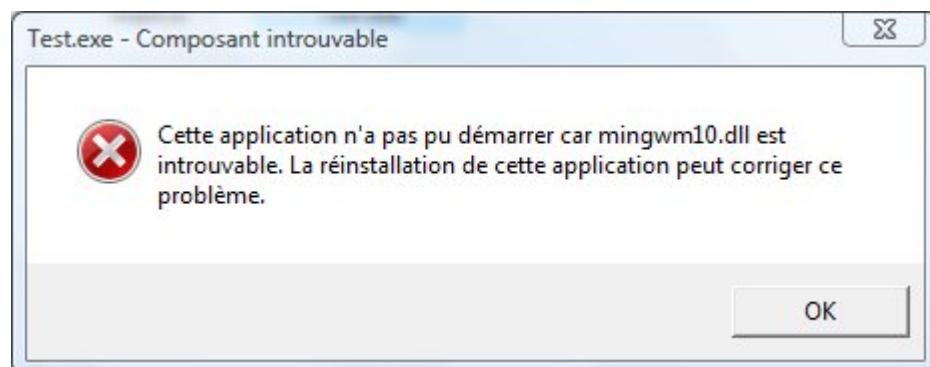
Je sais je sais, vous vous dites que lancer un programme depuis la console c'est un peu débile. Je suis d'accord, mais les programmes Qt ont besoin des fichiers DLL de Qt avec eux pour fonctionner.

Quand vous exécutez votre programme depuis la console spéciale, la position des DLL est "connue", donc votre programme se lance sans erreur.

Mais essayez de double-cliquer sur l'exécutable depuis l'explorateur pour voir !



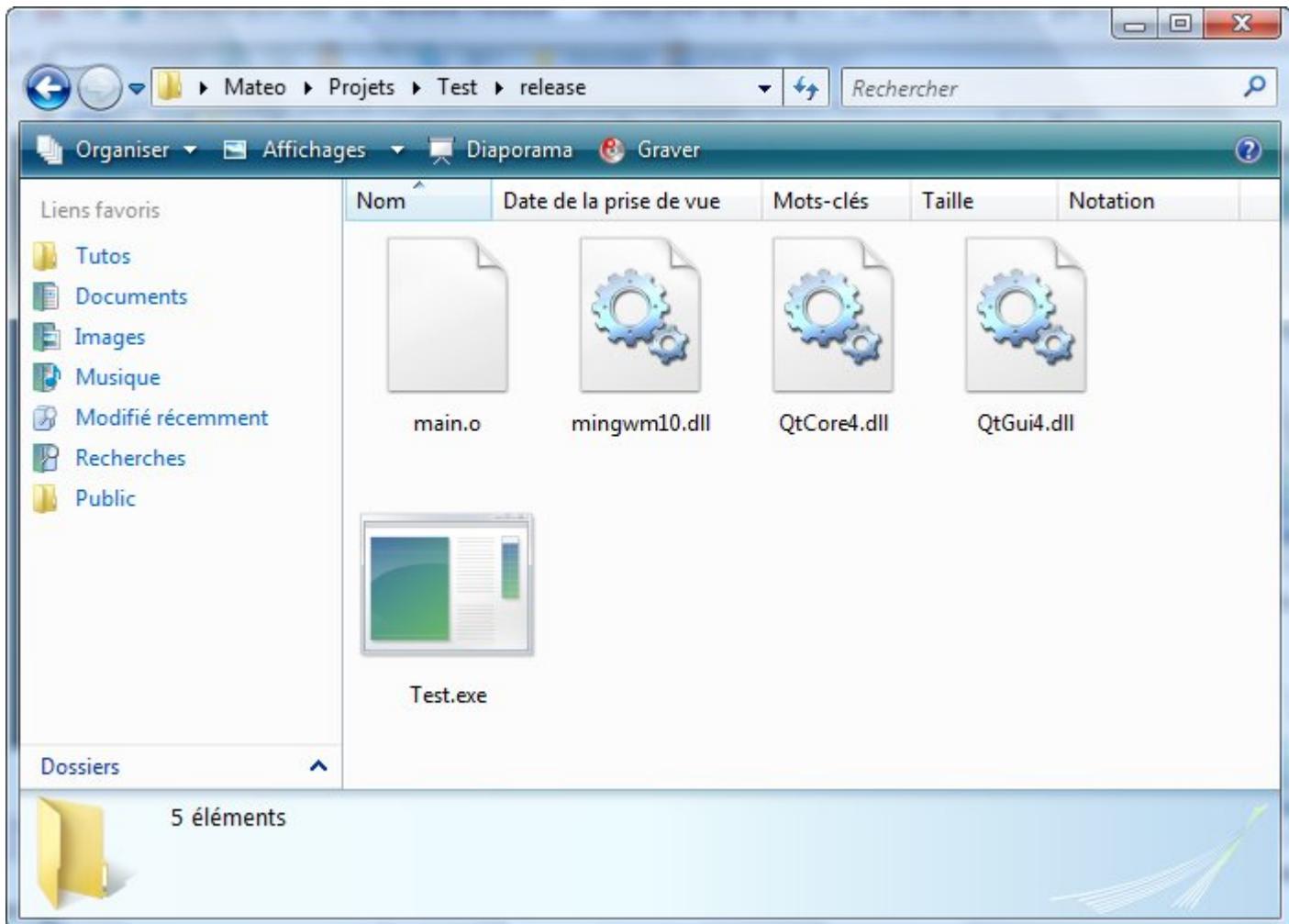
Le programme Test.exe dans le dossier "release". Double-cliquez dessus.



Miséricorde ! Ca ne marche pas !

En effet, sans quelques DLL à côté notre programme est perdu. Il a besoin de ces fichiers qui contiennent de quoi le guider. Vous avez déjà eu affaire aux DLL si vous avez utilisé la SDL dans le cours de C normalement 😊

Pour pouvoir lancer l'exécutable depuis l'explorateur (et aussi pour qu'il marche chez vos amis / clients), il faut placer les DLL qui manquent dans le même dossier que l'exécutable. A vous de les chercher, vous les avez sur votre disque (chez moi je les ai trouvés dans le dossier C:\MinGW\bin et C:\Qt\4.3.2\bin). En tout, vous devriez avoir eu besoin de mettre 3 DLL :



Vous pouvez lancer le programme maintenant !

Notre première fenêtre en action !

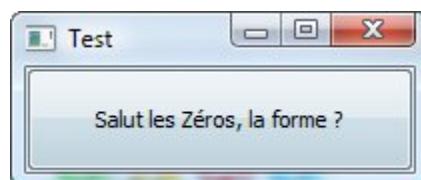
Ouf ! On est arrivé à compiler et lancer notre programme !

Coucou petite fenêtre, fais risette à la caméra !



Le bouton prend la taille du texte qui se trouve à l'intérieur, et la fenêtre qui est automatiquement créée prend la taille du bouton. Ca donne donc une toute petite fenêtre 😊

Mais... vous pouvez la redimensionner, voire même l'afficher en plein écran ! Rien ne vous en empêche, et le bouton s'adapte automatiquement à la taille de la fenêtre (ce qui peut donner un très gros bouton) :



Nous y sommes enfin arrivés, champagne ! 😊

Vous l'avez vu, le code nécessaire pour ouvrir une fenêtre toute simple constituée d'un bouton est ridicule. Quelques lignes à peine, et rien de bien compliqué à comprendre au final.

C'est ce qui fait la force de Qt : "un code simple est un beau code" dit-on. Qt s'efforce de respecter ce dicton à la lettre, vous vous en rendrez compte dans les prochains chapitres.

Le "défaut" de cette simplicité, c'est qu'on est obligé de passer par un utilitaire spécial appelé qmake pour générer le Makefile qui est parfois très complexe. Mais comme vous avez pu le constater, ça se fait sans problème 😊

Dans les prochains chapitres, nous allons voir comment changer l'apparence du bouton, comment faire une fenêtre un peu plus complexe. Nous découvrirons aussi le mécanisme des signaux et des slots, un des principes les plus importants de Qt qui permet de gérer les événements : un clic sur un bouton pourra par exemple provoquer l'ouverture d'une nouvelle fenêtre ou sa fermeture !

Personnaliser les widgets

La "fenêtre-bouton" que nous avons réalisée dans le chapitre précédent était un premier pas. Toutefois, nous avons passé plus de temps à expliquer les mécanismes de la compilation qu'à modifier le contenu de la fenêtre.

Par exemple, comment faire pour modifier la taille du bouton ? Comment placer le bouton où on veut sur la fenêtre ? Comment modifier les propriétés du bouton ? Changer la couleur, le curseur de la souris, la police, l'icône...

Dans ce chapitre, nous allons nous habituer à modifier les propriétés d'un widget : le bouton. Bien sûr, il existe des tonnes d'autres widgets (cases à cocher, listes déroulantes...) mais nous nous concentrerons sur le bouton pour nous habituer à éditer les propriétés d'un widget.

Une fois que vous saurez le faire pour le bouton, vous n'aurez aucun mal à le faire pour les autres widgets.

Enfin et surtout, nous reparlerons d'héritage dans ce chapitre. Nous apprendrons à créer un widget personnalisé qui "hérite" du bouton. C'est une technique extrêmement courante que l'on retrouve dans toutes les bibliothèques de création de GUI !

Allez hop, vous allez me personnaliser ce bouton tout gris !

"Yo man, on va te cus-to-mi-ser ton vieux bouton à la sauce west coast ! Aujourd'hui sur le Site du Zéro, c'est Pimp mon bouton !"

Pardonnez ce petit délire, je promets à l'avenir de ne plus regarder MTV avant de rédiger un tutoriel. Promis promis.



Modifier les propriétés d'un widget

Comme tous les éléments d'une fenêtre, on dit que le bouton est un widget.

Avec Qt, on crée un bouton à l'aide de la classe QPushButton.

Comme vous le savez, une classe est constituée de 2 éléments :

- Des attributs : ce sont les "variables" internes de la classe.
- Des méthodes : ce sont les "fonctions" internes de la classe.

La règle d'encapsulation dit que les utilisateurs de la classe ne doivent pas pouvoir modifier les attributs : ceux-ci doivent donc tous être privés.

Or, je ne sais pas si vous avez remarqué, mais nous sommes justement des utilisateurs des classes de Qt. Ce qui veut dire... que nous n'avons pas accès aux attributs puisque ceux-ci sont privés ! 😊

Hé, mais tu avais parlé d'un truc à un moment je crois... Les accesseurs, c'est pas ça ?

Ah... J'aime les gens qui ont de la mémoire 😊

Effectivement oui, j'avais dit que le créateur d'une classe devait rendre ses attributs privés, mais du coup proposer des méthodes accesseurs, c'est-à-dire des méthodes permettant de lire et de modifier les attributs de manière sécurisée (get et set ça vous dit rien ?).

Les accesseurs avec Qt

Justement, les gens qui ont créé Qt chez Trolltech sont des braves gars : ils ont codé proprement en respectant ces règles. Et il valait mieux qu'ils fassent bien les choses s'ils ne voulaient pas que leur bibliothèque devienne un véritable futoir !

Du coup, pour chaque propriété d'un widget, on a :

- **Un attribut : il est privé on ne peut pas le lire ni le modifier directement.**
Exemple : `text`
- **Un accesseur pour le lire : cet accesseur est une méthode constante qui porte le même nom que l'attribut (personnellement j'aurais plutôt mis un "get" devant pour ne pas confondre avec l'attribut, mais bon). Je vous rappelle qu'une méthode constante est une méthode qui s'interdit de modifier les attributs de la classe. Ainsi, vous êtes assuré que la méthode ne fait que lire l'attribut et qu'elle ne le modifie pas.**
Exemple : `text()`
- **Un accesseur pour le modifier : c'est une méthode qui se présente sous la forme `setAttribut()`. Elle modifie la valeur de l'attribut.**
Exemple : `setText()`

Cette technique, même si elle paraît un peu lourde parce qu'il faut créer 2 méthodes pour chaque attribut, a l'avantage d'être parfaitement sûre. Grâce à ça, Qt peut vérifier que la valeur que vous essayez de donner est valide. Cela permet d'éviter par exemple que vous ne donnez à une barre de progression la valeur "150%", alors que la valeur d'une barre de progression doit être comprise entre 0 et 100%.



Voyons voir sans plus tarder quelques propriétés des boutons que nous pouvons nous amuser à modifier à l'aide des accesseurs 😊

Quelques exemples de propriétés des boutons

Il existe un grand nombre de propriétés éditable pour chaque widget, y compris le bouton. Nous n'allons pas toutes les voir ici, ni même plus tard d'ailleurs, je vous apprendrai à lire la doc pour toutes les découvrir 😊
Cependant, je tiens à vous montrer les plus intéressantes d'entre elles pour que vous puissiez commencer à vous faire la main, et surtout pour que vous preniez l'habitude d'utiliser les accesseurs de Qt.

`text : le texte`

Cette propriété est probablement la plus importante : elle permet de modifier le texte présent sur le bouton.
En général, on définit le texte du bouton au moment de sa création car le constructeur accepte que l'on donne le texte du bouton dès sa création.

Toutefois, pour une raison ou une autre, vous pourriez être amené à modifier le texte présent sur le bouton au cours de l'exécution du programme. C'est là qu'il devient pratique d'avoir accès à l'attribut "text" du bouton via ses accesseurs.

Pour chaque attribut, la documentation de Qt nous dit à quoi il sert et quels sont ses accesseurs. Voyez par exemple [ce que ça donne pour l'attribut text des boutons](#).

On vous indique de quel type est l'attribut. Ici, `text` est de type `QString`, comme tous les attributs qui stockent du texte avec Qt. En effet, Qt n'utilise pas la classe "string" standard du C++ mais sa propre version de la gestion des chaînes de caractères. En gros, `QString` c'est un string amélioré.

Puis, on vous explique en quelques mots à quoi sert cet attribut (in english of course, il n'est jamais trop tard pour reprendre des cours d'anglais quel que soit votre âge 😊).

Enfin, on vous indique les accesseurs qui permettent de lire et de modifier l'attribut. Dans le cas présent, il s'agit de :

- `QString text () const` : c'est l'accesseur qui permet de lire l'attribut. Il retourne un `QString`, ce qui est logique puisque

l'attribut est de type `QString`. Vous noterez la présence du mot-clé "const" qui indique que c'est une méthode constante qui ne modifie aucun attribut.

- `void setText (const QString & text)` : c'est l'accesseur qui permet de modifier l'attribut. Il prend un paramètre : le texte que vous voulez mettre sur le bouton.

A la longue, vous ne devriez pas avoir besoin de la doc pour savoir quels sont les accesseurs d'un attribut. Ca suit toujours le même schéma :

`attribut()` : permet de lire l'attribut.
`setAttribut()` : permet de modifier l'attribut.

Essayons donc de modifier le texte du bouton après sa création :

Code : C++ - [Sélectionner](#)

```
#include <QApplication>
#include <QPushButton>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QPushButton bouton("Salut les Zéros, la forme ?");
    bouton.setText("Pimp mon bouton !");

    bouton.show();

    return app.exec();
}
```

Vous aurez noté que la méthode `setText` attend un `QString` et qu'on lui envoie une bête chaîne de caractères entre guillemets. En fait, ça fonctionne comme la classe `string` : les chaînes de caractères entre guillemets sont automatiquement converties en `QString`. Heureusement d'ailleurs, sinon ça serait lourd de devoir créer un objet de type `QString` juste pour ça !

Résultat :



Le résultat n'est peut-être pas très impressionnant, mais ça montre bien ce qui se passe :

1. On crée le bouton et on lui donne le texte "Salut les Zéros, la forme ?" à l'aide du constructeur.
2. On modifie le texte présent sur le bouton pour afficher "Pimp mon bouton !".

Au final, c'est "Pimp mon bouton !" qui s'affiche.

Pourquoi ? Parce que le nouveau texte a "écrasé" l'ancien. C'est exactement comme si on faisait :

Code : C++ - [Sélectionner](#)

```
int x = 1;
x = 2;
cout << x;
```

... Lorsqu'on affiche x, il vaut 2.

C'est pareil pour le bouton. Au final, c'est le tout dernier texte qui sera affiché.

Bien entendu, ce qu'on vient de faire est complètement inutile : autant donner le bon texte directement au bouton lors de l'appel du constructeur. Toutefois, `setText()` se révèlera utile plus tard lorsque vous voudrez modifier le contenu du bouton au cours de l'exécution. Par exemple, lorsque l'utilisateur aura donné son nom, le bouton pourra changer de texte pour dire "Bonjour M. Dupont !".

toolTip : l'infobulle

Il est courant d'afficher une petite aide sous la forme d'une infobulle qui apparaît lorsqu'on pointe sur un élément avec la souris.

L'infobulle peut afficher un court texte d'aide. On la définit à l'aide de la propriété `toolTip`.

Pour modifier l'infobulle, la méthode à appeler est donc... `setToolTip` ! Bah vous voyez, c'est facile quand on a compris comment Qt était organisé 😊

Code : C++ - [Sélectionner](#)

```
#include <QApplication>
#include <QPushButton>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QPushButton bouton("Pimp mon bouton !");
    bouton.setToolTip("Texte d'aide");

    bouton.show();

    return app.exec();
}
```



Une infobulle

font : la police

Avec la propriété `font`, les choses se compliquent. En effet, jusqu'ici on avait juste eu à envoyer une chaîne de caractères en paramètres, qui était en fait convertie en objet de type `QString`.

La propriété `font` est un peu plus complexe car elle contient 3 informations :

- Le nom de la police de caractères utilisée (Times New Roman, Arial, Comic Sans MS...)
- La taille du texte en pixels (12, 16, 18...)
- Le style du texte (gras, italique...)

La signature de la méthode `setFont` est :

```
void setFont ( const QFont & )
```

Cela veut dire que `setFont` attend un objet de type `QFont` !

Je rappelle, pour ceux qui auraient oublié la signification des symboles, que :

- `const` : signifie que l'objet que l'on envoie en paramètre ne sera pas modifié par la fonction
- `&` : signifie que la fonction attend une référence vers l'objet. En C, il aurait fallu envoyer un pointeur, mais comme en C++ on

dispose des références (qui sont plus simples à utiliser), on en profite 😊

Bon, comment on fait pour lui donner un objet de type QFont nous ?
Eh bien c'est simple : il... suffit de créer un objet de type QFont !

La doc nous indique [tout ce que nous avons besoin de savoir sur QFont](#), en particulier les informations qu'il faut donner à son constructeur. Je n'attends pas de vous encore que vous soyez capable de lire la doc de manière autonome, je vais donc vous mâcher le travail (mais profitez-en parce que ça ne durera pas éternellement 🤪).

Pour faire simple, le constructeur de QFont attend 4 paramètres. Voici son prototype :

```
QFont ( const QString & family, int pointSize = -1, int weight = -1, bool italic = false )
```

En fait, avec Qt il y a rarement un seul constructeur par classe. Les développeurs de Qt profitent des fonctionnalités du C++ et ont donc tendance à beaucoup surcharger les constructeurs. Certaines classes possèdent même plusieurs dizaines de constructeurs différents !

Pour QFont, celui que je vous montre là est néanmoins le principal et le plus utilisé. Et le plus simple aussi, tant qu'à faire.

Seul le premier argument est obligatoire : il s'agit du nom de la police à utiliser. Les autres, comme vous pouvez le voir, possèdent des valeurs par défaut donc nous ne sommes pas obligés de les indiquer.

Dans l'ordre, les paramètres signifient :

- family : le nom de la police de caractères à utiliser.
- pointSize : la taille des caractères en pixels.
- weight : le niveau d'épaisseur du trait (gras). Cette valeur peut être comprise entre 0 et 99 (du plus fin au plus gras). Vous pouvez aussi utiliser la constante QFont::Bold qui correspond à une épaisseur de 75.
- italic : un booléen pour dire si le texte doit être affiché en italique ou non.

On va faire quelques tests. Tout d'abord, il va falloir créer un objet de type QFont :

Code : C++ - [Sélectionner](#)

```
QFont maPolice( "Courier" );
```

J'ai appelé cet objet maPolice.

Maintenant, je dois envoyer l'objet maPolice de type QFont à la méthode setFont de mon bouton (suivez, suivez !) :

Code : C++ - [Sélectionner](#)

```
bouton.setFont(maPolice);
```

En résumé, j'ai donc dû écrire 2 lignes pour changer la police :

Code : C++ - [Sélectionner](#)

```
QFont maPolice( "Courier" );
bouton.setFont(maPolice);
```

C'est un peu fastidieux. Il existe une solution plus maligne, si on ne compte pas se resservir de la police plus tard, c'est de définir l'objet de type QFont au moment de l'appel à la méthode setFont. Ca nous évite d'avoir à donner un nom bidon à l'objet comme on l'a fait ici (maPolice), c'est plus court, ça va plus vite, bref c'est mieux en général 🤩

Code : C++ - [Sélectionner](#)

```
bouton.setFont(QFont( "Courier" ));
```

Voilà, en imbriquant comme ça ça marche très bien. La méthode setFont veut un objet de type QFont ? Qu'à cela ne tienne, on lui en crée un à la volée !

Voici le résultat :



Maintenant, on peut exploiter un peu plus le constructeur de QFont en utilisant une autre police plus fantaisiste et en augmentant la taille des caractères :

Code : C++ - [Sélectionner](#)

```
bouton.setFont(QFont("Comic Sans MS", 20));
```



Et voilà le même avec du gras et de l'italique !

Code : C++ - [Sélectionner](#)

```
bouton.setFont(QFont("Comic Sans MS", 20, QFont::Bold, true));
```



Bref, si vous avez compris le [principe des paramètres par défaut](#) (et j'espère que vous avez compris depuis le temps ! 😊), ça ne devrait vous poser aucun problème.

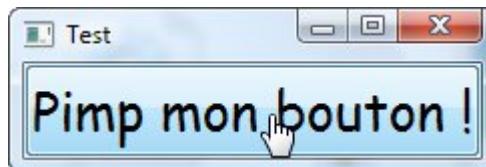
cursor : le curseur de la souris

Avec la propriété cursor, vous pouvez déterminer quel curseur de la souris doit s'afficher lorsqu'on pointe sur le bouton. Le plus simple est d'utiliser une des [constantes de curseurs prédéfinis](#) parmi la liste qui s'offre à vous.

Ce qui peut donner par exemple, si on veut qu'une main s'affiche :

Code : C++ - [Sélectionner](#)

```
bouton.setCursor(Qt::PointingHandCursor);
```



icon : l'icône du bouton

Après tout ce qu'on vient de voir, rajouter une icône au bouton va vous paraître très simple : la méthode setIcon attend juste un objet de type QIcon.

Un QIcon peut se construire très facilement en donnant le nom du fichier image à charger.

Prenons par exemple ce petit smiley souriant : 😊

Il s'agit d'une image au format PNG que sait lire Qt.

Code : C++ - [Sélectionner](#)

```
bouton.setIcon(QIcon("smile.png"));
```

Attention, sous Windows pour que cela fonctionne, votre icône smile.png doit se trouver dans le même dossier que l'exécutable (ou dans un sous-dossier si vous écrivez "dossier/smile.png").

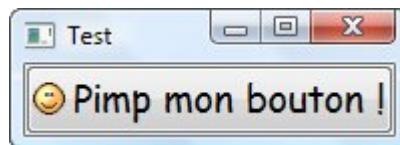
Sous Linux, il faut que votre icône soit dans votre répertoire HOME. Si vous voulez utiliser le chemin de votre application, comme cela se fait sous Windows par défaut, écrivez :

Code : C++ - [Sélectionner](#)

```
QIcon(QCoreApplication::applicationDirPath() + "/smile.png");
```

Cela aura pour effet d'afficher l'icône à condition que celle-ci se trouve dans le même répertoire que l'exécutable.

Si vous avez fait ce qu'il fallait, l'icône devrait alors apparaître comme ceci :



Qt et l'héritage

On aurait pu continuer à faire joujou longtemps avec les propriétés de notre bouton, mais il faut savoir s'arrêter au bout d'un moment et reprendre les choses sérieuses.

Quelles choses sérieuses ?

Si je vous dis "héritage", ça ne vous rappelle rien ? J'espère que ça ne vous donne pas des boutons en tout cas (oh oh oh), parce que si vous n'avez pas compris le [principe de l'héritage](#) vous ne pourrez pas aller plus loin.

De l'héritage en folie

L'héritage est probablement LA notion la plus intéressante de la programmation orientée objet. Le fait de pouvoir créer une classe de base, réutilisée par des sous-classes filles, qui ont elles-mêmes leurs propres sous-classes filles, ça donne à une bibliothèque comme Qt une puissance infinie (voire plus, même).

En fait... quasiment toutes les classes de Qt font appel à l'héritage.

Pour vous faire une idée, la documentation vous donne la [hiérarchie complète des classes](#). Chaque classe "à gauche" de cette liste à puces est une classe de base, et les classes qui sont décalées vers la droite sont des sous-classes.

Vous pouvez par exemple voir au début :

- QAbstractExtensionFactory
 - QExtensionFactory
- QAbstractExtensionManager
 - QExtensionManager

QAbstractExtensionFactory et QAbstractExtensionManager sont des classes dites "de base". Elles n'ont pas de classes parentes. En revanche, QExtensionFactory et QExtensionManager sont des classes-filles, qui héritent respectivement de QAbstractExtensionFactory et QAbstractExtensionManager.

Sympa hein ? 😊

Descendez plus bas sur la [page de la hiérarchie](#) à la recherche de la classe QObject.

Regardez un peu toutes ses classes filles.

Descendez.

Encore.

Encore.

Encore.

C'est bon vous avez pas trop pris peur ? 😊

Vous avez dû voir que certaines classes étaient carrément des sous-sous-sous-sous-sous-classes.

Wouaw mais comment je vais m'y retrouver là-dedans moi ? C'est pas possible je vais jamais m'en sortir !

C'est ce qu'on a tendance à se dire la première fois. En fait, vous allez petit à petit comprendre qu'au contraire tous ces héritages sont là pour vous simplifier la vie. Si ce n'était pas aussi bien architecturé, alors là vous ne vous en seriez jamais sortis !

QObject : une classe de base incontournable

[QObject](#) est la classe de base de tous les objets sous Qt.

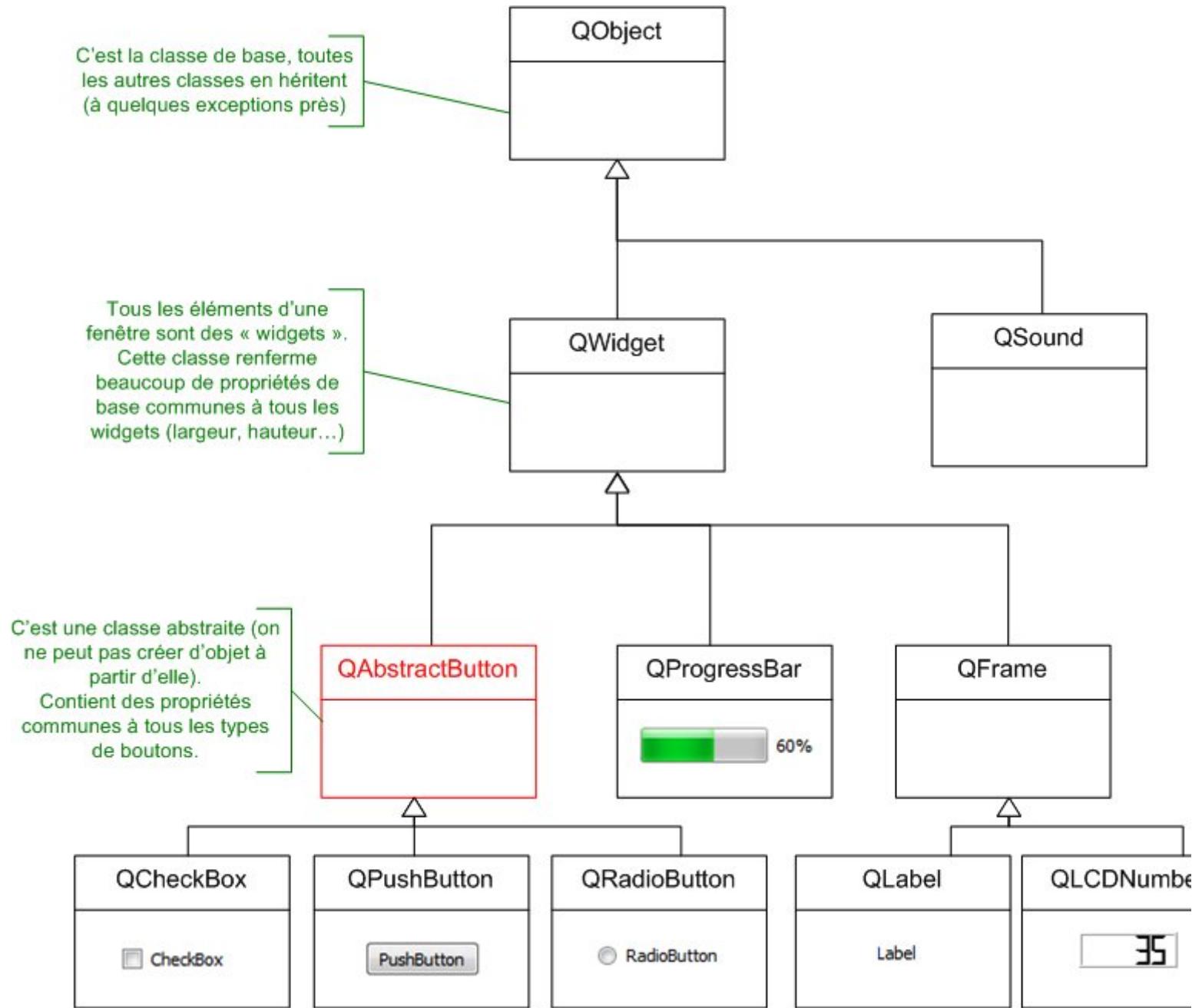
QObject ne correspond à rien de particulier, mais elle propose quelques fonctionnalités "de base" qui peuvent être utiles à toutes les autres classes.

Cela peut surprendre d'avoir une classe de base qui ne sait rien faire de particulier, mais en fait c'est ce qui donne beaucoup de puissance à la bibliothèque. Par exemple, il suffit de définir une fois dans QObject une méthode `objectName()` qui contient le nom de l'objet, et ainsi toutes les autres classes de Qt en héritent et possèderont donc cette méthode.

D'autre part, le fait d'avoir une classe de base comme QObject est indispensable pour réaliser le mécanisme des signaux et des slots qu'on verra dans le prochain chapitre. Ce mécanisme permet de faire en sorte par exemple que si un bouton est cliqué, alors une autre fenêtre s'ouvre (on dit qu'il envoie un signal à un autre objet).

Bref, tout cela doit vous sembler encore un peu abstrait et je le comprends parfaitement.

Je pense qu'un petit schéma simplifié des héritages de Qt s'impose. Cela devrait vous permettre de mieux visualiser la hiérarchie des classes :



Soyons clairs : je n'ai pas tout mis. J'ai juste mis quelques exemples, mais s'il fallait faire le schéma complet ça prendrait une place énorme vous vous en doutez !

On voit sur ce schéma que QObject est la classe mère principale, dont héritent toutes les autres classes. Comme je l'ai dit, elle propose quelques fonctionnalités qui se révèlent utiles pour toutes les classes, mais nous ne les verrons pas ici.

Certaines classes comme QSound (gestion du son) héritent directement de QObject.

Toutefois, comme je l'ai dit on s'intéresse plus particulièrement à la création de GUI, c'est-à-dire de fenêtres. Or, dans une fenêtre tout est considéré comme un widget (même la fenêtre est un widget).

C'est pour cela qu'il existe une classe de base QWidget pour tous les widgets. Elle contient énormément de propriétés communes à tous les widgets, comme :

- La largeur
- La hauteur
- La position en abscisse (x)
- La position en ordonnée (y)
- La police de caractères utilisée (eh oui, la méthode setFont est définie dans QWidget, et comme QPushButton en hérite, il possède lui aussi cette méthode)
- Le curseur de la souris (pareil, rebelotte, setCursor est en fait défini dans QWidget et non dans QPushButton, car il est aussi susceptible de servir sur tous les autres widgets)

- L'infobulle (toolTip)
- etc.

Vous commencez à percevoir un peu l'intérêt de l'héritage ?

Grâce à cette technique, il leur a suffi de définir une fois toutes les propriétés de base des widgets (largeur, hauteur...). Tous les widgets héritent de QWidget, donc ils possèdent tous ces propriétés. Vous savez donc par exemple que vous pouvez retrouver la méthode setCursor dans la classe QProgressBar.

Les classes abstraites

Vous avez pu remarquer sur mon schéma que j'ai écrit la classe QAbstractButton en rouge... Pourquoi ?

Il existe en fait un grand nombre de classes abstraites sous Qt, qui contiennent toutes le mot "Abstract" dans leur nom.

Les classes dites "abstraites" sont des classes qu'on ne peut pas instancier. C'est-à-dire... qu'on n'a pas le droit de créer d'objet à partir d'elles. Ainsi, on ne peut pas faire :

Code : C++ - [Sélectionner](#)

```
QAbstractButton bouton(); // Interdit car classe abstraite
```

Mais alors... à quoi ça sert de faire une classe si on ne peut pas créer d'objets à partir d'elle ?

Une classe abstraite sert de classe de base pour d'autres sous-classes. Ici, QAbstractButton définit un certain nombre de propriétés communes à tous les types de boutons (boutons classiques, cases à cocher, cases radio...). Par exemple, parmi les propriétés communes on trouve :

- text : le texte affiché
- icon : l'icône affichée à côté du texte du bouton
- shortcut : le raccourci clavier pour activer le bouton
- down : indique si le bouton est enfoncé ou non
- etc.

Bref, encore une fois tout ça n'est défini qu'une fois dans QAbstractButton, et on le retrouve ensuite automatiquement dans QPushButton, QCheckBox, etc.

Dans ce cas, pourquoi QObject et QWidget ne sont pas des classes abstraites elles aussi ? Après tout, elles ne représentent rien de particulier et servent juste de classes de base !

Oui, vous avez tout à fait raison, leur rôle est d'être des classes de base.

Mais... pour un certain nombre de raisons pratiques (qu'on ne détaillera pas ici), il est possible de les instancier quand même, donc de créer par exemple un objet de type QWidget.

Si on affiche un QWidget, qu'est-ce qui apparaît ? Une fenêtre !

En fait, un widget qui ne se trouve pas à l'intérieur d'un autre widget est considéré comme une fenêtre. Ce qui explique pourquoi, en l'absence d'autre information, Qt décide de créer une fenêtre.

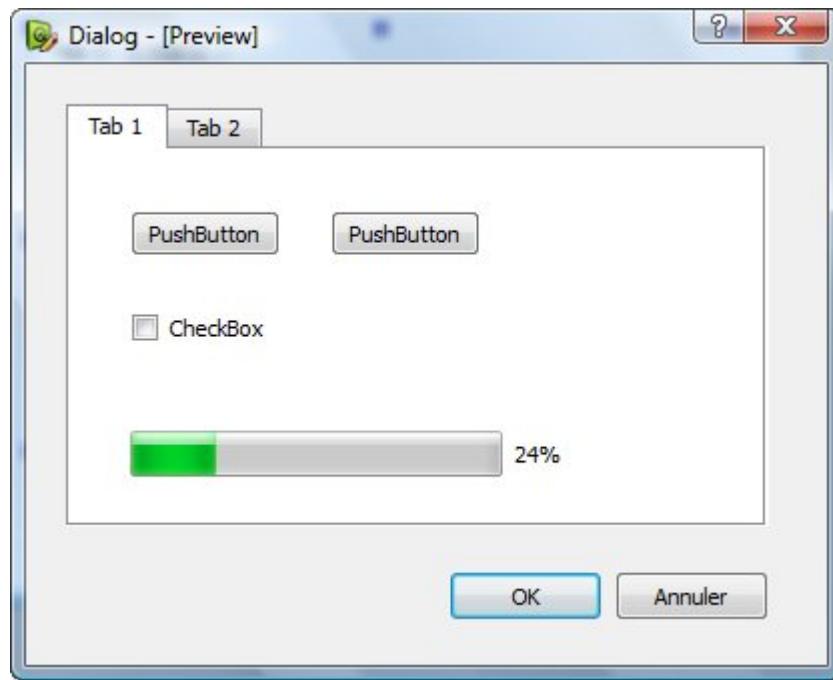
Un widget peut en contenir un autre

Nous attaquons maintenant une notion importante, pas très compliquée, qui est celle des widgets conteneurs.

Contenant et contenu

Il faut savoir qu'un widget peut en contenir un autre. Par exemple, une fenêtre (un QWidget) peut contenir 3 boutons (QPushButton), une case à cocher (QCheckBox), une barre de progression (QProgressBar), etc.

Ce n'est pas là de l'héritage, juste une histoire de contenant et de contenu.
Prenons un exemple :



Sur cette capture, la fenêtre contient 3 widgets :

- Un bouton OK
- Un bouton Annuler
- Un conteneur avec des onglets

Le conteneur avec des onglets est, comme son nom l'indique, un conteneur. Il contient à son tour des widgets :

- 2 boutons
- Une checkbox
- Une barre de progression

Les widgets sont donc imbriqués les uns dans les autres de cette manière :

- QWidget (la fenêtre)
 - QPushButton
 - QPushButton
 - QTabWidget (le conteneur à onglets)
 - QPushButton
 - QPushButton
 - QCheckBox
 - QProgressBar

Attention : ne confondez pas ceci avec l'héritage ! Dans cette partie, je suis en train de vous montrer qu'un widget peut en contenir d'autres. Le gros schéma qu'on a vu un peu plus haut n'a rien à voir avec la notion de widget conteneur.
Ici, on découvre qu'un widget peut en contenir d'autres, indépendamment du fait que ce soit une classe mère ou une classe fille.

Créer une fenêtre contenant un bouton

On ne va pas commencer par faire une fenêtre aussi compliquée que celle que nous venons de voir. Pour le moment on va s'entraîner à faire quelque chose de simple : créer une fenêtre qui contient un bouton.

Mais... c'est pas ce qu'on a fait tout le temps jusqu'ici ? 😊

Non, ce qu'on a fait jusqu'ici c'était juste afficher un bouton. Automatiquement, Qt a créé une fenêtre autour car on ne peut pas avoir de bouton qui "flotte" seul sur l'écran.

L'avantage de créer une fenêtre puis de mettre un bouton dedans, c'est que :

- On pourra mettre d'autres widgets à l'intérieur de la fenêtre à l'avenir.
- On pourra placer le bouton où on veut dans la fenêtre avec les dimensions qu'on veut (jusqu'ici le bouton avait toujours la même taille que la fenêtre).

Voilà comment il faut faire :

Code : C++ - Sélectionner

```
#include <QApplication>
#include <QPushButton>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

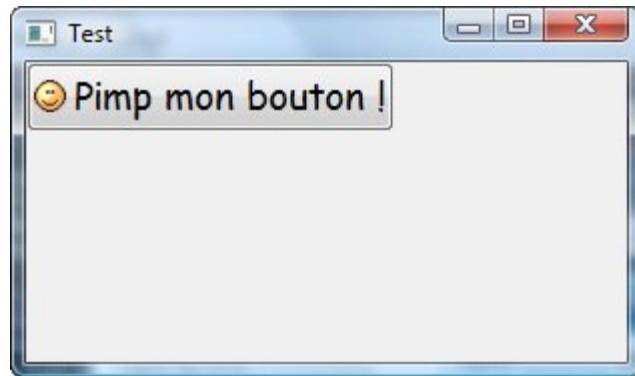
    // Crédation d'un widget qui servira de fenêtre
    QWidget fenetre;
    fenetre.setFixedSize(300, 150);

    // Crédation du bouton, ayant pour parent la "fenetre"
    QPushButton bouton("Pimp mon bouton !", &fenetre);
    // Customisation du bouton
    bouton.setFont(QFont("Comic Sans MS", 14));
    bouton.setCursor(Qt::PointingHandCursor);
    bouton.setIcon(QIcon("smile.png"));

    // Affichage de la fenêtre
    fenetre.show();

    return app.exec();
}
```

... et le résultat :



Qu'est-ce qu'on a fait ?

1. On a créé une fenêtre à l'aide d'un objet de type QWidget.
2. On a dimensionné notre widget (donc notre fenêtre) avec la méthode setFixedSize. La taille de la fenêtre sera fixée : on ne pourra pas la redimensionner.
3. On a créé un bouton, mais avec cette fois une nouveauté au niveau du constructeur : on a indiqué un pointeur vers le widget parent (en l'occurrence la fenêtre).

4. On a customisé un peu le bouton pour la forme.
5. On a déclenché l'affichage de la fenêtre (et donc du bouton qu'elle contenait).

Tous les widgets possèdent un constructeur surchargé qui permet d'indiquer quel est le parent du widget que l'on crée. Il suffit de donner un pointeur pour que Qt sache "qui contient qui".

Le paramètre "&fenetre" du constructeur permet donc d'indiquer que la fenêtre est le parent de notre bouton :

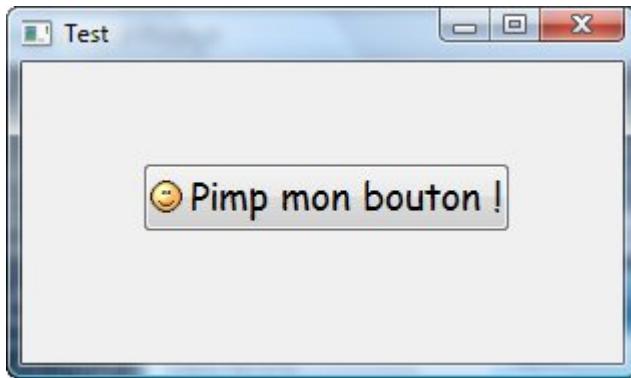
Code : C++ - [Sélectionner](#)

```
QPushButton bouton("Pimp mon bouton !", &fenetre);
```

Si vous voulez placer le bouton ailleurs dans la fenêtre, utilisez la méthode move :

Code : C++ - [Sélectionner](#)

```
bouton.move(60, 50);
```



A noter aussi la méthode setGeometry, qui prend 4 paramètres :

Code : C++ - [Sélectionner](#)

```
bouton.setGeometry(abscisse, ordonnee, largeur, hauteur);
```

La méthode setGeometry permet donc, en plus de déplacer le widget, de lui donner une dimension bien précise.

Tout widget peut en contenir d'autres

... même les boutons !

Quel que soit le widget, son constructeur accepte en dernier paramètre un pointeur vers un autre widget pour indiquer quel est le parent.

On peut faire le test si vous voulez en plaçant un bouton... dans notre bouton !

Code : C++ - [Sélectionner](#)

```

#include <QApplication>
#include <QPushButton>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QWidget fenetre;
    fenetre.setFixedSize(300, 150);

    QPushButton bouton("Pimp mon bouton !", &fenetre);
    bouton.setFont(QFont("Comic Sans MS", 14));
    bouton.setCursor(Qt::PointingHandCursor);
    bouton.setIcon(QIcon("smile.png"));
    bouton.setGeometry(60, 50, 180, 70);

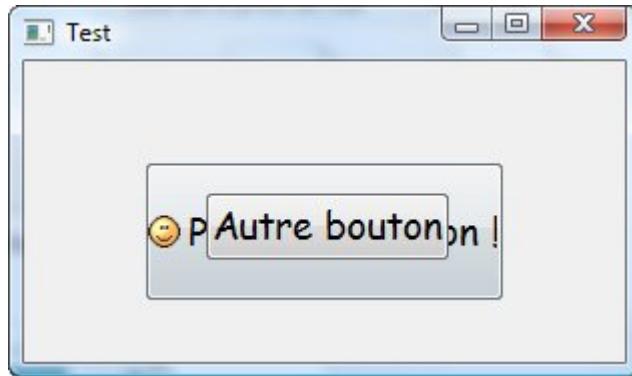
    // Cr ation d'un autre bouton ayant pour parent le premier bouton
    QPushButton autreBouton("Autre bouton", &bouton);
    autreBouton.move(30, 15);

    fenetre.show();

    return app.exec();
}

```

R sultat : notre bouton est plac  l'int rieur de l'autre bouton !



Cet exemple montre qu'il est donc possible de placer un widget dans n'importe quel autre widget, m me un bouton. Bien entendu, comme le montre ma capture d' cran, ce n'est pas tr s malin de faire cela, mais cela prouve que Qt est tr s flexible 😊

Des includes "oubli s"

Dans le code source précédent, nous avons utilis  les classes QWidget, QFont et QIcon pour cr er des objets. Normalement, nous devrions faire un include des fichiers headers de ces classes en plus de QPushButton et QApplication pour que le compilateur les connaisse :

Code : C++ - [S lectionner](#)

```

#include <QApplication>
#include <QPushButton>
#include <QWidget>
#include <QFont>
#include <QIcon>

```

Ah ben oui ! Si on n'a pas inclus le header de la classe QWidget, comment est-ce qu'on a pu cr er tout   l'heure un objet "fenetre" de type QWidget sans que le compilateur ne hurle   la mort ?

Coup de bol. En fait, on avait inclus QPushButton. Et comme QPushButton hérite de QWidget, il avait lui-même inclus QWidget dans son header.

Quant à QFont et QIcon, ils étaient inclus eux aussi car indirectement utilisés par QPushButton.

Bref, des fois comme ça ça marche et on a de la chance. Normalement, si on faisait très bien les choses, on devrait faire un include par classe utilisée.

C'est un peu lourd et il m'arrive d'en oublier. Comme ça marche, en général je ne me pose pas trop de questions.

Toutefois, si vous voulez être sûr d'inclure une bonne fois pour toutes toutes les classes du module "Qt GUI", il vous suffit de faire :

Code : C++ - [Sélectionner](#)

```
#include <QtGui>
```

Le header "QtGui" inclut à son tour toutes les classes du module GUI, donc QWidget, QPushButton, QFont, etc.
Attention toutefois, la compilation sera un peu ralentie du coup.

Hériter un widget

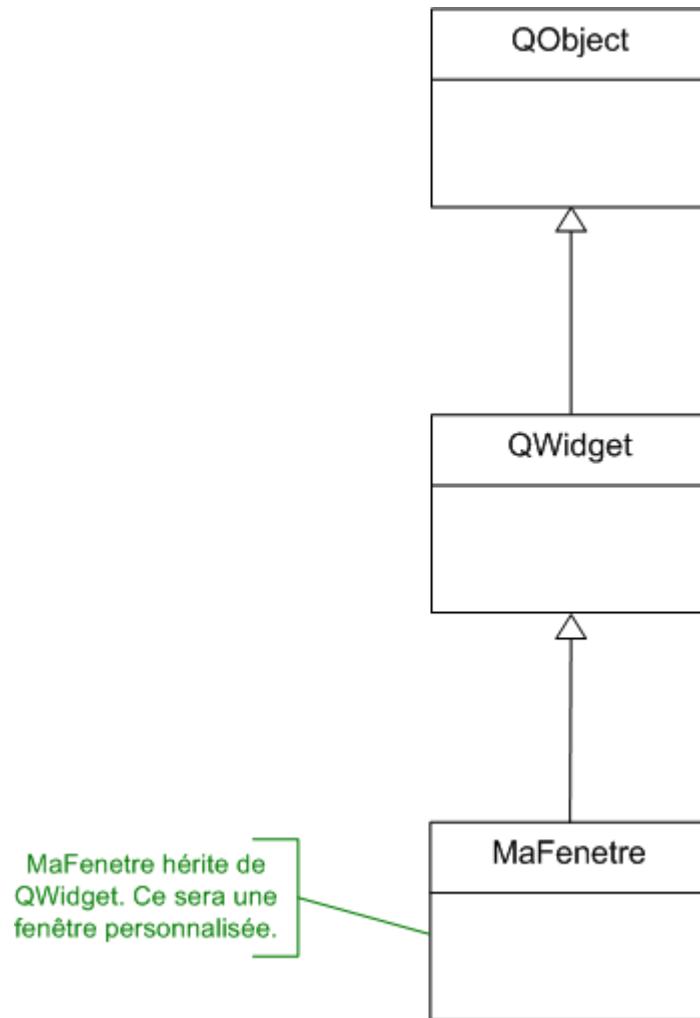
Bon résumons !

Jusqu'ici dans ce chapitre, nous avons :

- Appris à lire et modifier les propriétés d'un widget, en voyant quelques exemples de propriétés des boutons.
- Découvert de quelle façon étaient architecturées les classes de Qt, avec les multiples héritages.
- Découvert la notion de widget conteneur (un widget peut en contenir d'autres). Pour nous entraîner, nous avons créé une fenêtre puis inséré un bouton à l'intérieur.

Nous allons ici aller plus loin dans la personnalisation des widgets en "inventant" un nouveau type de widget. En fait, nous allons créer une nouvelle classe qui va hériter de QWidget et représenter notre fenêtre. Créer une classe pour gérer la fenêtre va peut-être vous paraître un peu lourd au premier abord, mais c'est pourtant comme ça qu'on fait à chaque fois que l'on crée des GUI en POO. Ça nous donnera une plus grande souplesse par la suite.

L'héritage que l'on va faire sera donc le suivant :



Allons-y 😊

Qui dit nouvelle classe dit 2 nouveaux fichiers :

- MaFenetre.h : contiendra la définition de la classe
- MaFenetre.cpp : contiendra l'implémentation des méthodes

Edition des fichiers

[MaFenetre.h](#)

Voici le code du fichier MaFenetre.h, nous allons le commenter tout de suite après :

Code : C++ - [Sélectionner](#)

```

#ifndef DEF_MAFENETRE
#define DEF_MAFENETRE

#include <QApplication>
#include <QWidget>
#include <QPushButton>

class MaFenetre : public QWidget // On hérite de QWidget (IMPORTANT)
{
public:
    MaFenetre();

private:
    QPushButton *m_bouton;
};

#endif

```

Quelques petites explications :

Code : C++ - Sélectionner

```

#ifndef DEF_MAFENETRE
#define DEF_MAFENETRE

// Contenu

#endif

```

Là, nous protégeons le header contre les inclusions infinies grâce à cette bonne vieille méthode du #ifndef.

Code : C++ - Sélectionner

```

#include <QApplication>
#include <QWidget>
#include <QPushButton>

```

Comme nous allons hériter de QWidget, il est nécessaire d'inclure la définition de cette classe.

Par ailleurs, nous allons utiliser un QPushButton, donc on inclut le header là aussi.

Quant à QApplication, on ne l'utilise pas ici, mais on en aura besoin dans le chapitre suivant, je prépare un peu le terrain 😊

Code : C++ - Sélectionner

```

class MaFenetre : public QWidget // On hérite de QWidget (IMPORTANT)
{

```

C'est le début de la définition de la classe. Si vous vous souvenez de l'héritage, ce que j'ai fait là ne devrait pas trop vous choquer. Le ": public QWidget" signifie que notre classe hérite de QWidget. Nous récupérons donc automatiquement toutes les propriétés de QWidget.

Code : C++ - Sélectionner

```

public:
    MaFenetre();

private:
    QPushButton *m_bouton;

```

Le contenu de la classe est très simple.

Nous écrivons le prototype du constructeur. C'est un prototype minimal (`MaFenetre()`), mais cela nous suffira. Le constructeur est public, car s'il était privé on ne pourrait jamais créer d'objet à partir de cette classe 😊

Nous créons un attribut "`m_bouton`" de type `QPushButton`. Notez que celui-ci est un pointeur, il faudra donc le "construire" de manière dynamique avec l'aide du mot-clé `new`. Tous les attributs devant être privés, nous avons fait précéder cette ligne d'un "private:" qui interdira aux utilisateurs de la classe de modifier directement le bouton.

MaFenetre.cpp

Le fichier .cpp contient l'implémentation des méthodes de la classe. Comme notre classe ne contient qu'une méthode (le constructeur), le fichier .cpp ne sera donc pas long à écrire :

Code : C++ - [Sélectionner](#)

```
#include "MaFenetre.h"

MaFenetre::MaFenetre() : QWidget()
{
    setFixedSize(300, 150);

    // Construction du bouton
    m_bouton = new QPushButton("Pimp mon bouton !", this);

    m_bouton->setFont(QFont("Comic Sans MS", 14));
    m_bouton->setCursor(Qt::PointingHandCursor);
    m_bouton->setIcon(QIcon("smile.png"));
    m_bouton->move(60, 50);
}
```

Quelques explications :

Code : C++ - [Sélectionner](#)

```
#include "MaFenetre.h"
```

C'est obligatoire pour inclure les définitions de la classe.

Tout ça ne devrait pas être nouveau pour vous, nous avons fait ça de nombreuses fois dans la partie précédente du cours 😊

Code : C++ - [Sélectionner](#)

```
MaFenetre::MaFenetre() : QWidget()
{
```

L'en-tête du constructeur. Il ne faut pas oublier de le faire précéder d'un "MaFenetre::" pour que le compilateur sache à quelle classe celui-ci se rapporte.

Le ": QWidget()" sert à appeler le constructeur de QWidget en premier lieu. Parfois, on en profitera pour envoyer au constructeur de QWidget quelques paramètres, mais là on va se contenter du constructeur par défaut.

Code : C++ - [Sélectionner](#)

```
setFixedSize(300, 150);
```

Rien d'extraordinaire : on définit la taille de la fenêtre de manière fixée, pour interdire son redimensionnement.

Vous noterez qu'on n'a pas eu besoin d'écrire `fenetre.setFixedSize(300, 150);`. Pourquoi ? Parce qu'on est dans la classe. On ne fait qu'appeler une des méthodes de la classe (`setFixedSize`), méthode qui appartient à `QWidget`, et donc qui appartient aussi à notre classe puisqu'on hérite de `QWidget` 😊

J'avoue j'avoue, ce n'est pas évident de bien se repérer au début. Pourtant, vous pouvez me croire, tout ceci est logique mais ça vous paraîtra plus clair à force de pratiquer. Pas de panique donc si vous vous dites "oh mon dieu j'aurais jamais pu deviner ça 😊".

Faites-moi confiance c'est tout 😊

Code : C++ - Sélectionner

```
m_bouton = new QPushButton("Pimp mon bouton !", this);
```

C'est la ligne la plus délicate de ce constructeur.

Ici nous construisons le bouton. En effet, dans le header nous n'avons fait que créer le pointeur, mais il ne pointait vers rien jusqu'ici !

Le new permet d'appeler le constructeur de la classe QPushButton et d'affecter une adresse au pointeur.

Autre détail un tout petit peu délicat : le mot-clé this. Je vous en avais parlé dans la partie précédente du cours, en vous disant "faites-moi confiance, même si ça vous paraît inutile maintenant, ça vous sera indispensable plus tard".

Bonne nouvelle : c'est maintenant que vous découvrez un cas où le mot-clé this nous est indispensable ! En effet, le second paramètre du constructeur doit être un pointeur vers le widget parent. Quand nous faisions tout dans le main, c'était simple : il suffisait de donner le pointeur vers l'objet fenêtre. Mais là, nous sommes dans la fenêtre ! En effet, nous écrivons la classe MaFenetre. C'est donc "moi", la fenêtre, qui sert de widget parent. Pour donner le pointeur vers moi, il suffit d'écrire le mot-clé this.

Et toujours... main.cpp

Bien entendu, que serait un programme sans son main ?

Ne l'oubliions pas celui-là !

La bonne nouvelle, c'est que comme bien souvent dans les gros programmes, notre main va être tout petit. Ridiculement petit. Microscopique. Microbique même.

Code : C++ - Sélectionner

```
#include <QApplication>
#include "MaFenetre.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    MaFenetre fenetre;
    fenetre.show();

    return app.exec();
}
```

On n'a besoin d'inclure que 2 headers car nous n'utilisons que 2 classes : QApplication et MaFenetre.

Le contenu du main est très simple : on crée un objet de type MaFenetre, et on l'affiche par un appel à la méthode "show()". C'est tout 😊

Lors de la création de l'objet fenêtre, le constructeur de la classe MaFenetre est appelé. Dans son constructeur, la fenêtre définit toute seule ses dimensions et les widgets qu'elle contient (en l'occurrence, juste un bouton).

La destruction automatique des widgets enfants

Minute papillon ! On a créé dynamiquement un objet de type QPushButton dans le constructeur de la classe MaFenetre... mais on n'a pas détruit cet objet avec un delete !

En effet, tout objet créé dynamiquement avec un new implique forcément un delete quelque part. Vous avez bien retenu la leçon. Normalement, on devrait écrire le destructeur de MaFenetre, qui contiendrait ceci :

Code : C++ - Sélectionner

```
MaFenetre::~MaFenetre()
{
    delete m_bouton;
}
```

C'est comme ça qu'on doit faire en temps normal. Toutefois, Qt supprimera automatiquement le bouton lors de la destruction de la fenêtre (à la fin du main).

En effet, quand on supprime un widget parent (ici notre fenêtre), Qt supprime automatiquement tous les widgets qui se trouvent à l'intérieur (tous les widgets enfants). C'est un des avantages d'avoir dit que le QPushButton avait pour "parent" la fenêtre. Dès qu'on supprime la fenêtre, hop, Qt supprime tout ce qu'elle contient, et donc fait le delete nécessaire du bouton.

Qt nous simplifie la vie en nous évitant d'avoir à écrire tous les delete des widgets enfants. N'oubliez pas néanmoins que tout new implique normalement un delete. Ici, on profite du fait que Qt le fasse pour nous.

Compilation

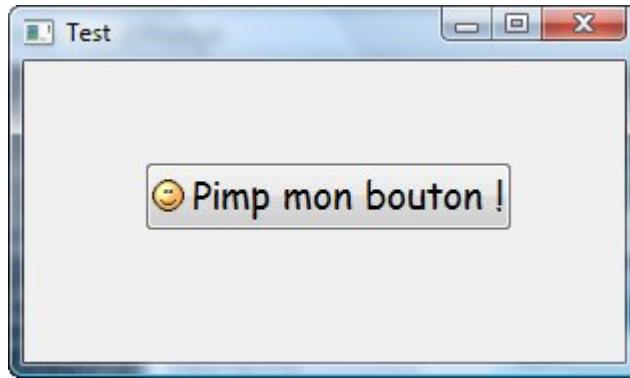
Pour la compilation, il ne faudra pas se contenter de faire un make comme les autres fois ! En effet, qu'est-ce que je vous avais dit ? "A chaque fois que la liste des fichiers de votre projet change, vous devez refaire `qmake -project` et `qmake` pour que le compilateur sache qu'il doit compiler les nouveaux fichiers".

Pensez donc à taper dans l'ordre :

1. qmake -project
2. qmake
3. make (ou mingw32-make sous Windows)

Si vous ne le faites pas, vous aurez une erreur de linker à coup sûr et la compilation échouera.

Le résultat, si tout va bien, devrait être le même que tout à l'heure :



QUOI ? TOUT CE BAZAR POUR FAIRE LA MÊME CHOSE AU FINAL ??? 😕

Mais non mais non 😕

En fait, on vient de créer des fondements beaucoup plus solides pour notre fenêtre en faisant ce qu'on vient de faire. On a déjà un peu plus découpé notre code (et avoir un code modulaire, c'est bien !) et on pourra par la suite plus facilement rajouter de nouveaux widgets et surtout... gérer les événements des widgets !

Mais tout ça, vous le découvrirez... dans le prochain chapitre !

Petit exercice : essayez de modifier (ou de surcharger) le constructeur de la classe MaFenetre pour qu'on puisse lui envoyer en paramètre la largeur et la hauteur de la fenêtre à créer.

Ainsi, vous pourrez alors définir les dimensions de la fenêtre lors de sa création dans le main.

Nous avançons dans notre découverte de Qt, c'est bien ! 😊

Vous commencez à mieux maîtriser le concept de widget et vous avez appris à organiser votre code de manière modulaire afin de servir de base solide pour les chapitres à venir.

Le programme de la suite ? Les signaux et les slots !
Nous allons faire en sorte que notre programme réagisse lorsqu'on clique sur le bouton !

Les signaux et les slots

Nous commençons à maîtriser petit à petit la création d'une fenêtre. Dans le chapitre précédent, nous avons posé de solides bases pour développer par la suite notre application. Nous avons réalisé une classe personnalisée, héritant de QWidget.

Nous allons maintenant découvrir le mécanisme des signaux et des slots, un principe propre à Qt qui est clairement un de ses points forts. Il s'agit d'une technique séduisante pour gérer les évènements au sein d'une fenêtre. Par exemple, si on clique sur un bouton, on voudrait qu'une fonction soit appelée pour réagir au clic. C'est précisément ce que nous apprendrons à faire dans ce chapitre, qui va enfin rendre votre application dynamique 😊

Le principe des signaux et slots

Le principe est plutôt simple à comprendre : une application de type GUI réagit à partir d'évènements. C'est ce qui rend votre fenêtre dynamique.

Ceux d'entre vous qui ont déjà essayé la bibliothèque SDL se souviennent peut-être de la gestion des évènements : interception des touches du clavier, des déplacements de la souris, du joystick, etc.

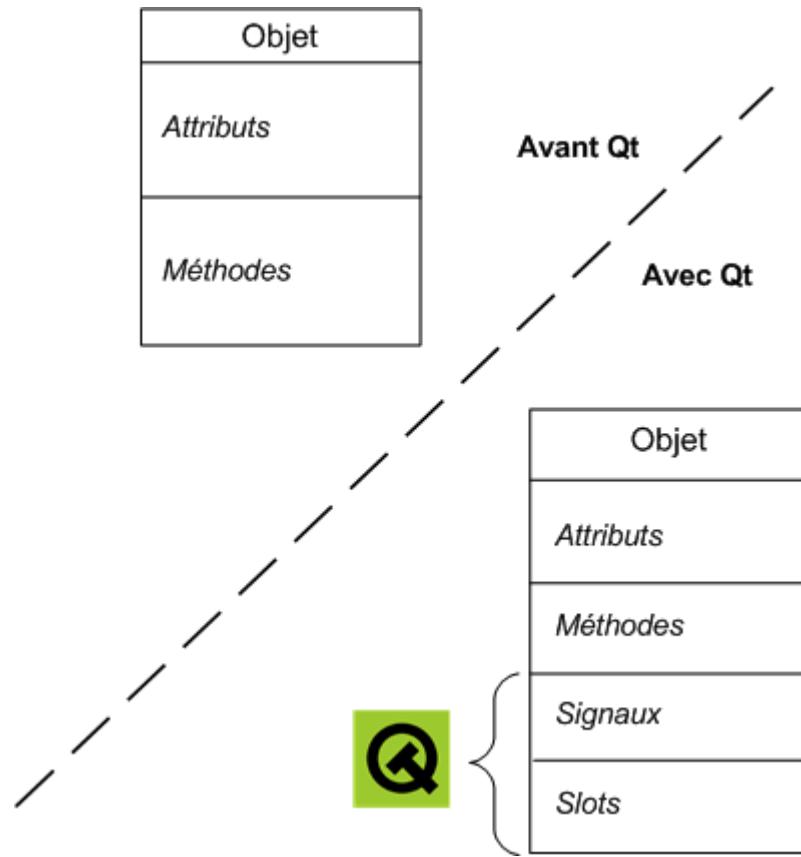
Ce que Qt propose, c'est la même chose mais à plus haut niveau : c'est donc beaucoup plus facile à gérer.

On parle de signaux et de slots, mais qu'est-ce que c'est concrètement ? C'est un concept inventé par Qt. Voici une petite définition en guise d'introduction :

- Un signal : c'est un message envoyé par un widget lorsqu'un évènement se produit.
Exemple : on a cliqué sur un bouton.
- Un slot : c'est la fonction qui est appelée lorsqu'un évènement s'est produit. On dit que le signal appelle le slot.
Concrètement, un slot est une méthode d'une classe.
Exemple : le slot quit() de la classe QApplication, qui provoque l'arrêt du programme.

Les signaux et les slots sont considérés par Qt comme des éléments d'une classe à part entière, en plus des attributs et des méthodes.

Voici un schéma qui montre ce qu'un objet pouvait contenir avant Qt, ainsi que ce qu'il peut contenir maintenant qu'on utilise Qt :



Qt rajoute des éléments appelés "Signaux" et "Slots" aux objets

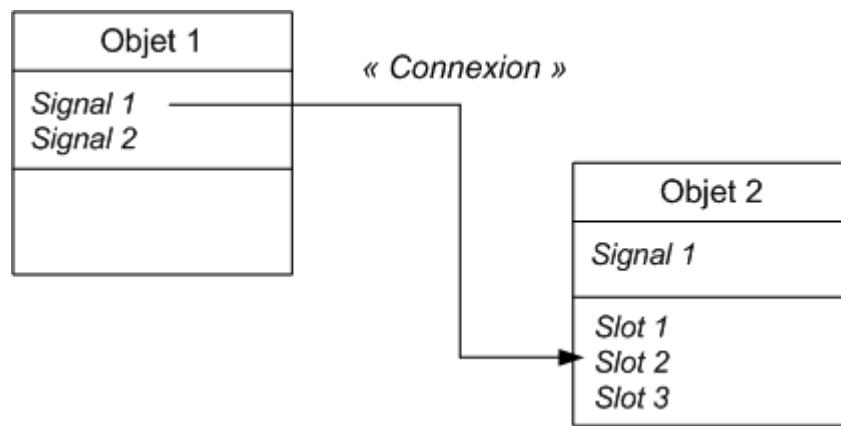
Avant Qt, un objet était constitué d'attributs et de méthodes. C'est tout.

Qt rajoute en plus la possibilité d'utiliser ce qu'il appelle des signaux et des slots pour gérer les évènements.

Un signal est un message envoyé par l'objet (par exemple "on a cliqué sur le bouton").

Un slot est une... méthode. En fait, c'est une méthode classique comme toutes les autres, à la différence près qu'elle a le droit d'être connectée à un signal.

Avec Qt, on dit que l'on connecte des signaux et des slots entre eux. Supposons que vous ayez deux objets, chacun ayant ses propres attributs, méthodes, signaux et slots (je n'ai pas représenté les attributs et les méthodes sur mon schéma pour simplifier) :



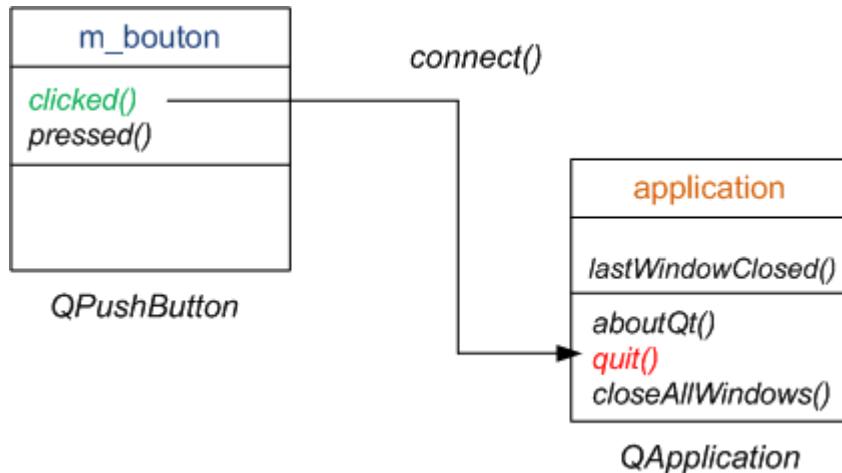
Sur le schéma ci-dessus, on a connecté le signal 1 de l'objet 1 avec le slot 2 de l'objet 2.

Il est possible de connecter un signal à plusieurs slots. Ainsi, un clic sur un bouton pourrait appeler non pas une mais plusieurs méthodes. Attention, si un signal est connecté à plusieurs slots, il est impossible de prédire dans quel ordre Qt appellera les slots.

Comble du raffinement, il est aussi possible de connecter un signal à un autre signal. Le signal d'un bouton peut donc provoquer la création du signal d'un autre widget, qui peut à son tour appeler des slots (voire appeler d'autres signaux pour provoquer une réaction en chaîne !). C'est un peu particulier et on ne verra pas ça dans ce chapitre.

Connexion d'un signal à un slot simple

Voyons un cas très concret. Je vais prendre 2 objets, l'un de type QPushButton, et l'autre de type QApplication. Dans le schéma ci-dessous, ce que vous voyez sont de vrais signaux et slots que vous allez pouvoir utiliser :



Regardez attentivement ce schéma. Nous avons d'un côté notre bouton appelé "m_bouton" (de type QPushButton), et de l'autre notre application (de type QApplication, utilisé dans le main).

Nous voudrions par exemple connecter le signal "bouton cliqué" au slot "quitter l'application". Ainsi, un clic sur le bouton provoquerait l'arrêt de l'application.

Pour ce faire, nous devons utiliser une méthode statique de la classe QObject : `connect()`.

Le principe de la méthode `connect()`

`connect()` est une méthode statique. Vous vous souvenez ce que ça veut dire ?

Une méthode statique est une méthode d'une classe que l'on peut appeler sans créer d'objet. C'est en fait exactement comme une fonction classique du langage C.

Si vous avez un trou de mémoire, allez vite relire le chapitre traitant des [méthodes statiques](#) !

Pour appeler une méthode statique, il faut faire précéder son nom du nom de la classe dans laquelle elle est déclarée. Comme `connect()` appartient à la classe QObject, il faut donc écrire :

Code : C++ - [Sélectionner](#)

```
QObject::connect();
```

La méthode `connect` prend 4 arguments :

- Un pointeur vers l'objet qui émet le signal.
- Le nom du signal que l'on souhaite "intercepter".
- Un pointeur vers l'objet qui contient le slot récepteur.
- Le nom du slot qui doit s'exécuter lorsque le signal se produit.

Pour que vous puissiez vous repérer, j'ai remis ci-contre le schéma qu'on a vu un peu plus haut. Les couleurs sont les mêmes, cela devrait vous permettre de bien visualiser à quoi correspond chaque attribut.

Il existe aussi une méthode disconnect() permettant de casser la connexion entre 2 objets, mais on n'en parlera pas ici car on en a rarement besoin.

Utilisation de la méthode connect() pour quitter

Revenons au code, et plus précisément au constructeur de MaFenetre (fichier MaFenetre.cpp). Ajoutez cette ligne :

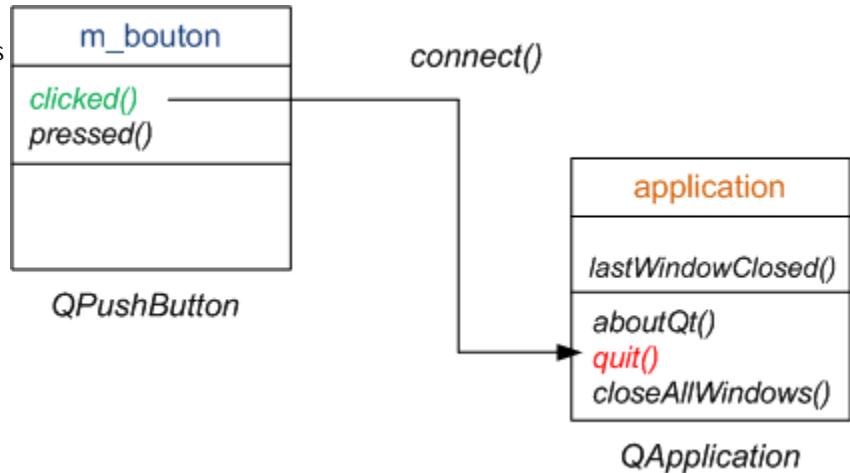
Code : C++ - [Sélectionner](#)

```
#include "MaFenetre.h"

MaFenetre::MaFenetre() : QWidget()
{
    setFixedSize(300, 150);

    m_bouton = new QPushButton("Quitter", this);
    m_bouton->setFont(QFont("Comic Sans MS", 14));
    m_bouton->move(110, 50);

    // Connexion du clic du bouton à la fermeture de l'application
    QObject::connect(m_bouton, SIGNAL(clicked()), qApp, SLOT(quit()));
}
```



connect() est une méthode de la classe QObject. Comme notre classe MaFenetre hérite de QObject indirectement, elle possède elle aussi cette méthode. Cela signifie que dans ce cas, et dans ce cas uniquement, on peut enlever le préfixe `QObject::` devant le connect() pour appeler la méthode statique.

J'ai choisi de conserver ce préfixe dans le cours pour rappeler qu'il s'agit d'une méthode statique, mais sachez donc qu'il n'a rien d'obligatoire si la méthode est appelée depuis une classe fille de QObject.

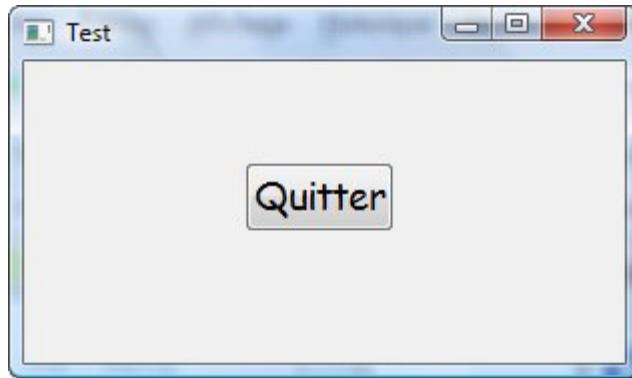
Etudions attentivement cette ligne et plus particulièrement les paramètres que l'on envoie à connect() :

- m_bouton : c'est un pointeur vers le bouton qui va émettre le signal. Facile.
 - SIGNAL(clicked()) : là c'est assez perturbant comme façon d'envoyer un paramètre. En fait, SIGNAL() est une macro du préprocesseur. Qt transformera ça en un code "acceptable" pour la compilation. Le but de cette technique est de vous faire écrire un code court et compréhensible. Ne cherchez pas à comprendre comment Qt fait pour transformer le code, on s'en fout 😊
 - qApp : c'est un pointeur vers l'objet de type QApplication que nous avons créé dans le main. D'où sort ce pointeur ? Euh... joker 😊
- En fait, Qt crée automatiquement un pointeur appelé qApp vers l'objet de type QApplication que nous avons créé. Ce pointeur est défini dans le header <QApplication>, que nous avons inclus dans "MaFenetre.h".
- SLOT(quit()) : c'est le slot qui doit être appelé lorsqu'on a cliqué sur le bouton. Là encore, il faut utiliser la macro SLOT() pour que Qt traduise ce code "bizarre" en quelque chose de compilable.

Le slot quit() de notre objet de type QApplication est un slot prédéfini. Il en existe d'autres, comme aboutQt() qui affiche une fenêtre "A propos de Qt".

Parfois, pour ne pas dire souvent, les slots prédéfinis par Qt ne nous suffiront pas. Nous apprendrons dans la suite de ce chapitre à créer les nôtres.

Testons notre code ! La fenêtre qui s'ouvre est la suivante :



Rien de bien extraordinaire à première vue. Sauf que... si vous cliquez sur le bouton "Quitter", le programme s'arrête !
Hourra, on vient de réussir à connecter notre premier signal à un slot ! 😊

Utilisation de la méthode connect() pour afficher "A propos"

On peut faire un autre essai pour se faire un peu plus la main si vous voulez. Je vous ai parlé d'un autre slot de QApplication : aboutQt().

Je vous propose de créer un second bouton qui se chargera d'afficher la fenêtre "A propos de Qt".

Je vous laisse rédiger le code tous seuls comme des grands.

...

...

C'est bon ?

Voici le code final 😊

Code : C++ - [Sélectionner](#)

```
#include "MaFenetre.h"

MaFenetre::MaFenetre() : QWidget()
{
    setFixedSize(300, 150);

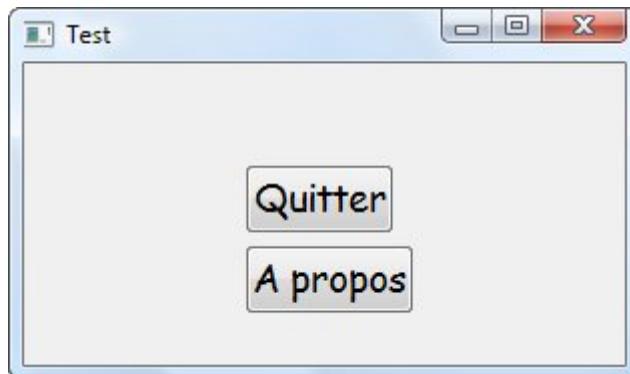
    m_quitter = new QPushButton("Quitter", this);
    m_quitter->setFont(QFont("Comic Sans MS", 14));
    m_quitter->move(110, 50);
    QObject::connect(m_quitter, SIGNAL(clicked()), qApp, SLOT(quit()));

    m_aPropos = new QPushButton("A propos", this);
    m_aPropos->setFont(QFont("Comic Sans MS", 14));
    m_aPropos->move(110, 90);
    QObject::connect(m_aPropos, SIGNAL(clicked()), qApp, SLOT(aboutQt()));
}
```

Vous noterez que j'ai pris la liberté de nommer les boutons avec des noms un peu plus compréhensibles.

Bien entendu, le fichier MaFenetre.h a un peu changé lui aussi du coup pour déclarer les attributs "m_quitter" et "m_aPropos", mais vous êtes assez grands pour le faire sans moi 😊

Le résultat est une fenêtre qui affiche 2 boutons :



Le bouton "Quitter" ferme toujours l'application.

Quant à "A propos", il provoque l'ouverture de la fenêtre "A propos de Qt".



Des paramètres dans les signaux et slots

La méthode statique `connect()` est assez originale, vous l'avez vu. Il s'agit justement d'une des particularités de Qt que l'on ne retrouve pas dans les autres bibliothèques.

Ces autres bibliothèques, comme wxWidgets par exemple, utilisent à la place de nombreuses macros et se servent du mécanisme un peu complexe et délicat des pointeurs de fonction (pour indiquer l'adresse de la fonction à appeler en mémoire).

Il y a d'autres avantages à utiliser la méthode `connect()` avec Qt. On va ici découvrir que les signaux et les slots peuvent s'échanger

des paramètres !

Dessin de la fenêtre

Dans un premier temps, nous allons placer de nouveaux widgets dans notre fenêtre.
Vous pouvez enlever les boutons, on ne va plus s'en servir ici.

A la place, je souhaite vous faire utiliser 2 nouveaux widgets :

- QSlider : un curseur qui permet de définir une valeur.
- QLCDNumber : un widget qui affiche un nombre.

On va aller un peu plus vite, je vous donne le code directement pour créer ça.
Tout d'abord, le header :

Code : C++ - [Sélectionner](#)

```
#ifndef DEF_MAFENETRE
#define DEF_MAFENETRE

#include <QApplication>
#include <QWidget>
#include <QPushButton>
#include <QLCDNumber>
#include <QSlider>

class MaFenetre : public QWidget
{
public:
    MaFenetre();

private:
    QLCDNumber *m_lcd;
    QSlider *m_slider;
};

#endif
```

J'ai donc enlevé les boutons comme vous pouvez le voir, et rajouté un QLCDNumber et un QSlider.
Surtout, n'oubliez pas d'inclure le header de ces classes pour pouvoir les utiliser. J'ai gardé l'include du QPushButton ici, ça ne fait pas de mal de le laisser mais si vous ne comptez pas le réutiliser vous pouvez le virer sans crainte.

Et le fichier .cpp :

Code : C++ - [Sélectionner](#)

```
#include "MaFenetre.h"

MaFenetre::MaFenetre() : QWidget()
{
    setFixedSize(200, 100);

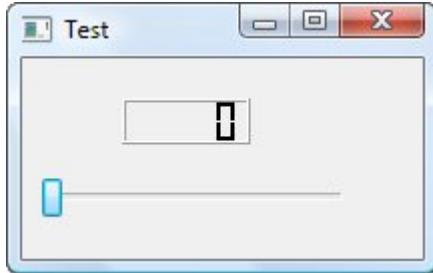
    m_lcd = new QLCDNumber(this);
    m_lcd->setSegmentStyle(QLCDNumber::Flat);
    m_lcd->move(50, 20);

    m_slider = new QSlider(Qt::Horizontal, this);
    m_slider->setGeometry(10, 60, 150, 20);
}
```

Les détails ne sont pas très importants. J'ai modifié le type d'afficheur LCD pour qu'il soit plus lisible (avec setSegmentStyle). Quant

au slider, j'ai rajouté un paramètre pour qu'il apparaisse horizontalement (sinon il est vertical).

Voilà qui est fait. Avec ce code, cette petite fenêtre devrait s'afficher :



Connexion avec des paramètres

Maintenant... connexionooooon !

C'est là que les choses deviennent intéressantes. On veut que l'afficheur LCD change de valeur en fonction de la position du curseur du slider.

On dispose du signal et du slot suivant :

- Le signal `valueChanged(int)` du `QSlider` : il est émis dès que l'on change la valeur du curseur du slider en le déplaçant. La particularité de ce signal est qu'il envoie un paramètre de type `int` (la nouvelle valeur du slider).
- Le slot `display(int)` du `QLCDNumber` : il affiche la valeur qui lui est passée en paramètre.

La connexion se fait avec le code suivant :

Code : C++ - [Sélectionner](#)

```
QObject::connect(m_slider, SIGNAL(valueChanged(int)), m_lcd, SLOT(display(int)));
```

Bizarre n'est-ce pas ? 😱

Il suffit d'indiquer le type du paramètre envoyé, ici un `int`, sans donner de nom à ce paramètre. Qt fait automatiquement la connexion entre le signal et le slot et "transmet" le paramètre au slot.

Le transfert de paramètre se fait comme ceci :

```
QObject::connect(m_slider, SIGNAL(valueChanged(int)), m_lcd, SLOT(display(int));
```

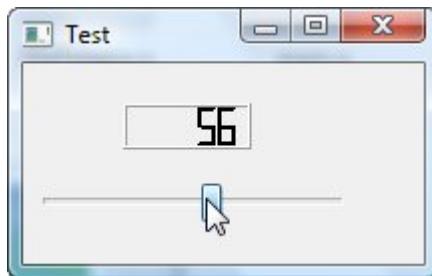
Ici il n'y a qu'un paramètre à transmettre, c'est donc simple. Sachez toutefois qu'il pourrait très bien y avoir plusieurs paramètres.

Le type des paramètres doivent correspondre absolument !

Vous ne pouvez pas connecter un signal qui envoie `(int, double)` à un slot qui reçoit `(int, int)`. C'est un des avantages du mécanisme des signaux et des slots : il respecte le type des paramètres. Veillez donc à ce que les signatures soient identiques entre votre signal et votre slot.

En revanche, un signal peut envoyer plus de paramètres à un slot que celui-ci ne peut en recevoir. Dans ce cas, les paramètres supplémentaires seront ignorés.

Résultat : quand on change la valeur du slider, le LCD affiche la valeur correspondante !



Mais comment je sais moi quels sont les signaux et les slots que proposent chacune des classes ? Et aussi, comment je sais qu'un signal envoie un int en paramètre ?

La réponse devrait vous paraître simple les amis : la doc, la doc, la doc ! 😊

Si vous regardez la [documentation de la classe QLCDNumber](#), vous pouvez voir au début la liste de ses propriétés (attributs) et ses méthodes. Un peu plus bas, vous avez la liste des slots ("Public Slots") et des signaux ("Signals") qu'elle possède !

Les signaux et les slots sont hérités comme les attributs et méthodes. Et ça, c'est génial, bien qu'un peu déroutant au début. Vous noterez donc qu'en plus des [slots propres à QLCDNumber](#), celui-ci propose de nombreux autres slots qui ont été définis dans sa classe parente QWidget, et même des slots issus de QObject ! Vous pouvez par exemple lire :

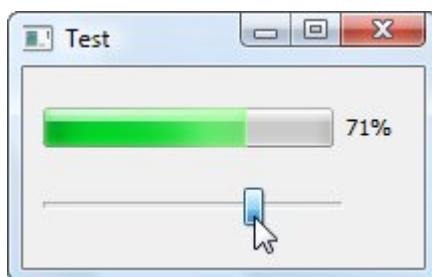
- 19 public slots inherited from QWidget
- 1 public slot inherited from QObject

N'hésitez pas à consulter les slots (ou signaux) qui sont hérités des classes parentes. Parfois on va vous demander d'utiliser un signal ou un slot que vous ne verrez pas dans la page de documentation de la classe : vérifiez donc si celui-ci n'est pas défini dans une classe parente !

Exercice

Pour vous entraîner, je vous propose de réaliser une petite variation du code source précédent.

Au lieu d'afficher le nombre avec un QLCDNumber, affichez-le sous la forme d'une jolie barre de progression comme ceci :



Je ne vous donne que 3 indications qui devraient vous suffire :

- La barre de progression est gérée par un QProgressBar
- Il faut donner des dimensions à la barre de progression pour qu'elle apparaisse correctement, à l'aide de la méthode setGeometry() que l'on a déjà vue auparavant.
- Le slot récepteur du QProgressBar est setValue(int). Il s'agit d'[un de ses slots](#), mais la documentation vous indique qu'il y en a d'autres. Par exemple, reset() remet à zéro la barre de progression. Pourquoi ne pas ajouter un bouton qui remetttrait à zéro la barre de progression ?

C'est tout. Bon courage 😊

Créer ses propres signaux et slots

Voici maintenant une partie très intéressante, bien que plus délicate. Nous allons créer nos propres signaux et slots.

En effet, si en général les signaux et slots par défaut suffisent, il n'est pas rare que l'on se dise "Zut, le signal (ou le slot) dont j'ai besoin n'existe pas". C'est dans un cas comme celui-là qu'il devient indispensable de créer son widget personnalisé.

Pour pouvoir créer son propre signal ou slot dans une classe, il faut que celle-ci dérive directement ou indirectement de QObject. C'est le cas de notre classe MaFenetre : elle hérite de QWidget, qui hérite de QObject. On a donc le droit de créer des signaux et des slots dans MaFenetre.

Nous allons commencer par créer notre propre slot, puis nous verrons comment créer notre propre signal.

Créer son propre slot

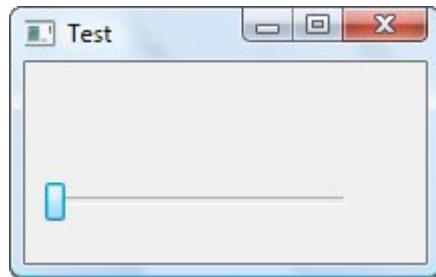
Je vous rappelle tout d'abord qu'un slot n'est rien d'autre qu'une méthode que l'on peut connecter à un signal. Nous allons donc créer une méthode, mais en suivant quelques règles un peu particulières...

Le but du jeu

Pour nous entraîner, nous allons inventer un cas où le slot dont on a besoin n'existe pas.

Je vous propose de conserver le QSlider (je l'aime bien celui-là 😊) et de ne garder que ça sur la fenêtre. Nous allons faire en sorte que le QSlider contrôle la largeur de la fenêtre.

Votre fenêtre doit ressembler à cela :



Nous voulons que le signal valueChanged(int) du QSlider puisse être connecté à un slot de notre fenêtre (de type MaFenetre). Ce nouveau slot aura pour rôle de modifier la largeur de la fenêtre.

Comme [il n'existe pas de slot "changerLargeur" dans la classe QWidget](#), nous allons devoir le créer.

Pour créer ce slot, il va falloir modifier un peu notre classe MaFenetre. Commençons par le header.

Le header (MaFenetre.h)

Dès que l'on doit créer un signal ou un slot personnalisé, il est nécessaire de définir une macro dans le header de la classe.

Cette macro porte le nom de Q_OBJECT (tout en majuscules) et doit être placée tout au début de la déclaration de la classe :

Code : C++ - [Sélectionner](#)

```

class MaFenetre : public QWidget
{
    Q_OBJECT

    public:
    MaFenetre();

    private:
    QSlider *m_slider;
};

```

Pour le moment, notre classe ne définit qu'un attribut (le QSlider, privé) et une méthode (le constructeur, public).

La macro Q_OBJECT "prépare" en quelque sorte le compilateur à accepter un nouveau mot-clé : "slot". Nous allons maintenant pouvoir créer une section "slots", comme ceci :

Code : C++ - [Sélectionner](#)

```

class MaFenetre : public QWidget
{
    Q_OBJECT

    public:
    MaFenetre();

    public slots:
    void changerLargeur(int largeur);

    private:
    QSlider *m_slider;
};

```

Vous noterez la nouvelle section "public slots". Je rends toujours mes slots publics. On peut aussi les mettre privés mais ils seront quand même accessibles de l'extérieur car Qt a besoin de pouvoir appeler un slot depuis n'importe quel autre widget.

A part ça, le prototype de notre slot-méthode est tout à fait classique. Il ne nous reste plus qu'à l'implémenter dans le .cpp.

L'implémentation (MaFenetre.cpp)

L'implémentation est d'une simplicité redoutable. Regardez :

Code : C++ - [Sélectionner](#)

```

void MaFenetre::changerLargeur(int largeur)
{
    setFixedSize(largeur, 100);
}

```

Le slot prend en paramètre un entier : la nouvelle largeur de la fenêtre.

Il se contente d'appeler la méthode setFixedSize de la fenêtre et de lui envoyer la nouvelle largeur qu'il a reçue.

Connexion

Bien, voilà qui est fait. Enfin presque : il faut encore connecter notre QSlider au slot de notre fenêtre. Où va-t-on faire ça ? Dans le constructeur de la fenêtre (toujours dans MaFenetre.cpp) :

Code : C++ - [Sélectionner](#)

```

MaFenetre::MaFenetre() : QWidget()
{
    setFixedSize(200, 100);

    m_slider = new QSlider(Qt::Horizontal, this);
    m_slider->setRange(200, 600);
    m_slider->setGeometry(10, 60, 150, 20);

    QObject::connect(m_slider, SIGNAL(valueChanged(int)), this, SLOT(changerLargeur(int)));
}

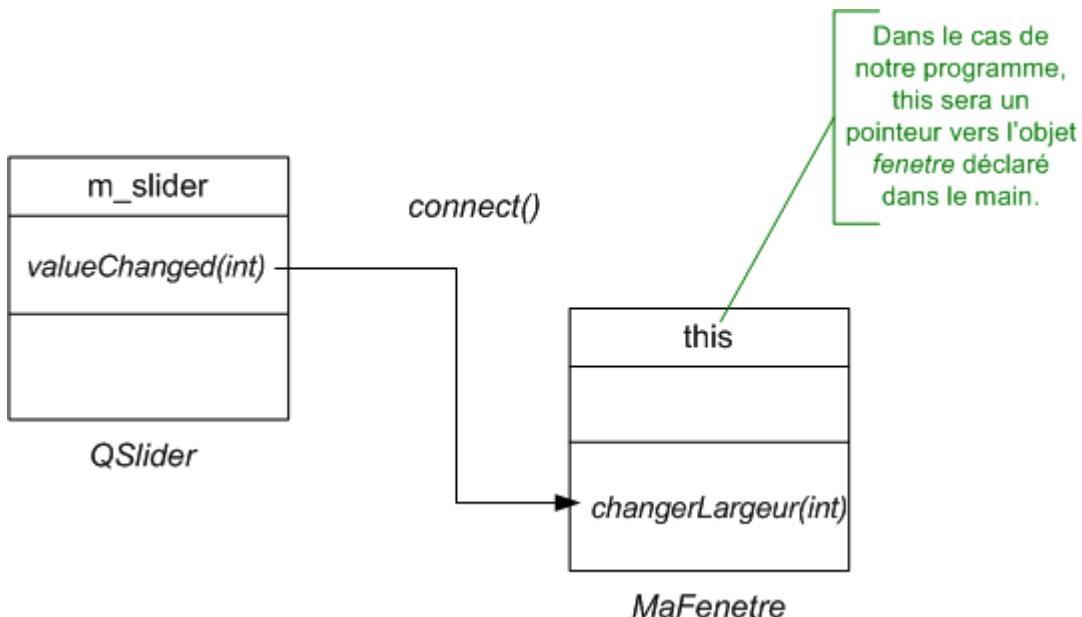
```

5
6
3
4

J'ai volontairement modifié les différentes valeurs que peut prendre notre slider pour le limiter entre 200 et 600 avec la méthode `setRange()`. Ainsi, on est sûr que notre fenêtre ne pourra ni être plus petite que 200 pixels de largeur, ni être plus grande que 600 pixels de largeur.

La connexion se fait entre le signal `valueChanged(int)` de notre `QSlider`, et le slot `changerLargeur(int)` de notre classe `MaFenetre`. Vous voyez là encore un exemple où `this` est indispensable : il faut pouvoir indiquer un pointeur vers l'objet actuel (la fenêtre) et seul `this` peut faire ça !

Schématiquement, on a réalisé la connexion suivante :



Compilation

Avec toutes les nouveautés que nous venons d'utiliser par rapport au C++, la compilation par un make ne suffira pas.

Je vous avais dit qu'il fallait refaire un qmake à chaque fois que les fichiers du projet changeaient. En fait j'ai un peu menti 😊
Comme vous utilisez la macro `Q_OBJECT`, Qt a besoin d'appeler un pré-compilateur qui lui est propre appelé le moc (Meta-Object Compiler).

Rassurez-vous, vous n'avez rien à faire de spécial. Relancez juste un qmake avant de faire votre make, et Qt fera le travail de "traduction" du slot en quelque chose de compréhensible pour le compilateur C++.
Vous noterez que le qmake a provoqué la création d'un fichier intermédiaire `moc_MaFenetre.cpp`, ce qui est parfaitement normal. Ce fichier fournit des informations indispensables au compilateur.

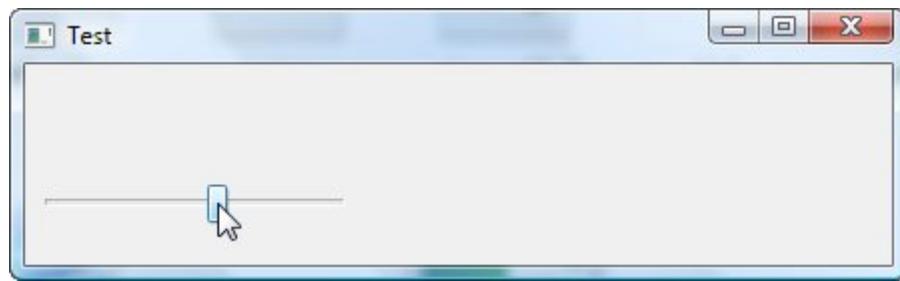
Vous pouvez ensuite faire un make, la compilation devrait bien se passer.

Souvenez-vous ! Si jamais lors de la compilation vous rencontrez l'erreur suivante :

`undefined reference to 'vtable for MaFenetre'`

... cela signifie que vous n'avez pas fait de qmake avant. Si le moc ne s'est pas exécuté auparavant, la compilation échouera.

Vous pouvez enfin admirer le résultat. Ouf ! 😊

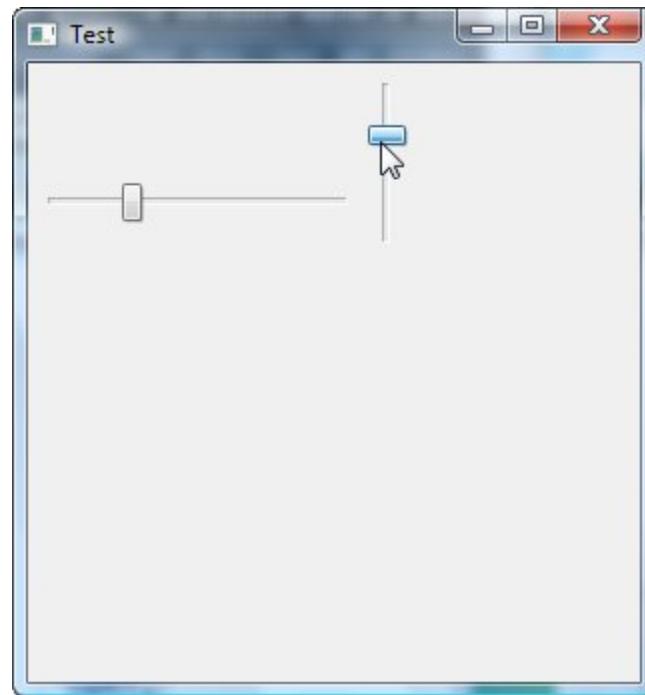


Amusez-vous à redimensionner la fenêtre comme bon vous semblera avec le slider. Comme nous avons fixé les limites du slider entre 200 et 600, la largeur de la fenêtre restera comprise entre 200 et 600 pixels.

Exercice : redimensionner la fenêtre en hauteur

Voici un petit exercice, mais qui va vous forcer à travailler (bande de fainéants, vous me regardez faire depuis tout à l'heure 😴). Je vous propose de créer un second QSlider, vertical cette fois, qui contrôlera la hauteur de la fenêtre. Pensez à bien définir des limites appropriées pour les valeurs de ce nouveau slider.

Vous devriez obtenir un résultat qui ressemblera à ça :



Si vous voulez "conserver" la largeur pendant que vous modifiez la hauteur, et inversement, vous aurez besoin d'utiliser les méthodes accesseur width() (largeur actuelle) et height() (hauteur actuelle).

Vous comprendrez très certainement l'intérêt de ces informations lorsque vous coderez. Au boulot !

Créer son propre signal

Il est plus rare d'avoir à créer son signal que son slot, mais cela peut arriver.

Je vous propose de réaliser le programme suivant : si le slider horizontal arrive à sa valeur maximale (600 dans notre cas), alors on émet un signal "agrandissementMax". Notre fenêtre doit pouvoir émettre l'information comme quoi elle est agrandie au maximum. Après, nous connecterons ce signal à un slot pour vérifier que notre programme réagit correctement.

Le header (MaFenetre.h)

Commençons par changer le header :

Code : C++ - Sélectionner

```
class MaFenetre : public QWidget
{
    Q_OBJECT

public:
    MaFenetre();

public slots:
    void changerLargeur(int largeur);

signals:
    void agrandissementMax();

private:
    QSlider *m_slider;
};
```

On a ajouté une section "signals". Les signaux se présentent en pratique sous forme de méthodes (comme les slots) à la différence près qu'on ne les implémente pas dans le .cpp. En effet, c'est Qt qui le fait pour nous. Si vous tentez d'implémenter un signal, vous aurez une erreur du genre "Multiple definition of...".

Un signal peut passer un ou plusieurs paramètres. Dans notre cas, il n'en envoie aucun.
Un signal doit toujours renvoyer void.

L'implémentation (MaFenetre.cpp)

Maintenant que notre signal est défini, il faut que notre classe puisse l'émettre à un moment.

Quand est-ce qu'on sait que la fenêtre a été agrandie au maximum ? Dans le slot changerLargeur ! Il suffit de tester dans ce slot si la largeur correspond au maximum (600), et d'émettre alors le signal "Youhou, j'ai été agrandie au maximum !".

Retournons dans MaFenetre.cpp et implémentons ce test qui émet le signal depuis changerLargeur :

Code : C++ - Sélectionner

```
void MaFenetre::changerLargeur(int largeur)
{
    setFixedSize(largeur, height());

    if (largeur == 600)
    {
        emit agrandissementMax();
    }
}
```

Notre méthode s'occupe toujours de redimensionner la fenêtre, mais vérifie en plus si la largeur a atteint le maximum (600). Si c'est le cas, elle émet le signal agrandissementMax().

Pour émettre un signal, on utilise le mot-clé `emit`, là encore un terme inventé par Qt qui n'existe pas en C++. L'avantage est que c'est très lisible, on comprend "Emettre le signal agrandissementMax()".

Ici, notre signal n'envoie pas de paramètres. Toutefois, sachez que si vous voulez envoyer un paramètre c'est très simple. Il suffit d'appeler votre signal comme ceci : `emit monSignal(parametre1, parametre2, ...);`

[Connexion](#)

Il ne nous reste plus qu'à connecter notre nouveau signal à un slot. Vous pouvez connecter ce signal au slot que vous voulez. Personnellement, je propose de le connecter à l'application (à l'aide du pointeur global qApp) pour provoquer l'arrêt du programme. Ca n'a pas trop de sens je suis d'accord, mais c'est juste pour s'entraîner et vérifier que ça fonctionne. Vous aurez l'occasion de faire des connexions plus logiques plus tard, je ne m'en fais pas pour ça 😊

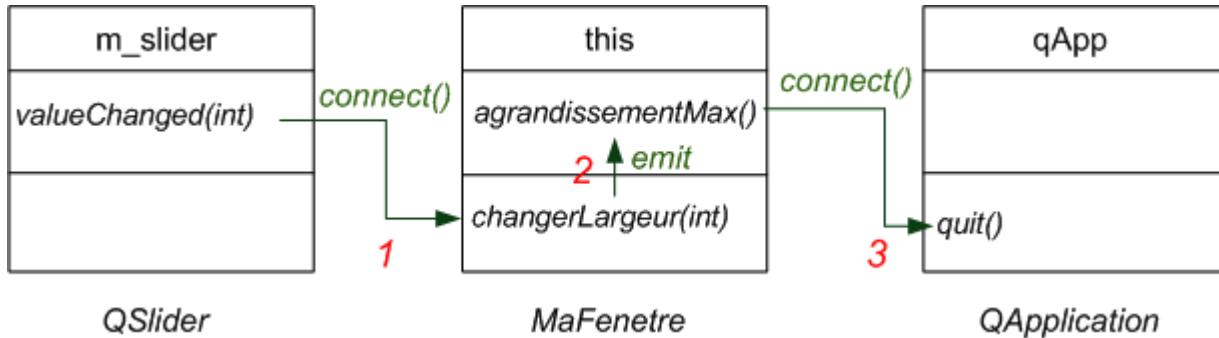
Dans le constructeur de MaFenetre, je rajoute donc :

Code : C++ - [Sélectionner](#)

```
QObject::connect(this, SIGNAL(agrandissementMax()), qApp, SLOT(quit()));
```

Vous pouvez tester le résultat : normalement le programme s'arrête quand la fenêtre est agrandie au maximum.

Le schéma des signaux qu'on vient d'émettre et connecter est le suivant :



Dans l'ordre, voici ce qui s'est passé :

1. Le signal `valueChanged` du slider a appelé le slot `changerLargeur` de la fenêtre.
2. Le slot a fait ce qu'il avait à faire (changer la largeur de la fenêtre) et a vérifié si la fenêtre était arrivée à sa taille maximale. Lorsque cela a été le cas, le signal personnalisé `agrandissementMax()` a été émis.
3. Le signal `agrandissementMax()` de la fenêtre était connecté au slot `quit()` de l'application, ce qui a provoqué la fermeture du programme.

Et voilà comment le déplacement du slider peut, par réaction en chaîne, provoquer la fermeture du programme !
Bien entendu, ce schéma peut être aménagé et complexifié selon les besoins de votre application.

Maintenant que vous savez créer vos propres slots et signaux, vous avez toute la souplesse nécessaire pour faire ce que vous voulez ! 😊

Eh ben dites donc les amis, que de nouveautés dans ce chapitre décidément !

Les signaux et les slots, c'est vraiment ce qui fait la force de Qt... mais ses détracteurs disent que c'est une erreur d'avoir voulu "modifier" le langage C++. En effet, la compilation est plus lourde car il y a des étapes de pré-compilation à effectuer impérativement si on veut que le code soit compilable. C'est un point de vue qui se défend.

L'avantage de ce système, et ça personne ne le discute, c'est qu'il est robuste. On dispose d'une extraordinaire souplesse pour faire communiquer des objets entre eux :

- Un signal peut appeler le slot d'un autre objet pour l'informer d'un évènement.
- Un signal peut appeler plusieurs slots d'objets différents si nécessaire pour faire plusieurs traitements.
- Un signal peut être connecté à un autre signal directement, qui lui-même peut être raccordé à un autre signal (réaction en chaîne) ou appeler un slot.
- La connexion entre un signal et un slot permet d'échanger un ou plusieurs paramètres.
- L'échange de paramètres entre le signal et le slot est sécurisé : Qt vérifie que la signature du signal correspond bien à celle du slot.

Les autres bibliothèques, comme wxWidgets, utilisent un ensemble de macros, moins lisibles mais qui ne nécessitent pas l'utilisation d'outils intermédiaires comme le moc.

Bref, profitez à fond des signaux et des slots, avec ça vous pouvez vraiment faire ce que vous voulez 😊

Les boîtes de dialogue usuelles

Après un chapitre sur les signaux et les slots riche en nouveaux concepts, on relâche ici un peu la pression. Nous allons découvrir les boîtes de dialogue usuelles, aussi appelées "common dialogs" par nos amis anglophones.

Qu'est-ce qu'une boîte de dialogue usuelle ? C'est une fenêtre qui sert à remplir une fonction bien précise. Par exemple, on connaît la boîte de dialogue "message" qui affiche un message et ne vous laisse d'autre choix que de cliquer sur le bouton OK. Ou encore la boîte de dialogue "ouvrir un fichier", "enregistrer un fichier", "sélectionner une couleur", etc. On ne s'amuse pas à recréer "à la main" ces fenêtres à chaque fois. On profite de fonctions système pour ouvrir des boîtes de dialogue pré-construites.

Qt s'adapte à l'OS pour afficher une boîte de dialogue qui corresponde aux formes habituelles de votre OS.

En clair : attendez-vous à un chapitre simple qui vous donnera de nombreux outils pour pouvoir interagir avec l'utilisateur de votre programme !

Afficher un message

Le premier type de boîte de dialogue que nous allons voir est le plus courant : la boîte de dialogue "afficher un message".

Nous allons créer un bouton sur notre fenêtre de type MaFenetre qui appellera un slot personnalisé. Ce slot ouvrira la boîte de dialogue. En clair, un clic sur le bouton doit pouvoir ouvrir la boîte de dialogue.

Les boîtes de dialogue "afficher un message" sont contrôlées par la classe QMessageBox. Vous pouvez commencer par faire l'include correspondant dans "MaFenetre.h" pour ne pas l'oublier : #include <QMessageBox>.

Quelques rappels et préparatifs

Pour que l'on soit sûr de travailler ensemble sur le même code, je vous donne le code source des fichiers MaFenetre.h et MaFenetre.cpp sur lesquels je vais travailler. Ils ont été simplifiés au maximum histoire d'éviter le superflu.

Code : C++ - [Sélectionner](#)

```
// MaFenetre.h

#ifndef DEF_MAFENETRE
#define DEF_MAFENETRE

#include <QApplication>
#include <QWidget>
#include <QPushButton>
#include <QMessageBox>

class MaFenetre : public QWidget
{
    Q_OBJECT

public:
    MaFenetre();

public slots:
    void ouvrirDialogue();

private:
    QPushButton *m_boutonDialogue;
};

#endif
```

Code : C++ - [Sélectionner](#)

```
// MaFenetre.cpp

#include "MaFenetre.h"

MaFenetre::MaFenetre() : QWidget()
{
    setFixedSize(230, 120);

    m_boutonDialogue = new QPushButton("Ouvrir la boîte de dialogue", this);
    m_boutonDialogue->move(40, 50);

    QObject::connect(m_boutonDialogue, SIGNAL(clicked()), this, SLOT(ouvrirDialogue()));
}

void MaFenetre::ouvrirDialogue()
{
    // Vous insérerez le code d'ouverture des boîtes de dialogue ici
}
```

C'est très simple. Nous avons créé un bouton dans la boîte de dialogue qui appelle le slot personnalisé `ouvrirDialogue()`. C'est dans ce slot que nous nous chargerons d'ouvrir une boîte de dialogue.

Au cas où certains se poseraient la question, notre `main.cpp` n'a pas changé. Allez, je vous le redonne. Je suis trop sympa je sais, ne me remerciez pas 

Code : C++ - [Sélectionner](#)

```
// main.cpp

#include <QApplication>
#include "MaFenetre.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    MaFenetre fenetre;
    fenetre.show();

    return app.exec();
}
```

Ouvrir une boîte de dialogue avec une méthode statique

Bien, place à l'action maintenant !

La classe QMessageBox permet de créer des objets de type QMessageBox (comme toute classe qui se respecte ) mais on utilise majoritairement ses méthodes statiques pour des raisons de simplicité. Nous commencerons donc par découvrir les méthodes statiques, qui se comportent je le rappelle comme de simples fonctions. Elles ne nécessiteront pas de créer d'objet.

QMessageBox::information

La méthode statique `information()` permet d'ouvrir une boîte de dialogue constituée d'une icône "information". Son prototype est le suivant :

Code : C++ - [Sélectionner](#)



StandardButton QMessageBox::information(QWidget *parent, const QString &title, const QString &text, const QMessageBox::StandardButtons &buttons)

Seuls les 3 premiers paramètres sont obligatoires, les autres ayant comme vous le voyez une valeur par défaut. Ces 3 premiers paramètres sont :

- parent : un pointeur vers la fenêtre parente (qui doit être de type QWidget ou hériter de QWidget). Vous pouvez envoyer NULL en paramètre si vous ne voulez pas que votre boîte de dialogue ait une fenêtre parente, mais ce sera plutôt rare.
- title : le titre de la boîte de dialogue (affiché en haut de la fenêtre).
- text : le texte affiché au sein de la boîte de dialogue.

Testons donc un code très simple. Voici le code du slot `ouvrirDialogue()` :

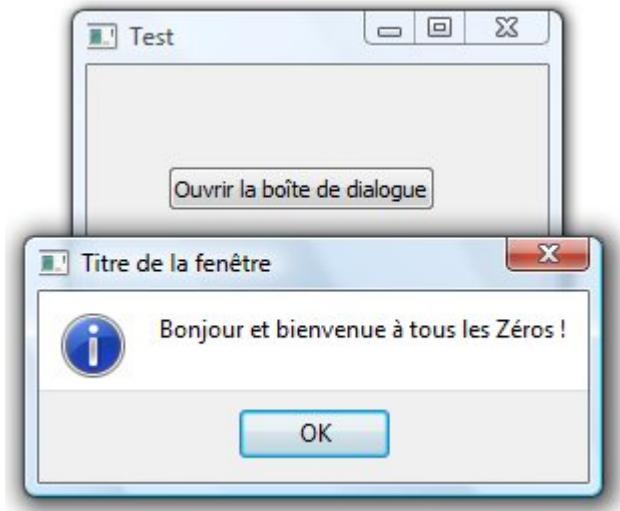
Code : C++ - [Sélectionner](#)



```
void MaFenetre::ouvrirDialogue()
{
    QMessageBox::information(this, "Titre de la fenêtre", "Bonjour et bienvenue à tous les Ze...");
}
```

L'appel de la méthode statique se fait donc comme celui d'une fonction classique, à la différence près qu'il faut mettre en préfixe le nom de la classe dans laquelle elle est définie (d'où le "QMessageBox::" avant).

Le résultat est une boîte de dialogue comme vous avez l'habitude d'en voir, constituée d'un bouton OK :



Vous noterez que lorsque la boîte de dialogue est ouverte, on ne peut plus accéder à sa fenêtre parente qui est derrière. On dit que la boîte de dialogue est une fenêtre modale : c'est une fenêtre qui "bloque" temporairement son parent en attente d'une réponse de l'utilisateur.

A l'inverse, on dit qu'une fenêtre est non modale quand on peut toujours accéder à la fenêtre derrière. C'est le cas en général des boîtes de dialogue "Rechercher un texte" dans les éditeurs de texte.

Comble du raffinement (j'aime bien cette expression 😊), il est même possible de mettre en forme son message à l'aide de balises (X)HTML pour ceux qui connaissent. Si vous ne connaissez pas, il est toujours temps d'[apprendre le HTML](#), j'ai fait un tuto il faut en profiter 😊

Exemple de boîte de dialogue "enrichie" avec du code HTML :

Code : C++ - Sélectionner

```
QMessa...Box::information(this, "Titre de la fenêtre", "Bonjour et bienvenue à tous les Zéros !")
```



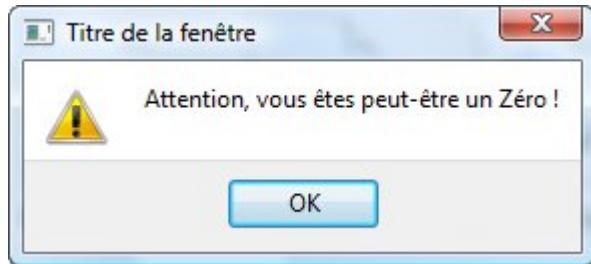
QMessageBox::warning

Si la boîte de dialogue "information" sert à informer l'utilisateur par un message, la boîte de dialogue warning le met en garde contre quelque chose. Elle est généralement accompagnée d'un "ding" caractéristique.

Elle s'utilise de la même manière que QMessageBox::information, mais cette fois l'icône change :

Code : C++ - Sélectionner

```
QMessa...Box::warning(this, "Titre de la fenêtre", "Attention, vous êtes peut-être un Zéro")
```



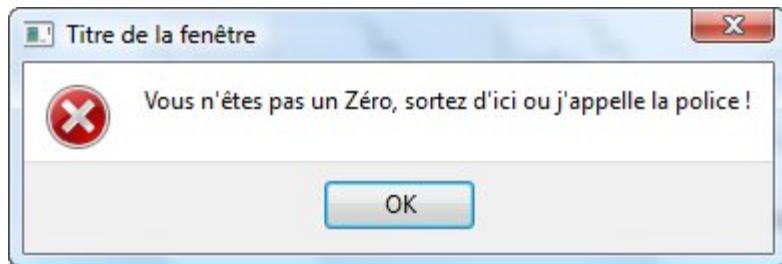
QMessageBox::critical

Quand c'est trop tard et qu'une erreur s'est produite, il ne vous reste plus qu'à utiliser la méthode statique critical() :

Code : C++ - [Sélectionner](#)

```
3 QMessageBox::critical(this, "Titre de la fenêtre", "Vous n'êtes pas un Zéro, sortez d'ici !")
```

4



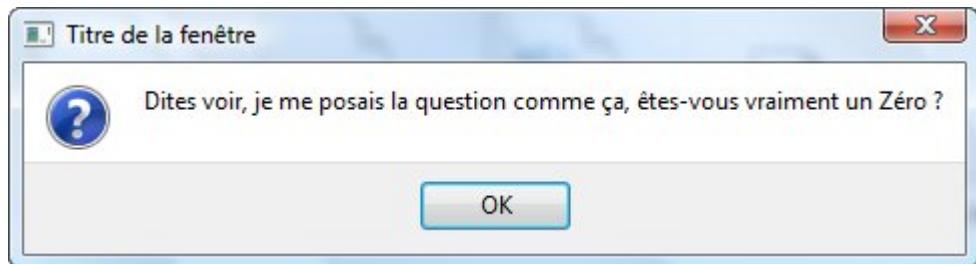
QMessageBox::question

Si vous avez une question à poser à l'utilisateur, c'est la boîte de dialogue qu'il vous faut !

Code : C++ - [Sélectionner](#)

```
3 QMessageBox::question(this, "Titre de la fenêtre", "Dites voir, je me posais la question comme ça, êtes-vous vraiment un Zéro ?")
```

4



C'est bien joli mais... comment peut-on répondre à la question avec un simple bouton OK ?

Par défaut, c'est toujours un bouton OK qui s'affiche. Mais dans certains cas, comme lorsqu'on pose une question, il faudra afficher d'autres boutons pour que la boîte de dialogue ait du sens.

Personnaliser les boutons de la boîte de dialogue

Pour personnaliser les boutons de la boîte de dialogue, il faut utiliser le 4ème paramètre de la méthode statique. Ce paramètre

accepte une combinaison de valeurs prédéfinies, séparées par un OR (la barre verticale |). On appelle cela des flags. Si vous avez déjà travaillé avec la SDL, vous connaissez cela. Sinon, vous vous y habituerez vite vous verrez, c'est juste une façon pratique d'envoyer des options à une fonction.

Pour ceux qui se poseraient la question, le 5ème et dernier paramètre de la fonction permet d'indiquer quel est le bouton par défaut. On change rarement cette valeur car Qt choisit généralement le bouton qui convient le mieux par défaut.

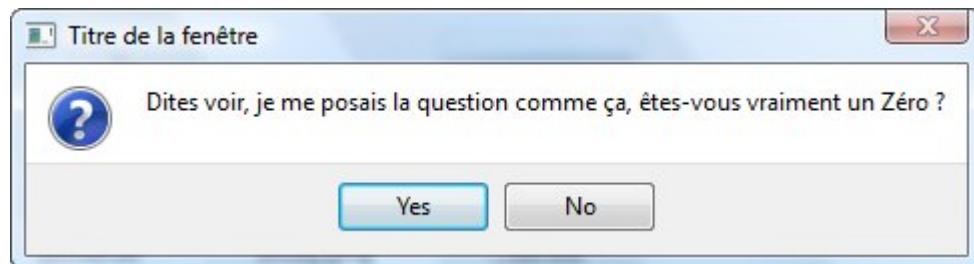
La [liste des flags disponibles](#) est donnée par la documentation. Vous avez du choix comme vous pouvez le voir.

Si on veut placer les boutons "Oui" et "Non", il nous suffit de combiner les valeurs "QMessageBox::Yes" et "QMessageBox::No"

Code : C++ - Sélectionner



Les boutons apparaissent alors :



Horrer ! Malédiction ! Enfer et damnation !

L'anglais me poursuit, les boutons sont écrits en anglais. Catastrophe qu'est-ce que je vais faire au secouuuuuurs !!!

En effet, les boutons sont écrits en anglais. Mais ce n'est pas grave du tout, les applications Qt peuvent être facilement traduites, je vous en avais parlé en introduction de cette partie.

On ne va pas rentrer dans les détails du fonctionnement de la traduction, on aura l'occasion d'en reparler plus longuement plus tard. Je vais vous donner un code à placer dans le fichier main.cpp, et vous allez l'utiliser gentiment sans poser de questions.

Attention, j'ai dit : sans poser de question. On n'aime pas trop les gens qui posent des questions ici. Un accident est si vite arrivé...



Code : C++ - Sélectionner

```
// main.cpp

#include < QApplication>
#include < QTranslator>
#include <QLocale>
#include <QLibraryInfo>
#include "MaFenetre.h"

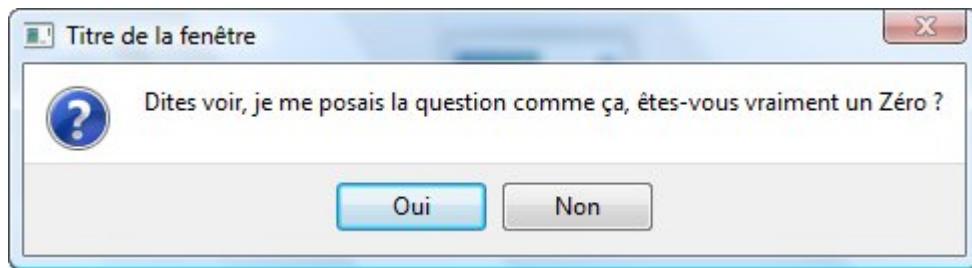
int main( int argc, char *argv[] )
{
    QApplication app(argc, argv);

    QString locale = QLocale::system().name().section('_', 0, 0);
    QTranslator translator;
    translator.load(QString("qt_") + locale, QLibraryInfo::location(QLibraryInfo::TranslationsPath));
    app.installTranslator(&translator);

    MaFenetre fenetre;
    fenetre.show();

    return app.exec();
```

Les lignes ajoutées ont été surlignées. Il y a plusieurs includes et quelques lignes de code supplémentaires dans le main. Normalement, votre application devrait maintenant afficher des boutons en français :



Et voilà le travail ! 😊

C'est cool, mais comment je fais pour savoir sur quel bouton l'utilisateur a cliqué ? Hein, hein ?

Quoi ? Encore une question ?

Vous savez, vous réduisez votre espérance de vie avec toutes les questions que vous posez aujourd'hui. Enfin moi j'dis ça comme ça 🍸

Bon ok, cette question est pertinente, je peux y répondre. Je dois y répondre même. Alors allons-y !

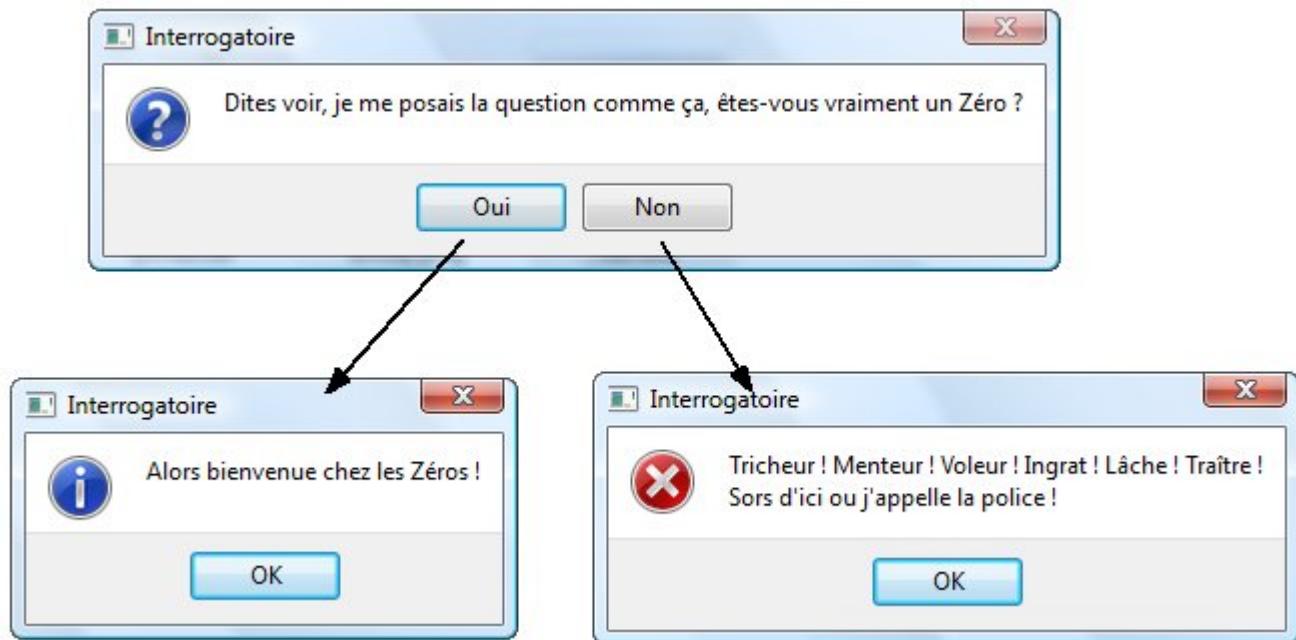
Récupérer la valeur de retour de la boîte de dialogue

Les méthodes statiques que nous venons de voir retournent un entier (int). On peut tester facilement la signification de ce nombre à l'aide des valeurs prédéfinies par Qt (comme quoi les énumérations c'est pratique !).

Code : C++ - Sélectionner

```
void MaFenetre::ouvrirDialogue()
{
    int reponse = QMessageBox::question(this, "Interrogatoire", "Dites voir, je me posais la question comme ça, êtes-vous vraiment un Zéro ?");
    if (reponse == QMessageBox::Yes)
    {
        QMessageBox::information(this, "Interrogatoire", "Alors bienvenue chez les Zéros !");
    }
    else if (reponse == QMessageBox::No)
    {
        QMessageBox::critical(this, "Interrogatoire", "Tricheur ! Menteur ! Voleur ! Ingrate !");
    }
}
```

Voici un schéma de ce qui peut se passer :



C'est ma foi clair, non ? 😊

Petite précision quand même : le type de retour exact de la méthode n'est pas int mais QMessageBox::StandardButton. Or, il s'agit là d'une énumération, et comme vous le savez probablement, une énumération n'est rien d'autre que le remplacement de nombres par des mots plus lisibles. Utiliser un int revient donc strictement au même.

Si un rappel sur les énumérations s'impose parce que je viens de vous parler en chinois, relisez donc le cours sur les énumérations issu du tutoriel du langage C.

Saisir une information

Les boîtes de dialogues précédentes étaient un peu limitées car, à part présenter différents boutons, on ne pouvait pas trop interagir avec l'utilisateur.

Si vous souhaitez que votre utilisateur saisisse une information, ou encore fasse un choix parmi une liste, les boîtes de dialogue de saisie sont idéales. Elles sont gérées par la classe QInputDialog, que je vous conseille d'inclure dès maintenant dans MaFenetre.h.

Les boîtes de dialogue "saisir une information" peuvent être de 4 types. Nous allons les voir dans l'ordre :

- I. Saisir un texte
- II. Saisir un entier
- III. Saisir un nombre décimal (double)
- IV. Choisir un élément parmi une liste

Chacune de ces fonctionnalités est assurée par une méthode statique différente.

Saisir un texte (QInputDialog::getText)

La méthode statique getText() ouvre une boîte de dialogue qui permet à l'utilisateur de saisir un texte.
Son prototype est :

Code : C++ - [Sélectionner](#)

```
3  QString QInputDialog::getText ( QWidget * parent, const QString & title, const QString &
```

4

Vous pouvez tout d'abord constater que la méthode retourne un `QString`, c'est-à-dire une chaîne de caractères de Qt. Les paramètres signifient, dans l'ordre :

- parent : pointeur vers la fenêtre parente. Peut être mis à NULL pour ne pas indiquer de fenêtre parente.
- title : titre de la fenêtre affiché en haut.
- label : texte affiché dans la fenêtre.
- mode : mode d'édition du texte. Permet de dire si on veut que les lettres s'affichent quand on tape, ou si elles doivent être remplacées par des astérisques (pour les mots de passe) ou si aucune lettre ne doit s'afficher. Toutes les options sont dans [la doc](#). Par défaut, les lettres s'affichent normalement (`QLineEdit::Normal`).
- text : le texte par défaut dans la zone de saisie.
- ok : un pointeur vers un booléen pour que Qt puisse vous dire si l'utilisateur a cliqué sur OK ou sur Annuler.
- f = quelques flags (options) permettant d'indiquer si la fenêtre est modale (bloquante) ou pas. Les valeurs possibles sont détaillées par [la doc](#).

Heureusement, comme vous pouvez le constater en lisant le prototype, certains paramètres possèdent des valeurs par défaut ce qui fait qu'ils ne sont pas obligatoires.

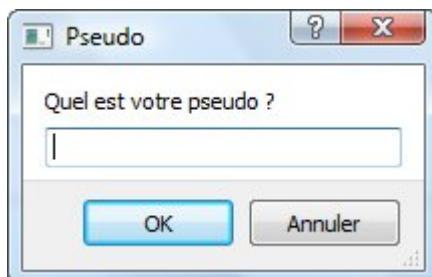
Reprenons notre code de tout à l'heure et cette fois, au lieu d'afficher une `QMessageBox`, nous allons afficher une `QInputDialog` lorsqu'on clique sur le bouton de la fenêtre.

Code : C++ - [Sélectionner](#)

```
void MaFenetre::ouvrirDialogue()
{
    QString pseudo = QInputDialog::getText(this, "Pseudo", "Quel est votre pseudo ?");
}
```

En une ligne, je crée un `QString` et je lui affecte directement la valeur renvoyée par la méthode `getText()`. J'aurais aussi bien pu faire la même chose en deux lignes, mais c'aurait été plus long et je suis une feignasse 😊

La boîte de dialogue devrait ressembler à cela :



On peut aller plus loin et vérifier si le bouton OK a été actionné, et si c'est le cas on peut alors afficher le pseudo de l'utilisateur dans une `QMessageBox`.

Code : C++ - [Sélectionner](#)

```
void MaFenetre::ouvrirDialogue()
{
    bool ok = false;
    QString pseudo = QInputDialog::getText(this, "Pseudo", "Quel est votre pseudo ?", QLineEdit::Normal, "Pseudo", &ok);

    if (ok && !pseudo.isEmpty())
    {
        QMessageBox::information(this, "Pseudo", "Bonjour " + pseudo + ", ça va ?");
    }
    else
    {
        QMessageBox::critical(this, "Pseudo", "Vous n'avez pas voulu donner votre nom...");
    }
}
```

Ici, on crée un booléen qui va recevoir l'information "Le bouton OK a-t-il été cliqué ?".

Pour pouvoir l'utiliser dans la méthode `getText`, il faut donner tous les paramètres avant qu'on ne souhaite pourtant pas changer ! C'est un des défauts des paramètres par défaut en C++ : si le paramètre que vous voulez renseigner est tout à la fin (à droite), il faudra alors absolument renseigner tous les paramètres qui sont avant ! J'ai donc envoyé des valeurs par défaut aux paramètres qui étaient avant, à savoir mode et text.

Comme j'ai donné un pointeur vers mon booléen à la méthode, celle-ci va le remplir pour indiquer si oui ou non le bouton a été cliqué.

Je peux ensuite faire un test, d'où la présence de mon if. Je vérifie 2 choses :

- Si le bouton OK a été cliqué
- Et si le texte n'est pas vide (la méthode `isEmpty` de `QString` sert à faire ça, vous ne pouvez pas la connaître, sauf en lisant la [doc de `QString`](#) bien sûr 😊).

Si un pseudo a été entré et que l'utilisateur a cliqué sur OK, alors une boîte de dialogue lui souhaite la bienvenue. Sinon, une erreur est affichée.

Ce schéma présente ce qui peut se produire :



Exercice : essayez d'afficher le pseudo de l'utilisateur quelque part sur la fenêtre mère, par exemple sur le bouton.

Saisir un entier (`QInputDialog::getInteger`)

La méthode `getInteger` devrait vous paraître simple maintenant que vous connaissez `getText`. Son prototype est :

Code : C++ - [Sélectionner](#)



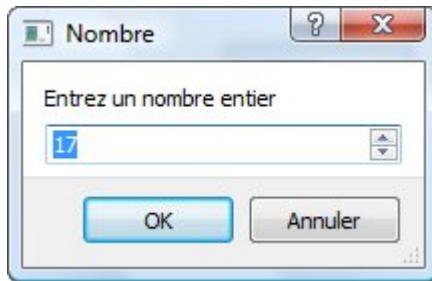
Elle retourne un int comme prévu.

Vous noterez les paramètres `value` (valeur par défaut), `minValue` (valeur minimale autorisée), `maxValue` (valeur maximale autorisée) et `step`, le pas d'incrémentation lorsqu'on clique sur les petites flèches (vous allez voir).

Testons ça avec les paramètres obligatoires, ça sera suffisant :

Code : C++ - Sélectionner

```
int entier = QInputDialog::getInteger(this, "Nombre", "Entrez un nombre entier");
```



Les petites flèches à droite permettent à l'utilisateur d'incrémenter (ou de décrémenter) le nombre affiché. Le rôle du paramètre step est d'indiquer la valeur du pas d'incrémentation. Par défaut il est de 1.

Par exemple si je clique sur la flèche vers le haut alors que le nombre saisi est 12 et que j'ai mis un pas d'incrémentation de 10, le nombre deviendra 22.

Le nombre saisi est retourné par la méthode dans un entier, à vous de le traiter pour faire ce que bon vous semblera avec 😊

Saisir un nombre décimal (QInputDialog::getDouble)

La saisie d'un double est pratiquement identique à celle d'un entier, à la différence près qu'il y a un paramètre qui permet d'indiquer le nombre maximal de chiffres après la virgule autorisés (paramètre decimals).

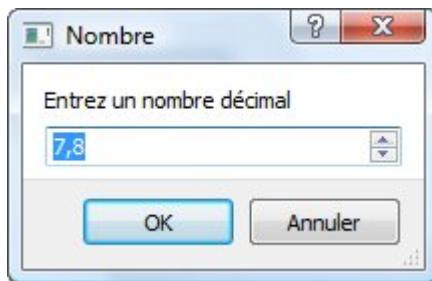
Code : C++ - Sélectionner

```
3 double QInputDialog::getDouble(QWidget * parent, const QString & title, const QString &3 4
```

Petit test :

Code : C++ - Sélectionner

```
3 double nombreDecimal = QInputDialog::getDouble(this, "Nombre", "Entrez un nombre décimal",4
```



Choix d'un élément parmi une liste (QInputDialog::getItem)

Si l'utilisateur doit faire son choix dans une liste, cette méthode permet d'afficher les choix possibles dans une boîte de dialogue avec un menu déroulant.

Son prototype est :

Code : C++ - Sélectionner

```
3 QString QInputDialog::getItem( QWidget * parent, const QString & title, const QString & 4
```

Il y a quelques nouveaux paramètres que je dois expliquer :

- list : la liste des choix possibles, envoyée via un objet de type QStringList (liste de chaînes) à construire au préalable.
- current : le numéro du choix qui doit être sélectionné par défaut.
- editable : un booléen qui indique si l'utilisateur a le droit d'entrer sa propre réponse (comme avec getText) ou s'il est obligé de faire un choix parmi la liste.

Toute la "difficulté", vous l'aurez compris, consiste à créer cette liste de choix. La doc nous dit qu'il faut envoyer un objet de type QStringList, allons donc voir la [doc de QStringList](#) !

Hmm...

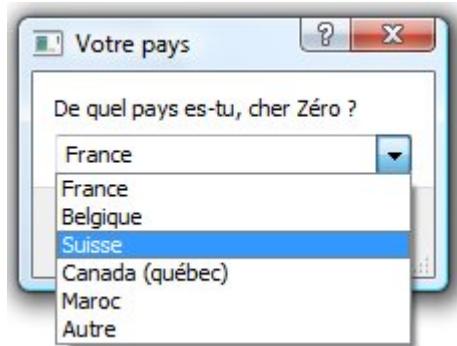
Hmm hmm...

Intéressant. Bon le constructeur ne permet pas d'envoyer un nombre infini de chaînes à la liste, par contre on peut voir dans la doc que l'opérateur << est surchargé. Cela va nous permettre de "remplir" notre liste de chaînes très facilement !

Code : C++ - Sélectionner

```
void MaFenetre::ouvrirDialogue()
{
    QStringList pays;
    pays << "France" << "Belgique" << "Suisse" << "Canada (québec)" << "Maroc" << "Autre";
    QInputDialog::getItem(this, "Votre pays", "De quel pays es-tu, cher Zéro ?", pays);
}
```

Pensez à inclure le header de la classe QStringList avant de vous en servir, sinon le compilateur vous dira que la classe QStringList est indéfinie !



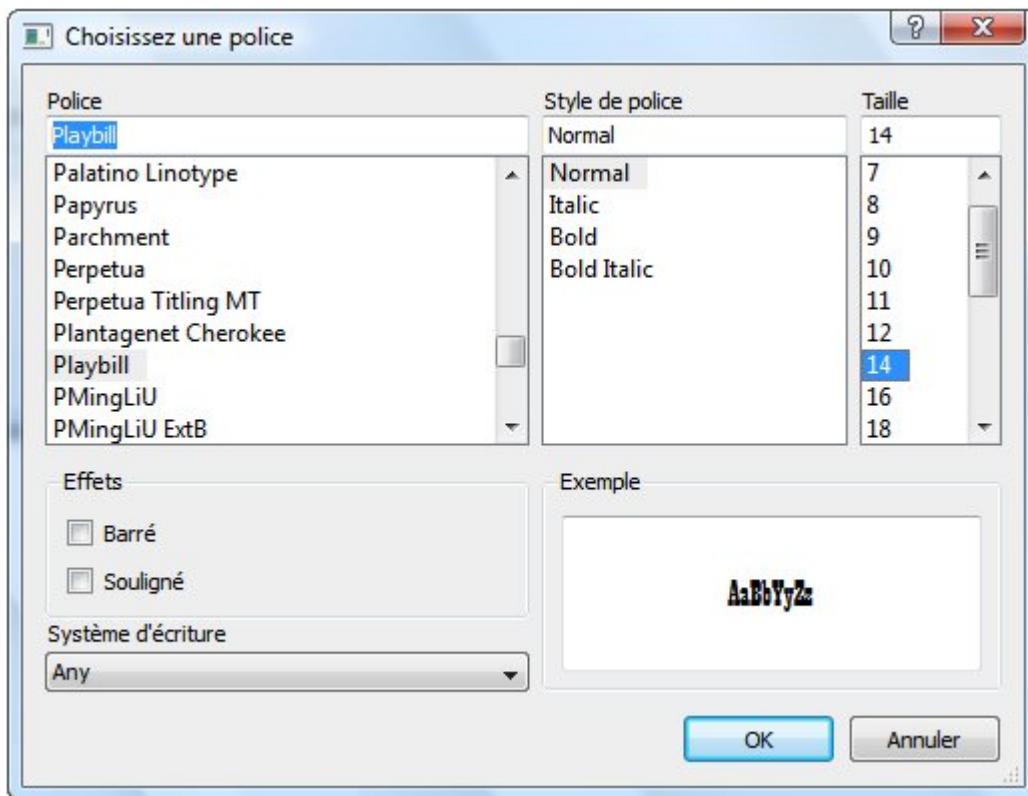
Et voilà, obstacle surmonté avec succès 😊

Si, pour une raison ou une autre, vous ne souhaitez pas utiliser l'opérateur surchargé <<, il existe des méthodes qui permettent d'ajouter des éléments un à un. Ces méthodes ne sont pas dans la classe [QStringList](#), mais dans sa classe mère [QList](#). On peut par exemple citer `append()` qui permet d'ajouter un élément à la fin de la liste.

Je dis ça pour vous rappeler de toujours regarder les méthodes de la classe mère si ce que vous cherchez n'est pas dans la liste des méthodes propres à votre classe.

Sélectionner une police

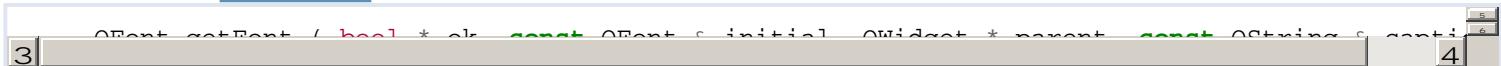
La boîte de dialogue "Sélectionner une police" est une des boîtes de dialogue standard les plus connues. Nul doute que vous l'avez déjà rencontrée dans l'un de vos programmes favoris.



La boîte de dialogue de sélection de police est gérée par la classe QFontDialog. Celle-ci propose en gros une seule méthode statique surchargée (il y a plusieurs façons de l'utiliser), comme vous pouvez le constater sur la [doc de QFontDialog](#).

Prenons le prototype le plus compliqué, juste pour la forme 😊

Code : C++ - Sélectionner



Les paramètres se comprennent normalement assez facilement.

On retrouve notre pointeur vers un booléen "ok" qui permet de savoir si l'utilisateur a cliqué sur OK ou a annulé.
On peut spécifier une police par défaut (initial), il faudra envoyer un objet de type QFont. Voilà justement que la classe QFont réapparaît 😊

Enfin, la chaîne caption correspond au message qui sera affiché en haut de la fenêtre.

Enfin, et surtout, la méthode retourne un objet de type QFont correspondant à la police qui a été choisie.

Testons ! Histoire d'aller un peu plus loin, je propose que la police que nous aurons sélectionnée soit immédiatement appliquée au texte de notre bouton, par l'intermédiaire de la méthode setFont() que nous avons appris à utiliser il y a quelques chapitres.

Code : C++ - Sélectionner

```
void MaFenetre::ouvrirDialogue()
{
    bool ok = false;

    QFont police = QFontDialog::getFont(&ok, m_boutonDialogue->font(), this, "Choisissez"

    if (ok)
    {
        m_boutonDialogue->setFont(police);
    }
}
```

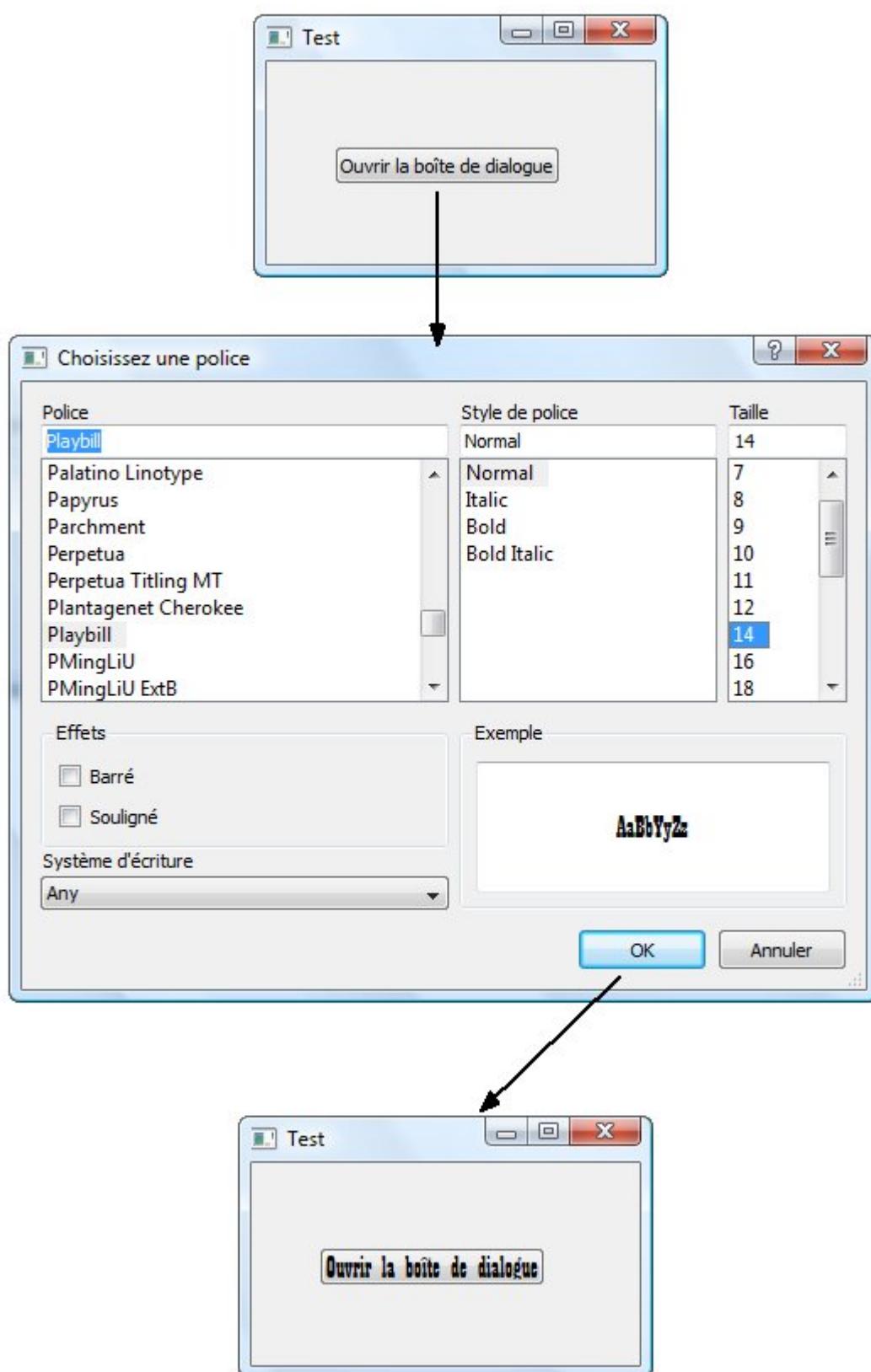


La méthode `getFont` prend comme police par défaut celle qui est utilisée par notre bouton `m_boutonDialogue` (rappelez-vous, `font()` est une méthode accesseur qui renvoie un `QFont`).

On teste si l'utilisateur a bien validé la fenêtre, et si c'est le cas on applique la police qui vient d'être choisie à notre bouton.

C'est l'avantage de travailler avec les classes de Qt : elles sont cohérentes. La méthode `getFont` renvoie un `QFont`, et ce `QFont` nous pouvons l'envoyer à notre tour à notre bouton pour qu'il change d'apparence.

Le résultat ? Le voici :

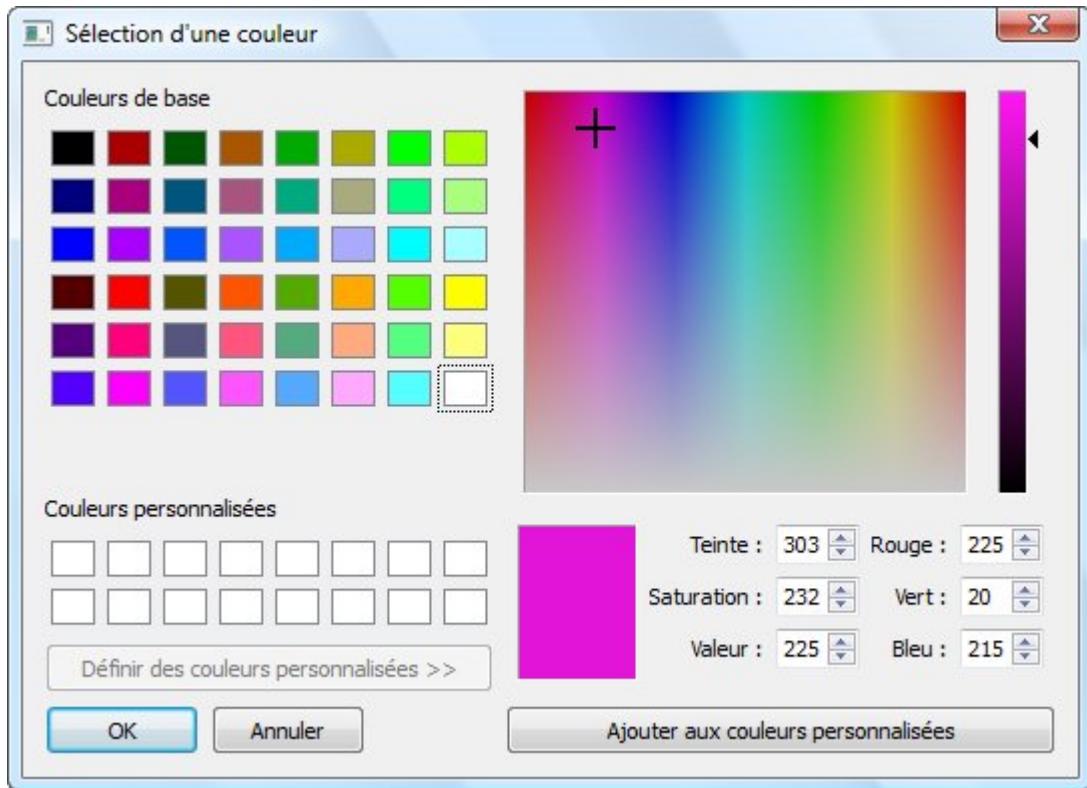


Attention le bouton ne se redimensionne pas tout seul. Vous pouvez le rendre plus large de base si vous voulez, ou bien le

redimensionner après le choix de la police.

Sélectionner une couleur

Dans la même veine que la sélection de police, on connaît probablement tous la boîte de dialogue "Sélection de couleur".



Utilisez la classe QColorDialog et sa méthode statique getColor().

Code : C++ - [Sélectionner](#)

```
3] QColor QColorDialog::getColor( const QColor &initial = Qt::white, QWidget *parent = 0, 4]
```

Elle retourne un objet de type QColor. Vous pouvez préciser une couleur par défaut, en envoyant un objet de type QColor ou en utilisant une des [constantes prédéfinies de couleur](#). En l'absence de paramètre, c'est la couleur blanche qui sera sélectionnée comme nous l'indique le prototype.

Si on veut tester le résultat en appliquant la nouvelle couleur au bouton, c'est un petit peu compliqué. En effet, il n'existe pas de méthode setColor pour les widgets, mais une méthode setPalette qui sert à indiquer une palette de couleurs. Je vous laisse vous renseigner plus amplement si vous le désirez sur la [classe QPalette](#) qui est intéressante.

Le code que je vous propose ci-dessous ouvre une boîte de dialogue de sélection de couleur, puis crée une palette dont la couleur du texte correspond à la couleur qu'on vient de sélectionner, et applique enfin cette palette au bouton :

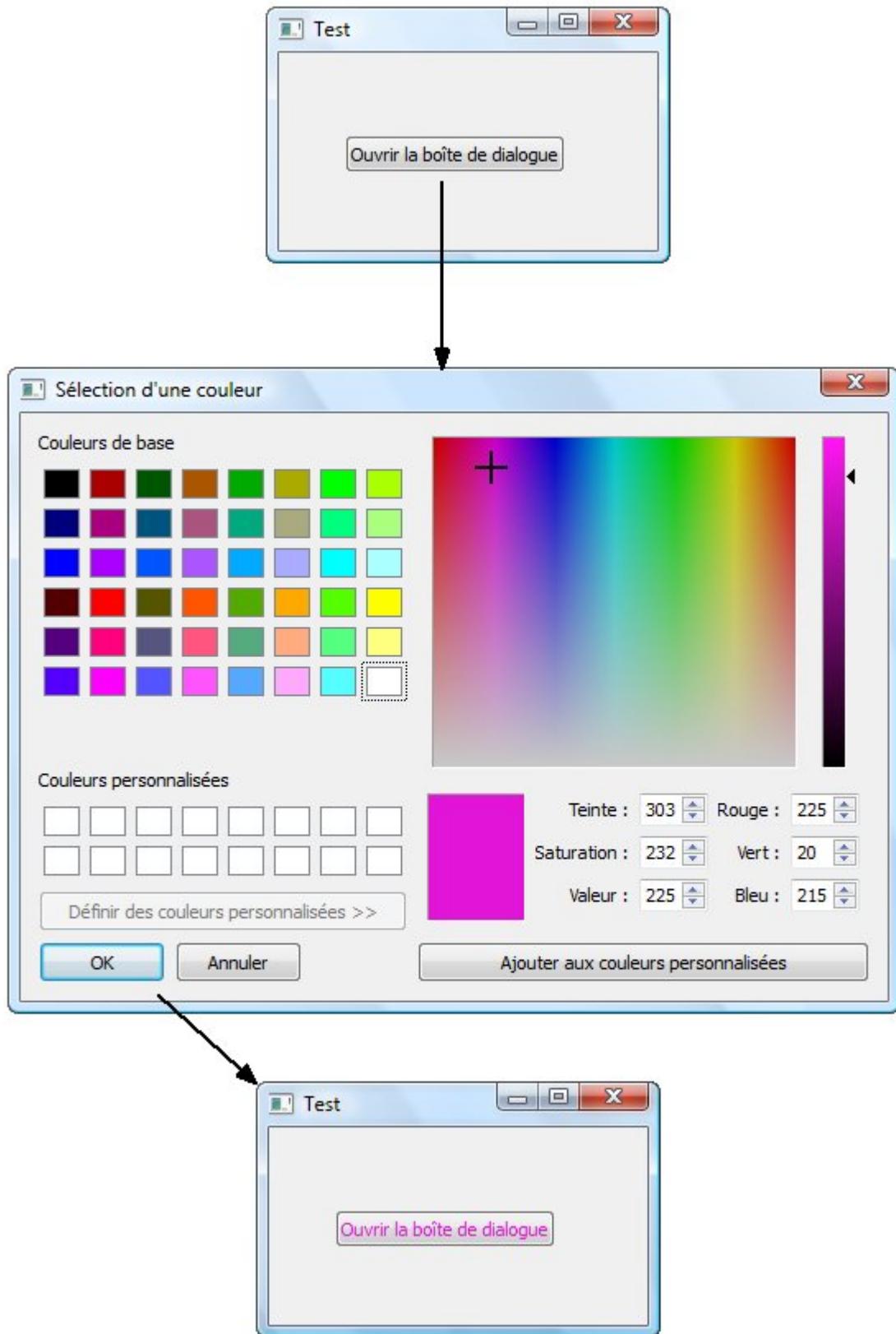
Code : C++ - [Sélectionner](#)

```
void MaFenetre::ouvrirDialogue()
{
    QColor couleur = QColorDialog::getColor(Qt::white, this);

    QPalette palette;
    palette.setColor(QPalette::ButtonText, couleur);
    m_boutonDialogue->setPalette(palette);
}
```

Je ne vous demande pas ici de comprendre comment fonctionne QPalette, qui est d'ailleurs une classe que je ne détaillerai pas plus dans le cours. A vous de vous renseigner sur elle si elle vous intéresse.

Le résultat de l'application est le suivant :



Sélection d'un fichier ou d'un dossier

Allez, plus que la sélection de fichiers et de dossiers et on aura fait le tour d'à peu près toutes les boîtes de dialogue usuelles qui existent ! 😊

La sélection de fichiers et de dossiers est gérée par la [classe QFileDialog](#) qui propose elle aussi des méthodes statiques faciles à utiliser.

Cette section sera divisée en 3 parties :

- Sélection d'un dossier existant
- Ouverture d'un fichier
- Enregistrement d'un fichier

Sélection d'un dossier existant (QFileDialog::getExistingDirectory)

Bon je ne vous donne plus le prototype, vous devriez être assez grands pour le retrouver dans la doc 😊

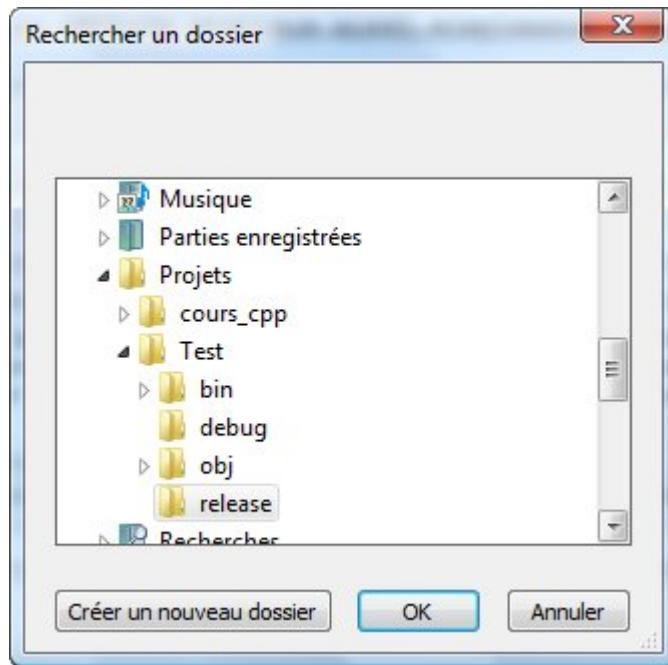
On peut utiliser la méthode statique aussi simplement que comme ceci :

Code : C++ - [Sélectionner](#)

```
QString dossier = QFileDialog::getExistingDirectory(this);
```

Elle retourne un QString contenant le chemin complet vers le dossier demandé.

La fenêtre qui s'ouvre devrait ressembler à cela :



Ouverture d'un fichier (QFileDialog::getOpenFileName)

La célèbre boîte de dialogue "Ouverture d'un fichier" est gérée par `getOpenFileName()`. Sans paramètres particuliers, la boîte de dialogue permet d'ouvrir n'importe quel fichier.

Vous pouvez néanmoins créer un filtre (4ème paramètre) pour afficher par exemple uniquement les images.

Ce code demande d'ouvrir un fichier image. Le chemin vers le fichier est stocké dans un QString, que l'on affiche ensuite via une QMessageBox :

Code : C++ - [Sélectionner](#)

```

void MaFenetre::ouvrirDialogue()
{
    QString fichier = QFileDialog::getOpenFileName(this, "Ouvrir un fichier", QString(),
    QMessageBox::information(this, "Fichier", "Vous avez sélectionné :\n" + fichier);
}

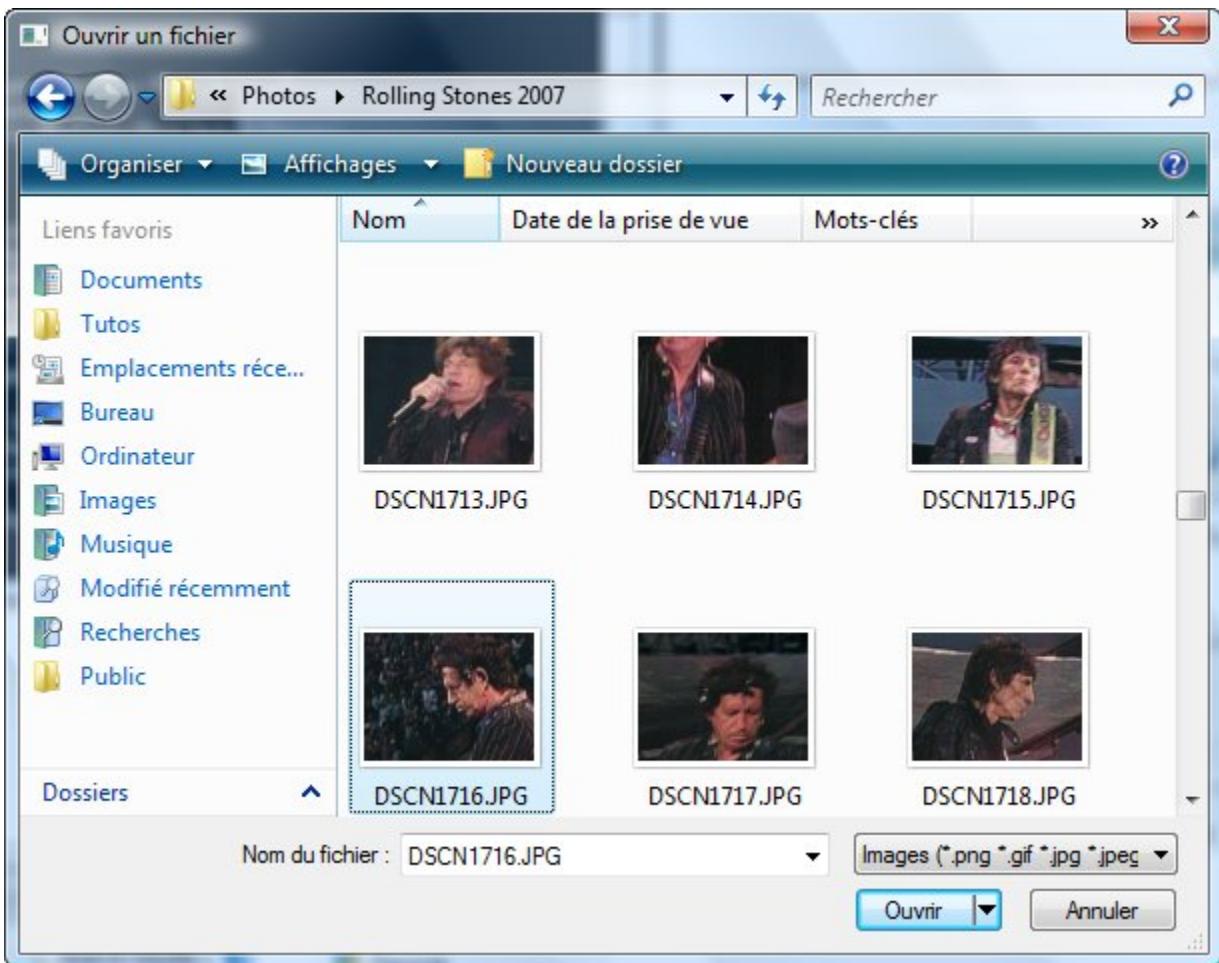
```

5
6
3
4

Le troisième paramètre de getOpenFileName est le nom du répertoire par défaut dans lequel l'utilisateur est placé. J'ai laissé la valeur par défaut (QString()), ce qui est équivalent à écrire "", donc la boîte de dialogue affichera par défaut le répertoire dans lequel est situé le programme.

Grâce au 4ème paramètre j'ai choisi de filtrer les fichiers. Seules les images de type PNG, GIF, JPG et JPEG s'afficheront.

Résultat :



La fenêtre bénéficie de toutes les options que propose votre OS, dont l'affichage des images sous forme de miniatures. Lorsque vous cliquez sur "Ouvrir", le chemin est enregistré dans un QString qui s'affiche ensuite dans une boîte de dialogue :



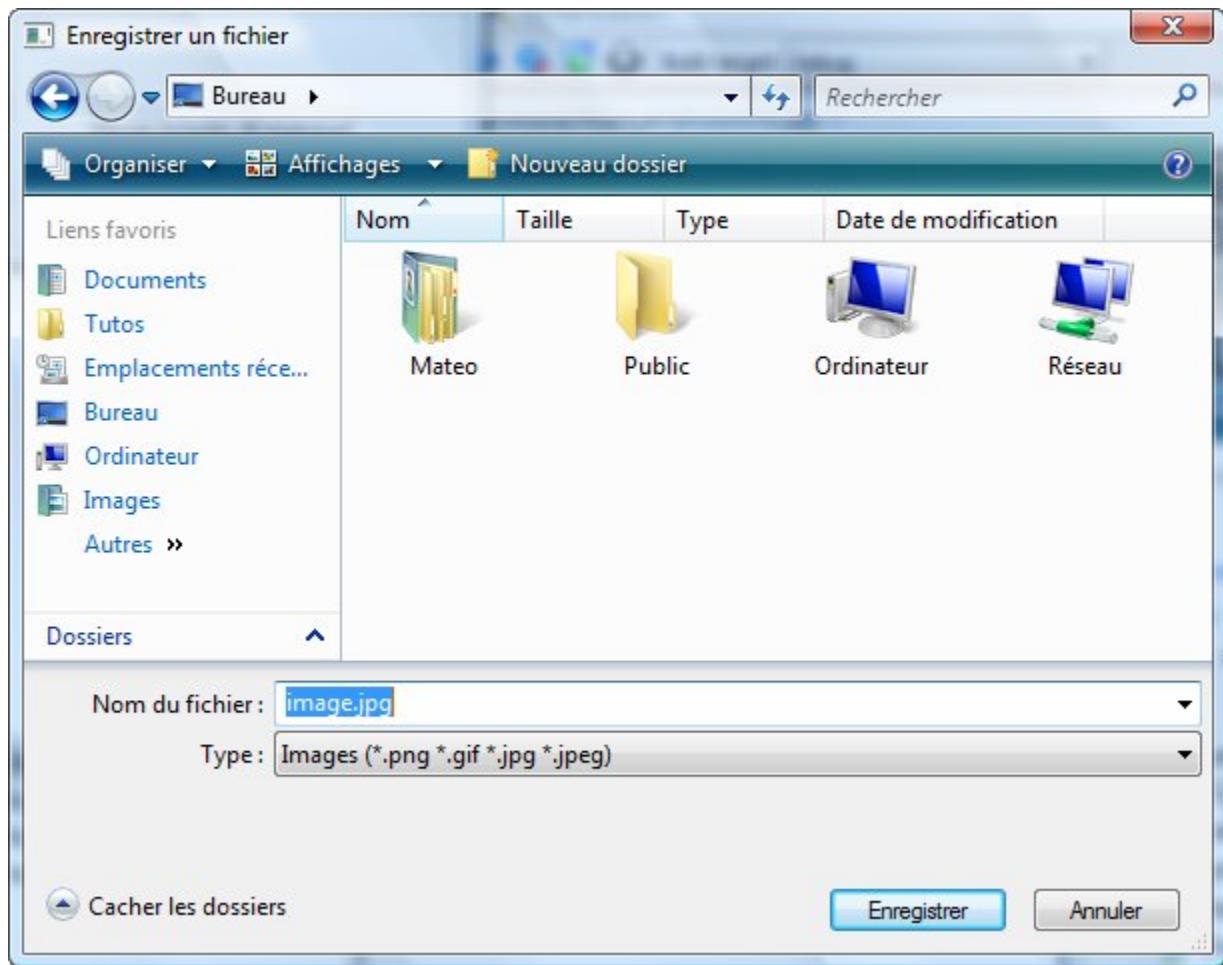
Le principe de cette boîte de dialogue est de vous donner le chemin complet vers le fichier, mais pas de vous ouvrir ce fichier. C'est à vous ensuite de faire les opérations nécessaires pour ouvrir le fichier et l'afficher dans votre programme.

A noter aussi la fonction `getOpenFileNames` (notez le "s" à la fin) qui autorise la sélection de plusieurs fichiers. La principale différence est qu'au lieu de retourner un `QString`, elle retourne un `QStringList` (liste de chaînes). Tiens, comme on se retrouve !

Enregistrement d'un fichier (`QFileDialog::getSaveFileName`)

C'est le même principe que la méthode précédente, à la différence près que la personne peut cette fois spécifier un nom de fichier qui n'existe pas pour l'enregistrement. Le bouton "Ouvrir" est remplacé par "Enregistrer".

Code : C++ - [Sélectionner](#)



Je vous avais promis un chapitre simple, vous avez eu un chapitre simple !

En effet, les méthodes statiques ne sont rien d'autre que des "fonctions" comme en langage C, elles ne nécessitent donc pas de créer d'objets. Comme quoi, parfois le modèle objet est inadapté et ici c'était clairement le cas. Pour la plupart des classes que nous avons vues, on peut s'en sortir sans créer le moindre objet.

Ces considérations mises à part, le modèle objet reste quoiqu'il en soit très pratique lorsqu'on crée des GUI comme on le fait là. Et je peux vous dire qu'on n'a pas fini de tout découvrir 😊

A titre informatif, il existe quelques autres boîtes de dialogue usuelles un peu plus rares et surtout un peu plus complexes à utiliser. Je pense notamment à :

- `QProgressDialog` : affiche une boîte de dialogue avec une barre de progression et un bouton "Annuler". Cela permet de faire patienter l'utilisateur le temps qu'une longue opération s'exécute. Cette classe est très intéressante mais il vaut mieux qu'on la voie en pratique si on a l'occasion, car Qt cherche à estimer le temps restant pour savoir s'il doit afficher ou non la fenêtre. C'est plus intéressant de le voir dans un cas très concret donc.
- `QWizard` : affiche un assistant, avec les boutons "Suivant", "Précédent", "Terminer"... Là encore il vaut mieux avoir un projet

concret pour apprendre à utiliser cette classe car elle est assez complexe.

Ceci étant, vous pouvez aussi lire la documentation si vous en avez besoin maintenant, il y a tout ce qu'il faut dessus.

Mais... mais... je sais pas lire une doc moi, je sais pas où chercher l'information dont j'ai besoin, je suis perdu j'y comprends rien 😕

Ah ouais ? C'est ce qu'on va voir !

On vous a pas encore fait de tuto pour vous apprendre à lire une doc à ce que je sache ? Alors c'est le moment d'apprendre !

Apprendre à lire la documentation de Qt

Voilà le chapitre le plus important de toute la partie sur Qt : celui qui va vous apprendre à lire la documentation de Qt.

Pourquoi est-ce que c'est si important de savoir lire la documentation ?

Parce que la documentation, c'est la bible du programmeur. Elle explique toutes les possibilités d'un langage ou d'une bibliothèque. La documentation de Qt contient la liste des fonctionnalités de Qt. Toute la liste.

La documentation, c'est donc ce qu'il y a de plus complet mais... ça n'a rien à voir avec un tutoriel du Site du Zéro.

Déjà, il faudra vous y faire : la doc n'est disponible qu'en anglais (c'est valable pour Qt et pour la quasi-totalité des autres docs). Il faudra donc faire l'effort de lire de l'anglais, même si vous y êtes allergiques. En programmation, on peut rarement s'en sortir si on ne lit pas un minimum d'anglais technique.

D'autre part, la documentation est construite de manière assez déroutante quand on débute. Il faut être capable de "lire" et naviguer dans une documentation.

C'est précisément ce que ce chapitre va vous apprendre à faire 😊

Où trouver la doc ?

On vous dit que Qt propose une superbe documentation très complète qui vous explique tout son fonctionnement.

Oui, mais où peut-on trouver cette documentation au juste ? 😊

Il y a en fait 2 moyens d'accéder à la doc :

- si vous avez internet : vous pouvez aller sur le site de Trolltech (l'entreprise qui édite Qt) ;
- si vous n'avez pas internet : vous pouvez utiliser le programme Qt Assistant qui contient toute la doc.

Avec internet : sur le site de Trolltech

Personnellement, si j'ai accès à internet, j'ai tendance à préférer utiliser cette méthode pour lire la documentation. Il suffit d'aller sur le site web de Trolltech, section documentation. L'adresse est simple à retenir :

<http://doc.trolltech.com>

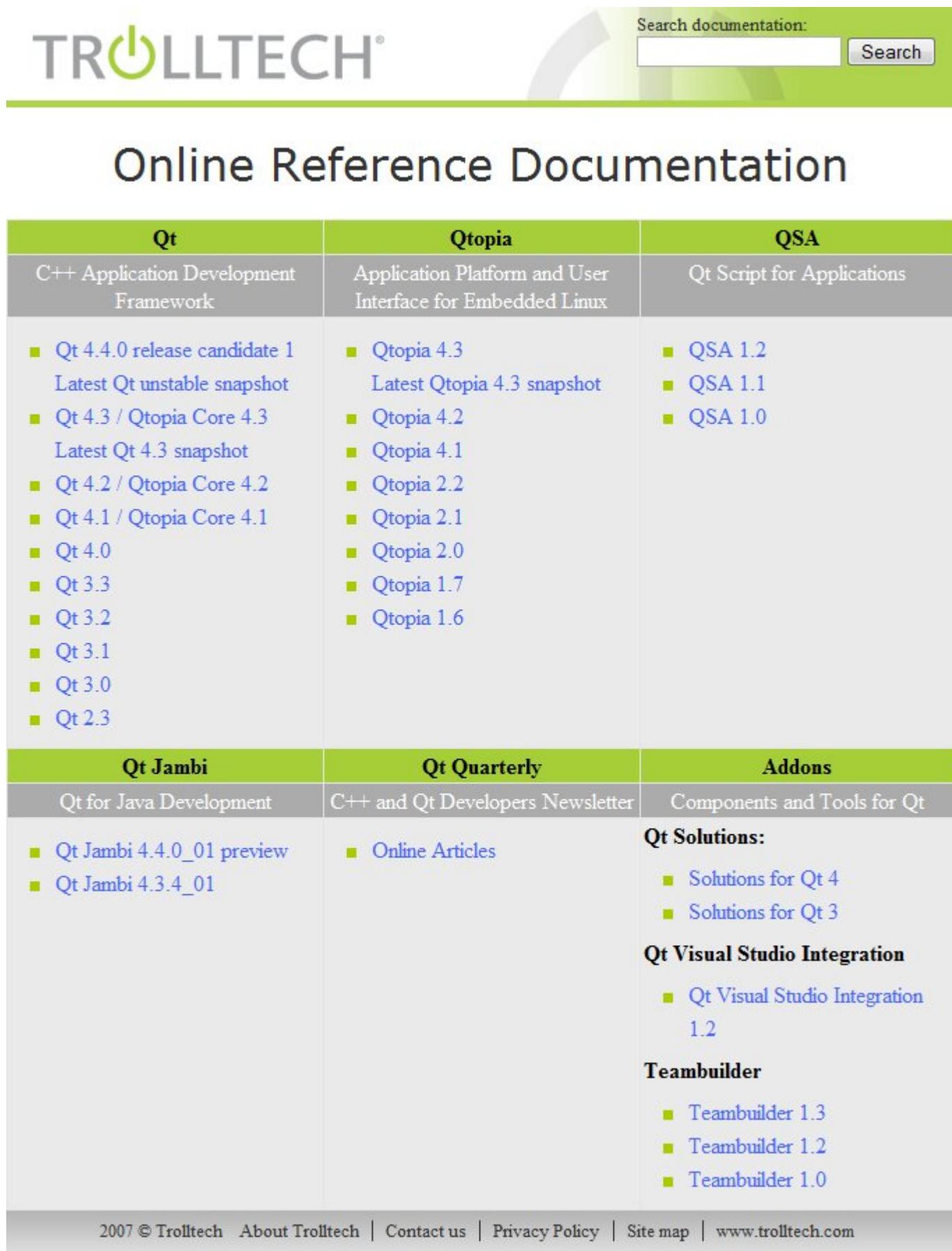
Je vous conseille très fortement d'ajouter ce site dans vos favoris, et de faire en sorte qu'il soit visible !

Si vous ne faites pas un raccourci visible vers la doc, vous serez moins tentés d'y aller... or le but c'est justement que vous preniez le réflexe d'y aller 😊

Un des principaux avantages à aller chercher la doc sur internet, c'est que l'on est assuré d'avoir la doc la plus à jour. En effet, s'il y

a des nouveautés ou des erreurs, on est certain en allant sur le net d'en avoir la dernière version.

Lorsque vous arrivez sur la doc, la page suivante s'affiche :



The screenshot shows the Trolltech Online Reference Documentation homepage. At the top left is the Trolltech logo. To its right is a search bar with the placeholder "Search documentation:" and a "Search" button. Below the header is a large section titled "Online Reference Documentation". Underneath this are two tables:

Qt	Qtopia	QSA
C++ Application Development Framework	Application Platform and User Interface for Embedded Linux	Qt Script for Applications
<ul style="list-style-type: none">■ Qt 4.4.0 release candidate 1■ Latest Qt unstable snapshot■ Qt 4.3 / Qtopia Core 4.3■ Latest Qt 4.3 snapshot■ Qt 4.2 / Qtopia Core 4.2■ Qt 4.1 / Qtopia Core 4.1■ Qt 4.0■ Qt 3.3■ Qt 3.2■ Qt 3.1■ Qt 3.0■ Qt 2.3	<ul style="list-style-type: none">■ Qtopia 4.3■ Latest Qtopia 4.3 snapshot■ Qtopia 4.2■ Qtopia 4.1■ Qtopia 2.2■ Qtopia 2.1■ Qtopia 2.0■ Qtopia 1.7■ Qtopia 1.6	<ul style="list-style-type: none">■ QSA 1.2■ QSA 1.1■ QSA 1.0

Qt Jambi	Qt Quarterly	Addons
Qt for Java Development	C++ and Qt Developers Newsletter	Components and Tools for Qt
<ul style="list-style-type: none">■ Qt Jambi 4.4.0_01 preview■ Qt Jambi 4.3.4_01	<ul style="list-style-type: none">■ Online Articles	<p>Qt Solutions:</p> <ul style="list-style-type: none">■ Solutions for Qt 4■ Solutions for Qt 3 <p>Qt Visual Studio Integration</p> <ul style="list-style-type: none">■ Qt Visual Studio Integration 1.2 <p>Teambuilder</p> <ul style="list-style-type: none">■ Teambuilder 1.3■ Teambuilder 1.2■ Teambuilder 1.0

At the bottom of the page is a footer bar containing links: "2007 © Trolltech | About Trolltech | Contact us | Privacy Policy | Site map | www.trolltech.com".

C'est la liste des produits de Trolltech. Dans le lot on trouve Qt bien évidemment, mais aussi Qtopia, une version "light" de Qt pour

les appareils mobiles, Qt Jambi, une version de Qt pour le langage Java, etc.

Nous nous intéressons au premier cadre en haut à gauche intitulé Qt.

Vous pouvez voir la liste des différentes versions de Qt, depuis Qt 2.3.

Sélectionnez la version de Qt qui correspond à celle que vous avez installée. Vous pouvez retrouver le numéro de votre version dans le raccourci du menu Démarrer par exemple.

Dans mon cas, j'ai la version 4.3.2, je vais donc ouvrir Qt 4.3 (la version 4.3.2 n'est en général qu'une correction de bugs de la 4.3, c'est un sous-numéro de version).

Voici la page qui devrait s'afficher maintenant :



Qt Reference Documentation (Open Source Edition)

Note: This edition is for the development of **Free and Open Source** software only; see [Qt Commercial Editions](#).

Getting Started	General	Developer Resources
<ul style="list-style-type: none">• What's New in Qt 4.3• How to Learn Qt• Installation• Tutorial and Examples• Porting from Qt 3 to Qt 4	<ul style="list-style-type: none">• About Qt• About Trolltech• Commercial Edition• Open Source Edition• Frequently Asked Questions	<ul style="list-style-type: none">• Mailing Lists• Qt Community Web Sites• Qt Quarterly• How to Report a Bug• Other Online Resources
API Reference	Core Features	Key Technologies
<ul style="list-style-type: none">• All Classes• Main Classes• Grouped Classes• Annotated Classes• Qt Classes by Module• Inheritance Hierarchy• All Functions• Qtopia Core• All Overviews and HOWTOs• Qt Widget Gallery• Class Chart	<ul style="list-style-type: none">• Signals and Slots• Object Model• Layout Management• Paint System• Graphics View• Accessibility• Tool and Container Classes• Internationalization• Plugin System• Inter-process Communication• Unit Testing Framework	<ul style="list-style-type: none">• Multithreaded Programming• Main Window Architecture• Rich Text Processing• Model/View Programming• Style Sheets• Network Module• OpenGL Module• SQL Module• SVG Module• XML Module• Script Module• ActiveQt Framework
Add-ons & Services	Tools	Licenses & Credits
<ul style="list-style-type: none">• Qt Solutions• Partner Add-ons• Third-Party Qt Components (qt-apps.org)• Support• Training	<ul style="list-style-type: none">• Qt Designer• Qt Assistant• Qt Linguist• qmake• All Tools	<ul style="list-style-type: none">• GNU General Public License• Third-Party Licenses Used in Qt• Other Licenses Used in Qt• Trademark Information• Credits

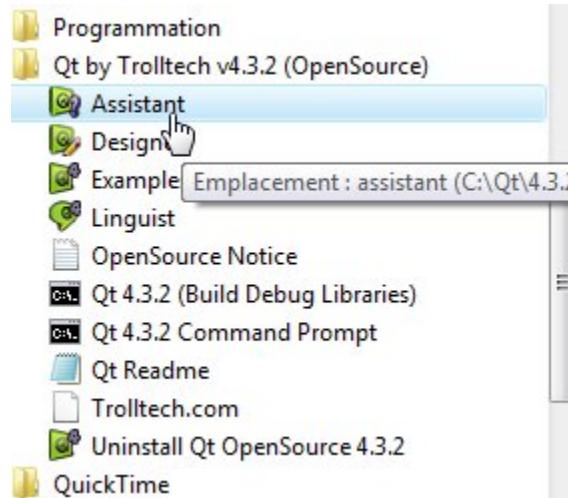
sur votre PC, il est inutile d'aller lire les docs des autres versions.

Nous allons détailler les différentes sections de cette page.
Mais avant... voyons voir comment accéder à la doc quand on n'a pas internet !

Sans internet : avec Qt Assistant

Si vous n'avez pas internet, pas de panique !

Qt a installé toute la documentation sur votre disque dur. Vous pouvez y accéder grâce au programme "Assistant" que vous retrouverez par exemple dans le menu Démarrer :



Qt Assistant se présente sous la forme d'un mini-navigateur qui contient la documentation de Qt :

The screenshot shows the Qt Assistant application window. The title bar reads "Qt Assistant by Trolltech - Qt Reference Documentation (Open Source Edition)". The menu bar includes File, Edit, View, Go, Bookmarks, and Help. The toolbar contains icons for back, forward, search, and other functions. A sidebar on the left lists "Qt Assistant Manual", "Qt Designer Manual", "Qt Linguist Manual", "Qt Reference Document", and "qmake Manual". The main content area displays the "Qt Reference Documentation (Open Source Edition)" page. The header features the Trolltech logo. Below the header, there's a navigation bar with links to Home, All Classes, Main Classes, Grouped Classes, Modules, and Functions. The main title is "Qt Reference Documentation (Open Source Edition)". A note below the title states: "Note: This edition is for the development of Free and Open Source software only; see [Qt Commercial Editions](#)". The page is organized into several sections:

Getting Started	General	Developer Resources
<ul style="list-style-type: none">• What's New in Qt 4.3• How to Learn Qt• Installation• Tutorial and Examples• Porting from Qt 3 to Qt 4	<ul style="list-style-type: none">• About Qt• About Trolltech• Commercial Edition• Open Source Edition• Frequently Asked Questions	<ul style="list-style-type: none">• Mailing Lists• Qt Community Web Sites• Qt Quarterly• How to Report a Bug• Other Online Resources

API Reference	Core Features	Key Technologies
<ul style="list-style-type: none">• All Classes• Main Classes• Grouped Classes• Annotated Classes• Qt Classes by Module• Inheritance Hierarchy	<ul style="list-style-type: none">• Signals and Slots• Object Model• Layout Management• Paint System• Accessibility• Tool and Container	<ul style="list-style-type: none">• Multithreaded Programming• Main Window Architecture• Rich Text Processing• Model/View Programming• Style Sheets

Vous ne disposez que de la documentation de Qt correspondant à la version que vous avez installée (c'est logique dans un sens). Si vous voulez lire la documentation d'anciennes versions de Qt (ou de futures versions en cours de développement) il faut obligatoirement aller sur internet.

Le logiciel Qt Assistant vous permet d'ouvrir plusieurs onglets différents en cliquant sur le bouton "+". Vous pouvez aussi effectuer une recherche grâce au menu à gauche de l'écran et rajouter des pages en favoris.

Les différentes sections de la doc

Lorsque vous arrivez à l'accueil de la doc, la page suivante s'affiche comme nous l'avons vu :



Qt Reference Documentation (Open Source Edition)

Note: This edition is for the development of [Free and Open Source](#) software only; see [Qt Commercial Editions](#).

Getting Started	General	Developer Resources
<ul style="list-style-type: none">• What's New in Qt 4.3• How to Learn Qt• Installation• Tutorial and Examples• Porting from Qt 3 to Qt 4	<ul style="list-style-type: none">• About Qt• About Trolltech• Commercial Edition• Open Source Edition• Frequently Asked Questions	<ul style="list-style-type: none">• Mailing Lists• Qt Community Web Sites• Qt Quarterly• How to Report a Bug• Other Online Resources
API Reference	Core Features	Key Technologies
<ul style="list-style-type: none">• All Classes• Main Classes• Grouped Classes• Annotated Classes• Qt Classes by Module• Inheritance Hierarchy• All Functions• Qtopia Core• All Overviews and HOWTOs• Qt Widget Gallery• Class Chart	<ul style="list-style-type: none">• Signals and Slots• Object Model• Layout Management• Paint System• Graphics View• Accessibility• Tool and Container Classes• Internationalization• Plugin System• Inter-process Communication• Unit Testing Framework	<ul style="list-style-type: none">• Multithreaded Programming• Main Window Architecture• Rich Text Processing• Model/View Programming• Style Sheets• Network Module• OpenGL Module• SQL Module• SVG Module• XML Module• Script Module• ActiveQt Framework
Add-ons & Services	Tools	Licenses & Credits
<ul style="list-style-type: none">• Qt Solutions• Partner Add-ons• Third-Party Qt Components (qt-apps.org)• Support• Training	<ul style="list-style-type: none">• Qt Designer• Qt Assistant• Qt Linguist• qmake• All Tools	<ul style="list-style-type: none">• GNU General Public License• Third-Party Licenses Used in Qt• Other Licenses Used in Qt• Trademark Information• Credits

Getting Started

Getting Started contient des informations vous permettant de débuter avec Qt. C'est là-dedans que tout débutant devrait commencer par jeter un oeil. On trouve dans cette section :

- What's new in Qt 4.x : qu'est-ce qu'il y a de nouveau dans votre version de Qt par rapport aux précédentes ? Cette page intéressera surtout ceux qui connaissent Qt depuis quelques temps et qui sont curieux de voir ce qui a été ajouté depuis les versions précédentes. Je vous conseille d'y jeter un oeil, c'est plutôt bien illustré et clair et ça vous donne une idée des fonctionnalités sur lesquelles Trolltech travaille.
- How to Learn Qt : comment apprendre Qt ? Cette page vous donnera une série de conseils pour apprendre Qt dans les meilleures conditions. On vous conseillera notamment de suivre un tutoriel (ils ne parlent pas des tutoriels du SdZ hélas ).
- Installation : comment installer Qt. A ce stade du cours vous devriez déjà l'avoir fait, donc ça ne devrait pas vous avoir posé de problème.
- Tutorial and Examples : le tutoriel officiel de Qt et une série de programmes d'exemple avec leur code source. C'est une section TRES intéressante, je vous recommande d'y jeter un oeil. J'ai moi-même débuté avec Qt grâce à leur tutoriel qui est bien fait. Ce tutoriel permet en revanche seulement de démarrer et n'ira pas autant dans le détail que le tutoriel du Site du Zéro. Lorsque vous serez un peu plus expérimentés, je vous recommande de regarder les programmes d'exemple aussi, à partir d'un moment on apprend beaucoup mieux en lisant le code source des programmes .
- Porting from Qt 3 and Qt 4 : quels sont les changements majeurs entre Qt 3 et Qt 4 ? Il s'agit de 2 "grandes" versions de Qt très différentes. Cette section n'est destinée qu'aux anciens développeurs qui utilisaient Qt 3 et qui veulent passer à Qt 4. A priori, elle ne vous concerne donc absolument pas. Allez zou, on passe !

General

Ici, on trouvera des informations très générales relatives à Qt et Trolltech. Ce sont des pages "à propos" qui ne devraient pas vraiment vous intéresser.

Notez qu'on vous informe de la différence entre la version commerciale de Qt et la version open-source. Pour résumer simplement : seule la version open-source est gratuite, mais elle implique que vous publiez le code source de votre programme si vous le distribuez au public.

Jetez un oeil aussi à la FAQ (Frequently Asked Questions, les questions fréquemment posées), vous seriez surpris du nombre de petites choses que l'on peut apprendre sur ce genre de pages.

Developer Resources

Cette section un peu technique indique aux développeurs comment participer à Qt (en rapportant des bugs) et propose des articles et des forums pour ceux qui veulent aller plus loin. C'est encore un peu tôt pour vous, nous n'irons pas dans le détail de cette section.

API Reference

Contient la liste des classes de Qt. C'est probablement LA section la plus importante. C'est en passant par là que vous pourrez savoir tout ce que vous voulez sur une classe précise.

C'est en général à ça que sert la doc. Vous connaissez la classe que vous voulez utiliser, mais vous ne savez pas vous en servir complètement. Vous lisez donc son mode d'emploi.

C'est bien beau tout ça, mais si je ne connais pas le nom de la classe que je veux utiliser ? Si je sais par exemple que je veux créer un menu dans ma fenêtre, comment je fais pour retrouver le nom de la classe qui correspond ?

C'est la question que l'on se pose le plus souvent quand on débute. Ce n'est pas toujours facile de retrouver la classe que l'on cherche dans une doc.

Dans ce cas, soit vous faites une "recherche" dans la doc comme on va le voir, soit vous découvrez le nom de la classe dans un tutoriel (mon tutoriel est fait pour ça, il vous montre les classes et vous apprend à les utiliser un peu, mais si vous voulez aller dans

le détail il faudra lire la doc).

La section API Reference propose les liens suivants :

- [All Classes](#) : affiche TOUTES les classes de Qt, triées par ordre alphabétique. Notez que comme toutes les classes de Qt commencent par la lettre Q, on considère que c'est la seconde lettre qui détermine l'ordre alphabétique. Ainsi, QWidget se trouvera dans la section de la lettre "W".
- [Main Classes](#) : c'est une version épurée de "All Classes" qui ne contient que les classes les plus fréquemment utilisées. Je vous conseille de commencer par cette section, amplement suffisante dans un premier temps. Inutile de vous assommer directement avec la liste de toutes les classes 😊
- [Grouped Classes](#) : ici, les classes sont groupées par thèmes. C'est une section TRES intéressante pour vous qui débutez. Justement, si vous recherchez une classe dont vous ne connaissez pas le nom, c'est là qu'il faut aller.
- [Annotated Classes](#) : contient toutes les classes (comme All Classes) mais avec une courte description devant chacune d'elles. C'est pratique pour retrouver une classe, mais pas autant que la section "Grouped Classes" dont je viens de vous parler 😞
- [Qt Classes by Module](#) : les classes sont triées suivant les grands modules de Qt. Comme je vous l'avais dit, nous nous intéresserons surtout au plus gros module dédié à la création d'interfaces graphiques, à savoir QtGui.
- [Inheritance Hierarchy](#) : hiérarchie des classes sous forme de liste à puces. On peut voir qui hérite de qui, qui est le "parent" de qui. Ce n'est pas très exploitable à mon avis, je vous conseille de voir le "Class Chart" (plus bas).
- [All Functions](#) : liste toutes les fonctions (méthodes) utilisées par toutes les classes de Qt. A utiliser si vous vous souvenez du nom d'une fonction mais pas du nom de la classe.
- [Qtokia Core](#) : documentation de la version "light" de Qt destinée aux appareils mobiles (téléphones portables par exemple). Ca ne nous intéresse pas ici.
- [All Overviews and HOWTOs](#) : liste de tous les articles de la doc de Qt. Il y a beaucoup de choses dedans et c'est un peu fouilli et inexploitable si vous voulez mon avis, vous pouvez accéder à ces articles par d'autres moyens plus "logiques" depuis l'accueil de la doc.
- [Qt Widget Gallery](#) : une galerie des principaux widgets de Qt. Vous pouvez comparer l'apparence des widgets en fonction des systèmes d'exploitation. Si vous cliquez par exemple sur "Windows Vista Style Widget Gallery", vous aurez un très bon aperçu de la plupart des widgets que Qt propose (avec des captures d'écran faites sous Vista). Je vous rappelle que tous ces widgets fonctionnent sur tous les autres OS, c'est juste l'apparence qui change à chaque fois.
- [Class Chart](#) : un diagramme de toutes les classes de Qt qui permet de voir la relation d'héritage entre chacune des classes. Ce diagramme (au format PDF) fait peur au premier abord, mais il est en fait vraiment intéressant. Vous voyez qui hérite de qui. Vous constatez par exemple comme je vous l'ai dit que la plupart des classes héritent directement ou indirectement de QObject.
Par exemple, on voit que QPushButton hérite des propriétés de QAbstractButton, qui hérite de QWidget, qui hérite de QObject ! Ah les joies du C++ 😊
Je vous aurais bien conseillé de l'imprimer et de l'afficher dans votre chambre, mais c'est un peu trop gros pour être imprimé sur une feuille A4. Quoi qu'il en soit, ça peut être intéressant pour se repérer parmi les classes de Qt.

Voici le diagramme des classes au format PNG (au lieu de PDF) pour ceux qui aimeraient y jeter un coup d'oeil :



Ne vous laissez pas impressionner hein, avec un peu de méthode on s'y retrouve tout à fait 😊
 Notez que les classes sont colorées d'un fond de couleur différent en fonction du module auquel elles appartiennent. Les classes de QtGui sont colorées en vert clair sur ce schéma. On voit d'un seul coup d'œil que ce sont les plus nombreuses.

Core Features

Vous trouverez ici des articles sur les plus importantes fonctionnalités de Qt. C'est un peu comme un tutoriel à thèmes.

Par exemple, il y a une section qui explique le fonctionnement des signaux et des slots avec Qt. Je vous ai déjà fait un tuto à ce sujet, mais sachez pour information que quand moi j'ai débuté avec Qt, j'ai lu cet article qui m'a permis de comprendre ce que c'était et comment ça fonctionnait.

A vous de voir si un de ces thèmes vous intéresse plus particulièrement. Par exemple si vous voulez en savoir plus sur la gestion du dessin avec Qt, allez dans "Paint System". Si vous êtes intéressés par l'accessibilité, regardez du côté de "Accessibility". Si vous voulez traduire votre programme en plusieurs langues, la section "Internationalization" vous donnera de précieux conseils.

Key Technologies

C'est un peu comme la section précédente : il s'agit de tutoriels à thèmes.

Ici, vous trouverez notamment un article sur la gestion de la fenêtre principale d'un programme (Main Window Architecture), ou encore sur la programmation Modèle / Vue dont on reparlera dans un prochain chapitre.

On retrouve plus particulièrement une introduction à chacun des principaux modules de Qt, à l'exception de Qt GUI qui fait déjà l'objet d'un tutoriel dans la section "Getting Started".

Si vous voulez démarrer avec le module réseau de Qt par exemple, il faut lire "Network Module". Si vous comptez faire appel à des bases de données depuis votre programme, consultez "SQL Module".

Add-ons & Services

On vous propose ici des services autour de Qt, à savoir :

- du support

- des formations
- etc.

Bref autant le dire, des choses qui intéressent des entreprises qui ont le moyen de payer pour ça.

On trouvera aussi un lien vers <http://qt-apps.org>, un site sur lequel vous trouverez des applications réalisées avec Qt ainsi que de nouveaux widgets que vous pouvez librement utiliser dans vos applications.

Tools

Comme vous le savez déjà, Qt est pré-installé avec un certain nombre d'outils que je vous avais présentés.
On trouve ici des tutoriels pour savoir utiliser chacun de ces logiciels :

- Qt Designer : le logiciel qui permet de dessiner à la souris une interface graphique ;
 - Qt Assistant : le logiciel qui contient la documentation dont j'ai parlé au début du chapitre ;
 - Qt Linguist : le logiciel qui permet de traduire une application de Qt vers une autre langue ;
 - qmake : l'utilitaire indispensable pour compiler une application Qt. Il propose beaucoup d'options et nous n'en avons vu qu'une petite partie pour le moment (la partie essentielle). Vous pourrez en apprendre plus sur la syntaxe des fichiers de projet .pro.
- Vous aurez peut-être besoin de lire cette section lorsque vous ferez des programmes un peu plus complexes ;
- All Tools : la liste de tous les outils proposés par Qt (y compris les précédents que je viens de citer).

Licenses & Credits

Voilà une section contenant des informations légales à propos de Qt. Vous y trouverez notamment un exemplaire de la GNU GPL, la licence libre qui vous autorise à utiliser librement et gratuitement Qt à condition que vous fassiez à votre tour un programme libre.

Bref, ce n'est pas une section très lisible par les programmeurs, il faut plutôt réserver ça aux juristes 

Comprendre la documentation d'une classe

Voilà la section la plus importante et la plus intéressante de ce chapitre : nous allons étudier la documentation d'une classe de Qt au hasard.

Chaque classe possède sa propre page, plus ou moins longue selon la complexité de la classe. Vous pouvez donc retrouver tout ce dont vous avez besoin de savoir sur une classe en lisant une seule page.

Bon, j'ai dit qu'on allait prendre une classe de Qt au hasard. Alors, voyons voir... sur qui ça va tomber... ah ! Je sais :

QLineEdit

Lorsque vous connaissez le nom de la classe et que vous voulez lire sa documentation, vous pouvez passer par le lien "All Classes" depuis le sommaire, ou encore taper directement dans votre navigateur <http://doc.trolltech.com/nomdelaclass.html> (exemple : <http://doc.trolltech.com/qlineedit.html>).

Vous devriez avoir une longue page qui s'affiche sous vos yeux ébahis, et qui commence par quelque chose comme ça :



QLineEdit Class Reference

[[QtGui module](#)]

The QLineEdit widget is a one-line text editor. [More...](#)

```
#include <QLineEdit>
```

Inherits [QWidget](#).

- [List of all members, including inherited members](#)
- [Qt 3 support members](#)

Public Types

- enum [EchoMode](#) { Normal, NoEcho, Password, PasswordEchoOnEdit }

Properties

Chaque documentation de classe suit exactement la même structure. Vous retrouverez donc les mêmes sections, les mêmes titres, etc.

Analysons à quoi correspond chacune de ces sections ! 😊

Introduction

Au tout début, vous pouvez lire une très courte introduction qui explique en quelques mots à quoi sert la classe.

Ici, nous avons : "The QLineEdit widget is a one-line text editor.", ce qui signifie, si vous avez bien révisé votre anglais, que ce widget est un éditeur de texte sur une ligne, comme le montre la capture d'écran ci-contre.

Le lien "[More...](#)" vous amène vers une description plus détaillée de la classe. En général, il s'agit d'un mini-tutoriel pour apprendre à utiliser la classe. Je vous recommande de toujours lire cette introduction quand vous travaillez avec une classe que vous ne connaissiez pas jusqu'alors.

Ça vous fera gagner beaucoup de temps car vous saurez "par où commencer" et "quelles sont les principales méthodes de la classe".

Ensuite, on vous donne le header à inclure pour pouvoir utiliser la classe dans votre code, en l'occurrence il s'agit de :

Code : C++ - [Sélectionner](#)

```
#include <QLineEdit>
```

Puis, vous avez une information très importante à côté de laquelle on passe souvent : la classe dont hérite votre classe. Ici, on voit que QWidget est le parent de QLineEdit. Donc QLineEdit récupère toutes les propriétés de QWidget. Ça a son importance comme nous allons le voir...

Voilà pour l'intro ! 😊

Maintenant, voyons voir les sections qui suivent...

Public Types

Les classes définissent parfois des types de données personnalisés, sous la forme de ce qu'on appelle des énumérations (j'en ai parlé dans mon cours de C pour ceux qui auraient un trou de mémoire !).

Ici, QLineEdit définit l'énumération EchoMode qui propose plusieurs valeurs : Normal, NoEcho, Password, etc.

Une énumération ne s'utilise pas "telle quelle". C'est juste une liste de valeurs, que vous pouvez renvoyer à une méthode spécifique qui en a besoin. Dans le cas de QLineEdit, c'est la méthode `setEchoMode(EchoMode)` qui en a besoin, car elle n'accepte que des données de type EchoMode..

Pour envoyer la valeur "Password", il faudra écrire : `setEchoMode(QLineEdit::Password)` .

Properties

Vous avez là toutes les propriétés d'une classe que vous pouvez lire et modifier.

Euh, ce ne sont pas des attributs ça par hasard ?

Si. Mais la doc ne vous affiche que les attributs pour lesquels Qt définit des accesseurs. Il y a de nombreux attributs "internes" à chaque classe que la doc ne vous montre pas car ils ne vous concernent pas.

Toutes les propriétés sont donc des attributs intéressants de la classe que vous pouvez lire et modifier. Comme je vous l'avais dit dans un chapitre précédent, Qt suit cette convention pour le nom des accesseurs :

- `propriete()` : c'est la méthode accesseur qui vous permet de lire la propriété ;
- `setPropriete()` : c'est la méthode accesseur qui vous permet de modifier la propriété.

Prenons par exemple la propriété `text`. C'est la propriété qui stocke le texte rentré par l'utilisateur dans le champ de texte QLineEdit.

Comme indiqué dans la doc, `text` est de type `QString`. Vous devez donc récupérer la valeur dans un `QString`. Pour récupérer le texte entré par l'utilisateur dans une variable `contenu`, on fera donc :

Code : C++ - [Sélectionner](#)

```
QLineEdit monChamp( "Contenu du champ" );
QString contenu = monChamp.text();
```

Pour modifier le texte présent dans le champ, on écrira :

Code : C++ - [Sélectionner](#)

```
QLineEdit monChamp;
monChamp.setText("Entrez votre nom ici");
```

Vous remarquerez que dans la doc, la propriété `text` est un lien. [Cliquez dessus](#). Cela vous amènera plus bas sur la même page vers une description de la propriété (que fait-elle ? à quoi sert-elle ?).
On vous y donne aussi le prototype des accesseurs :

- `QString text () const`
- `void setText (const QString &)`

Et enfin, parfois vous verrez comme là une mention "See also" (voir aussi) qui vous invite à aller voir d'autres propriétés ou méthodes de la classe qui ont un rapport avec celle que vous êtes en train de lire. Ici, on vous dit que les méthodes `insert()` et `clear()` pourraient vous intéresser. En effet, par exemple `clear()` vide le contenu du champ de texte, c'est donc une méthode intéressante en rapport avec la propriété qu'on était en train de lire.

TRES IMPORTANT : dans la liste des propriétés en haut de la page, notez les mentions "56 properties inherited from QWidget", et "1 property inherited from QObject". Comme `QLineEdit` hérite de `QWidget`, qui lui-même hérite de `QObject`, il possède du coup toutes les propriétés et toutes les méthodes de ses classes parentes !

En clair, les propriétés que vous voyez là ne sont qu'un tout petit bout des possibilités offertes par `QLineEdit`. Si vous cliquez sur le lien `QWidget`, on vous amène vers la [liste des propriétés de QWidget](#). Vous disposez aussi de toutes ces propriétés dans un [QLineEdit](#) !

Vous pouvez donc utiliser la propriété `width` (largeur) qui est définie dans `QWidget` pour modifier la largeur de votre `QLineEdit`. Toute la puissance de l'héritage est là ! Tous les widgets possèdent donc ces propriétés "de base", ils n'ont plus qu'à définir des propriétés qui leur sont spécifiques.

J'insiste bien dessus car au début je me disais souvent : "Mais pourquoi il y a aussi peu de choses dans cette classe ?". En fait, il ne faut pas s'y fier et toujours regarder les classes parentes dont hérite la classe qui vous intéresse. Tout ce que les classes parentes possèdent, vous y avez accès aussi.

Public Functions

C'est bien souvent la section la plus importante. Vous y trouverez toutes les méthodes publiques (parce que les privées ne vous concernent pas) de la classe. On trouve dans le lot :

- le (ou les) constructeur(s) de la classe. Très intéressant pour savoir comment créer un objet à partir de cette classe ;
- les accesseurs de la classe (comme `text()` et `setText()` qu'on vient de voir), basés sur les attributs ;
- et enfin d'autres méthodes publiques qui ne sont ni des constructeurs ni des accesseurs et qui effectuent diverses opérations sur l'objet. Par exemple : `home()`, qui ramène le curseur au début du champ de texte.

Cliquez sur le nom d'une méthode pour en savoir plus sur son rôle et son fonctionnement.

Lire et comprendre le prototype

A chaque fois, il faut que vous lisiez attentivement le prototype de la méthode, c'est très important ! Le prototype à lui seul vous donne une grosse quantité d'informations sur la méthode.

Prenons l'exemple du constructeur. On voit qu'on a 2 prototypes :

- `QLineEdit (QWidget * parent = 0)`
- `QLineEdit (const QString & contents, QWidget * parent = 0)`

Vous noterez que certains paramètres sont facultatifs.

Si vous cliquez sur un de ces constructeurs, par exemple [le second](#), on vous explique la signification de chacun de ces paramètres.

On apprend que `parent` est un pointeur vers le widget qui "contiendra" notre `QLineEdit` (par exemple une fenêtre), et que `contents` est le texte qui doit être écrit dans le `QLineEdit` par défaut.

Cela veut dire, si on prend en compte que le paramètre `parent` est facultatif, qu'on peut créer un objet de type `QLineEdit` de 4 façons différentes :

Code : C++ - [Sélectionner](#)

```
QLineEdit monChamp(); // Appel du premier constructeur
QLineEdit monChamp(fenetre); // Appel du premier constructeur
QLineEdit monChamp("Entrez un texte"); // Appel du second constructeur
QLineEdit monChamp("Entrez un texte", fenetre); // Appel du second constructeur
```

C'est fou tout ce qu'un prototype peut raconter hein ? 😊

Quand la méthode attend un paramètre d'un type que vous ne connaissez pas...

Je viens de voir la méthode [setAlignment](#), mais elle demande un paramètre de type Qt::Alignment. Comment je lui donne ça moi, je connais pas les Qt::Alignment !

Pas de panique. Il vous arrivera très souvent de tomber sur une méthode qui attend un paramètre d'un type qui vous est inconnu. Par exemple, vous n'avez jamais entendu parler de Qt::Alignment. Qu'est-ce que c'est que ce type ?

La solution pour savoir comment envoyer un paramètre de type Qt::Alignment consiste à cliquer dans la doc sur le lien [Qt::Alignment](#) (eh oui, ce n'est pas un lien par hasard !).

Ce lien vous amènera vers une page qui vous explique ce qu'est le type Qt::Alignment.

Il peut y avoir 2 types différents :

- Les énumérations : Qt::Alignment en est une. Les énumérations sont très simples à utiliser, c'est une série de valeurs. Il suffit d'écrire la valeur que l'on veut, comme le donne la [documentation de Qt::Alignment](#), par exemple Qt::AlignCenter. La méthode pourra donc être appelée comme ceci :

Code : C++ - [Sélectionner](#)

```
monChamp.setAlignment(Qt::AlignCenter);
```

- Les classes : parfois, la méthode attend un objet issu d'une classe précise pour travailler. Là c'est un peu plus compliqué : il va falloir créer un objet de cette classe et l'envoyer à la méthode.

Prenons par exemple [setValidator](#), qui attend un pointeur vers un QValidator. La méthode setValidator vous dit qu'elle permet de vérifier si l'utilisateur a rentré un texte valide, ce qui peut être utile si vous voulez vérifier que l'utilisateur a bien rentré un nombre entier et non pas "Bonjour ça va ?" quand vous lui demandez son âge...

Si vous cliquez sur le lien [QValidator](#), on vous emmène vers la page qui explique comment utiliser la classe QValidator. Lisez le texte d'introduction pour comprendre ce que cette classe est censée faire, puis regardez les constructeurs afin de savoir comment créer un objet de type QValidator.

Parfois, comme là, c'est même un peu plus délicat. QValidator est une classe abstraite (c'est ce que vous dit l'intro de sa doc), ce qui signifie qu'on ne peut pas créer d'objet de type QValidator et qu'il faut utiliser une de ses classes filles 😊. Au tout début, la page de la doc de QValidator vous dit "Inherited by QDoubleValidator, QIntValidator, and QRegExpValidator". Cela signifie que ces classes héritent de QValidator et que vous pouvez les utiliser aussi. En effet, une classe fille est compatible avec la classe mère, comme nous l'avons déjà vu dans le [chapitre sur l'héritage](#).

Nous, nous voulons autoriser uniquement la personne à rentrer un nombre entier, nous allons donc utiliser [QIntValidator](#). Il faut créer un objet de type QIntValidator. Regardez ses constructeurs et choisissez celui qui vous convient.

Au final (ouf !), pour utiliser setValidator, on peut faire comme ceci :

Code : C++ - [Sélectionner](#)

```
QValidator *validator = new QIntValidator(0, 150, this);
monChamp.setValidator(validator);
```

... pour s'assurer que la personne ne rentrera qu'un nombre compris entre 0 et 150 ans (ça laisse de la marge 😊).

La morale de l'histoire, c'est qu'il ne faut pas avoir peur d'aller lire la documentation d'une classe dont a besoin la classe sur laquelle vous travaillez, et même des fois là d'aller voir les classes filles.

Ça peut faire un peu peur au début, mais c'est une gymnastique de l'esprit à acquérir. N'hésitez donc pas à sauter de lien en lien dans la doc pour arriver enfin à envoyer à cette \$%@#\$% de méthode un objet du type qu'elle attend ! 😊

Public Slots

Les slots sont des méthodes comme les autres, à la différence près qu'on peut aussi les connecter à un signal comme on l'a vu dans le chapitre sur les signaux et les slots.

Notez que rien ne vous interdit d'appeler un slot directement, comme si c'était une méthode comme une autre.

Par exemple, le [slot undo\(\)](#) annule la dernière opération de l'utilisateur.

Vous pouvez l'appeler comme une bête méthode :

Code : C++ - [Sélectionner](#)

```
monChamp.undo();
```

... mais la particularité du fait que `undo()` soit un slot, c'est que vous pouvez aussi le connecter à un autre widget. Par exemple, on peut imaginer un menu Edition / Annuler dont le signal "cliqué" sera connecté au slot "undo" du champ de texte 😊

Tous les slots offerts par `QLineEdit` ne sont pas dans cette liste. Je me permets de vous rappeler une fois de plus qu'il faut penser à regarder les mentions comme "19 public slots inherited from `QWidget`", qui vous invitent à aller voir les slots de `QWidget` auxquels vous avez aussi accès.

C'est ainsi que vous découvrez que vous disposez du slot `hide()` qui permet de masquer votre `QLineEdit`.

Signals

C'est la liste des signaux que peut envoyer un `QLineEdit`.

Un signal est un évènement qui s'est produit et que l'on peut connecter à un slot (le slot pouvant appartenir à cet objet ou à un autre).

Par exemple, le signal `textChanged()` est émis à chaque fois que l'utilisateur modifie le texte à l'intérieur du `QLineEdit`. Si vous le voulez, vous pouvez connecter ce signal à un slot pour qu'une action soit effectuée à chaque fois que le texte est modifié.

Attention encore une fois à bien regarder les signaux hérités de `QWidget` et `QObject`, car ils appartiennent aussi à la classe `QLineEdit`. Je sais que je suis lourd à force de répéter ça, inutile de me le dire, je le fais exprès pour que ça rentre 🍪

Protected Functions

Ce sont des méthodes protégées. Elles ne sont ni public, ni private, mais protected.

Comme on l'a vu dans le chapitre sur l'héritage, ce sont des méthodes privées (auxquelles vous ne pouvez pas accéder directement en tant qu'utilisateur de la classe) mais qui seront héritées et donc réutilisables si vous créez une classe basée sur `QLineEdit`.

Il est très fréquent d'hériter des classes de Qt, on l'a d'ailleurs déjà fait avec `QWidget` pour créer une fenêtre personnalisée. Si vous héritez de `QLineEdit`, sachez donc que vous disposerez aussi de ces méthodes.

Additional Inherited Members

Si des éléments hérités n'ont pas été listés jusqu'ici, on les retrouvera dans cette section à la fin.

Par exemple, la classe `QLineEdit` ne définit pas de méthodes statiques, mais elle en possède quelques-unes héritées de `QWidget` et `QObject`.

Je vous rappelle qu'une méthode statique est une méthode qui peut être appelée sans avoir eu à créer d'objet. C'est un peu comme une fonction.

Il n'y a rien de bien intéressant avec QLineEdit, mais sachez par exemple que la classe QString possède de nombreuses méthodes statiques, comme number() qui convertit le nombre donné en une chaîne de caractères de type QString.

Code : C++ - [Sélectionner](#)

```
QString maChaine = QString::number(12);
```

Une méthode statique s'appelle comme ceci : NomDeLaClasse::nomDeLaMethode().

On a déjà vu tout ça dans les chapitres précédents, je ne fais ici que des rappels 😊

Ce chapitre était absolument nécessaire car je suis convaincu que vous ne pouvez pas passer à côté de la doc.

Toutes les informations dont vous avez besoin y sont, le tout est d'être capable de les retrouver et de les comprendre. C'est, je l'espère, ce que ce chapitre vous aura aidés à faire. Il s'agissait de faire une sorte de "guide" pour rassurer les débutants qui n'ont jamais vraiment touché à une documentation.

Le concept pour apprendre un langage ou une bibliothèque est donc le suivant :

1. d'abord on lit des tutoriels qui nous permettent de savoir comment débuter et dans quelle direction chercher ;
2. et ensuite on consulte la doc pour connaître le détail des fonctions et des classes.

Il n'existe pas de tutoriel qui vous apprendra tout de A à Z sur une bibliothèque comme Qt par exemple. Ca n'aurait pas de sens et ce serait complètement stupide de chercher à faire ça étant donné que cela représenterait un travail énorme qui deviendrait obsolète dès la prochaine mise à jour de Qt.

Le but de mon tutoriel n'est donc pas de "tout vous apprendre" mais de vous apprendre à apprendre. Bien sûr, je ne vous lâche pas dans la nature comme ça : j'ai encore beaucoup de choses à vous expliquer dans ce tutoriel. Mais pensez à lire la doc en parallèle de mes cours, et une fois que vous aurez fini de lire ma prose, ayez le réflexe de consulter la doc à chaque fois que vous en avez besoin.

C'est un vrai réflexe de programmeur 😊

Positionner ses widgets avec les layouts

Comme vous le savez, une fenêtre peut contenir toutes sortes de widgets : des boutons, des champs de texte, des cases à cocher...

Placer ces widgets sur la fenêtre est une science à part entière. Je veux dire par là qu'il faut vraiment y aller avec méthode, si on ne veut pas que la fenêtre ressemble rapidement à un champ de bataille 😱

Comment bien placer les widgets sur la fenêtre ?

Comment gérer les redimensionnements de la fenêtre ?

Comment s'adapter automatiquement à toutes les résolutions d'écran ?

On distingue 2 techniques différentes pour positionner des widgets :

- Le positionnement absolu : c'est celui que nous avons vu jusqu'ici, avec l'appel à la méthode setGeometry (ou move)... Ce positionnement est très précis, car on place les widgets au pixel près, mais cela comporte un certain nombre de défauts comme nous allons le voir.
- Le positionnement relatif : c'est le plus flexible et c'est celui que je vous recommande d'utiliser autant que possible. Nous allons l'étudier dans ce chapitre.

Le positionnement absolu et ses défauts

Nous allons commencer par voir le code Qt de base que nous allons utiliser dans ce chapitre, puis nous ferons quelques rappels sur le positionnement absolu que vous avez déjà utilisé sans savoir exactement ce que c'était 😊

Le code Qt de base

Dans les chapitres précédents, nous avions créé un projet Qt constitué de 3 fichiers :

- main.cpp : contenait le main qui se chargeait juste d'ouvrir la fenêtre principale.
- MaFenetre.h : contenait l'en-tête de notre classe MaFenetre qui héritait de QWidget.
- MaFenetre.cpp : contenait l'implémentation des méthodes de MaFenetre, notamment du constructeur.

C'est l'architecture que l'on utilisera dans la plupart de nos projets Qt.

Toutefois, pour ce chapitre nous n'avons pas besoin d'une architecture aussi complexe, et nous allons faire comme dans les tous premiers chapitre Qt : nous allons juste utiliser un main (1 seul fichier : main.cpp).

Voici le code de votre projet, sur lequel nous allons commencer :

Code : C++ - [Sélectionner](#)

```
#include < QApplication>
#include < QPushButton>

int main( int argc, char *argv[] )
{
    QApplication app( argc, argv );

    QWidget fenetre;

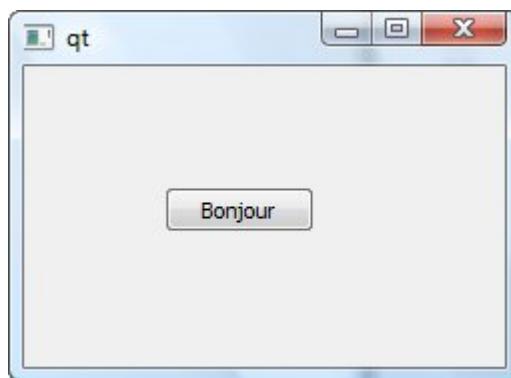
    QPushButton bouton( "Bonjour", &fenetre );
    bouton.move( 70, 60 );

    fenetre.show();

    return app.exec();
}
```

C'est très simple : nous créons une fenêtre, et nous affichons un bouton que nous plaçons aux coordonnées (70, 60) sur la fenêtre.

Le résultat est le suivant :



Les défauts du positionnement absolu

Dans le code précédent, nous avons positionné notre bouton de manière absolue en faisant `bouton.move(70, 60);`
Le bouton a été très précisément placé 70 pixels sur la droite et 60 pixels plus bas.

Le problème... c'est que ce n'est pas flexible du tout. Imaginez que l'utilisateur s'amuse à redimensionner la fenêtre :



C'est moche, non ?

Le bouton ne bouge pas de place. Du coup, si on réduit la taille de la fenêtre, il sera coupé en deux, et pourra même disparaître si on réduit trop la taille.

Dans ce cas, pourquoi ne pas empêcher l'utilisateur de redimensionner la fenêtre ? On avait fait ça grâce à `setFixedSize` dans les chapitres précédents...

Oui, vous pouvez faire cela. C'est d'ailleurs ce que font le plus souvent les développeurs de logiciels qui positionnent leurs widgets en absolu. Cependant, l'utilisateur apprécie aussi de pouvoir redimensionner sa fenêtre. Ce n'est qu'une demi-solution.

D'ailleurs, il y a un autre problème que `setFixedSize` ne peut pas régler : le cas des résolutions d'écran plus petites que la vôtre. Imaginez que vous placiez un bouton 1200 pixels sur la droite parce que vous avez une grande résolution (1600 x 1200), et que l'utilisateur soit dans une résolution plus petite que vous (1024 x 768). Il ne pourra jamais voir le bouton, parce qu'il ne pourra jamais agrandir autant sa fenêtre !

Alors quoi ? Le positionnement absolu c'est mal ? Où veux-tu en venir ?
Et surtout, comment peut-on faire autrement ?

Non, le positionnement absolu ce n'est pas "mal". Il sert parfois quand on a vraiment besoin de positionner au pixel près. Vous pouvez l'utiliser dans certains de vos projets, mais autant que possible, préférez l'autre méthode : le positionnement relatif.

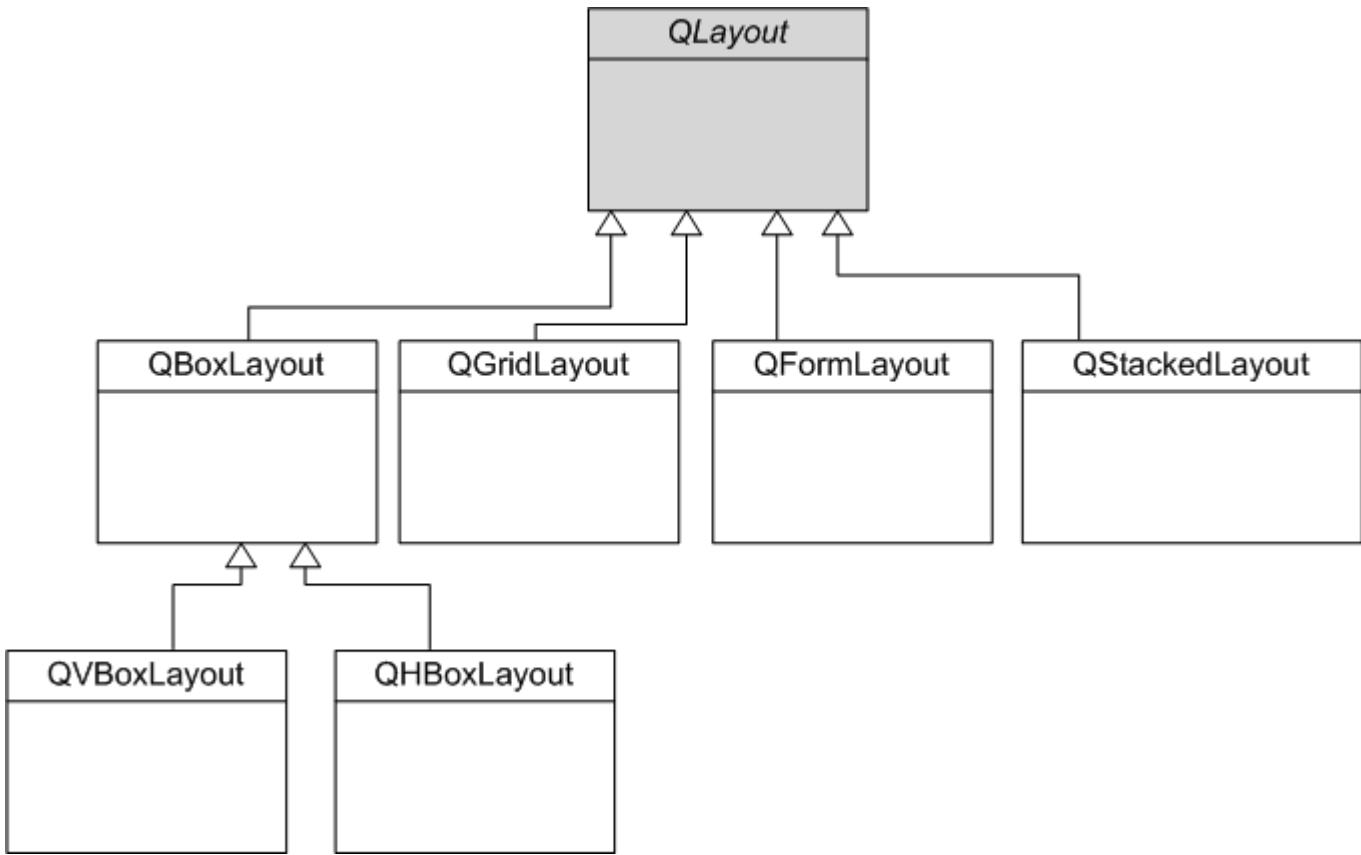
Le positionnement relatif, cela consiste à expliquer comment les widgets sont agencés les uns par rapport aux autres, plutôt que d'utiliser une position en pixels. Par exemple, on peut dire "Le bouton 1 est en-dessous du bouton 2, qui est à gauche du bouton 3".

Le positionnement relatif est géré par ce qu'on appelle les layouts avec Qt. Ce sont des conteneurs de widgets.
C'est justement l'objet principal de ce chapitre 😊

L'architecture des classes de layout

Pour positionner intelligemment nos widgets, nous allons utiliser des classes de Qt gérant les layouts.
Il existe par exemple des classes gérant le positionnement horizontal et vertical des widgets (ce que nous allons étudier en premier), ou encore le positionnement sous forme de grille.

Pour que vous y voyiez plus clair, je vous propose de regarder ce schéma de mon cru :



Ce sont les classes gérant les layouts de Qt.

Toutes les classes héritent de la classe de base `QLayout`.

On compte donc en gros les classes :

- `QBoxLayout`
- `QHBoxLayout`
- `QVBoxLayout`
- `QGridLayout`
- `QFormLayout`
- `QStackedLayout`

Nous allons étudier chacune de ces classes dans ce chapitre, à l'exception de `QStackedLayout` (gestion des widgets sur plusieurs pages) qui est un peu trop complexe pour qu'on puisse travailler dessus ici. On utilisera plutôt des widgets qui le réutilisent, comme `QWizard` qui permet de créer des assistants.

Euh... Mais pourquoi tu as écrit `QLayout` en italique, et pourquoi tu as grisé la classe ? 🤔

`QLayout` est ce qu'on appelle une classe abstraite. Je ne vous en ai pas trop parlé jusqu'ici.

En fait, une classe abstraite est une classe "de base" qu'on ne peut pas instancier. C'est-à-dire qu'on ne peut pas créer d'objets de type `QLayout`, il faut forcément créer un objet à partir d'une des classes filles (`QBoxLayout`, `QGridLayout`...).

A quoi ça sert de créer une classe qui ne nous permette pas de créer d'objet ? 🤔

Ca sert pour le programmeur, pour avoir juste une classe "de base".

Cependant, comme ça n'a pas de sens de créer d'objet de type `QLayout`, la classe a été définie comme étant abstraite.

Je ne vais pas rentrer dans les détails de "comment faire pour créer une classe abstraite en C++". Ce serait un peu trop compliqué et

hors-sujet.

Tout ce que vous avez besoin de retenir, c'est que vous pouvez créer des objets de type QBoxLayout, QGridLayout, etc, mais pas d'objets de type QLayout. En quelque sorte, QLayout sert de "modèle" de base pour les autres classes mais on ne peut rien faire avec elle seule 😞

Les layouts horizontaux et verticaux

Attaquons sans plus tarder l'étude de nos premiers layouts (les plus simples), vous allez mieux comprendre à quoi tout cela sert 😊

Nous allons travailler sur 2 classes :

- [QHBoxLayout](#)
- [QVBoxLayout](#)

QHBoxLayout et QVBoxLayout héritent de [QBoxLayout](#). Ce sont des classes très similaires (la doc Qt parle de "convenience classes", des classes qui sont là pour vous aider à aller plus vite mais qui sont en fait quasiment identiques à QBoxLayout). Nous n'allons pas utiliser QBoxLayout, mais juste ses classes filles QHBoxLayout et QVBoxLayout (ça revient au même).

Le layout horizontal

L'utilisation d'un layout se fait en 3 temps :

1. On crée les widgets
2. On crée le layout et on place les widgets dedans
3. On dit à la fenêtre d'utiliser le layout qu'on a créé

1/ Créer les widgets

Pour les besoins de ce tutoriel, nous allons créer plusieurs boutons de type QPushButton :

Code : C++ - [Sélectionner](#)

```
QPushButton *bouton1 = new QPushButton("Bonjour");
QPushButton *bouton2 = new QPushButton("les");
QPushButton *bouton3 = new QPushButton("Zéros");
```

Vous remarquerez que j'utilise des pointeurs. En effet, j'aurais très bien pu faire sans pointeurs comme ceci :

Code : C++ - [Sélectionner](#)

```
QPushButton bouton1("Bonjour");
QPushButton bouton2("les");
QPushButton bouton3("Zéros");
```

... cette méthode a l'air plus simple, mais vous verrez que c'est plus pratique de travailler directement avec des pointeurs par la suite 😊

La différence entre ces 2 codes, c'est que bouton1 est un pointeur dans le premier code, tandis que c'est un objet dans le second code.

On va donc utiliser la première méthode avec les pointeurs.

Bon, on a 3 boutons, c'est bien. Mais les plus perspicaces d'entre vous auront remarqué qu'on n'a pas indiqué quelle était la fenêtre parente, comme on aurait fait avant :

Code : C++ - [Sélectionner](#)

```
QPushButton *bouton1 = new QPushButton("Bonjour", &fenetre);
```

On n'a pas fait comme ça, et c'est fait exprès justement. Nous n'allons pas placer les boutons dans la fenêtre directement, mais dans un conteneur : le layout.

2/ Créer le layout et placer les widgets dedans

Créons justement ce layout, un layout horizontal :

Code : C++ - [Sélectionner](#)

```
QHBoxLayout *layout = new QHBoxLayout;
```

Le constructeur de cette classe est simple, on n'a pas besoin d'indiquer de paramètre.

Maintenant que notre layout est créé, rajoutons nos widgets à l'intérieur :

Code : C++ - [Sélectionner](#)

```
layout->addWidget(bouton1);
layout->addWidget(bouton2);
layout->addWidget(bouton3);
```

La méthode addWidget du layout attend que vous lui donnez en paramètre un pointeur vers le widget à ajouter au conteneur. Voilà pourquoi je vous ai fait utiliser des pointeurs (sinon il aurait fallu écrire layout->addWidget(&bouton1) ; à chaque fois).

3/ Indiquer à la fenêtre d'utiliser le layout

Maintenant, dernière chose : il faut placer le layout dans la fenêtre. Il faut dire à la fenêtre : "tu vas utiliser ce layout, qui contient mes widgets".

Code : C++ - [Sélectionner](#)

```
fenetre.setLayout(layout);
```

La méthode setLayout de la fenêtre attend un pointeur vers le layout à utiliser.

Et voilà, notre fenêtre contient maintenant notre layout, qui contient les widgets. Le layout se chargera d'organiser les widgets horizontalement tout seul.

Résumé du code

Voici le code complet de notre fichier main.cpp :

Code : C++ - [Sélectionner](#)

```

#include <QApplication>
#include <QPushButton>
#include <QHBoxLayout>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QWidget fenetre;

    QPushButton *bouton1 = new QPushButton("Bonjour");
    QPushButton *bouton2 = new QPushButton("les");
    QPushButton *bouton3 = new QPushButton("Zéros");

    QHBoxLayout *layout = new QHBoxLayout;
    layout->addWidget(bouton1);
    layout->addWidget(bouton2);
    layout->addWidget(bouton3);

    fenetre.setLayout(layout);

    fenetre.show();

    return app.exec();
}

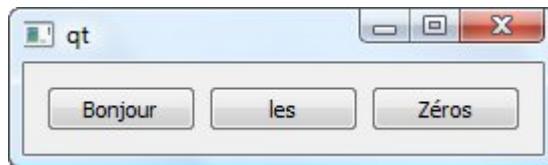
```

J'ai surligné les principales nouveautés.

En particulier, comme d'hab' lorsque vous utilisez une nouvelle classe Qt, pensez à l'inclure au début de votre code : `#include <QHBoxLayout>`

Résultat

Voilà à quoi ressemble la fenêtre maintenant que l'on utilise un layout horizontal :



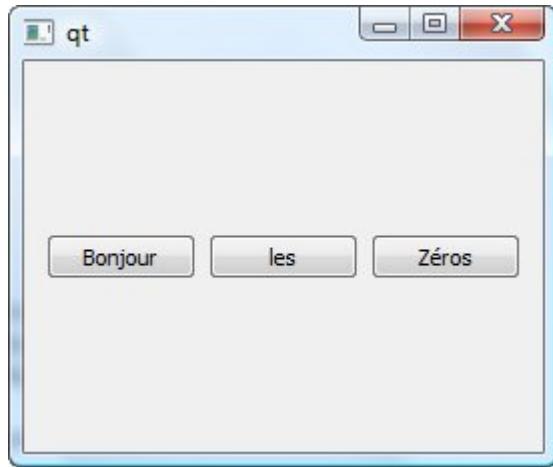
Les boutons sont automatiquement disposés de manière horizontale ! 😊

L'intérêt principal du layout, c'est son comportement face aux redimensionnements de la fenêtre.
Essayons de l'élargir :



Les boutons continuent de prendre l'espace en largeur.

On peut aussi l'agrandir en hauteur :

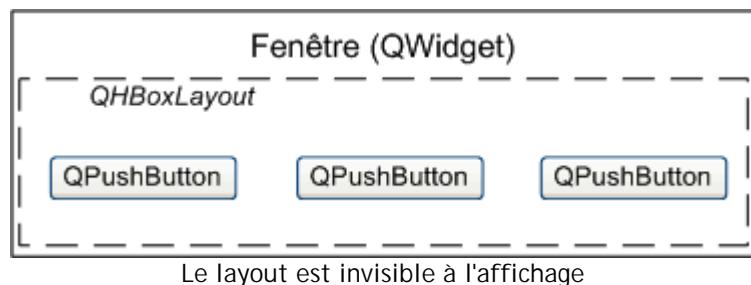


On remarque que les widgets restent centrés verticalement.

Vous pouvez aussi essayer de réduire la taille de la fenêtre. On vous interdira de la réduire si les boutons ne peuvent plus être affichés, ce qui vous garantit que les boutons ne risquent plus de disparaître comme avant ! 😊

Schéma des conteneurs

En résumé, la fenêtre contient le layout qui contient les widgets. Le layout se charge d'organiser les widgets. Schématiquement, ça se passe donc comme ça :



On vient de voir le layout QHBoxLayout qui organise les widgets horizontalement.

Il y en a un autre qui les organise verticalement (c'est quasiment la même chose) : QVBoxLayout.

Le layout vertical

Pour utiliser un layout vertical, il suffit de remplacer QHBoxLayout par QVBoxLayout dans le code précédent. Oui oui, c'est aussi simple que ça 😊

Code : C++ - [Sélectionner](#)

```

#include <QApplication>
#include <QPushButton>
#include <QVBoxLayout>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QWidget fenetre;

    QPushButton *bouton1 = new QPushButton("Bonjour");
    QPushButton *bouton2 = new QPushButton("les");
    QPushButton *bouton3 = new QPushButton("Zéros");

    QVBoxLayout *layout = new QVBoxLayout;
    layout->addWidget(bouton1);
    layout->addWidget(bouton2);
    layout->addWidget(bouton3);

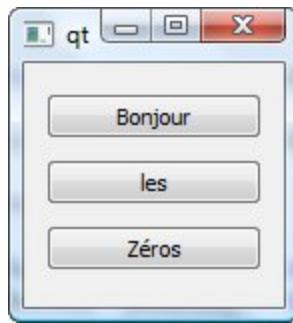
    fenetre.setLayout(layout);
    fenetre.show();

    return app.exec();
}

```

N'oubliez pas d'inclure QVBoxLayout.

Compilez et exécutez ce code, et admirez le résultat :



Amusez-vous à redimensionner la fenêtre. Vous voyez là encore que la layout adapte les widgets qu'il contient à toutes les dimensions. Il empêche en particulier la fenêtre de devenir trop petite, ce qui aurait empêché l'affichage des boutons.

La suppression automatique des widgets

Eh ! Je viens de me rendre compte que tu fais des new dans tes codes, mais il n'y a pas de delete ! Si tu alloues des objets sans les supprimer, ils vont pas rester en mémoire ?

Si, mais comme je vous l'avais dit plus tôt, Qt est intelligent 😊

En fait, les widgets sont placés dans un layout, qui est lui-même placé dans la fenêtre. Lorsque la fenêtre est supprimée (ici à la fin du programme), tous les widgets contenus dans son layout sont supprimés par Qt. C'est donc Qt qui se charge de faire les delete pour nous.

Bien, vous devriez commencer à comprendre comment fonctionnent les layouts 😊

Comme on l'a vu au début du chapitre, il y a de nombreux layouts, qui ont chacun leurs spécificités ! Intéressons-nous maintenant au puissant (mais complexe) QGridLayout.

Le layout de grille

Les layouts horizontaux et verticaux sont gentils, mais il ne permettent pas de créer des dispositions très complexes sur votre fenêtre.

C'est là qu'entre en jeu [QGridLayout](#), qui est en fait un peu un assemblage de QHBoxLayout et QVBoxLayout. Il s'agit d'une disposition en grille, comme un tableau avec des lignes et des colonnes.

Schéma de la grille

Il faut imaginer que votre fenêtre peut être découpée sous la forme d'une grille avec une infinité de cases, comme ceci :

0, 0	0, 1	0, 2	...
1, 0	1, 1	1, 2	...
2, 0	2, 1	2, 2	...
...

Si on veut placer un widget en haut à gauche, il faudra le placer à la case de coordonnées (0, 0).
Si on veut en placer un autre en-dessous, il faudra utiliser les coordonnées (1, 0).

Ainsi de suite 😊

Utilisation basique de la grille

Essayons d'utiliser un QGridLayout simplement pour commencer (oui parce qu'on peut aussi l'utiliser de manière compliquée 😱).

Nous allons placer un bouton en haut à gauche, un à sa droite et un en-dessous.

La seule différence réside en fait dans l'appel à la méthode addWidget. Celle-ci accepte 2 paramètres supplémentaires : les coordonnées où placer le widget sur la grille.

Code : C++ - [Sélectionner](#)

```

#include <QApplication>
#include <QPushButton>
#include <QGridLayout>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QWidget fenetre;

    QPushButton *bouton1 = new QPushButton("Bonjour");
    QPushButton *bouton2 = new QPushButton("les");
    QPushButton *bouton3 = new QPushButton("Zéros");

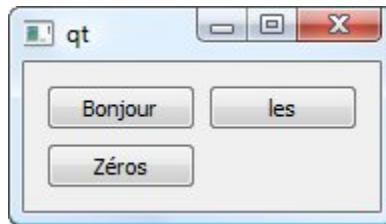
    QGridLayout *layout = new QGridLayout;
    layout->addWidget(bouton1, 0, 0);
    layout->addWidget(bouton2, 0, 1);
    layout->addWidget(bouton3, 1, 0);

    fenetre.setLayout(layout);
    fenetre.show();

    return app.exec();
}

```

Résultat :



Si vous comparez avec le schéma de la grille que j'ai fait plus haut, vous voyez que les boutons ont bien été disposés selon les bonnes coordonnées.

D'ailleurs en parlant du schéma plus haut, il y a un truc que je comprends pas, c'est tous ces points de suspension "..." là. Ca veut dire que la taille de la grille est infinie ? Dans ce cas, comment je fais pour placer un bouton en bas à droite ?

Qt "sait" quel est le widget à mettre en bas à droite en fonction des coordonnées des autres widgets. Le widget qui a les coordonnées les plus élevées sera placé en bas à droite.

Petit test, rajoutons un bouton aux coordonnées (1, 1) :

Code : C++ - [Sélectionner](#)

```

#include <QApplication>
#include <QPushButton>
#include <QGridLayout>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QWidget fenetre;

    QPushButton *bouton1 = new QPushButton("Bonjour");
    QPushButton *bouton2 = new QPushButton("les");
    QPushButton *bouton3 = new QPushButton("Zéros");
    QPushButton *bouton4 = new QPushButton("!!!");

    QGridLayout *layout = new QGridLayout;
    layout->addWidget(bouton1, 0, 0);
    layout->addWidget(bouton2, 0, 1);
    layout->addWidget(bouton3, 1, 0);
    layout->addWidget(bouton4, 1, 1);

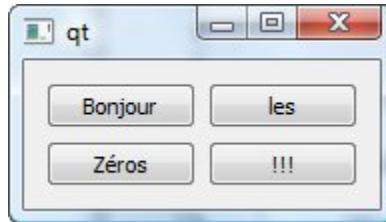
    fenetre.setLayout(layout);

    fenetre.show();

    return app.exec();
}

```

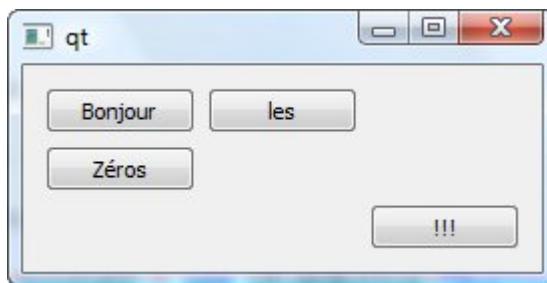
Résultat :



Si on veut, on peut aussi décaler le bouton encore plus en bas à droite dans une nouvelle ligne et une nouvelle colonne :

Code : C++ - [Sélectionner](#)

```
layout->addWidget(bouton4, 2, 2);
```



C'est compris ? 😊

Un widget qui occupe plusieurs cases

L'avantage de la disposition en grille, c'est qu'on peut faire en sorte qu'un widget occupe plusieurs cases à la fois. On parle de spanning (ceux qui font du HTML doivent avoir entendu parler des attributs rowspan et colspan sur les tableaux).

Pour faire cela, il faut appeler une version surchargée de addWidget qui accepte 2 paramètres supplémentaires : le rowspan et le columnSpan.

- rowspan : nombre de lignes qu'occupe le widget (par défaut 1)
- columnSpan : nombre de colonnes qu'occupe le widget (par défaut 1)

Imaginons un widget placé en haut à gauche, aux coordonnées (0, 0). Si on lui donne un rowspan de 2, il occupera alors l'espace suivant :

rowSpan = 2			
			...
			...
...

Si on lui donne un columnSpan de 3, il occupera cet espace :

columnSpan = 3			...
			...
			...
...

L'espace pris par le widget au final dépend de la nature du widget (les boutons s'agrandissent en largeur mais pas en hauteur par exemple), et dépend du nombre de widgets sur la grille. En pratiquant vous allez rapidement comprendre comment ça fonctionne.

Essayons de faire en sorte que le bouton "Zéros" prenne 2 colonnes de largeur :

Code : C++ - [Sélectionner](#)

```

#include <QApplication>
#include <QPushButton>
#include <QGridLayout>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QWidget fenetre;

    QPushButton *bouton1 = new QPushButton("Bonjour");
    QPushButton *bouton2 = new QPushButton("les");
    QPushButton *bouton3 = new QPushButton("Zéros");

    QGridLayout *layout = new QGridLayout;
    layout->addWidget(bouton1, 0, 0);
    layout->addWidget(bouton2, 0, 1);
    layout->addWidget(bouton3, 1, 0, 1, 2);

    fenetre.setLayout(layout);
    fenetre.show();

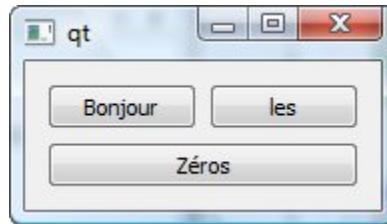
    return app.exec();
}

```

Notez la ligne : `layout->addWidget(bouton3, 1, 0, 1, 2);`

Les 2 derniers paramètres correspondent respectivement au `rowSpan` et au `columnSpan`. Le `rowSpan` est ici de 1, c'est la valeur par défaut on ne change donc rien, mais le `columnSpan` est de 2.

Le bouton va donc "occuper" 2 colonnes :



Essayez en revanche de monter le `columnSpan` à 3 : vous ne verrez aucun changement.

En effet, il aurait fallu qu'il y ait un troisième widget sur la première ligne pour que le `columnSpan` puisse fonctionner.

Faites des tests avec le spanning pour vous assurer que vous avez bien compris comment ça marche 😊

Le layout de formulaire

Le layout de formulaire [QFormLayout](#) est un layout assez récent spécialement fait pour les fenêtres qui contiennent des formulaires.

Un formulaire est en général une suite de libellés ("Votre prénom :") associés à des champs de formulaire (zone de texte par exemple) :

Votre prénom :	<input type="text" value="Anna"/>
Votre nom :	<input type="text" value="Conda"/>
Votre âge :	<input type="text" value="26"/>

Normalement, pour écrire du texte dans la fenêtre, on utilise le widget QLabel (libellé), dont on parlera plus en détail dans le prochain chapitre.

L'avantage du layout que nous allons utiliser, c'est qu'il simplifie notre travail en créant automatiquement des QLabel pour nous.

Vous noterez d'ailleurs que la disposition correspond à celle d'un QGridLayout à 2 colonnes et plusieurs lignes. En effet, le QFormLayout n'est en fait rien d'autre qu'une version spéciale du QGridLayout pour les formulaires, avec quelques particularités : il s'adapte en fonction des habitudes des OS, pour certains les libellés sont alignés à gauche, pour d'autres ils sont alignés à droite, etc.

L'utilisation d'un QFormLayout est très simple. La différence, c'est qu'au lieu d'utiliser une méthode addWidget, nous allons utiliser une méthode addRow qui prend 2 paramètres :

- Le texte du libellé
- Un pointeur vers le champ du formulaire

Pour faire simple, nous allons créer 3 champs de formulaire de type "Zone de texte à une ligne" (QLineEdit), puis nous allons les placer dans un QFormLayout au moyen de la méthode addRow :

Code : C++ - Sélectionner

```
#include <QApplication>
#include <QLineEdit>
#include <QFormLayout>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QWidget fenetre;

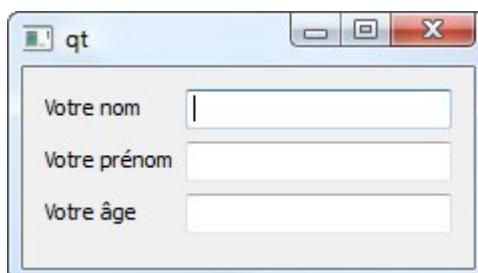
    QLineEdit *nom = new QLineEdit;
    QLineEdit *prenom = new QLineEdit;
    QLineEdit *age = new QLineEdit;

    QFormLayout *layout = new QFormLayout;
    layout->addRow("Votre nom", nom);
    layout->addRow("Votre prénom", prenom);
    layout->addRow("Votre âge", age);

    fenetre.setLayout(layout);
    fenetre.show();

    return app.exec();
}
```

Résultat :



Sympa, non ? 😊

On peut aussi définir des raccourcis clavier pour accéder rapidement aux champs du formulaire. Pour ce faire, placez un symbole "&" devant la lettre du libellé que vous voulez transformer en raccourci.

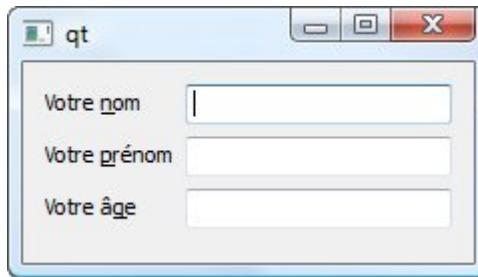
Explication en image (ehu, en code) :

Code : C++ - [Sélectionner](#)

```
layout->addRow( "Votre &nom" , nom );
layout->addRow( "Votre &prénom" , prenom );
layout->addRow( "Votre â&ge" , age );
```

La lettre "p" est désormais un raccourci vers le champ du prénom.
"n" pour le champ nom.
"g" pour le champ âge.

L'utilisation du raccourci dépend de votre système d'exploitation. Sous Windows, il faut faire Alt puis la touche raccourci. Lorsque vous appuyez sur Alt, les lettres raccourcis apparaissent soulignées :



Faites Alt + N pour accéder directement au champ du nom ! 😊

Souvenez-vous de ce symbole &, il est très souvent utilisé en GUI Design (design de fenêtre) pour indiquer quelle lettre sert de raccourci. On le réutilisera notamment pour avoir des raccourcis dans les menus de la fenêtre.

Ah, et si vous voulez par contre vraiment afficher un symbole & dans un libellé, tapez-en deux : "&&".
Exemple : "Bonnie && Clyde".

Combiner les layouts

Avant de terminer ce chapitre, il me semble important que nous jetions un oeil aux layouts combinés, une fonctionnalité qui va vous faire comprendre toute la puissance des layouts.

Commençons comme il se doit par une question que vous devriez vous poser :

Les layouts c'est bien joli, mais c'est pas un peu limité ? Si je veux faire une fenêtre un peu complexe, ce n'est pas à grands coups de QVBoxLayout ou même de QGridLayout que je vais m'en sortir !

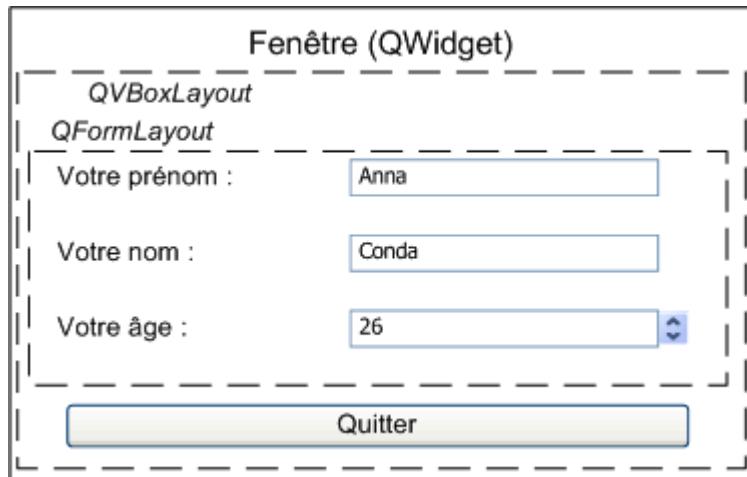
C'est vrai que mettre ses widgets les uns en-dessous des autres peut sembler limité. Même la grille fait un peu "rigide", je reconnais. Mais assurez-vous, tout a été pensé. La magie apparaît lorsque nous commençons à combiner les layouts, c'est-à-dire à placer un layout dans un autre layout.

Un cas concret

Prenons par exemple notre joli formulaire. Supposons que l'on veuille ajouter un bouton "Quitter". Si vous voulez placer ce bouton en bas du formulaire, comment faire ?

Il va falloir d'abord créer un layout vertical (QVBoxLayout), et placer à l'intérieur notre layout de formulaire puis notre bouton "Quitter".

Cela donne le schéma suivant :



On voit que notre QVBoxLayout contient 2 choses, dans l'ordre :

1. Un QFormLayout (qui contient lui-même d'autres widgets)
2. Un QPushButton

Un layout peut donc contenir aussi bien des layouts que des widgets.

Utilisation de addLayout

Pour insérer un layout dans un autre, on utilise `addLayout` au lieu de `addWidget` (c'est logique me direz-vous 😊).

Voici un bon petit code pour se faire la main :

Code : C++ - [Sélectionner](#)

```
#include <QApplication>
#include <QLineEdit>
#include <QPushButton>
#include <QVBoxLayout>
#include <QFormLayout>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QWidget fenetre;

    // Création du layout de formulaire et de ses widgets

    QLineEdit *nom = new QLineEdit;
    QLineEdit *prenom = new QLineEdit;
    QLineEdit *age = new QLineEdit;

    QFormLayout *formLayout = new QFormLayout;
    formLayout->addRow( "Votre &nom", nom);
    formLayout->addRow( "Votre &prenom", prenom);
    formLayout->addRow( "Votre âge", age);

    // Création du layout principal de la fenêtre (vertical)

    QVBoxLayout *layoutPrincipal = new QVBoxLayout;
    layoutPrincipal->addLayout(formLayout); // Ajout du layout de formulaire

    QPushButton *boutonQuitter = new QPushButton("Quitter");
    QWidget::connect(boutonQuitter, SIGNAL(clicked()), &app, SLOT(quit()));
    layoutPrincipal->addWidget(boutonQuitter); // Ajout du bouton

    fenetre.setLayout(layoutPrincipal);

    fenetre.show();

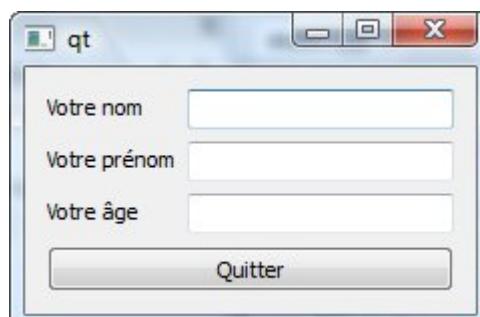
    return app.exec();
}
```

J'ai surligné les ajouts au layout vertical principal :

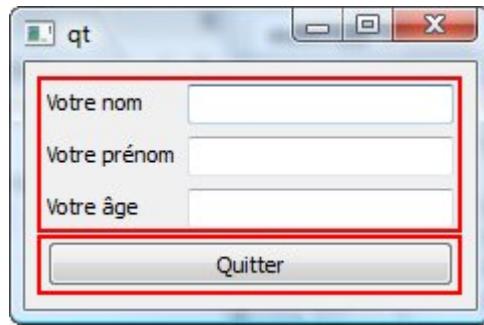
- L'ajout du sous-layout de formulaire (addLayout)
 - L'ajout du bouton (addWidget)

Vous remarquerez que je fais les choses un peu dans l'ordre inverse : d'abord je crée les widgets et layouts "enfants" (le QFormLayout), et ensuite je crée le layout principal (le QVBoxLayout) et j'y ajoute le layout enfant que j'ai créé.

Au final, la fenêtre qui apparaît est la suivante :

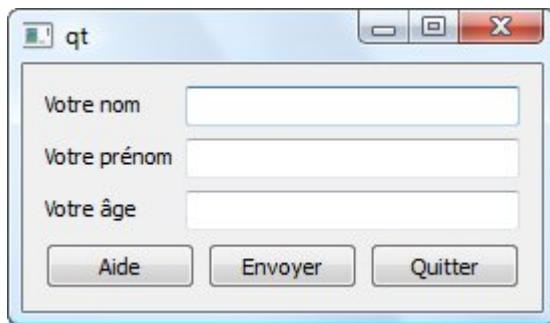


On ne le voit pas, mais la fenêtre contient d'abord un QVBoxLayout, qui contient lui-même un layout de formulaire et un bouton :



Exercice

Essayez d'obtenir le rendu suivant :



Si vous voulez mettre plusieurs boutons en bas sur la même ligne, vous pouvez créer un QHBoxLayout et ajouter ce QHBoxLayout au QVBoxLayout !

Vous pouvez aussi utiliser plus simplement un QGridLayout en utilisant un columnSpan. En effet, un QGridLayout n'est rien d'autre qu'un assemblage de QVBoxLayout et de QHBoxLayout.

Plusieurs méthodes sont donc possibles, libre à vous d'utiliser un QGridLayout ou des QVBoxLayout et QHBoxLayout.

Ce ne devrait pas être un exercice difficile si vous avez bien suivi ce chapitre. Ce sera en tout cas l'occasion de vous assurer que vous avez bien compris 😊

Dans mon exemple, les boutons "Aide" et "Envoyer" ne font rien (je n'ai pas géré de signaux et de slots pour eux). Le résultat que vous devez obtenir est juste visuel, n'essayez pas de tenter d'envoyer le formulaire sur internet et de le stocker dans une base de données, il est un peu trop tôt encore 🍻

Les layouts sont la base du positionnement de widgets en GUI Design. Ils nous donnent un maximum de flexibilité pour que nos fenêtres s'adaptent à toutes les conditions.

Bien entendu, je vous mentirais si je vous disais qu'absolument tout le monde les utilise. Pour certains logiciels simples, il n'est parfois pas nécessaire de recourir aux layouts. Il est néanmoins recommandé de s'en servir autant que possible.

Nous n'avons pas pu absolument tout voir à propos des layouts. La différence, c'est que maintenant je vous ai appris à vous servir de la doc et vous pouvez aller compléter ce que vous savez si besoin est.

Je vous recommande de lire leur page d'[explication générale sur les layouts](#) puis de regarder les différentes classes de layouts.

N'oubliez pas de consulter les classes parentes à chaque fois, ce sont souvent elles qui contiennent les méthodes et attributs qui semblent manquer.

Jetez un œil aux "stretch factors", qui permettent de définir des tailles proportionnelles pour les widgets, ainsi qu'à l'alignement des widgets.

Dans le prochain chapitre, nous passerons en revue la plupart des widgets courants et simples. En effet, cela fait un moment que je

vous fais utiliser pour le besoin du cours quelques widgets comme les boutons et les champs de texte, mais il est maintenant temps de faire un tour d'horizon plus général pour que vous sachiez quels sont les principaux widgets qui peuvent peupler une fenêtre.

Les principaux widgets

Voilà un moment que nous avons commencé à nous intéresser à Qt, je vous parle en long en large et en travers de widgets, mais jusqu'ici nous n'avions toujours pas pris le temps de faire un tour d'horizon rapide de ce qui existait.

C'était voulu. Je voulais dans un premier temps vous faire manipuler un ou deux widgets simples pour vous faire comprendre les concepts de base comme :

- La création de la fenêtre
- Les signaux et les slots
- Les layouts

Il est maintenant temps de faire une "pause" et de regarder ce qui existe comme widgets. Nous étudierons cependant seulement les principaux widgets ici.

Pourquoi ne les verrons-nous pas tous ? Parce qu'il existe un grand nombre de widgets et que certains sont rarement utilisés. D'autres sont parfois tellement complexes qu'ils nécessiteront un chapitre entier pour les étudier.

Néanmoins, avec ce que vous allez voir, vous aurez largement de quoi faire pour être capables de créer la quasi-totalité des fenêtres que vous voulez ! 😊

Pour information, je me base sur la page "[liste des widgets](#)" (ici avec l'apparence de vista, mais peu importe l'apparence, ça sera adapté à votre OS).

Je ne compte pas remplacer la doc. Je vous inviterai donc à consulter la doc à chaque fois pour en savoir plus. Mon rôle sera surtout de vous introduire à utiliser de manière basique ces widgets. Je vous fais confiance, je sais que vous saurez en faire une utilisation plus avancée si besoin est. 😊

Les fenêtres

Avec Qt, tout élément de la fenêtre est appelé un widget. La fenêtre elle-même est considérée comme un widget.

Dans le code, les widgets sont des classes qui héritent toujours de [QWidget](#) (directement ou indirectement). C'est donc une classe de base très importante, et vous aurez probablement très souvent besoin de lire la doc de cette classe.

Quelques rappels sur l'ouverture d'une fenêtre

Cela fait plusieurs chapitres que l'on crée une fenêtre dans nos programmes à l'aide d'un objet de type QWidget. Cela signifie-t-il que QWidget = Fenêtre ?

Non. En fait, un widget qui n'est contenu dans aucun autre widget est considéré comme une fenêtre. Donc quand on fait juste ce code très simple :

Code : C++ - [Sélectionner](#)

```

#include <QApplication>
#include <QWidget>

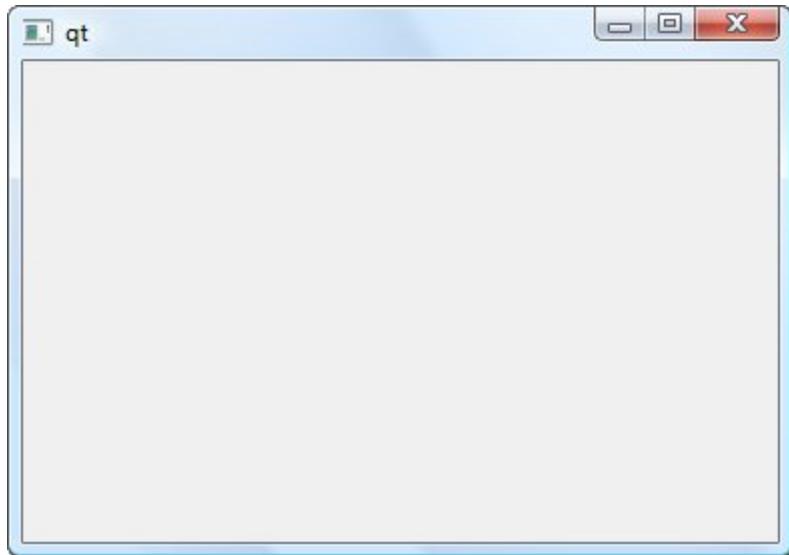
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QWidget fenetre;
    fenetre.show();

    return app.exec();
}

```

... cela affiche une fenêtre (vide) :



C'est comme cela que Qt fonctionne. C'est un peu déroutant au début, mais après on apprécie au contraire que ça ait été pensé comme ça.

Donc si je comprends bien, il n'y a pas de classe QFenetre ou quelque chose du genre ?

Tout à fait, il n'y a pas de classe du genre "QFenetre" car n'importe quel widget peut servir de fenêtre. Si vous vous souvenez bien, on avait créé un bouton dans les premiers exemples du cours sur Qt. On avait demandé à afficher ce bouton. Comme le bouton n'avait pas de parent, une fenêtre avait été ouverte :

Code : C++ - [Sélectionner](#)

```

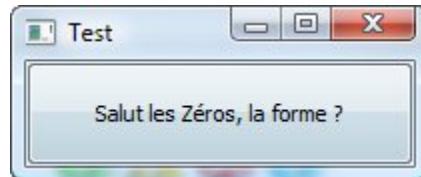
#include <QApplication>
#include <QPushButton>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QPushButton bouton("Salut les Zéros, la forme ?");
    bouton.show();

    return app.exec();
}

```



Dans la pratique, on ne crée pas de fenêtre-bouton comme là. On crée d'abord une fenêtre, et on place ensuite des widgets à l'intérieur (ces widgets étant parfois organisés grâce aux layouts comme on l'a vu).

Code : C++ - Sélectionner

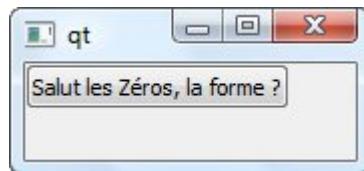
```
#include <QApplication>
#include <QWidget>
#include <QPushButton>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QWidget fenetre;
    QPushButton bouton("Salut les Zéros, la forme ?", &fenetre);
    fenetre.show();

    return app.exec();
}
```

Note : je n'ai pas utilisé de layouts dans ce code pour le rendre court et simple. Le bouton est donc positionné de manière absolue au coin en haut à gauche, de coordonnées (0, 0).



Le QPushButton a donc pour parent un QWidget, car on a indiqué en second paramètre de son constructeur un pointeur vers le QWidget : `&fenetre`.

Le QWidget n'a pas de parent car on n'a pas envoyé de pointeur vers un autre widget dans son constructeur, donc c'est une fenêtre.

Ne confondez pas :

- En termes C++ : une classe parente est une classe mère (quand on fait un héritage).
- En termes Qt : un widget parent est un widget qui en contient d'autres. Un widget fils est un widget qui n'en contient aucun autre.

Ici, je suis en train de parler de widgets parents en termes Qt.

Quelques classes particulières pour les fenêtres

Résumons ce que je viens de dire : tout widget peut servir de fenêtre.

C'est le widget qui n'a pas de parent qui sera considéré comme étant la fenêtre.

A ce titre, un QPushButton ou un QLineEdit peuvent être considérés comme des fenêtres s'ils n'ont pas de widget parent.

Toutefois, il y a 2 classes de widgets que j'aimerais mettre en valeur :

- QMainWindow : c'est un widget spécial qui permet de créer la fenêtre principale de l'application. Une fenêtre principale peut contenir des menus, une barre d'outils, une barre d'état, etc.
- QDialog : c'est une classe de base utilisée par toutes les classes de boîtes de dialogue qu'on a vues il y a quelques chapitres. On peut aussi s'en servir directement pour ouvrir des boîtes de dialogue personnalisées.

La fenêtre principale QMainWindow mérite un chapitre entier à elle toute seule. Et elle en aura un. Nous pourrons alors tranquillement passer en revue la gestion des menus, de la barre d'outils et de la barre d'état.

La fenêtre QDialog peut être utilisée pour ouvrir une boîte de dialogue personnalisée générique. Une boîte de dialogue est une fenêtre généralement de petite taille dans laquelle il y a peu d'informations.

La classe QDialog hérite de QWidget comme tout widget qui se respecte, et elle y est même très similaire. Elle y ajoute peu de choses, parmi lesquelles la gestion des fenêtres modales (une fenêtre par-dessus toutes les autres qui doit être remplie avant de pouvoir accéder aux autres fenêtres de l'application).

Nous allons ici étudier ce que l'on peut faire d'intéressant avec la classe de base QWidget qui permet déjà de réaliser la plupart des fenêtres que l'on veut.

Nous verrons ensuite ce qu'on peut faire avec les fenêtres de type QDialog. Quant à QMainWindow, ce sera pour un autre chapitre comme je vous l'ai dit. 😊

Une fenêtre avec QWidget

Pour commencer, je vous invite à ouvrir la [doc de QWidget](#) en même temps que vous lisez ce chapitre.

Vous remarquerez que QWidget est la classe mère d'un grand nombre d'autres classes. 😊

Les QWidget disposent de beaucoup de propriétés et de méthodes. Donc tous les widgets disposent de ces propriétés et méthodes.

On peut découper les propriétés en 2 catégories :

- Celles qui valent pour tous les types de widgets et pour les fenêtres
- Celles qui n'ont de sens que pour les fenêtres

Jetons un œil à celles qui me semblent les plus intéressantes. Pour avoir la liste complète, il faudra recourir à la doc, je ne compte pas tout répéter ici ! 😊

Les propriétés utilisables pour tous les types de widgets, y compris les fenêtres

Je vous fais une liste rapide pour extraire quelques propriétés qui pourraient vous intéresser. Pour savoir comment vous servir de toutes ces propriétés, lisez le prototype que vous donne la doc.

N'oubliez pas qu'on peut modifier une propriété en appelant une méthode du même nom commençant par "set". Par exemple, si la propriété est cursor, la méthode sera setCursor().

- cursor : curseur de la souris à afficher lors du survol du widget. La méthode setCursor attend que vous lui envoyiez un objet de type QCursor. Certains curseurs classiques (comme le sablier) sont prédefinis dans une énumération. La doc vous fait un lien vers cette énumération.
- enabled : indique si le widget est activé, si on peut le modifier. Un widget désactivé est généralement grisé. Si vous appliquez setEnabled(false) à toute la fenêtre, c'est toute la fenêtre qui deviendra inutilisable.
- height : hauteur du widget.
- size : dimensions du widget. Vous devrez indiquer la largeur et la hauteur.
- visible : contrôle la visibilité du widget.
- width : largeur.

N'oubliez pas : pour modifier une de ces propriétés, préfixez la méthode par un "set". Exemple :

Code : C++ - Sélectionner

```
maFenetre.setWidth(200);
```

Ces propriétés sont donc valables pour tous les widgets, y compris les fenêtres. Si vous appliquez un `setWidth` sur un bouton, ça modifiera la largeur du bouton. Si vous appliquez cela sur une fenêtre, c'est la largeur de la fenêtre qui sera modifiée.

Les propriétés utilisables uniquement sur les fenêtres

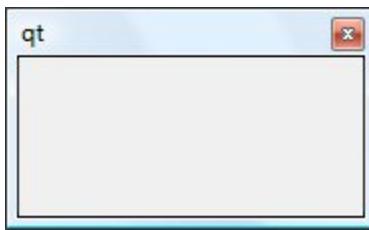
Ces propriétés sont faciles à reconnaître, elles commencent toutes par "window". Elles n'ont de sens que si elles sont appliquées aux fenêtres.

- `windowFlags` : une série d'options contrôlant le comportement de la fenêtre. Il faut consulter l'énumération [Qt::WindowType](#) pour savoir les différents types disponibles. Vous pouvez aussi consulter l'exemple Window Flags du programme "Qt Examples and Demos".

Par exemple pour afficher une fenêtre de type "Outil" avec une petite croix et pas de possibilité d'agrandissement ou de réduction :

Code : C++ - Sélectionner

```
fenetre.setWindowFlags(Qt::Tool);
```



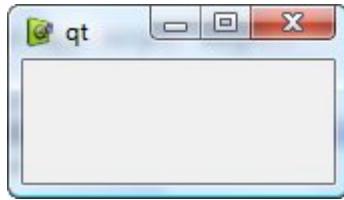
Une fenêtre de type "Tool"

C'est par là aussi qu'on passe pour que la fenêtre reste par-dessus toutes les autres fenêtres du système (avec le flag `Qt::WindowStaysOnTopHint`).

- `windowIcon` : l'icône de la fenêtre. Il faut envoyer un objet de type `QIcon`, qui lui-même accepte un nom de fichier à charger. Cela donne le code suivant pour charger le fichier [icone.png](#) situé dans le même dossier que l'application :

Code : C++ - Sélectionner

```
fenetre.setWindowIcon(QIcon("icone.png"));
```

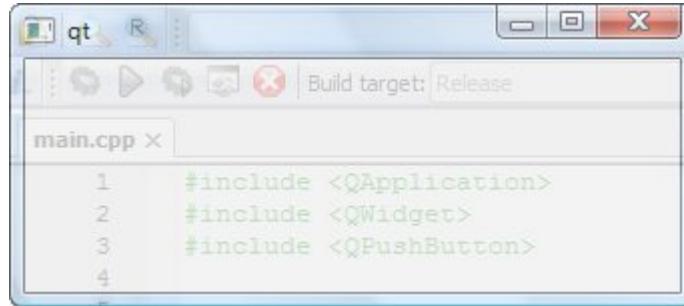


Une icône pour la fenêtre

- `windowOpacity` : contrôle la transparence de la fenêtre (ne fonctionne pas sur tous les OS). La valeur à envoyer est un nombre décimal compris entre 0 (transparent) et 1 (complètement opaque). Ici, avec la valeur 0.8 :

Code : C++ - Sélectionner

```
fenetre.setWindowOpacity(0.8);
```



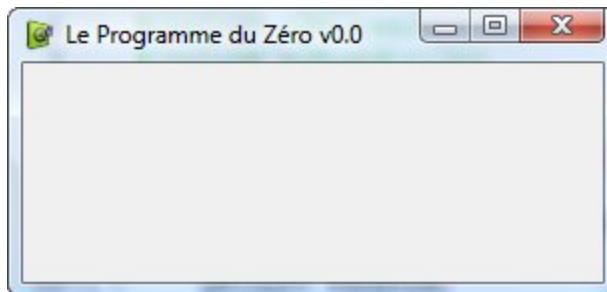
```
main.cpp x
1 #include <QApplication>
2 #include <QWidget>
3 #include <QPushButton>
4
```

Une fenêtre transparente

- `windowTitle` : le titre de la fenêtre, affiché en haut.

Code : C++ - [Sélectionner](#)

```
fenetre.setWindowTitle( "Le Programme du Zéro v0.0" );
```



Une fenêtre avec un titre

Une fenêtre avec QDialog

[QDialog](#) est un widget spécialement créé pour générer des fenêtres de type "boîte de dialogue".

Quelle est la différence avec une fenêtre créée à partir d'un `QWidget` ? 😊

En général les `QDialog` sont des petites fenêtres secondaires : des boîtes de dialogue.
Elles proposent le plus souvent un choix simple entre :

- Valider
- Annuler

Les `QDialog` sont rarement utilisées pour gérer la fenêtre principale. Pour la fenêtre principale on préfère utiliser `QWidget`, ou carrément `QMainWindow` si on a besoin de l'artillerie lourde.

Les `QDialog` peuvent être de 2 types :

- Modales : on ne peut pas accéder aux autres fenêtres de l'application lorsqu'elles sont ouvertes.
- Non modales : on peut toujours accéder aux autres fenêtres.

Par défaut, les `QDialog` sont modales.

Elles disposent en effet d'une méthode `exec()` qui ouvre la boîte de dialogue de manière modale. Il s'avère d'ailleurs qu'`exec()` est un slot (très pratique pour effectuer une connexion ça !).

Je vous propose d'essayer de pratiquer de la manière suivante : nous allons ouvrir une fenêtre principale QWidget qui contiendra un bouton. Lorsqu'on cliquera sur ce bouton, il ouvrira une fenêtre secondaire de type QDialog.

Notre objectif est d'ouvrir une fenêtre secondaire après un clic sur un bouton de la fenêtre principale. La fenêtre secondaire, de type QDialog, affichera juste une image pour cet exemple.

Code : C++ - Sélectionner

```
#include <QApplication>
#include <QtGui>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QWidget fenetre;
    QPushButton *bouton = new QPushButton("Ouvrir la fenêtre", &fenetre);

    QDialog secondeFenetre (&fenetre);
    QVBoxLayout *layout = new QVBoxLayout;
    QLabel *image = new QLabel(&secondeFenetre);
    image->setPixmap(QPixmap("icone.png"));
    layout->addWidget(image);
    secondeFenetre.setLayout(layout);

    QWidget::connect(bouton, SIGNAL(clicked()), &secondeFenetre, SLOT(exec()));
    fenetre.show();

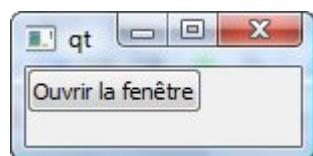
    return app.exec();
}
```

Mon code est indenté de manière bizarroïde je sais. Je trouve que c'est plus lisible : vous pouvez mieux voir comme cela à quelles fenêtres se rapportent les opérations que je fais.

Vous voyez ainsi immédiatement que dans la première fenêtre je n'ai fait que placer un bouton, tandis que dans la seconde j'ai mis un QLabel affichant une image que j'ai placée dans un QVBoxLayout.

D'autre part, j'ai tout fait dans le main pour cet exemple, mais dans la pratique, comme nous le verrons dans les TP, on a en général un fichier .cpp par fenêtre, c'est plus facile à gérer.

Au départ, la fenêtre principale s'affiche, comme ceci :



Si vous cliquez sur le bouton, la boîte de dialogue s'ouvre :



Comme elle est modale, vous remarquerez que vous ne pouvez pas accéder à la fenêtre principale tant qu'elle est ouverte.

Bon intéressons-nous au code. J'ai surligné les 2 lignes qui me paraissaient les plus pertinentes :

- Ligne 12 : la création de la QDialog. Rien de bien extraordinaire à première vue, mais si vous regardez bien vous devriez voir que j'ai mentionné dans le constructeur l'adresse de la fenêtre parente. Cela permet à la QDialog de savoir quelle est la fenêtre qui l'a appelée. Entre autres choses, la QDialog se placera automatiquement de manière centrée par rapport à sa fenêtre mère.
- Ligne 20 : je connecte le clic sur le bouton à la méthode exec() de la QDialog pour ouvrir la boîte de dialogue (de manière modale).

Cela devrait vous donner des bases suffisantes pour désormais savoir comment ouvrir des fenêtres secondaires de type QDialog. Nous n'avons cependant pas tout vu sur cette classe : on peut rendre les QDialog non modales ou encore utiliser les slots accept() et reject() pour les connecter respectivement à des boutons "OK" et "Annuler" et ainsi informer la fenêtre parente afin qu'elle sache si l'opération a été validée ou refusée par l'utilisateur.

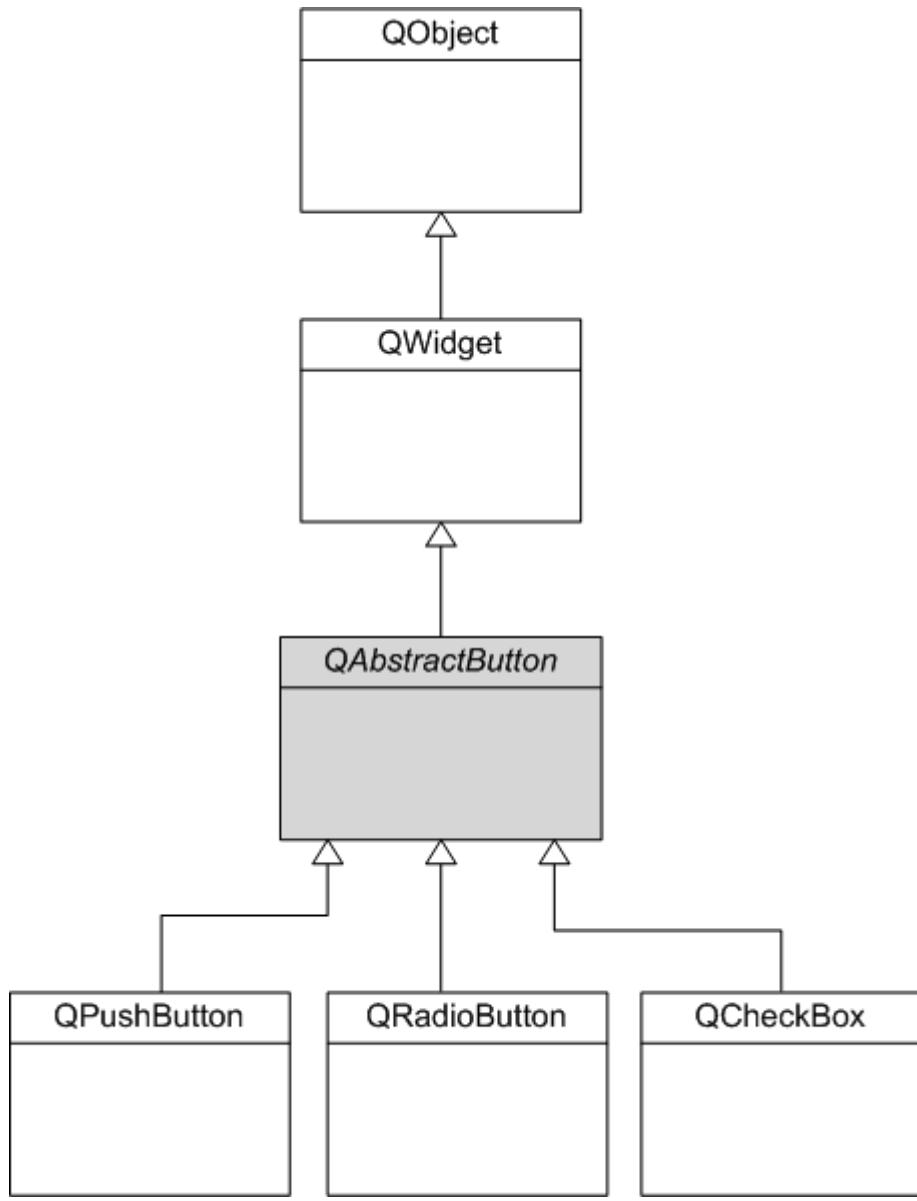
Pour savoir faire tout cela, vous savez ce qu'il vous reste à faire. Tout est dans la doc. 😊

Les boutons

Nous allons maintenant étudier la catégorie des widgets "boutons". Nous allons passer en revue :

- QPushButton : un bouton classique, que vous avez déjà largement eu l'occasion de manipuler.
- QRadioButton : un bouton "radio", pour un choix à faire parmi une liste.
- QCheckBox : une case à cocher (on considère que c'est un bouton en GUI Design).

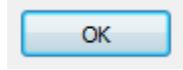
Tous ces widgets héritent de [QAbstractButton](#) qui lui-même hérite de QWidget, qui finalement hérite de QObject :



Comme l'indique son nom, `QAbstractButton` est une classe abstraite. Si vous vous souvenez des épisodes précédents de notre passionnant feuilleton, une classe abstraite est une classe... qu'on ne peut pas instancier, bravo ! 😊
 On ne peut donc pas créer d'objets de type `QAbstractButton`, il faut forcément utiliser une des classes filles. `QAbstractButton` sert donc juste de modèle de base pour ses classes filles.

QPushButton : un bouton

Le [QPushButton](#) est l'élément le plus classique et le plus commun des fenêtres :



Je ne vous fais pas l'offense de vous expliquer à quoi sert un bouton 😊

Commençons par un rappel important, indiqué par la doc : `QPushButton` hérite de [QAbstractButton](#). Et c'est vraiment une info importante, car vous serez peut-être surpris de voir que `QPushButton` contient peu de méthodes qui lui sont propres. C'est normal, une grande partie d'entre elles se trouvent dans sa classe parente `QAbstractButton`.

Il faudra donc absolument consulter aussi `QAbstractButton`, et même sa classe mère [QWidget](#) (et éventuellement aussi `QObject` mais c'est plus rare), si vous voulez connaître toutes les possibilités offertes au final par un `QPushButton`. Par exemple, [setEnabled](#) permet d'activer / désactiver le bouton, et cette propriété se trouve dans `QWidget`.

Les signaux du bouton

Un bouton émet un signal `clicked()` quand on l'active. C'est le signal le plus communément utilisé.

On note aussi les signaux `pressed()` (bouton enfoncé) et `released()` (bouton relâché), mais ils sont plus rares.

Les boutons à 2 états

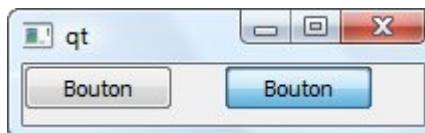
Un bouton peut parfois avoir 2 états : enfoncé et relâché (normal).

Pour activer un bouton à 2 états, utilisez `setCheckable(true)` :

Code : C++ - [Sélectionner](#)

```
QPushButton *bouton = new QPushButton("Bouton", &fenetre);
bouton->setCheckable(true);
```

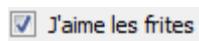
Désormais, un clic sur le bouton le laissera enfoncé, et un nouveau clic le rétablira dans son état normal. Utilisez les signaux `pressed()` et `released()` pour récupérer les changements d'état du bouton.



A gauche un bouton normal, à droite un bouton pressé

QCheckBox : une case à cocher

Une case à cocher [QCheckBox](#) est généralement associée à un texte de libellé comme ceci :



On définit le libellé de la case lors de l'appel du constructeur :

Code : C++ - [Sélectionner](#)

```
#include < QApplication>
#include < QWidget>
#include < QCheckBox>

int main( int argc, char *argv[] )
{
    QApplication app(argc, argv);

    QWidget fenetre;
    QCheckBox *checkbox = new QCheckBox("J'aime les frites", &fenetre);
    fenetre.show();

    return app.exec();
}
```

La case à cocher émet le signal `stateChanged(bool)` lorsqu'on modifie son état. Le booléen en paramètre nous permet de savoir si la

case est maintenant cochée ou décochée.

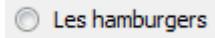
Si vous voulez vérifier à un autre moment si la case est cochée, appelez isChecked() qui renvoie un booléen.

On peut aussi faire des cases à cocher à 3 états (le troisième état étant l'état grisé). Renseignez-vous sur la propriété tristate pour savoir faire cela. Notez que ce type de case à cocher est relativement rare.

Enfin, sachez que si vous avez plusieurs cases à cocher, vous pouvez les regrouper au sein d'une [QGroupBox](#).

QRadioButton : les boutons radio

C'est une case à cocher particulière : une seule case peut être cochée à la fois parmi une liste.



Les radio buttons qui ont le même widget parent sont mutuellement exclusifs. Si vous en cochez un, les autres seront automatiquement décochés.

En général, on place les radio buttons dans une QGroupBox. Utiliser des QGroupBox différentes vous permet de séparer les groupes de radio buttons.

Voici un exemple d'utilisation d'une QGroupBox (qui contient un layout, qui contient les QRadioButton) :

Code : C++ - [Sélectionner](#)

```
#include < QApplication>
#include < QtGui>

int main( int argc, char *argv[] )
{
    QApplication app( argc, argv );

    QWidget fenetre;
    QGroupBox *groupbox = new QGroupBox( "Votre plat préféré", &fenetre );

    QRadioButton *steacks = new QRadioButton( "Les steacks" );
    QRadioButton *hamburgers = new QRadioButton( "Les hamburgers" );
    QRadioButton *nuggets = new QRadioButton( "Les nuggets" );

    steacks->setChecked( true );

    QVBoxLayout *vbox = new QVBoxLayout;
    vbox->addWidget( steacks );
    vbox->addWidget( hamburgers );
    vbox->addWidget( nuggets );

    groupbox->setLayout( vbox );
    groupbox->move( 5, 5 );

    fenetre.show();

    return app.exec();
}
```

J'en profite pour signaler que vous pouvez inclure QtGui pour automatiquement inclure tous les widgets : `#include < QtGui>`. C'est un peu bourrin mais ça marche. 😊

Cela vous évite d'avoir à rajouter un nouveau widget à la liste des includes à chaque fois. Attention par contre, la compilation sera un peu plus longue.

Les radio buttons sont placés dans un layout qui est lui-même placé dans la groupbox, qui est elle-même placée dans la fenêtre.

Pfiou ! Le concept des widgets conteneurs est ici utilisé à fond !
Et encore, je n'ai pas fait de layout pour la fenêtre (la flème, et je ne voulais pas trop encombrer le code), ce qui fait que la taille initiale de la fenêtre est un peu petite, mais ce n'est pas grave c'est pour l'exemple.

Voilà le résultat :



Nota : j'ai une nourriture plus équilibrée que ne le laisse suggérer cette dernière capture d'écran quand même, je vous rassure. 🍔

Les afficheurs

Parmi les widgets afficheurs, on compte principalement :

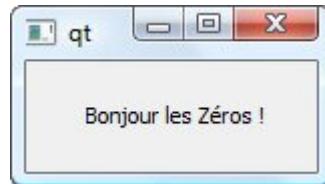
- QLabel : le plus important, un widget permettant d'afficher du texte ou une image.
- QProgressBar : une barre de progression.

Etudions-les en choeur, sans heurts, dans la joie et la bonne humeur. 🎉

QLabel : afficher du texte ou une image

C'est vraiment LE widget de base pour afficher du texte à l'intérieur de la fenêtre. 😊
Nous l'avons déjà utilisé indirectement auparavant, via les cases à cocher ou encore les layouts de formulaire.

Voici un libellé :



... du moins, UN des types de libellés possibles comme nous allons le voir.

QLabel hérite de QFrame, qui est un widget de base permettant d'afficher des bordures. Renseignez-vous auprès de QFrame pour savoir gérer les différents types de bordure.
Par défaut, un QLabel n'a pas de bordure.

Un QLabel peut afficher plusieurs types d'éléments :

- Du texte simple,
- Du texte enrichi (gras, italique, souligné, coloré, avec des liens...),
- Une image,
- Et même une image animée !

Nous allons étudier chacun de ces types de contenu, à l'exception de l'image animée qui sort un peu du cadre du chapitre. Et puis de toute façon, ça ne sert qu'à afficher en pratique des GIF animés, ça nous sera donc peu utile.

Afficher un texte simple

Rien de plus simple, on utilise setText() :

Code : C++ - [Sélectionner](#)

```
label->setText( "Bonjour les Zéros !" );
```

Mais on peut aussi afficher un texte simple dès l'appel au constructeur comme ceci :

Code : C++ - [Sélectionner](#)

```
#include < QApplication>
#include < QtGui>

int main( int argc, char *argv[] )
{
    QApplication app( argc, argv );

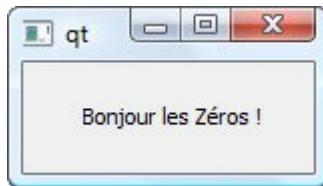
    QWidget fenetre;

    QLabel *label = new QLabel( "Bonjour les Zéros !", &fenetre );
    label->move( 30, 20 );

    fenetre.show();

    return app.exec();
}
```

Le résultat est le même que la capture d'écran que je vous ai montrée plus haut :



Vous pouvez jeter aussi un oeil à la propriété alignment qui permet de définir l'alignement du texte dans le libellé.

Afficher un texte enrichi

Vous pouvez envoyer du texte enrichi (formaté) au QLabel, avec du HTML :

Code : C++ - [Sélectionner](#)

```

#include <QApplication>
#include <QtGui>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QWidget fenetre;

    QLabel *label = new QLabel("Bonjour les Zéros !<br />Etes-vous allés sur le Site du Zéro aujourd'hui ?");
    label->move(30, 20);

    fenetre.show();

    return app.exec();
}

```

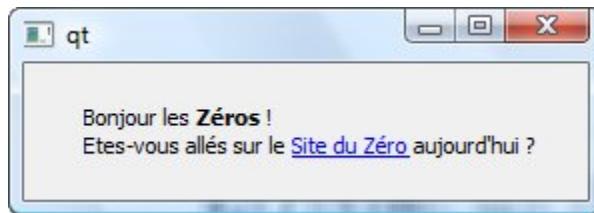
3

4

5

6

Magie, magie, le texte est correctement mis en forme !



C'est beau la technologie quand même. 😊

Et encore, vous n'avez pas tout vu !

Afficher une image

Vous pouvez demander à ce que le QLabel affiche une image.

Comme il n'y a pas de constructeur qui accepte une image en paramètre, on va appeler le constructeur qui prend juste un pointeur vers la fenêtre parente.

Nous demanderons ensuite à ce que le libellé affiche une image à l'aide de setPixmap().

Cette méthode attend un objet de type [QPixmap](#). Après lecture de la doc sur QPixmap, il s'avère que cette classe a un constructeur qui accepte le nom du fichier à charger sous forme de chaîne de caractères.

Nous allons donc afficher notre belle icône de tout à l'heure, mais cette fois en grand et dans la fenêtre :

Code : C++ - [Sélectionner](#)

```

#include <QApplication>
#include <QtGui>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QWidget fenetre;

    QLabel *label = new QLabel(&fenetre);
    label->setPixmap(QPixmap("icone.png"));
    label->move(30, 20);

    fenetre.show();

    return app.exec();
}

```

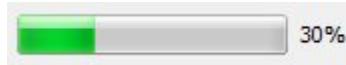
L'icône doit se trouver dans le même dossier que l'exécutable pour que cela fonctionne.

Et voilà le résultat !



QProgressBar : une barre de progression

Les barres de progression sont gérées par [QProgressBar](#). Cela permet d'indiquer à l'utilisateur l'avancement des opérations.



Voici quelques propriétés utiles de la barre de progression :

- maximum : la valeur maximale que peut prendre la barre de progression.
- minimum : la valeur minimale que peut prendre la barre de progression.
- value : la valeur actuelle de la barre de progression.

On utilisera donc setValue pour changer la valeur de la barre de progression. Par défaut les valeurs sont comprises entre 0 et 100%.

Qt ne peut pas deviner où en sont vos opérations. C'est à vous de calculer le pourcentage d'avancement de vos opérations. La QProgressBar se contente juste d'afficher le résultat.

Une QProgressBar envoie un signal valueChanged() lorsque sa valeur a été modifiée.

A part ça, rien de bien spécial à signaler. Je vous avais déjà fait manipuler les barres de progression dans le chapitre sur les signaux et les slots pour tester la connexion entre les widgets.

Les champs

Nous allons maintenant faire le tour des widgets qui permettent d'entrer des données. C'est la catégorie de widgets la plus importante.

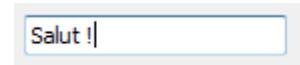
Encore une fois, on ne verra pas tout, mais les principaux d'entre eux :

- QLineEdit : champ de texte à une seule ligne.
- QTextEdit : champ de texte à plusieurs lignes pouvant afficher du texte mis en forme.
- QSpinBox : champ de texte adapté à la saisie de nombres entiers.
- QDoubleSpinBox : champ de texte adapté à la saisie de nombres décimaux.
- QSlider : un curseur qui permet de sélectionner une valeur.
- QComboBox : une liste déroulante.

QLineEdit : champ de texte à une seule ligne

Nous avons utilisé ce widget comme classe d'exemple lors du chapitre sur la lecture de la doc de Qt, vous vous souvenez ?

Un [QLineEdit](#) est un champ de texte sur une seule ligne :



Son utilisation est dans la plupart des cas assez simple. Voici quelques propriétés à connaître :

- `text` : permet de récupérer / modifier le texte contenu dans le champ.
- `alignment` : l'alignement du texte à l'intérieur.
- `echoMode` : type d'affichage du texte. Il faudra utiliser l'énumération `EchoMode` pour indiquer le type d'affichage. Par défaut les lettres entrées s'affichent, mais on peut aussi faire en sorte que les lettres soient masquées pour les mots de passe.

Code : C++ - [Sélectionner](#)

```
lineEdit->setEchoMode(QLineEdit::Password);
```



- `inputMask` : permet de définir un masque de saisie, pour obliger l'utilisateur à rentrer une chaîne précise (par exemple un numéro de téléphone ne doit pas contenir de lettres). Vous pouvez aussi jeter un oeil aux validators qui sont un autre moyen de valider la saisie de l'utilisateur.
- `maxLength` : le nombre de caractères maximum qui peuvent être entrés.
- `readOnly` : le contenu du champ de texte ne peut être modifié. Cette propriété ressemble à `enabled` (définie dans `QWidget`), mais avec `readOnly` on peut quand même copier-coller le contenu du `QLineEdit`, tandis qu'avec `enabled` le champ est complètement grisé et on ne peut pas récupérer son contenu.

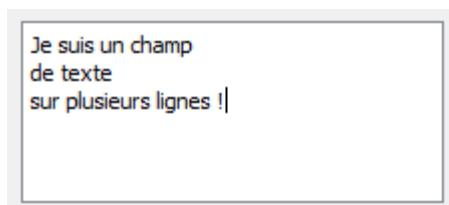
On note aussi plusieurs slots qui permettent de couper / copier / coller / vider / annuler le champ de texte.

Enfin, certains signaux comme `returnPressed()` (l'utilisateur a appuyé sur Entrée) ou `textChanged()` (l'utilisateur a modifié le texte) peuvent être utiles dans certains cas.

QTextEdit : champ de texte à plusieurs lignes

Ce type de champ est similaire à celui qu'on vient de voir, à l'exception du fait qu'il gère l'édition sur plusieurs lignes et, en particulier, qu'il autorise l'affichage de texte enrichi (HTML).

Voici un [QTextEdit](#) :



Il y a un certains nombre de choses que l'on pourrait voir sur les QTextEdit mais ce serait un peu trop long pour ce chapitre qui est plutôt là pour faire une revue rapide des widgets.

Notez les propriétés plainText et html qui permettent respectivement de récupérer & modifier le contenu sous forme de texte simple et sous forme de texte enrichi en HTML. Tout dépend de l'utilisation que vous en faites, normalement dans la plupart des cas vous utiliserez plutôt plainText.

Si vous vous intéressez à l'utilisation du HTML avec Qt, je vous invite à consulter la [liste des balises HTML et propriétés CSS supportées](#). Vous remarquerez qu'un grand nombre d'éléments sont supportés.

QSpinBox : champ de texte de saisie d'entiers

Une [QSpinBox](#) est un champ de texte (type QLineEdit) qui permet d'entrer uniquement un nombre entier et qui dispose de petits boutons pour augmenter ou diminuer la valeur :



QSpinBox hérite de [QAbstractSpinBox](#) (tiens, encore une classe abstraite !). Vérifiez donc aussi la doc de QAbstractSpinBox pour connaître toutes les propriétés de la spinbox.

Voici quelques propriétés intéressantes :

- accelerated : permet d'autoriser la spinbox à accélérer la vitesse d'augmentation du nombre si on appuie longtemps sur le bouton.
- minimum : valeur minimale que peut prendre la spinbox.
- maximum : valeur maximale que peut prendre la spinbox.
- singleStep : pas d'incrémentation (par défaut de 1). Si vous voulez que les boutons fassent augmenter la spinbox de 100 en 100, c'est cette propriété qu'il faut modifier !
- value : valeur contenue dans la spinbox.
- prefix : texte à afficher avant le nombre.
- suffix : texte à afficher après le nombre.

QDoubleSpinBox : champ de texte de saisie de nombres décimaux

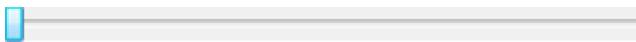
Le [QDoubleSpinBox](#) est très similaire au QSpinBox, à la différence près qu'il travaille sur des nombres décimaux (des double) :



On retrouve la plupart des propriétés de QSpinBox. On peut rajouter la propriété decimals qui gère le nombre de chiffres après la virgule affichés par le QDoubleSpinBox.

QSlider : un curseur pour sélectionner une valeur

Un [QSlider](#) se présente sous la forme d'un curseur permettant de sélectionner une valeur numérique :



QSlider hérite de [QAbstractSlider](#) (damned, encore une classe abstraite !) qui propose déjà un grand nombre de fonctionnalités de base.

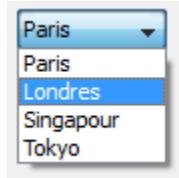
Beaucoup de propriétés sont les mêmes que QSpinBox, je ne les relisterai donc pas ici. Notons la propriété orientation qui permet de définir l'orientation du slider (verticale ou horizontale).

Jetez un oeil en particulier à ses signaux, car on connecte en général le signal valueChanged(int) au slot d'autre widget pour répercuter la saisie de l'utilisateur.

Nous avions d'ailleurs manipulé ce widget lors du chapitre sur les signaux et les slots.

QComboBox : une liste déroulante

Une [QComboBox](#) est une liste déroulante :



On ajoute des valeurs à la liste déroulante avec la méthode addItem :

Code : C++ - [Sélectionner](#)

```
#include < QApplication>
#include <QtGui>

int main( int argc, char *argv[] )
{
    QApplication app(argc, argv);

    QWidget fenetre;

    QComboBox *liste = new QComboBox(&fenetre);
    liste->addItem( "Paris" );
    liste->addItem( "Londres" );
    liste->addItem( "Singapour" );
    liste->addItem( "Tokyo" );
    liste->move( 30, 20 );

    fenetre.show();

    return app.exec();
}
```

On dispose de propriétés permettant de contrôler le fonctionnement de la QComboBox :

- count : nombre d'éléments dans la liste déroulante.
- currentIndex : numéro d'indice de l'élément actuellement sélectionné. Les indices commencent à 0. Ainsi, si currentItem renvoie 2, c'est que "Singapour" a été sélectionné dans l'exemple précédent.
- currentText : texte correspondant à l'élément sélectionné. Si on a sélectionné "Singapour", cette propriété contient donc "Singapour".
- editable : indique si le widget autorise l'ajout de valeurs personnalisées ou non. Par défaut, l'ajout de nouvelles valeurs est interdit.

Si le widget est éditable, l'utilisateur pourra rentrer de nouvelles valeurs dans la liste déroulante. Elle se comportera donc aussi comme un champ de texte. L'ajout d'une nouvelle valeur se fait par appui sur la touche "Entrée". Les nouvelles valeurs sont placées par défaut à la fin de la liste.

La QComboBox émet des signaux comme currentIndexChanged() qui indique qu'un nouvel élément a été sélectionné et highlighted() qui indique l'élément survolé par la souris (ces signaux peuvent envoyer un int pour donner l'indice de l'élément ou un QString pour le texte).

A noter aussi le widget fils [QFontComboBox](#) qui permet de sélectionner une police parmi une liste proposant une prévisualisation de la police.

Les conteneurs

Normalement, n'importe quel widget peut en contenir d'autres.

Cependant, certains widgets ont été vraiment créés spécialement pour pouvoir en contenir d'autres :

- QFrame : un widget pouvant avoir une bordure.
- QGroupBox : un widget (que nous avons déjà utilisé) adapté à la gestion des cases à cocher et de boutons radio.
- QTabWidget : un widget gérant plusieurs pages d'onglets.

Nous allons apprendre à les manipuler, en nous intéressant en particulier à celui qui propose des onglets qui est un petit peu délicat.



QFrame : une bordure

[QFrame](#) est très proche de QWidget. En fait, la seule nouveauté c'est qu'il peut générer une bordure. C'est donc un QWidget basique avec une bordure.



QFrame est une classe de base pour de nombreux widgets qui peuvent avoir une bordure, comme les QLabel. Tout ce que nous allons faire avec les QFrame ici, tous les widgets qui en héritent peuvent le faire aussi.

Dans la doc de [QFrame](#), regardez au début le paragraphe "Inherited by...". C'est la liste des classes qui héritent de QFrame, et qui disposent donc aussi des fonctionnalités de QFrame.

Un QFrame possède quelques propriétés pour gérer la forme de la bordure :

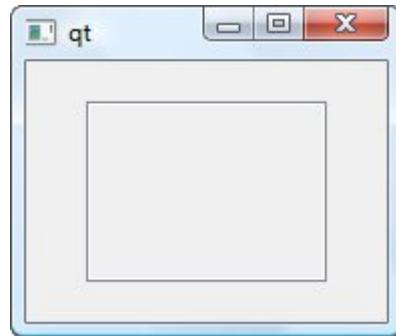
- frameShape : le type de bordure. Il faut utiliser une énumération définie par QFrame pour sélectionner la bordure. Consultez la doc pour avoir la liste des types de bordure.
De manière générale je recommande d'utiliser QFrame::StyledPanel (comme sur ma capture d'écran) car cela crée une bordure dans un style adapté à votre OS. Notez aussi QFrame::HLine et QFrame::VLine, un peu particuliers, qui ne créent pas un rectangle mais juste une ligne horizontale ou verticale. Très utile pour séparer les éléments dans sa fenêtre.
- frameShadow : l'ombre de la bordure. Par défaut il n'y en a pas, mais vous pouvez définir une ombre qui donne l'impression que le widget est surélevé ou enfoncé.
Regardez les énumérations définies par QFrame pour avoir la liste des possibilités.
- lineWidth : l'épaisseur de la ligne de la bordure.
- midLineWidth : l'épaisseur de la ligne intermédiaire (utilisé uniquement pour certaines bordures complexes avec une ombre).

Testons la propriété frameShape :

Code : C++ - [Sélectionner](#)

```
QFrame *frame = new QFrame(&fenetre);
frame->setFrameShape(QFrame::StyledPanel);
frame->setGeometry(30, 20, 120, 90);
```

Résultat :



Dans la pratique, le QFrame sert à contenir d'autres widgets (à moins que vous aimiez dessiner des rectangles partout pour le plaisir). Il est donc probable que vous définissiez un layout pour le QFrame et que vous placiez des widgets à l'intérieur.
Allez je me fends d'un petit exemple, ça vous rappellera le chapitre sur les layouts :

Code : C++ - [Sélectionner](#)

```
#include <QApplication>
#include <QtGui>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QWidget fenetre;

    QFrame *frame = new QFrame(&fenetre);
    frame->setFrameShape(QFrame::StyledPanel);
    frame->setGeometry(30, 20, 120, 90);

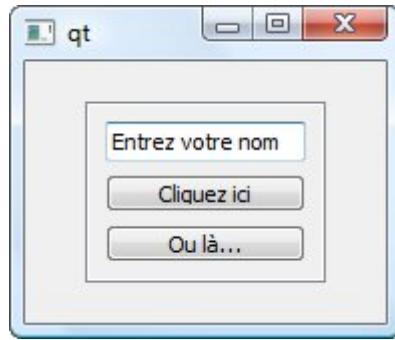
    QLineEdit *lineEdit = new QLineEdit("Entrez votre nom");
    QPushButton *bouton1 = new QPushButton("Cliquez ici");
    QPushButton *bouton2 = new QPushButton("Ou là...");

    QVBoxLayout *vbox = new QVBoxLayout;
    vbox->addWidget(lineEdit);
    vbox->addWidget(bouton1);
    vbox->addWidget(bouton2);

    frame->setLayout(vbox);

    fenetre.show();

    return app.exec();
}
```

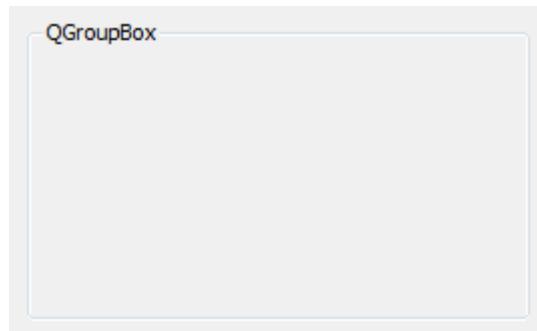


On n'a rien fait de bien nouveau. Le QFrame contient un layout vertical qui organise ses widgets enfants : un QLineEdit et deux QPushButton.

Le QFrame lui-même est placé de manière absolue sur la fenêtre (ses coordonnées sont définies dans setGeometry). Je fais ça pour simplifier l'exemple, mais dans la pratique le QFrame devrait être lui-même placé dans un autre layout (le layout principal de la fenêtre).

QGroupBox : un groupe de widgets

Nous avons déjà utilisé [QGroupBox](#) en pratique plus haut, lorsque nous avons utilisé les boutons radio. Je ne vais donc pas trop m'éterniser dessus.



Un QGroupBox propose de créer une bordure comme QFrame, mais il a en plus l'avantage de permettre de définir un titre pour le conteneur.

QGroupBox n'hérite pas de QFrame. On n'a donc pas le choix dans le type de bordure, mais on a une propriété flat qui permet d'aplatiser la bordure.

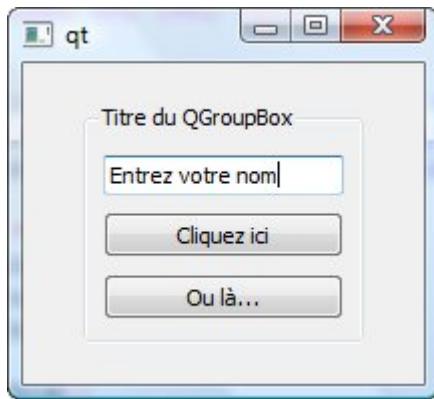
Le titre du conteneur est défini via sa propriété title, ou directement lors de l'appel au constructeur :

Code : C++ - [Sélectionner](#)

```
QGroupBox *groupBox = new QGroupBox("Titre du QGroupBox", &fenetre);
```

Exercice : essayez d'adapter l'exemple précédent sur QFrame pour utiliser cette fois un QGroupBox. C'est facile, mais attention aux propriétés spécifiques au QFrame qui ne sont ici plus valables.

Le résultat devrait être le suivant :



Il est aussi possible d'ajouter une case à cocher devant le QGroupBox. Pour cela, mettez sa propriété checkable à true :

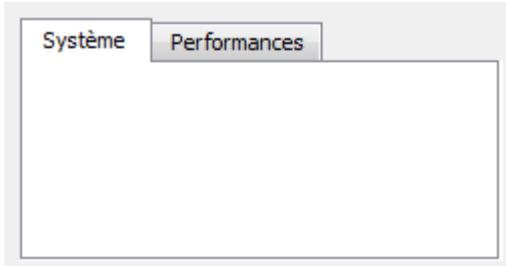
Code : C++ - [Sélectionner](#)

```
groupBox->setCheckable(true);
```

Lorsque la case est cochée, les widgets à l'intérieur sont activés. Lorsqu'elle est décochée, les widgets sont désactivés. Essayez.

QTabWidget : des pages d'onglets

Le [QTabWidget](#) propose une gestion de plusieurs pages de widgets, organisées sous forme d'onglets :



Ce widget-conteneur est sensiblement plus difficile à utiliser que les autres. En effet, il ne peut contenir qu'un widget par page.

Quoi ? On ne peut pas afficher plus d'un widget par page ???

Mais c'est tout nul !

Sauf... qu'un widget peut en contenir d'autres ! 😊

Et si on utilise un layout pour organiser le contenu de ce widget, on peut arriver rapidement à une super présentation. Le tout est de savoir combiner tout ce qu'on a appris jusqu'ici.

D'après le texte d'introduction de la doc de QTabWidget, ce conteneur doit être utilisé de la façon suivante :

1. Créer un QTabWidget.
2. Créer un QWidget pour chacune des pages (chacun des onglets) du QTabWidget, sans leur indiquer de widget parent.
3. Placer des widgets enfants dans chacun de ces QWidget pour peupler le contenu de chaque page. Utiliser un layout pour positionner les widgets de préférence.
4. Appeler plusieurs fois addTab() pour créer les pages d'onglets en indiquant l'adresse du QWidget qui contient la page à chaque fois.

Bon, c'est un peu plus délicat comme vous pouvez le voir, mais il faut bien un peu de difficulté, ce chapitre était trop facile. 😊

Si on fait tout dans l'ordre, vous allez voir que l'on n'aura pas de problème.

Je vous propose de lire ce code que j'ai créé qui montre un exemple d'utilisation du QTabWidget. Il est un peu long mais il est commenté et vous devriez arriver à le digérer. 😊

Code : C++ - [Sélectionner](#)

```

#include <QApplication>
#include <QtGui>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QWidget fenetre;

    // 1 : Créer le QTabWidget
    QTabWidget *onglets = new QTabWidget(&fenetre);
    onglets->setGeometry(30, 20, 240, 160);

    // 2 : Créer les pages, en utilisant un widget parent pour contenir chacune des pages
    QWidget *page1 = new QWidget;
    QWidget *page2 = new QWidget;
    QLabel *page3 = new QLabel; // Comme un QLabel est aussi un QWidget (il en hérite), on peut l'utiliser comme page

    // 3 : Créer le contenu des pages de widgets

    // Page 1

    QLineEdit *lineEdit = new QLineEdit("Entrez votre nom");
    QPushButton *bouton1 = new QPushButton("Cliquez ici");
    QPushButton *bouton2 = new QPushButton("Ou là...");

    QVBoxLayout *vbox1 = new QVBoxLayout;
    vbox1->addWidget(lineEdit);
    vbox1->addWidget(bouton1);
    vbox1->addWidget(bouton2);

    page1->setLayout(vbox1);

    // Page 2

    QProgressBar *progress = new QProgressBar;
    progress->setValue(50);
    QSlider *slider = new QSlider(Qt::Horizontal);
    QPushButton *bouton3 = new QPushButton("Valider");

    QVBoxLayout *vbox2 = new QVBoxLayout;
    vbox2->addWidget(progress);
    vbox2->addWidget(slider);
    vbox2->addWidget(bouton3);

    page2->setLayout(vbox2);

    // Page 3 (je ne vais afficher qu'une image ici, pas besoin de layout)

    page3->setPixmap(QPixmap("icone.png"));
    page3->setAlignment(Qt::AlignCenter);

    // 4 : ajouter les onglets au QTabWidget, en indiquant la page qu'ils contiennent
    onglets->addTab(page1, "Coordonnées");
    onglets->addTab(page2, "Progression");
    onglets->addTab(page3, "Image");

    fenetre.show();

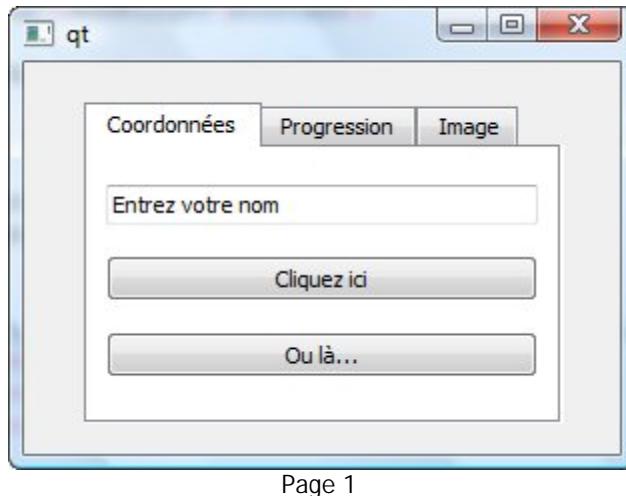
    return app.exec();

```

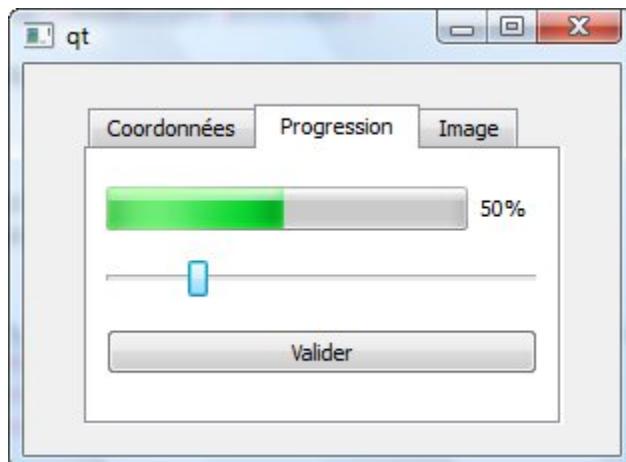
Vous devriez retrouver chacune des étapes que j'ai mentionnées plus haut :

1. Je crée d'abord le QTabWidget que je positionne ici de manière absolue sur la fenêtre (mais je pourrais aussi utiliser un layout).
2. Ensuite, je crée les pages pour chacun de mes onglets. Ces pages sont matérialisées par des QWidget. Vous noterez que pour la dernière page je n'utilise pas un QWidget mais un QLabel. Ca revient au même et c'est compatible car QLabel hérite de QWidget. Sur la dernière page, je me contenterai d'afficher une image.
3. Je crée ensuite le contenu de chacune de ces pages que je dispose à l'aide de layouts verticaux (sauf pour la page 3 qui n'est constituée que d'un widget). Là il n'y a rien de nouveau.
4. Enfin, j'ajoute les onglets avec la méthode addTab(). Je dois indiquer le libellé de l'onglet ainsi qu'un pointeur vers le "widget-page" de cette page.

Résultat, on a un super système d'onglets à 3 pages avec tout plein de widgets dedans !



Page 1



Page 2



Page 3

Je vous conseille de vous entraîner à créer vous aussi un QTabWidget. C'est un bon exercice, et c'est l'occasion de réutiliser la plupart des widgets que l'on a vus dans ce chapitre. 😊

Il faut pratiquer et pratiquer. Ce qu'on fait là ne devrait pas être bien compliqué si vous avez correctement suivi le cours jusqu'ici. Mais il faut quand même pratiquer pour faire des erreurs et se rendre compte qu'on n'avait pas parfaitement tout compris. C'est à partir de ce moment-là seulement que vous commencerez à maîtriser les widgets. 😊

Pfiou !

Il s'agit peut-être du chapitre le plus long que j'aie jamais écrit... mais je tenais à le faire. En fait, c'est un peu paradoxal car tout cela se trouve dans la doc et je vous ai déjà appris à lire la doc, mais j'estimais qu'il devait quand même forcément y avoir un tour d'horizon des principaux widgets dans mon tutoriel, sinon j'aurais eu l'impression d'avoir été incomplet.

Servez-vous de ce chapitre pour vous faire une idée de ce qui existe, mais ensuite je vous conseille très fortement de passer plus de temps sur la doc que sur ce chapitre. En effet, nous sommes loin d'avoir vu toutes les fonctionnalités de ces widgets, de même que nous n'avons pas vu tous les widgets qui existent !

J'ai réservé les widgets les plus complexes pour de futurs chapitres, qui introduiront pour l'occasion un concept de programmation important lorsqu'on programme des GUI : le modèle MVC.

Bon, vous pensez pas qu'il serait temps de pratiquer tout ce qu'on a appris avec un petit TP là ? 😊

TP : ZeroClassGenerator

Je pense que le moment est bien choisi pour vous exercer avec un petit TP. En effet, vous avez déjà vu suffisamment de choses sur Qt pour être en mesure de faire déjà des programmes intéressants.

Quand je me suis dit "Je vais leur faire faire un TP", les idées de sujet ne manquaient pas... mais elles étaient toutes un peu "bateau". J'ai pensé à une calculatrice par exemple, et en effet pourquoi pas, mais ce n'était pas très original.

Finalement, après réflexion, j'ai trouvé une idée qui sort un peu de l'ordinaire et qui pourra même vous être utile à vous, programmeurs. 😊

Notre programme s'intitulera le ZeroClassGenerator... un programme qui génère le code de base des classes C++ automatiquement en fonction des options que vous choisissez !

Notre objectif

Ne vous laissez pas impressionner par le nom "ZeroClassGenerator". Ce TP ne sera pas bien difficile et réutilisera toutes les connaissances que vous avez apprises pour les mettre à profit dans un projet concret.

Ce TP est volontairement modulaire : je vais vous proposer de réaliser un programme de base assez simple, que je vous laisserai coder et que je corrigerais ensuite avec vous. Puis, je vous proposerai un certain nombre d'améliorations intéressantes (non corrigées) pour lesquelles il faudra vous creuser un peu plus les méninges si vous êtes motivés. 😊

Notre ZeroClassGenerator est un programme qui génère le code de base des classes C++. Qu'est-ce que ça veut dire ?

Un générateur de classe C++

Ce programme est un outil graphique qui va créer automatiquement le code source d'une classe en fonction des options que vous aurez choisies.

Vous n'avez jamais remarqué que les classes avaient en général une structure de base similaire qu'il fallait réécrire à chaque fois ? C'est un peu laborieux parfois. Par exemple :

Code : C++ - Sélectionner

```
#ifndef HEADER_MAGICIEN
#define HEADER_MAGICIEN

class Magicien : public Personnage
{
public:
    Magicien();
    ~Magicien();

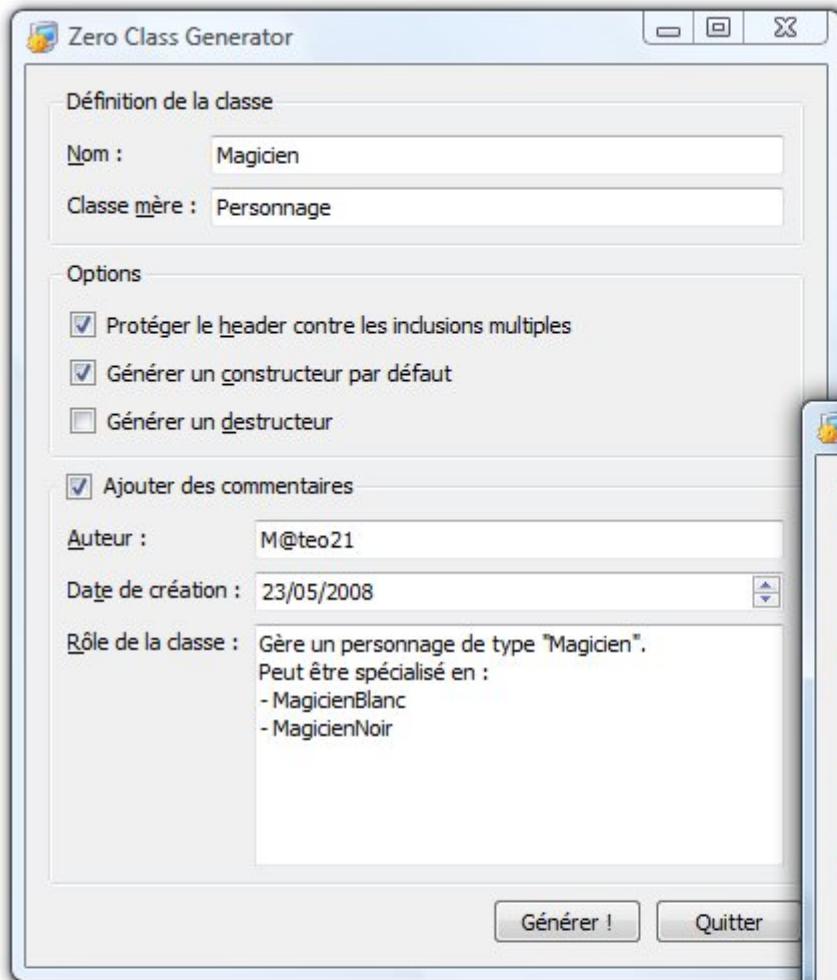
protected:

private:
};

#endif
```

Rien que ça, ça serait bien si on avait un programme capable de générer le squelette de la classe, de définir les portées public, protected et private, de définir un constructeur par défaut et un destructeur, etc.

Nous allons réaliser un GUI (une fenêtre) contenant plusieurs options. Plutôt que de faire une longue liste, je vous propose une capture d'écran du programme final à réaliser :



ZeroClassGenerator

```
/*
Auteur : M@teo21
Date de création : ven. mai 23 2008

Rôle :
Gère un personnage de type "Magicien".
Peut être spécialisé en :
- MagicienBlanc
- MagicienNoir
*/

#ifndef HEADER_MAGICIEN
#define HEADER_MAGICIEN

class Magicien : public Personnage
{
public:
    Magicien();

protected:

private:

```

Fermer

La fenêtre principale est en haut à gauche, en arrière-plan. L'utilisateur renseigne obligatoirement le champ "Nom", pour indiquer le nom de la classe. Il peut aussi donner le nom de la classe mère.

On propose quelques cases à cocher pour choisir des options comme "Protéger le header contre les inclusions multiples" (la fameuse technique du `#ifndef`, pratique mais un peu lourde à écrire à chaque fois). Il faudra que le nom du `define` soit généré automatiquement à partir du nom de la classe, et mis en majuscules. Pour la mise en majuscules, renseignez-vous auprès de la doc de la classe [QString](#) qui propose plein de choses.

Enfin, on donne la possibilité d'ajouter des commentaires en haut du fichier pour indiquer quel est l'auteur, quelle est la date de création et quel est le rôle de la classe. C'est une bonne habitude en effet que de commenter un peu le début de ses classes pour que l'on ait une idée de ce à quoi elle sert.

Lorsqu'on clique sur le bouton "Générer" en bas, une nouvelle fenêtre s'ouvre (une `QDialog`). Elle affiche le code généré dans un `QTextEdit`, et vous pouvez à partir de là copier/coller ce code dans votre IDE comme `Code::Blocks`.

C'est un début, et je vous proposerai à la fin du chapitre des améliorations intéressantes à ajouter à ce programme. Essayez déjà de réaliser ça correctement, ça représente un peu de travail je peux vous le dire !

Quelques conseils techniques

Avant de vous lâcher tels des fauves dans la jungle, je voudrais vous donner quelques conseils techniques pour vous guider un peu.



Architecture du projet

Je vous recommande de faire une classe par fenêtre. Comme on a 2 fenêtres, et qu'on met toujours le main à part, ça fait 5 fichiers :

- main.cpp : contiendra uniquement le main qui ouvre la fenêtre principale (très court).
- FenPrincipale.h : header de la fenêtre principale.
- FenPrincipale.cpp : l'implémentation des méthodes de la fenêtre principale.
- FenCodeGenere.h : le header de la fenêtre secondaire qui affiche le code généré.
- FenCodeGenere.cpp : ... et l'implementation de ses méthodes.

Pour la fenêtre principale, vous pourrez hériter de QWidget comme on l'a toujours fait, ça me semble le meilleur choix.

Pour la fenêtre secondaire, je vous conseille d'hériter de QDialog. La fenêtre principale ouvrira la QDialog en appelant sa méthode exec().

La fenêtre principale

Je vous conseille très fortement d'utiliser des layouts. Mon layout principal, si vous regardez bien ma capture d'écran, est un layout vertical. Il contient des QGroupBox.

A l'intérieur des QGroupBox, j'utilise à nouveau des layouts. Je vous laisse le choix du layout qui vous semble le plus adapté à chaque fois.

Pour le QGroupBox "Ajouter des commentaires", il faudra ajouter une case à cocher. Si cette case est cochée, les commentaires seront ajoutés. Sinon, on ne mettra pas de commentaires. Renseignez-vous sur l'utilisation des cases à cocher dans les QGroupBox.

Pour le champ "Date de création", je vous propose d'utiliser un [QDateEdit](#). Ce widget n'a pas été vu dans le chapitre précédent mais je vous fais confiance, il est proche de la QSpinBox et après lecture de la doc vous devriez savoir vous en servir sans problème.

Vous "dessinerez" le contenu de la fenêtre dans le constructeur de FenPrincipale. Pensez à faire de vos champ de formulaire des attributs de la classe (les QLineEdit, QCheckbox...), afin que toutes les autres méthodes de la classe aient accès à leur valeur.

Lors d'un clic sur le bouton "Générer !", appelez un slot personnalisé. Dans ce slot personnalisé (qui ne sera rien d'autre qu'une méthode de FenPrincipale), vous récupérerez toutes les infos contenues dans les champs de la fenêtre pour générer le code dans une chaîne de caractères (de type QString de préférence).

C'est là qu'il faudra un peu réfléchir sur la génération du code, mais c'est tout à fait faisable. 😊

Une fois le code généré, votre slot appellera la méthode exec() d'un objet de type FenCodeGenere que vous aurez créé pour l'occasion. La fenêtre du code généré s'affichera alors...

La fenêtre du code généré

Beaucoup plus simple, cette fenêtre est constituée d'un QTextEdit et d'un bouton de fermeture.

Pour le QTextEdit, essayez de définir une police à pas fixe (comme "Courier") pour que ça ressemble à du code (parce que le Times New Roman pour rédiger du code c'est moche 😊). Personnellement, j'ai rendu le QTextEdit en mode readOnly pour qu'on ne puisse pas modifier son contenu (juste le copier), mais vous faites comme vous voulez.

Vous connecterez le bouton "Fermer" à un slot spécial de la QDialog qui demande la fermeture et qui indique que tout s'est bien

passé. Je vous laisse trouver dans la doc duquel il s'agit. 😊

Minute euh... Comment je passe le code généré (de type QString si j'ai bien compris) à la seconde fenêtre de type QDialog ?

Le mieux est de passer cette QString en paramètre du constructeur. Votre fenêtre récupèrera ainsi le code et n'aura plus qu'à l'afficher dans son QTextEdit !

Allez hop hop hop, au boulot, à vos éditeurs ! Vous aurez besoin de lire la doc plusieurs fois pour trouver la bonne méthode à appeler à chaque fois, donc n'ayez pas peur d'y aller.

On se retrouve dans la partie suivante pour la... correction !

Correction

Ding !

C'est l'heure de ramasser les copies. 🦁

Bien que je vous aie donné quelques conseils techniques, je vous ai volontairement laissé le choix pour certains petits détails (comme "quelles cases sont cochées par défaut"). Vous pouviez même présenter la fenêtre un peu différemment si vous vouliez. Tout ça pour dire que ma correction n'est pas la correction ultime. Si vous avez fait différemment, ce n'est pas grave. Si vous n'avez pas réussi, ce n'est pas grave non plus, pas de panique : prenez le temps de bien lire mon code et d'essayer de comprendre ce que je fais. Vous devrez être capable par la suite de refaire ce TP sans regarder la correction. 😊

main.cpp

Comme prévu, ce fichier est tout bête et ne mérite même pas d'explication. 😊

Code : C++ - [Sélectionner](#)

```
#include < QApplication>
#include "FenPrincipale.h"

int main( int argc, char* argv[] )
{
    QApplication app(argc, argv);

    FenPrincipale fenetre;
    fenetre.show();

    return app.exec();
}
```

Je signale juste qu'on aurait pu charger la langue française comme on l'avait fait dans le chapitre sur les boîtes de dialogue, afin que les menus contextuels et certains boutons automatiques soient traduits en français. Mais c'est du détail, ça ne se verra pas vraiment sur ce projet.

FenPrincipale.h

La fenêtre principale hérite de QWidget comme prévu. Elle utilise la macro Q_OBJECT car nous définissons un slot personnalisé :

Code : C++ - [Sélectionner](#)

```
#ifndef HEADER_FENPRINCIPALE
#define HEADER_FENPRINCIPALE

#include <QtGui>

class FenPrincipale : public QWidget
{
    Q_OBJECT

public:
    FenPrincipale();

private slots:
    void genererCode();

private:
    QLineEdit *nom;
    QLineEdit *classeMere;
    QCheckBox *protections;
    QCheckBox *genererConstructeur;
    QCheckBox *genererDestructeur;
    QGroupBox *groupCommentaires;
    QLineEdit *auteur;
    QDateEdit *date;
    QTextEdit *role;
    QPushButton *generer;
    QPushButton *quitter;

};

#endif
```

Ce qui est intéressant, ce sont tous les champs de formulaire que j'ai mis en tant qu'attributs (privés) de la classe. Il faudra les initialiser dans le constructeur. L'avantage d'avoir défini les champs en attributs, c'est que toutes les méthodes de la classe y auront accès, et ça nous sera bien utile pour récupérer les valeurs des champs dans notre méthode qui générera le code source.

Notre classe est constituée de 2 méthodes, ce qui est ici largement suffisant :

- FenPrincipale() : c'est le constructeur. Il initialisera les champs de la fenêtre, jouera avec les layouts et placera les champs à l'intérieur. Il fera des connexions entre les widgets et indiquera la taille de la fenêtre, son titre, son icône...
- genererCode() : c'est une méthode (plus précisément un slot) qui sera connectée au signal "Le bouton Générer a été cliqué". Dès qu'on cliquera sur le bouton, cette méthode sera appelée.
J'ai mis le slot en privé car il n'y a pas de raison qu'une autre classe l'appelle, mais j'aurais aussi bien pu le mettre public.

FenPrincipale.cpp

Bon là c'est le plus gros morceau. Il n'y a que 2 méthodes mais elles sont grosses, ne vous laissez pas impressionner pour autant. Prenez le temps de bien les comprendre. 😊

Code : C++ - [Sélectionner](#)

```
#include "FenPrincipale.h"
#include "FenCodeGenere.h"

FenPrincipale::FenPrincipale()
{
    // Groupe : Définition de la classe
    nom = new QLineEdit;
    classeMere = new QLineEdit;

    QFormLayout *definitionLayout = new QFormLayout;
    definitionLayout->addRow( "&Nom :" , nom );
    definitionLayout->addRow( "Classe &mère :" , classeMere );

    QGroupBox *groupDefinition = new QGroupBox( "Définition de la classe" );
    groupDefinition->setLayout(definitionLayout);

    // Groupe : Options

    protections = new QCheckBox( "Protéger le &header contre les inclusions multiples" );
    protections->setChecked(true);
    genererConstructeur = new QCheckBox( "Générer un &constructeur par défaut" );
    genererDestructeur = new QCheckBox( "Générer un &destructeur" );

    QVBoxLayout *optionsLayout = new QVBoxLayout;
    optionsLayout->addWidget(protections);
    optionsLayout->addWidget(genererConstructeur);
    optionsLayout->addWidget(genererDestructeur);

    QGroupBox *groupOptions = new QGroupBox( "Options" );
    groupOptions->setLayout(optionsLayout);

    // Groupe : Commentaires

    auteur = new QLineEdit;
    date = new QDateEdit;
    date->setDate(QDate::currentDate());
    role = new QTextEdit;

    QFormLayout *commentairesLayout = new QFormLayout;
    commentairesLayout->addRow( "&Auteur :" , auteur );
    commentairesLayout->addRow( "Da&te de création :" , date );
    commentairesLayout->addRow( "&Rôle de la classe :" , role );

    groupCommentaires = new QGroupBox( "Ajouter des commentaires" );
    groupCommentaires->setCheckable(true);
    groupCommentaires->setChecked(false);
    groupCommentaires->setLayout(commentairesLayout);

    // Layout : boutons du bas (générer, quitter...)
    generer = new QPushButton( "&Générer !" );
    quitter = new QPushButton( "&Quitter" );

    QHBoxLayout *boutonsLayout = new QHBoxLayout;
    boutonsLayout->setAlignment(Qt::AlignRight);

    boutonsLayout->addWidget(generer);
    boutonsLayout->addWidget(quitter);

    // Définition du layout principal, du titre de la fenêtre, etc.

    QVBoxLayout *layoutPrincipal = new QVBoxLayout;
    layoutPrincipal->addWidget(groupDefinition);
```

Vous noterez que j'appelle directement la méthode connect(), au lieu d'écrire QWidget::connect(). En effet, si on est dans une classe qui hérite de QWidget (et c'est le cas), on peut se passer de mettre le préfixe "QWidget::".

Pour le constructeur, je pense ne rien avoir à ajouter, je ne fais rien de bien nouveau. Il faut juste être organisé parce qu'il y a pas mal de lignes pour générer la fenêtre.

Par contre, le slot genererCode a demandé du travail, même s'il n'est pas si compliqué que ça au final. Il récupère la valeur des champs de la fenêtre (via des méthodes comme text() pour les QLineEdit). J'ai dû lire la doc plusieurs fois pour chacun de ces widgets afin de savoir comment récupérer le texte, la valeur (si la case est cochée ou pas), etc. Là, c'est juste de la lecture de la doc.

Une QString code se génère en fonction des choix que vous avez fait.

Une erreur se produit et la méthode s'arrête s'il n'y a pas au moins un nom de classe défini.

Tout à la fin de genererCode(), on n'a plus qu'à appeler la fenêtre secondaire et à lui envoyer le code généré :

Code : C++ - [Sélectionner](#)

```
FenCodeGenere *fenetreCode = new FenCodeGenere(code, this);
fenetreCode->exec();
```

Le code est envoyé lors de la construction de l'objet. La fenêtre sera affichée lors de l'appel à exec().

FenCodeGenere.h

La fenêtre du code généré est beaucoup plus simple que sa parente :

Code : C++ - [Sélectionner](#)

```
#ifndef HEADER_FENCODEGENERE
#define HEADER_FENCODEGENERE

#include <QtGui>

class FenCodeGenere : public QDialog
{
public:
    FenCodeGenere(QString &code, QWidget *parent);

private:
    QTextEdit *codeGenere;
    QPushButton *fermer;
};

#endif
```

Il y a juste un constructeur et deux petits widgets de rien du tout. 😊

FenCodeGenere.cpp

Le constructeur prend 2 paramètres :

- Une référence vers la QString qui contient le code.
- Un pointeur vers la fenêtre parente.

Code : C++ - Sélectionner

```
#include "FenCodeGenere.h"

FenCodeGenere::FenCodeGenere(QString &code, QWidget *parent = 0) : QDialog(parent)
{
    codeGenere = new QTextEdit();
    codeGenere->setPlainText(code);
    codeGenere->setReadOnly(true);
    codeGenere->setFont(QFont("Courier"));
    codeGenere->setLineWrapMode(QTextEdit::NoWrap);

    fermer = new QPushButton("Fermer");

    QVBoxLayout *layoutPrincipal = new QVBoxLayout;
    layoutPrincipal->addWidget(codeGenere);
    layoutPrincipal->addWidget(fermer);

    resize(350, 450);
    setLayout(layoutPrincipal);

    connect(fermer, SIGNAL(clicked()), this, SLOT(accept()));
}
```

C'est un rappel, mais je pense qu'il ne fera pas de mal : le paramètre parent est transféré au constructeur de la classe-mère QDialog dans cette ligne :

Code : C++ - Sélectionner

```
FenCodeGenere::FenCodeGenere(QString &code, QWidget *parent = 0) : QDialog(parent)
```

Schématiquement, le transfert se fait comme ceci :

```
FenCodeGenere::FenCodeGenere(QString &code, QWidget *parent = 0) : QDialog(parent)
```



Je pense que s'il y avait juste ça vous comprendrez tous :

Code : C++ - Sélectionner

```
FenCodeGenere::FenCodeGenere(QString &code, QWidget *parent = 0)
```

La nouveauté (enfin, on en a parlé dans le chapitre sur l'héritage quand même 😊), c'est qu'on appelle aussi le constructeur de la classe-mère QDialog et on lui transfère le paramètre parent avec le code : QDialog(parent)

Pourquoi avoir appelé le constructeur de QDialog et pourquoi lui avoir envoyé en paramètre un pointeur vers la fenêtre mère ?

Même si ce n'est pas obligatoire en fait, il est conseillé lors de la création d'une QDialog d'indiquer quelle est la fenêtre mère. La QDialog se centre automatiquement par rapport à la fenêtre mère, entre autres choses.

Télécharger le projet

Vous pouvez aussi télécharger le projet zippé :

[Télécharger le projet ZeroClassGenerator \(25 Ko\)](#)

Ce zip contient :

- Les fichiers source .cpp et .h
- Le projet .cbp pour ceux qui utilisent Code::Blocks
- L'exécutable Windows et son icône. Attention, il faudra mettre les DLL de Qt dans le même dossier si vous voulez que le programme puisse s'exécuter.

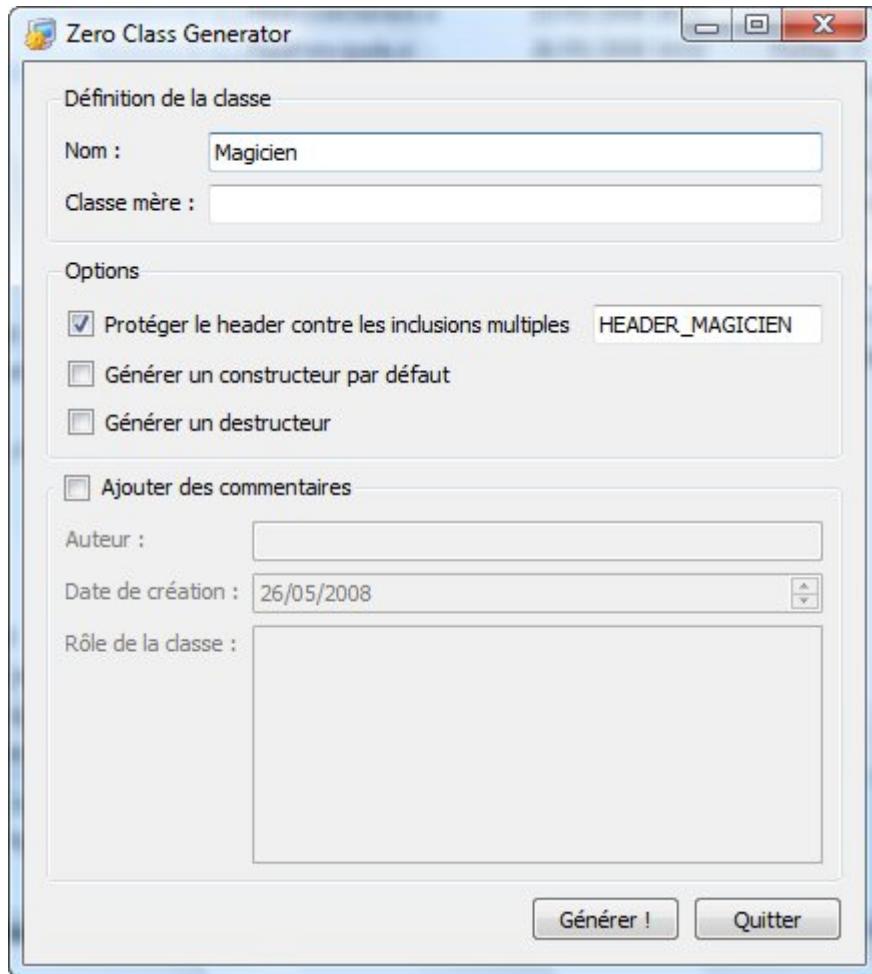
Des idées d'améliorations

Vous pensiez en avoir fini ?

Que nenni ! Un tel TP n'attend qu'une seule chose : être amélioré !

Voici une liste de suggestions qui me passent par la tête pour améliorer le ZeroCodeGenerator, mais vous pouvez inventer les vôtres :

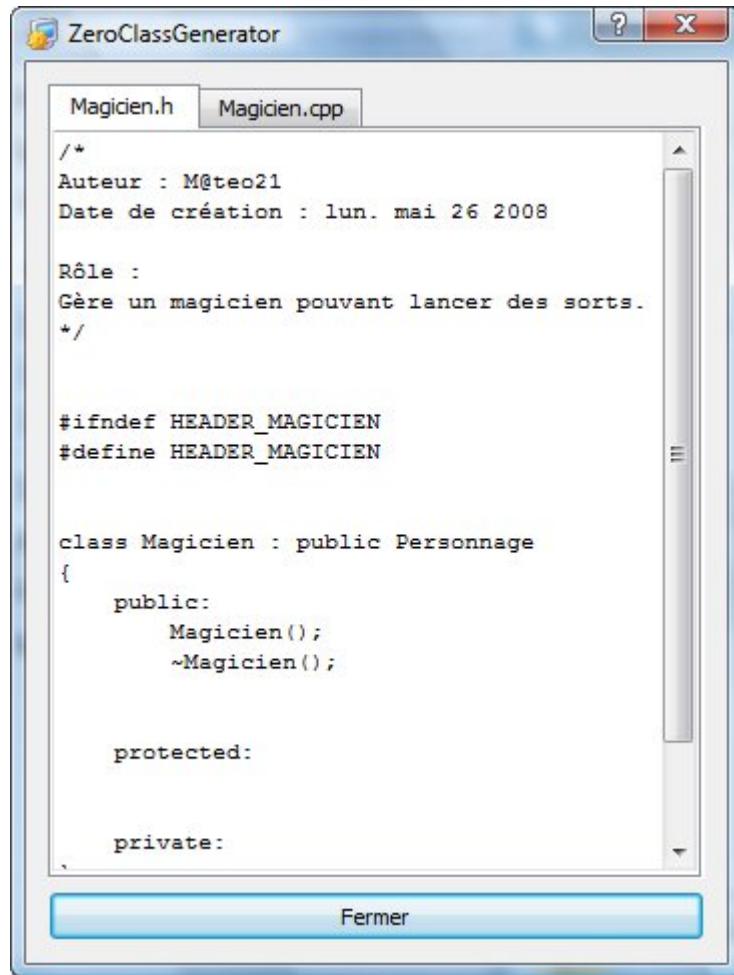
- Lorsqu'on coche "Protéger le header contre les inclusions multiples", un define (aussi appelé "header guard") est généré. Par défaut, ce header guard est de la forme HEADER_NOMCLASSE. Pourquoi ne pas l'afficher en temps réel dans un libellé lorsqu'on tape le nom de la classe ? Ou, mieux, affichez-le en temps réel dans un QLineEdit pour que la personne puisse le modifier si elle le désire.
Le but est de vous faire travailler les signaux et les slots.



- Ajoutez d'autres options de génération de code. Par exemple, vous pouvez proposer d'inclure le texte légal d'une licence libre (comme la GPL) dans les commentaires d'en-tête si la personne fait un logiciel libre, vous pouvez demander quels headers inclure, la liste des attributs, générer automatiquement les accesseurs pour ces attributs, etc. Attention, il faudra peut-être

utiliser des widgets de liste un peu plus complexes, comme le [QListWidget](#). Je ne vous l'ai pas encore présenté, mais rien ne vous interdit de prendre de l'avance. 😊

- Pour le moment on ne génère que le code du fichier .h. Même s'il y a moins de travail, ça serait bien de générer aussi le .cpp. Je vous propose d'utiliser un QTabWidget (des onglets) pour afficher le code .h et le .cpp dans la boîte de dialogue du code généré.



- On ne peut que voir et copier / coller le code généré. C'est bien, mais comme vous je pense que si on pouvait enregistrer le résultat dans des fichiers ce serait du temps gagné pour l'utilisateur. Je vous propose d'ajouter dans la QDialog un bouton pour sauvegarder dans des fichiers.

Ce bouton ouvrira une fenêtre qui demandera dans quel dossier enregistrer les fichiers .h et .cpp. Le nom de ces fichiers sera automatiquement généré en fonction du nom de la classe.

Pour l'enregistrement dans des fichiers, regardez du côté de la classe [QFile](#). Bon courage. 😊

- C'est un détail, mais les menus contextuels (quand on fait un clic droit sur un champ de texte par exemple) sont en anglais. Je vous avais parlé dans un des chapitres précédents d'une technique permettant de les avoir en français, un code à placer au début du main(). Je vous laisse le retrouver !
- On vérifie si le nom de la classe n'est pas vide, mais on ne vérifie pas s'il contient des caractères invalides (comme un espace, des accents, des guillemets...). Il faudrait afficher une erreur si le nom de la classe n'est pas valide.
Pour valider le texte saisi, vous avez 2 techniques : utiliser un inputMask(), ou un validator(). L'inputMask est peut-être le plus simple des deux, mais ça vaut le coup d'avoir pratiqué les deux. Pour savoir faire ça, direction la [doc de QLineEdit](#).

Voilà pour un petit début d'idées d'améliorations. Il y a déjà de quoi faire pour que vous ne puissiez pas dormir pendant quelques nuits, gnark gnark gnark. 😊

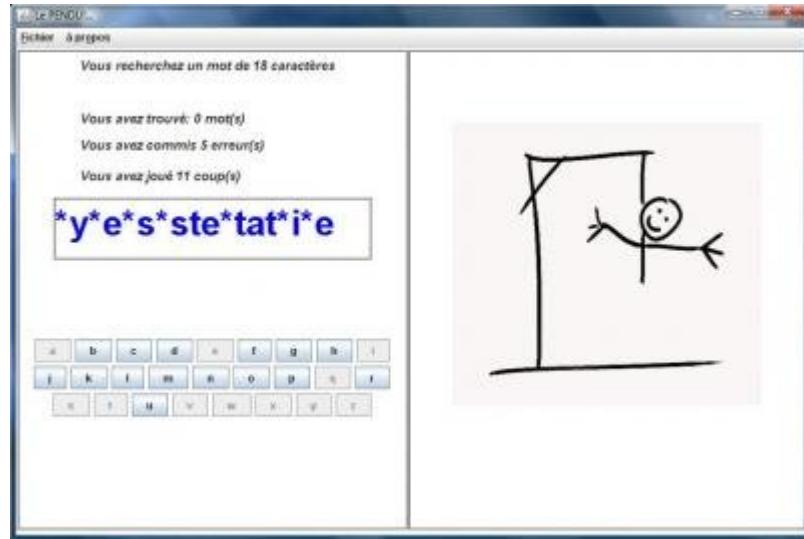
Comme toujours pour les TP, si vous êtes bloqués rendez-vous sur les forums du Site du Zéro pour demander de l'aide. Bon courage à tous !

J'espère que le sujet de ce TP vous a plu, j'ai essayé de trouver quelque chose d'original et d'éviter les sujets habituels comme "Faire une calculatrice".

Ceci étant, si vous avez envie de faire une calculatrice, surtout foncez ! Plus vous pratiquez, plus vous progressez (même si vous êtes parfois confrontés à des difficultés). C'est une règle immuable.

Parmi les TP sur lesquels vous pourriez vous entraîner à votre niveau, je vous suggère de regarder du côté de ce qu'on avait fait dans le cours de C en console. C'est l'occasion idéale de faire évoluer ces programmes de la console au GUI :

- [Le jeu du Plus ou Moins](#) : faites deviner un nombre à l'utilisateur. Ce devrait être assez simple à réaliser, et ça vous fera pratiquer un peu plus.
- [Le jeu du Pendu](#) : un peu plus complexe mais toujours très intéressant, le Pendu sous forme de GUI peut être un très bon exercice.
Ci-dessous, un exemple de jeu de pendu sous forme de GUI, ici réalisé en Java par cysboy [dans son tutoriel](#) (même si c'est du Java, ça ne change rien, c'est pour vous donner une idée de l'interface à réaliser) :



Amusez-vous bien. 😊

La fenêtre principale

Intéressons-nous maintenant à la fenêtre principale de vos applications.

Pour le moment, nous avons créé des fenêtres plutôt basiques en héritant de QWidget. C'est en effet largement suffisant pour de petites applications, mais au bout d'un moment on a besoin de plus d'outils.

La classe QMainWindow a été spécialement créée pour gérer la fenêtre principale de votre application quand celle-ci est complexe. Parmi les fonctionnalités offertes par QMainWindow, on trouve :

- Les menus
- La barre d'outils
- Les docks
- La barre d'état

A la fin de ce chapitre, vous pourrez vraiment faire tout ce que vous voulez de votre fenêtre principale !

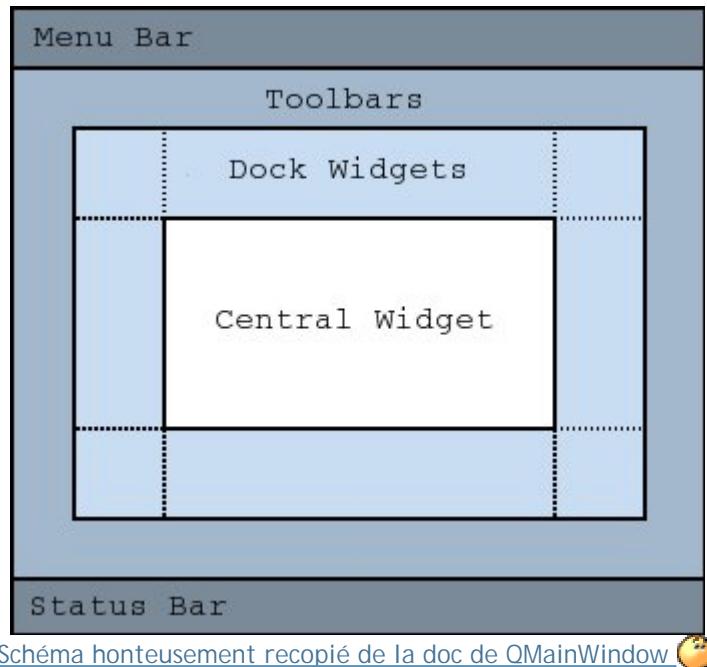
Présentation de QMainWindow

La classe QMainWindow hérite directement de QWidget. C'est un widget généralement utilisé une seule fois par programme, et qui sert uniquement à créer la fenêtre principale de l'application.

Certaines applications simples n'ont pas besoin de recourir à la QMainWindow. On va supposer ici que vous vous attaquez à un programme complexe et d'envergure. 😊

Structure de la QMainWindow

Avant toute chose, il me semble indispensable de vous présenter l'organisation d'une QMainWindow. Commençons par analyser le schéma ci-dessous :



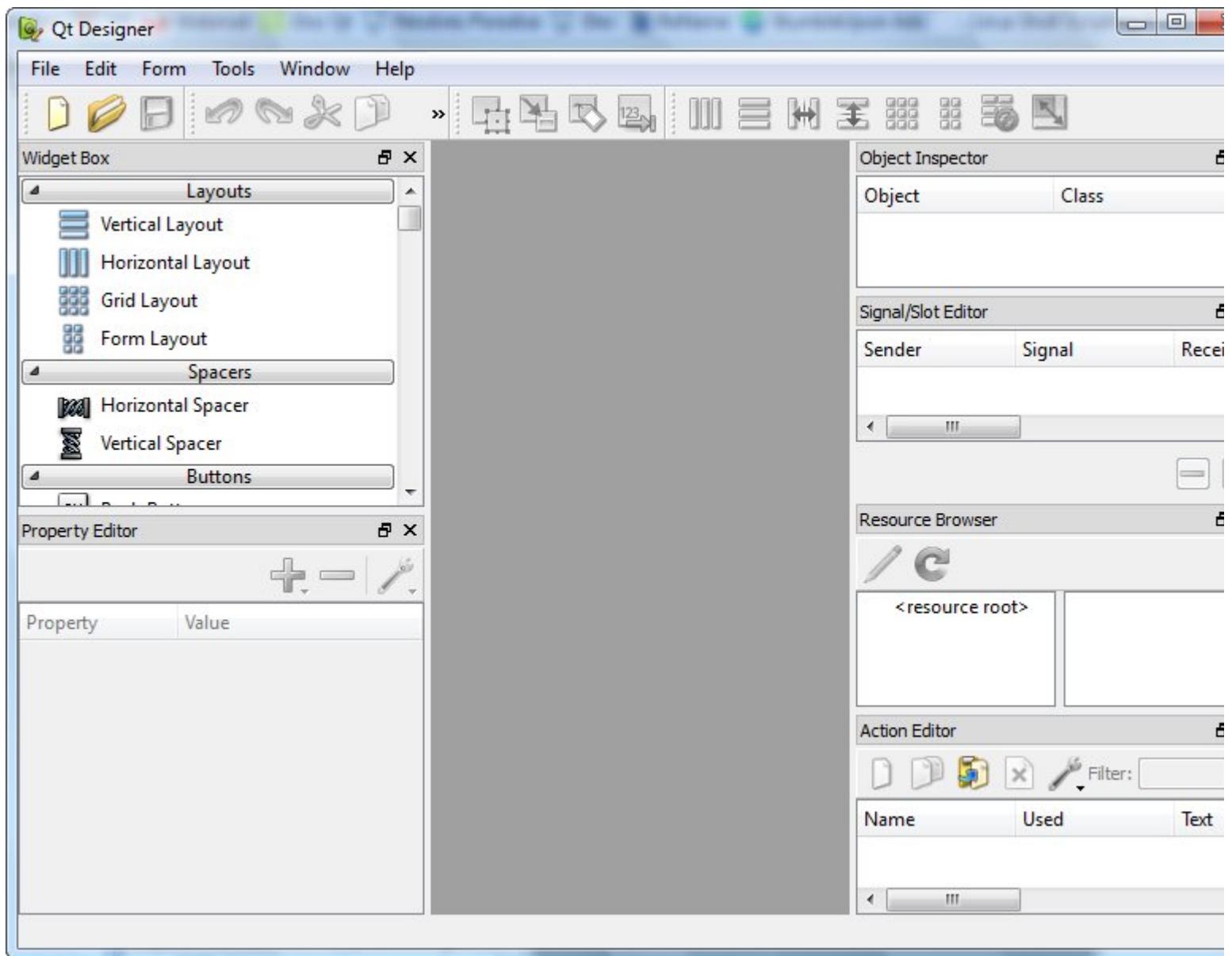
Une fenêtre principale peut être constituée de tout cela. Et j'ai bien dit peut, car rien ne vous oblige à utiliser à chaque fois chacun de ces éléments.

Détaillons les éléments :

- **Menu Bar** : c'est la barre de menus. C'est là que vous allez pouvoir créer votre menu Fichier, Edition, Affichage, Aide, etc.
- **Toolbars** : les barres d'outils. Dans un éditeur de texte, on a par exemple des icônes pour créer un nouveau fichier, pour enregistrer, etc.
- **Dock Widgets** : plus complexes et plus rarement utilisés, ces docks sont des conteneurs que l'on place autour de la fenêtre principale. Ils peuvent contenir des outils, par exemple les différents types de pinceaux que l'on peut utiliser quand on fait un logiciel de dessin.
- **Central Widget** : c'est le cœur de la fenêtre, là où il y aura le contenu proprement dit.
- **Status Bar** : la barre d'état. Elle affiche en général l'état du programme (Prêt / Enregistrement en cours, etc.).

Exemple de QMainWindow

Pour imaginer ces éléments en pratique, je vous propose de prendre pour exemple le programme Qt Designer :



Vous repérez en haut la barre de menus : File, Edit, Form...

En dessous, on a la barre d'outils, avec les icônes pour créer un nouveau projet, ouvrir un projet, enregistrer, annuler...

Autour (sur la gauche et la droite), on a les fameux docks. Ils servent ici à sélectionner le widget que l'on veut utiliser, ou à éditer les propriétés du widget par exemple.

Au centre, dans la partie grise où il n'y a rien, c'est la zone centrale. Lorsqu'un document est ouvert, cette zone l'affiche. La zone centrale peut afficher un ou plusieurs documents à la fois, comme on le verra plus loin.

Enfin, en bas il y a normalement la barre de statut, mais Qt Designer n'en utilise pas vraiment visuellement (en tout cas rien n'est affiché en bas).

Je vous propose de passer en revue chacune de ces sections dans ce chapitre. Nous commencerons par parler du "Central Widget", car c'est quand même lui le plus important et c'est le seul véritablement indispensable. 

Le code de base

Pour suivre ce chapitre, il va falloir créer un projet en même temps que moi. Nous allons créer notre propre classe de fenêtre principale qui héritera de QMainWindow, car c'est comme cela qu'on fait dans 99,99 % des cas.

Notre projet contiendra 3 fichiers :

- main.cpp : la fonction main().
- FenPrincipale.h : définition de notre classe FenPrincipale, qui héritera de QMainWindow.
- FenPrincipale.cpp : implémentation des méthodes de la fenêtre principale.

main.cpp

Code : C++ - Sélectionner

```
#include <QApplication>
#include <QtGui>
#include "FenPrincipale.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    FenPrincipale fenetre;
    fenetre.show();

    return app.exec();
}
```

FenPrincipale.h

Code : C++ - Sélectionner

```
#ifndef HEADER_FENPRINCIPALE
#define HEADER_FENPRINCIPALE

#include <QtGui>

class FenPrincipale : public QMainWindow
{
public:
    FenPrincipale();

private:
};

#endif
```

FenPrincipale.cpp

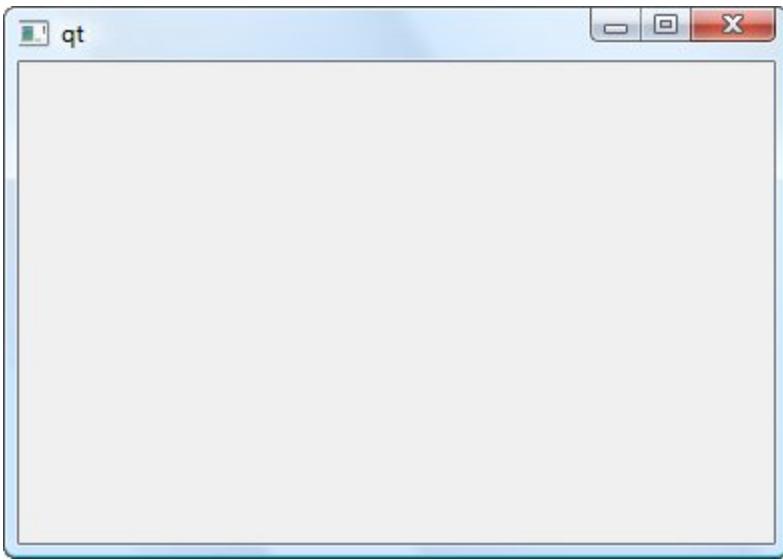
Code : C++ - Sélectionner

```
#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
}
```

Résultat

Si tout va bien, ce code devrait avoir pour effet d'afficher une fenêtre vide, toute bête :



Si c'est ce qui s'affiche chez vous, c'est bon, nous pouvons commencer. 😊

La zone centrale (SDI et MDI)

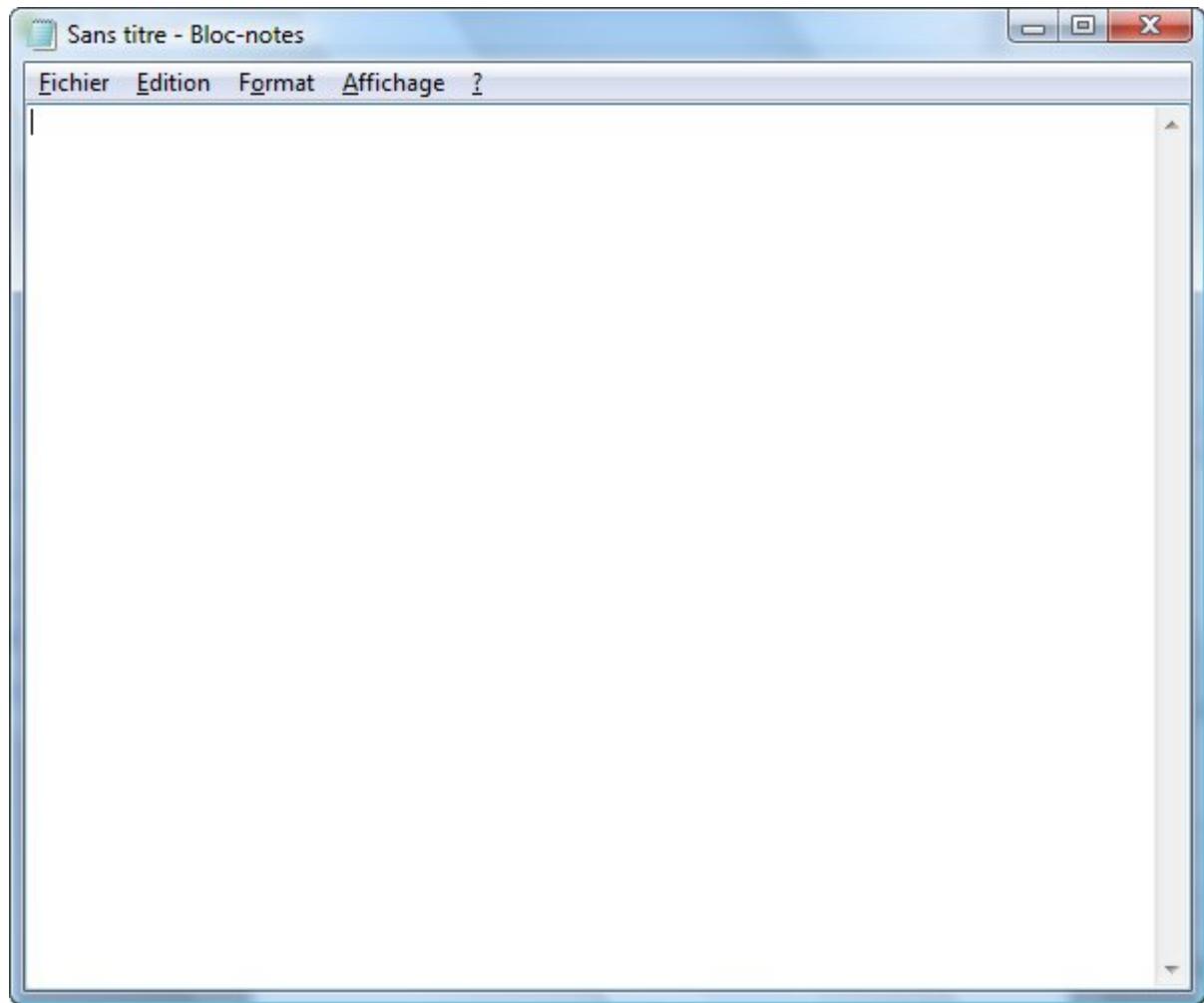
La zone centrale de la fenêtre principale est prévue pour contenir un et un seul widget.

C'est comme pour les onglets. On y insère un QWidget (ou une de ses classes filles) et on s'en sert comme conteneur pour mettre d'autres widgets à l'intérieur si besoin est.

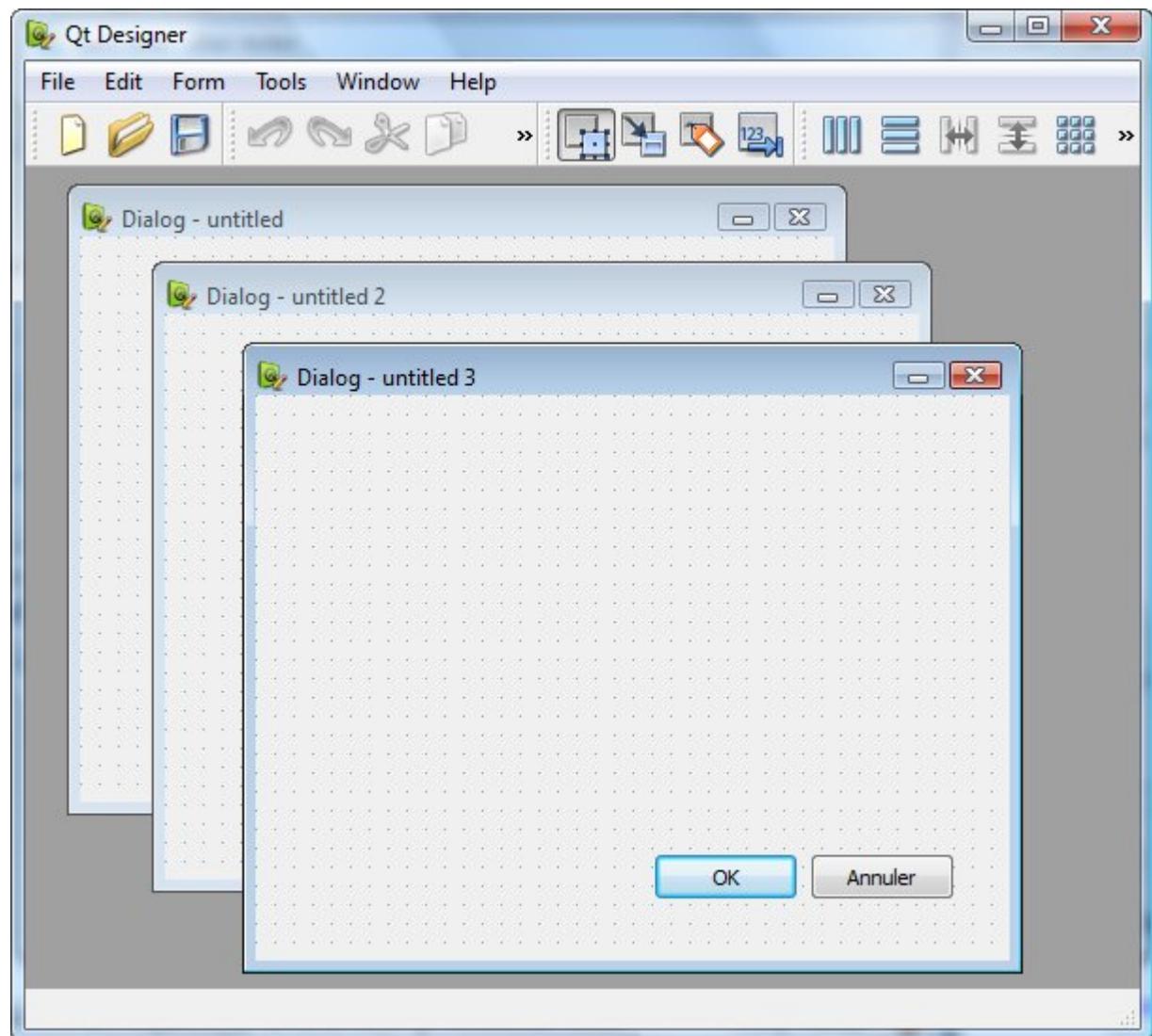
Nous allons refaire la manipulation ici pour s'assurer que tout le monde comprend comment cela fonctionne.

Sachez tout d'abord qu'on distingue 2 types de QMainWindow :

- Les SDI (Single Document Interface) : elles ne peuvent afficher qu'un document à la fois. C'est le cas de Bloc-Notes par exemple :



- Les MDI (Multiple Document Interface) : elles peuvent afficher plusieurs documents à la fois. Elles affichent des sous-fenêtres dans la zone centrale. C'est le cas par exemple de Qt Designer :



Définition de la zone centrale (type SDI)

On utilise la méthode `setCentralWidget()` de la `QMainWindow` pour indiquer quel widget contiendra la zone centrale. Faisons cela dans le constructeur de `FenPrincipale` :

Code : C++ - [Sélectionner](#)

```
#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
    QWidget *zoneCentrale = new QWidget;
    setCentralWidget(zoneCentrale);
}
```

Visuellement, ça ne change rien pour le moment. Par contre ce qui est intéressant, c'est qu'on a maintenant un `QWidget` qui sert de conteneur pour les autres widgets de la zone centrale de la fenêtre.

On peut donc y insérer des widgets au milieu :

Code : C++ - [Sélectionner](#)

```

#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
    QWidget *zoneCentrale = new QWidget;

    QLineEdit *nom = new QLineEdit;
    QLineEdit *prenom = new QLineEdit;
    QLineEdit *age = new QLineEdit;

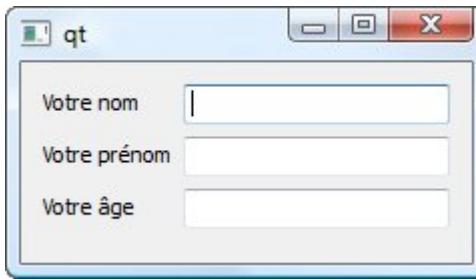
    QFormLayout *layout = new QFormLayout;
    layout->addRow("Votre nom", nom);
    layout->addRow("Votre prénom", prenom);
    layout->addRow("Votre âge", age);

    zoneCentrale->setLayout(layout);

    setCentralWidget(zoneCentrale);
}

```

Vous noterez que j'ai repris le code du chapitre sur les layouts. J'ai un poil dans la main aujourd'hui, pas envie d'inventer de nouveaux exemples surtout que ça fait exactement la même chose. 😊



Bon, je reconnais qu'on ne fait rien de bien excitant pour le moment. Mais maintenant vous savez au moins comment définir un widget central pour une QMainWindow, et ça mine de rien c'est important. 😊

Définition de la zone centrale (type MDI)

Les choses se compliquent un peu (mais pas trop 😊) si vous voulez créer un programme MDI... par exemple un éditeur de texte qui peut gérer plusieurs documents à la fois.

Nous allons utiliser pour cela une [QMdiArea](#), qui est une sorte de gros widget conteneur capable d'afficher plusieurs sous-fenêtres.

On peut se servir du QMdiArea comme de widget conteneur pour la zone centrale :

Code : C++ - [Sélectionner](#)

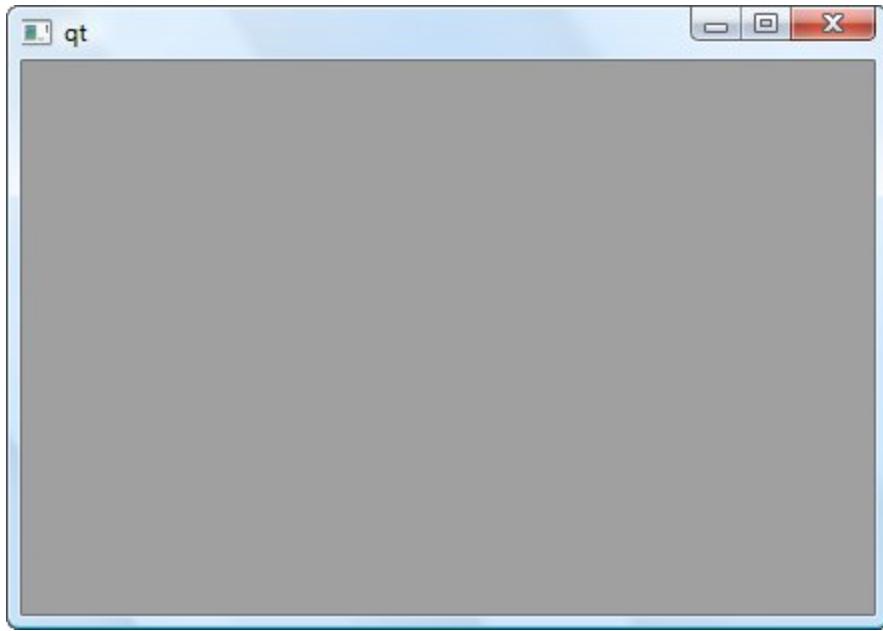
```

#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
    QMdiArea *zoneCentrale = new QMdiArea;
    setCentralWidget(zoneCentrale);
}

```

La fenêtre est maintenant prête à accepter des sous-fenêtres :



Si le fond gris par défaut ne vous plaît pas, vous pouvez changer le fond (en mettant une autre couleur ou une image) en appelant `setBackgroundColor()`.

On crée ces sous-fenêtres en appelant la méthode `addSubWindow()` du `QMdiArea`. Cette méthode attend en paramètre le widget que la sous-fenêtre doit afficher à l'intérieur.

Là encore, vous pouvez créer un `QWidget` générique qui contiendra d'autres widgets, eux-mêmes organisés selon un layout.

On va faire plus simple dans notre exemple : on va faire en sorte que les sous-fenêtres contiennent juste un `QTextEdit` (pour notre éditeur de texte) :

Code : C++ - [Sélectionner](#)

```
#include "FenPrincipale.h"

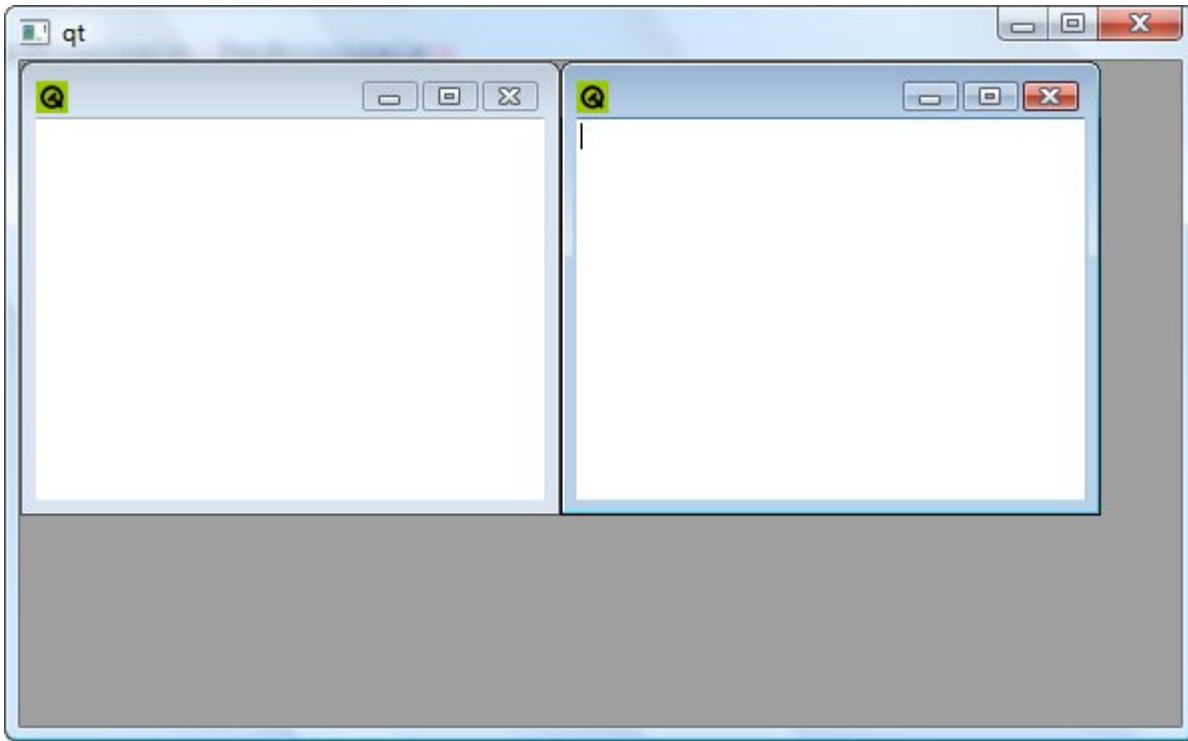
FenPrincipale::FenPrincipale()
{
    QMdiArea *zoneCentrale = new QMdiArea;

    QTextEdit *zoneTexte1 = new QTextEdit;
    QTextEdit *zoneTexte2 = new QTextEdit;

    QMdiSubWindow *sousFenetre1 = zoneCentrale->addSubWindow(zoneTexte1);
    QMdiSubWindow *sousFenetre2 = zoneCentrale->addSubWindow(zoneTexte2);

    setCentralWidget(zoneCentrale);
}
```

Résultat, on a une fenêtre principale qui contient plusieurs sous-fenêtres à l'intérieur :



Ces fenêtres peuvent étre réduites ou agrandies à l'intérieur même de la fenêtre principale.
On peut leur définir un titre et une icône avec les bonnes vieilles méthodes setWindowTitle, setWindowIcon, etc.

C'est quand même dingue tout ce qu'on peut faire en quelques lignes de code avec Qt ! 😊

Sur cet exemple, j'ai créé des "fenêtres-zones_de_texte", un peu comme j'avais fait des "fenêtres-boutons" dans les premiers chapitres sur Qt.

Bien sûr, dans la pratique, les sous-fenêtres seront peut-être un peu plus complexes. On n'utilisera pas un QTextEdit directement mais plutôt un QWidget qui contiendra un layout qui contiendra des widgets. Bref, vous m'avez compris. 😊

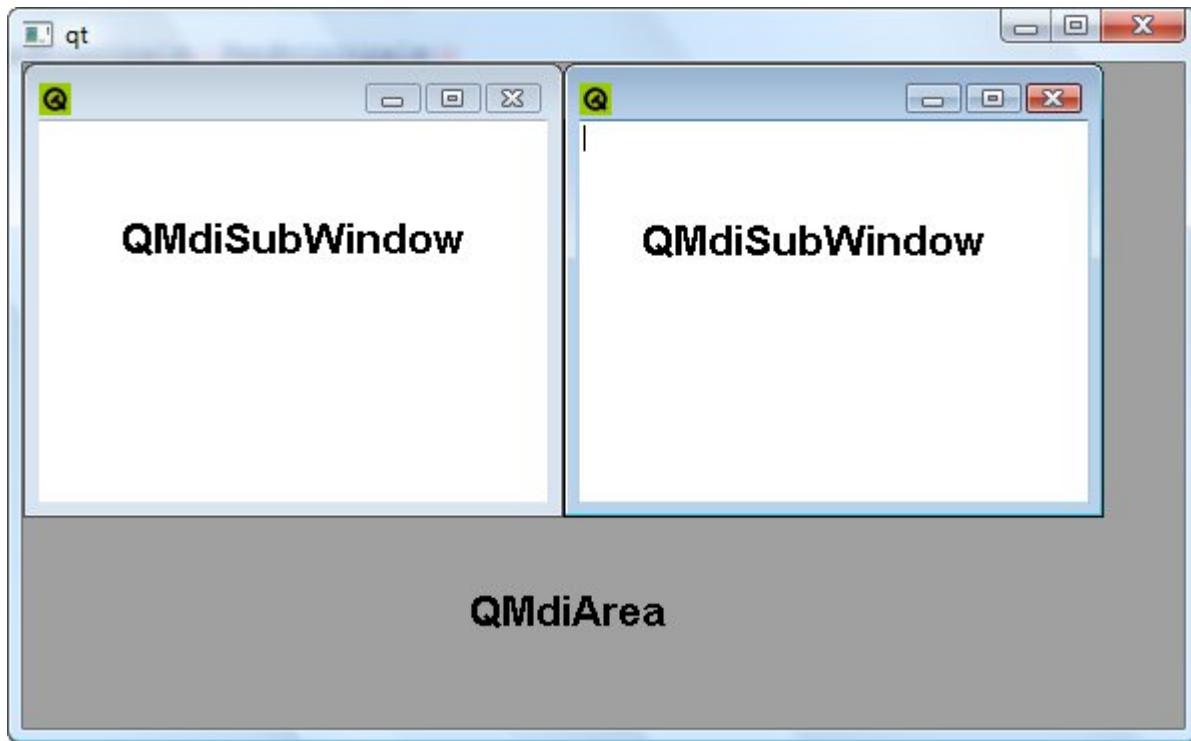
Vous remarquerez que addSubWindow() renvoie un pointeur sur une QMdiSubWindow : ce pointeur représente la sous-fenêtre qui a été créée. Ça peut étre une bonne idée de garder ce pointeur pour la suite. Vous pourrez ainsi supprimer la fenêtre en appelant removeSubWindow().

Sinon, sachez que vous pouvez retrouver à tout moment la liste des sous-fenêtres créées en appelant subWindowList(). Cette méthode renvoie la liste des QMdiSubWindow contenues dans la QMdiArea.

...

Ça va vous vous y retrouvez ? 😊

Allez un petit schéma pour étre sûr que vous avez compris :



On a donc une zone centrale QMdiArea qui contient plusieurs sous-fenêtres, représentées par des QMdiSubWindow. Chacune de ces sous-fenêtres est indépendante et peut contenir ses propres widgets.

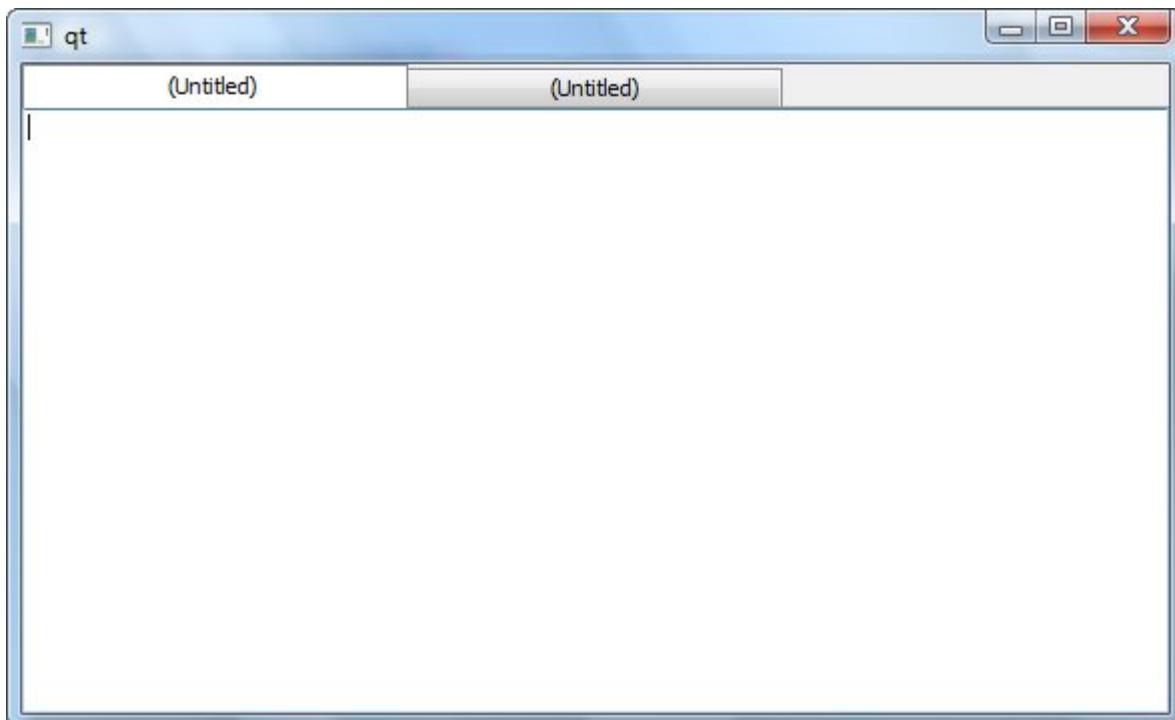
Notons que ce type de fenêtre MDI est de moins en moins utilisée. On a plutôt tendance aujourd'hui à recourir aux onglets lorsque plusieurs documents sont ouverts.

Ça tombe bien, les QMdiArea de Qt peuvent changer de mode d'affichage en une seule ligne grâce à setViewMode(). Par exemple, avec ce code :

Code : C++ - [Sélectionner](#)

```
zoneCentrale->setViewMode(QMdiArea::TabbedView);
```

... les sous-fenêtres seront organisées sous forme d'onglets :



Ça ne revient pas au même qu'un QTabWidget ça ? 😊

Si, en effet, on peut faire pareil avec un QTabWidget. L'avantage là c'est que Qt gère chaque onglet comme une fenêtre, et vous pouvez proposer à l'utilisateur de passer en un clic du mode de vue MDI classique au mode onglets. 😊

Les menus

La QMainWindow peut afficher une barre de menus, comme par exemple : Fichier, Edition, Affichage, Aide...
Comment fait-on pour les créer ?

Créer un menu pour la fenêtre principale

La barre de menus est accessible depuis la méthode menuBar(). Cette méthode renvoie un pointeur sur un QMenuBar, qui vous propose une méthode addMenu(). Cette méthode renvoie un pointeur sur le QMenu créé.

Puisqu'un petit code vaut tous les discours du monde, voici comment faire :

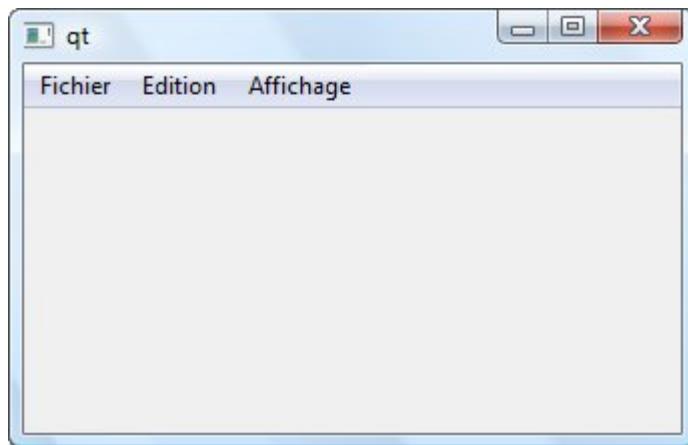
Code : C++ - [Sélectionner](#)

```
#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
    QMenu *menuFichier = menuBar()->addMenu( "&Fichier" );
    QMenu *menuEdition = menuBar()->addMenu( "&Edition" );
    QMenu *menuAffichage = menuBar()->addMenu( "&Affichage" );
}
```

Avec ça, nous avons créé 3 menus dont nous gardons les pointeurs (menuFichier, menuEdition, menuAffichage). Vous noterez qu'on utilise ici aussi le symbole & pour définir des raccourcis clavier (les lettres F, E et A seront donc des raccourcis vers leurs menus respectifs).

Nous avons maintenant 3 menus sur notre fenêtre :



Mais... ces menus n'affichent rien ! En effet, ils ne contiennent pour le moment aucun élément.

Création d'actions pour les menus

Un élément de menu est représenté par une action. C'est la classe [QAction](#) qui gère ça.

Pourquoi avoir créé une classe QAction au lieu de... je sais pas moi... QSubMenu pour dire "sous-menu" ?

En fait, les QAction sont des éléments de menu génériques. Ils peuvent être utilisés à la fois pour les menus et pour la barre d'outils. Par exemple, imaginons l'élément "Nouveau" qui permet de créer un nouveau document. On peut en général y accéder depuis 2 endroits différents :

- Le menu Fichier / Nouveau.
- Le bouton de la barre d'outils "Nouveau", généralement représenté par une icône de document vide.

Une seule QAction peut servir à définir ces 2 éléments à la fois.

Les développeurs de Qt se sont en effet rendus compte que les actions des menus étaient souvent dupliquées dans la barre d'outils, d'où la création de la classe QAction que nous réutiliserons lorsque nous créerons la barre d'outils.

Pour créer une action vous avez 2 possibilités :

- Soit vous la créez d'abord, puis vous créez l'élément de menu qui correspond.
- Soit vous créez l'élément de menu directement, et celui-ci vous renvoie un pointeur vers la QAction créée automatiquement.

Nous allons tester ces 2 possibilités.

Créer une QAction, puis créer l'élément de menu

Nous allons tout d'abord créer une QAction, puis nous l'ajouterons au menu "Fichier" :

Code : C++ - [Sélectionner](#)

```
#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
    QMenu *menuFichier = menuBar()->addMenu( "&Fichier" );

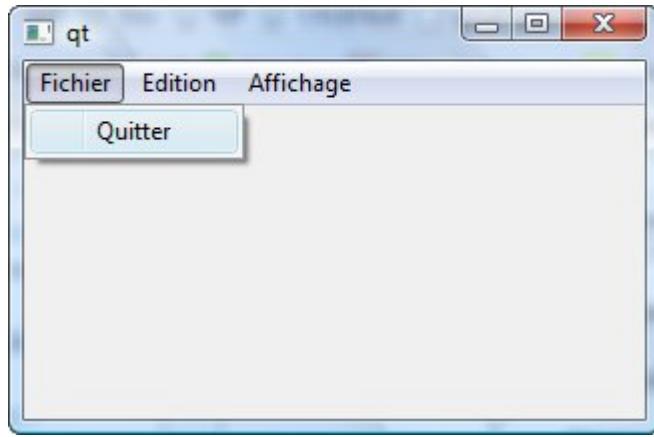
    QAction *actionQuitter = new QAction( "&Quitter", this );
    menuFichier->addAction(actionQuitter);

    QMenu *menuEdition = menuBar()->addMenu( "&Edition" );
    QMenu *menuAffichage = menuBar()->addMenu( "&Affichage" );

}
```

Dans l'exemple de code ci-dessus, nous créons d'abord une QAction correspondant à l'action "Quitter". Nous définissons en second paramètre de son constructeur un pointeur sur la fenêtre principale (this), qui servira de parent à l'action. Puis, nous ajoutons l'action au menu "Fichier".

Résultat, l'élément de menu est créé :



Créer l'élément de menu et récupérer la QAction

Il y a une autre façon de faire.

Parfois, vous trouverez que créer une QAction avant de générer l'élément de menu est un peu lourd. Dans ce cas, vous pouvez passer par une des versions surchargées de la méthode addAction :

Code : C++ - Sélectionner

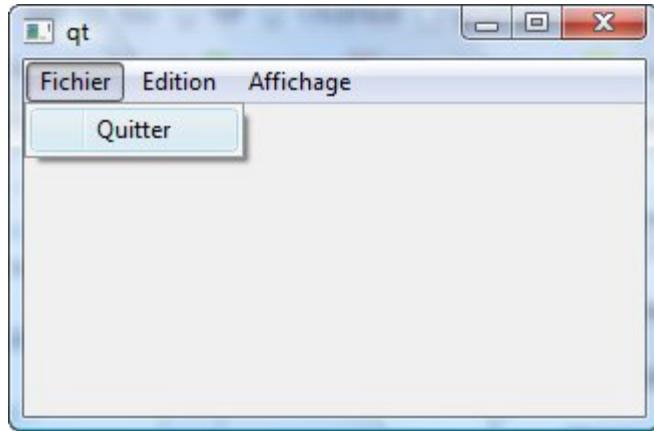
```
#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
    QMenu *menuFichier = menuBar()->addMenu( "&Fichier" );
    QAction *actionQuitter = menuFichier->addAction( "&Quitter" );

    QMenu *menuEdition = menuBar()->addMenu( "&Edition" );
    QMenu *menuAffichage = menuBar()->addMenu( "&Affichage" );

    connect(actionQuitter, SIGNAL(triggered()), qApp, SLOT(quit()));
}
```

Le résultat est strictement le même :



Les sous-menus

Les sous-menus sont gérés par la classe [QMenu](#).

Imaginons que nous voulions créer un sous-menu "Fichiers récents" au menu "Fichier". Ce sous-menu affichera une liste de fichiers

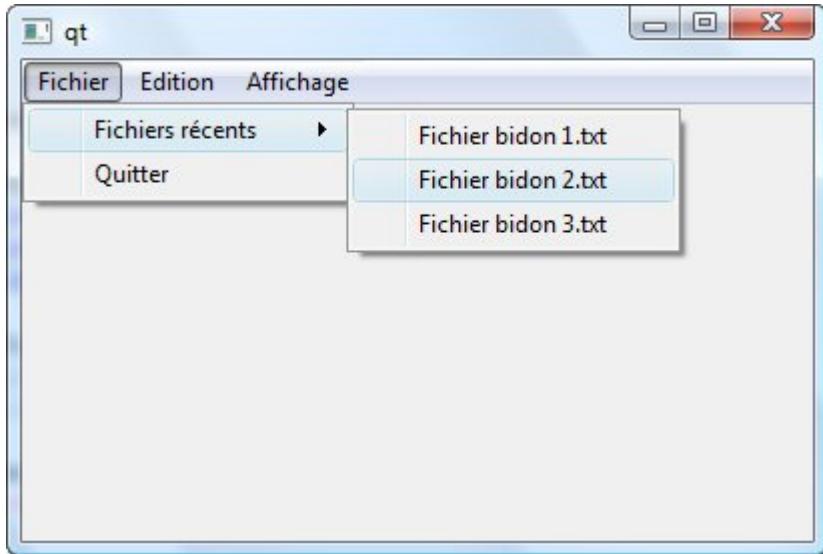
récemment ouverts par le programme (des fichiers bidons pour cet exemple).

Au lieu d'appeler addAction() de la QMenuBar, appelez cette fois addMenu() qui renvoie un pointeur vers un QMenu :

Code : C++ - Sélectionner

```
QMenu *fichiersRecents = menuFichier->addMenu("Fichiers &récents");
fichiersRecents->addAction("Fichier bidon 1.txt");
fichiersRecents->addAction("Fichier bidon 2.txt");
fichiersRecents->addAction("Fichier bidon 3.txt");
```

Vous voyez que j'ajoute ensuite de nouvelles actions pour peupler le sous-menu "Fichiers récents". Résultat :



Je n'ai pas récupéré de pointeur vers les QAction créées à chaque fois. J'aurais dû le faire si je voulais ensuite connecter les signaux des actions à des slots, mais je ne l'ai pas fait ici pour simplifier le code.

Vous pouvez créer des menus contextuels personnalisés de la même façon, avec des QMenu. Un menu contextuel est un menu qui s'affiche lorsqu'on fait un clic droit sur un widget. C'est un petit peu plus complexe. Je vous laisse lire la doc de QWidget à propos des menus contextuels pour savoir comment faire ça si vous en avez besoin.

Manipulations plus avancées des QAction

Une QAction est au minimum constituée d'un texte descriptif. Mais ce serait dommage de la limiter à ça. Voyons un peu ce qu'on peut faire avec les QAction...

Connecter les signaux et les slots

Le premier rôle d'une QAction est de générer des signaux, que l'on aura connectés à des slots.

La QAction propose plusieurs signaux intéressants. Le plus utilisé d'entre eux est triggered() qui indique que l'action a été choisie par l'utilisateur.

On peut connecter notre action "Quitter" au slot quit() de l'application :

Code : C++ - Sélectionner

```
connect(actionQuitter, SIGNAL(triggered()), qApp, SLOT(quit()));
```

Désormais, un clic sur "Fichier / Quitter" fermera l'application. 😊

Vous avez aussi un évènement `hovered()` qui s'active lorsqu'on passe la souris sur l'action. A tester !

Ajouter un raccourci

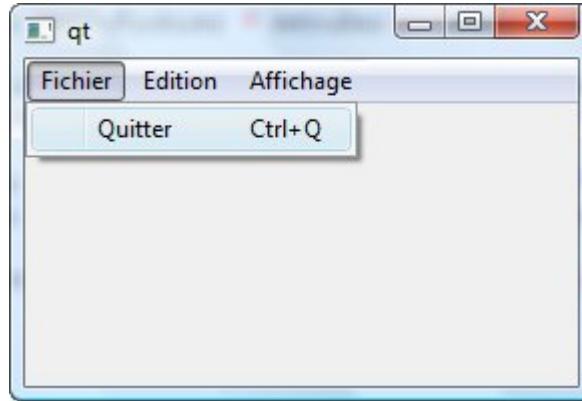
On peut définir un raccourci clavier pour l'action. On passe pour cela par la méthode `addShortcut()`.

Cette méthode peut être utilisée de plusieurs manières différentes. La technique la plus simple est de lui envoyer une `QKeySequence` représentant le raccourci clavier :

Code : C++ - [Sélectionner](#)

```
actionQuitter->setShortcut(QKeySequence("Ctrl+Q"));
```

Voilà, il suffit d'écrire dans le constructeur de la `QKeySequence` le raccourci approprié, Qt se chargera de comprendre le raccourci tout seul.



Vous pouvez faire le raccourci clavier Ctrl + Q n'importe où dans la fenêtre maintenant, cela activera l'action "Quitter" !

Sachez que `QKeySequence` accepte d'autres syntaxes :

Code : C++ - [Sélectionner](#)

```
actionQuitter->setShortcut(Qt::CTRL + Qt::Key_Q);
```

... créera le même raccourci "Ctrl + Q", sauf que cette fois nous sommes passés par des symboles pour le définir.

Vous pouvez aussi utiliser une séquence prédefinie, qui s'adapte en fonction des habitudes de l'OS.

Par exemple, la séquence prédefinie `QKeySequence::HelpContents` est faite pour représenter un raccourci clavier qui amène à l'aide.

Code : C++ - [Sélectionner](#)

```
actionQuitter->setShortcut(QKeySequence(QKeySequence::HelpContents));
```

Sous Windows, ce sera la touche F1, sous Mac OS X, ce sera le raccourci "Ctrl + ?".

Pour avoir la liste des séquences prédefinies, c'est dans la doc. 😊

Ajouter une icône

Chaque action peut avoir une icône.

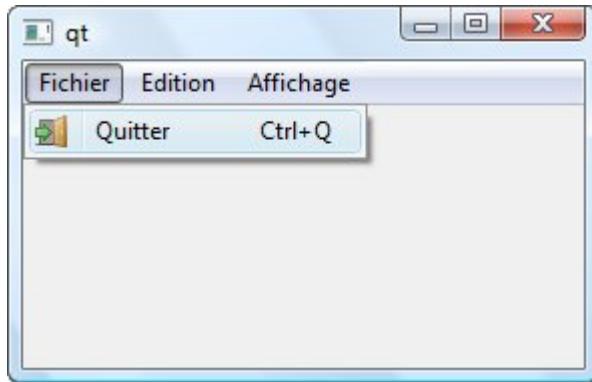
Lorsque l'action est associée à un menu, l'icône est affichée à gauche de l'élément de menu. Mais, souvenez-vous, une action peut aussi être associée à une barre d'outils comme on le verra plus tard. L'icône peut donc aussi être réutilisée dans la barre d'outils.

Pour ajouter une icône, appelez setIcon() et envoyez-lui un QIcon :

Code : C++ - Sélectionner

```
actionQuitter->setIcon(QIcon( "quitter.png" ));
```

Résultat :



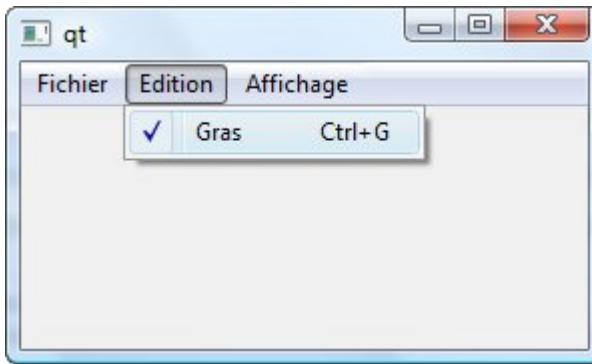
Pouvoir cocher une action

Lorsqu'une action peut avoir 2 états (activée, désactivée), vous pouvez la rendre "cochable" grâce à setCheckable(). Imaginons par exemple le menu Edition / Gras :

Code : C++ - Sélectionner

```
actionGras->setCheckable(true);
```

Le menu a maintenant 2 états et peut être précédé d'une case à cocher :



On vérifiera dans le code si l'action est cochée avec isChecked().



Lorsque l'action est utilisée sur une barre d'outils, le bouton reste enfoncé lorsque l'action est "cochée". C'est ce que vous avez l'habitude de voir dans un traitement de texte par exemple (cf image ci-contre 😊).

Ah, puisqu'on parle de barre d'outils, il serait temps d'apprendre à en créer une !

La barre d'outils

La barre d'outils est généralement constituée d'icônes et située sous les menus.

Avec Qt, la barre d'outils utilise des actions pour construire chacun des éléments de la barre. Étant donné que vous avez appris à manipuler des actions juste avant, vous devriez donc être capables de créer une barre d'outils très rapidement. 😊

Pour ajouter une barre d'outils, vous devez tout d'abord appeler la méthode addToolBar() de la QMainWindow. Il faudra donner un nom à la barre d'outils, même si celui-ci ne s'affiche pas.

Vous récupérez un pointeur vers la QToolBar :

Code : C++ - [Sélectionner](#)

```
QToolBar *toolBarFichier = addToolBar("Fichier");
```

Maintenant que nous avons notre QToolBar, nous pouvons commencer !

Ajouter une action

Le plus simple est d'ajouter une action à la QToolBar. On utilise comme pour les menus une méthode appelée addAction() qui prend comme paramètre une QAction.

Le gros intérêt que vous devriez saisir maintenant, c'est que vous pouvez réutiliser ici vos QAction créées pour les menus !

Code : C++ - [Sélectionner](#)

```
#include "FenPrincipale.h"

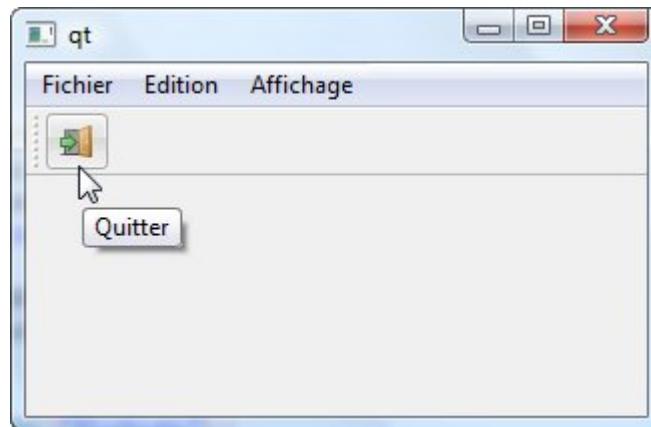
FenPrincipale::FenPrincipale()
{
    // Crédation des menus
    QMenu *menuFichier = menuBar()->addMenu("&Fichier");
    QAction *actionQuitter = menuFichier->addAction("&Quitter");
    actionQuitter->setShortcut(QKeySequence("Ctrl+Q"));
    actionQuitter->setIcon(QIcon("quitter.png"));

    QMenu *menuEdition = menuBar()->addMenu("&Edition");
    QMenu *menuAffichage = menuBar()->addMenu("&Affichage");

    // Crédation de la barre d'outils
    QToolBar *toolBarFichier = addToolBar("Fichier");
    toolBarFichier->addAction(actionQuitter);

    connect(actionQuitter, SIGNAL(triggered()), qApp, SLOT(quit()));
}
```

Dans ce code, on voit qu'on crée d'abord une QAction pour un menu (ligne 7), puis plus loin on réutilise cette action pour l'ajouter à la barre d'outils (ligne 16).



Comme l'action est la même que celle utilisée pour le menu "Quitter", on retrouve :

- Son icône : ici affichée dans la barre d'outils.
- Son texte : ici affiché lorsqu'on pointe sur l'icône.
- Son action : elle est toujours connectée au slot quit() de l'application, ce qui a pour effet de mettre fin au programme.

Et voilà comment Qt fait d'une pierre deux coups grâce aux QAction ! 😊

Ajouter un widget

Les barres d'outils contiennent le plus souvent des QAction, mais il arrivera que vous ayez besoin d'insérer des éléments plus complexes.

La QToolBar gère justement tous types de widgets.

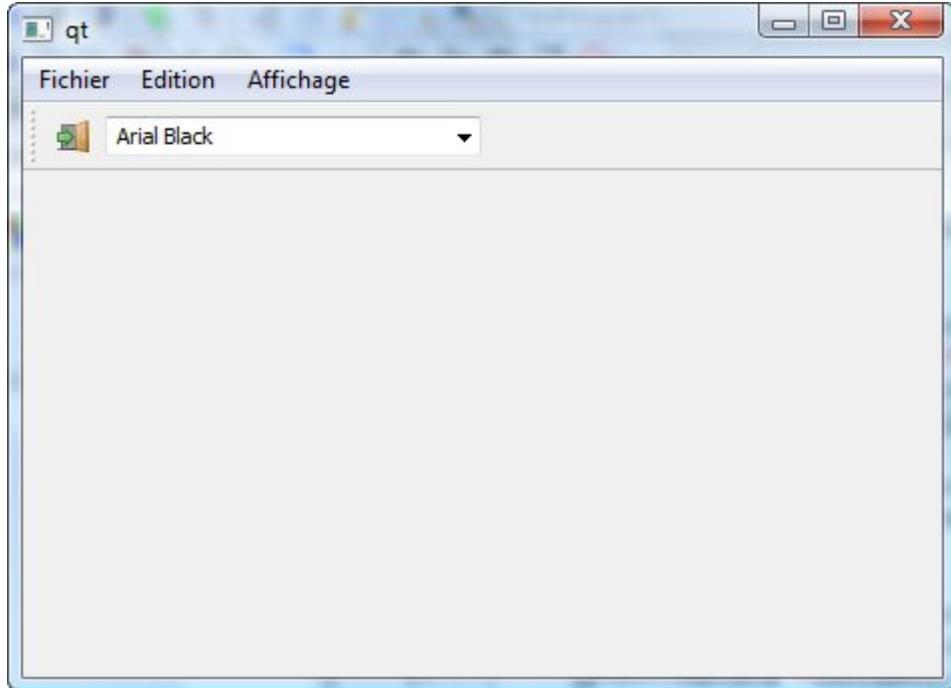
Vous pouvez ajouter des widgets avec la méthode addWidget(), comme vous le faisiez avec les layouts :

Code : C++ - [Sélectionner](#)

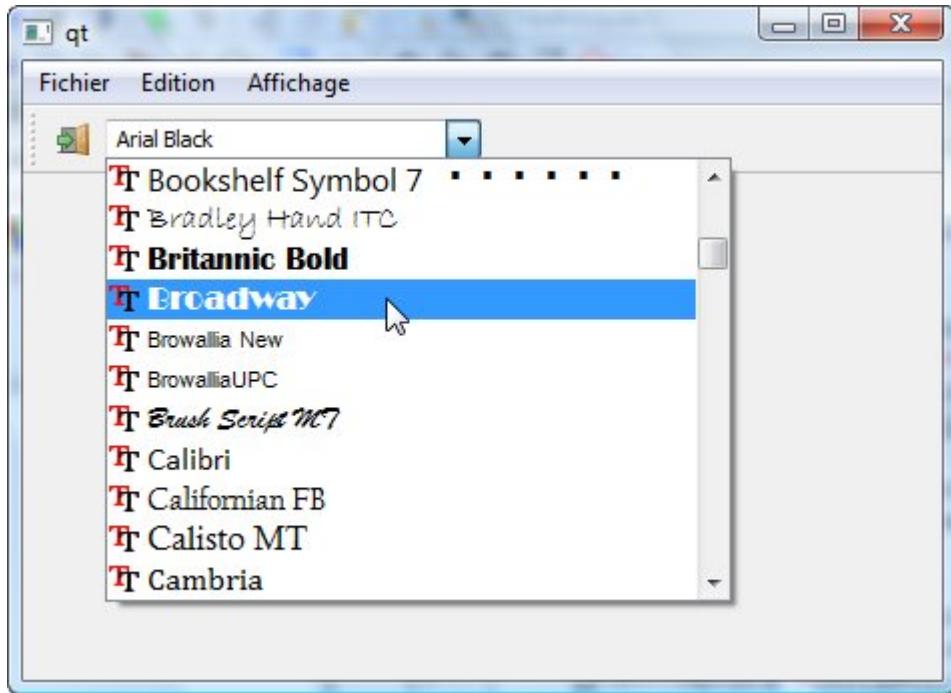
```
QFontComboBox *choixPolice = new QFontComboBox;
toolBarFichier->addWidget(choixPolice);
```

Ici, on insère une liste déroulante (plus précisément une QFontComboBox, une liste déroulante spécialisée dans le choix d'une police).

Le widget s'insère alors dans la barre d'outils :



... et vous pouvez l'utiliser comme n'importe quel widget normal, gérer ses signaux, ses slots, etc.



La méthode addWidget() crée une QAction automatiquement. Elle renvoie un pointeur vers cette QAction créée. Ici, on n'a pas récupéré le pointeur, mais vous pouvez le faire si vous avez besoin d'effectuer des opérations ensuite sur la QAction.

Ajouter un séparateur

Si votre barre d'outils commence à comporter trop d'éléments, ça peut être une bonne idée de les séparer. C'est pour cela que Qt propose des separators (séparateurs).

Cela vous permettra par exemple de regrouper les boutons "Annuler" et "Rétablir", pour ne pas les confondre avec les boutons "Gras", "Italique", "Souligné".

Il suffit simplement d'appeler la méthode addSeparator() au moment où vous voulez insérer un séparateur :

Code : C++ - Sélectionner

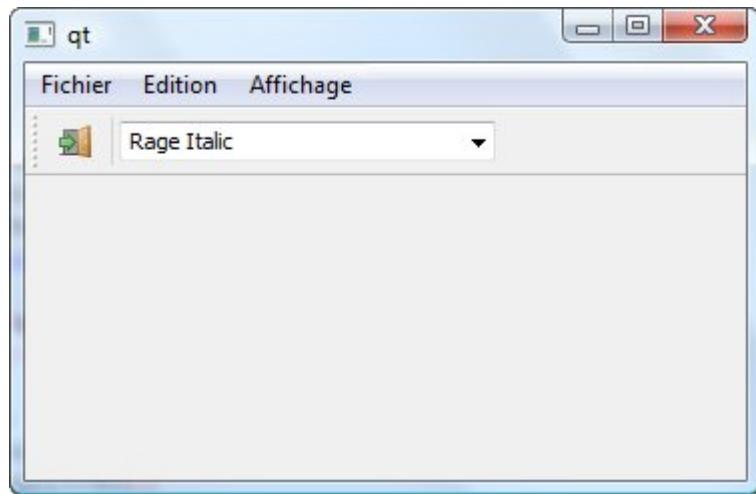
```
#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
    // Cr ation des menus
    QMenu *menuFichier = menuBar()->addMenu("&Fichier");
    QAction *actionQuitter = menuFichier->addAction("&Quitter");
    actionQuitter->setShortcut(QKeySequence("Ctrl+Q"));
    actionQuitter->setIcon(QIcon("quitter.png"));

    QMenu *menuEdition = menuBar()->addMenu("&Edition");
    QMenu *menuAffichage = menuBar()->addMenu("&Affichage");

    // Cr ation de la barre d'outils
    QToolBar *toolBarFichier = addToolBar("Fichier");
    toolBarFichier->addAction(actionQuitter);
    toolBarFichier->addSeparator();
    QFontComboBox *choixPolice = new QFontComboBox;
    toolBarFichier->addWidget(choixPolice);

    connect(actionQuitter, SIGNAL(triggered()), qApp, SLOT(quit()));
}
```

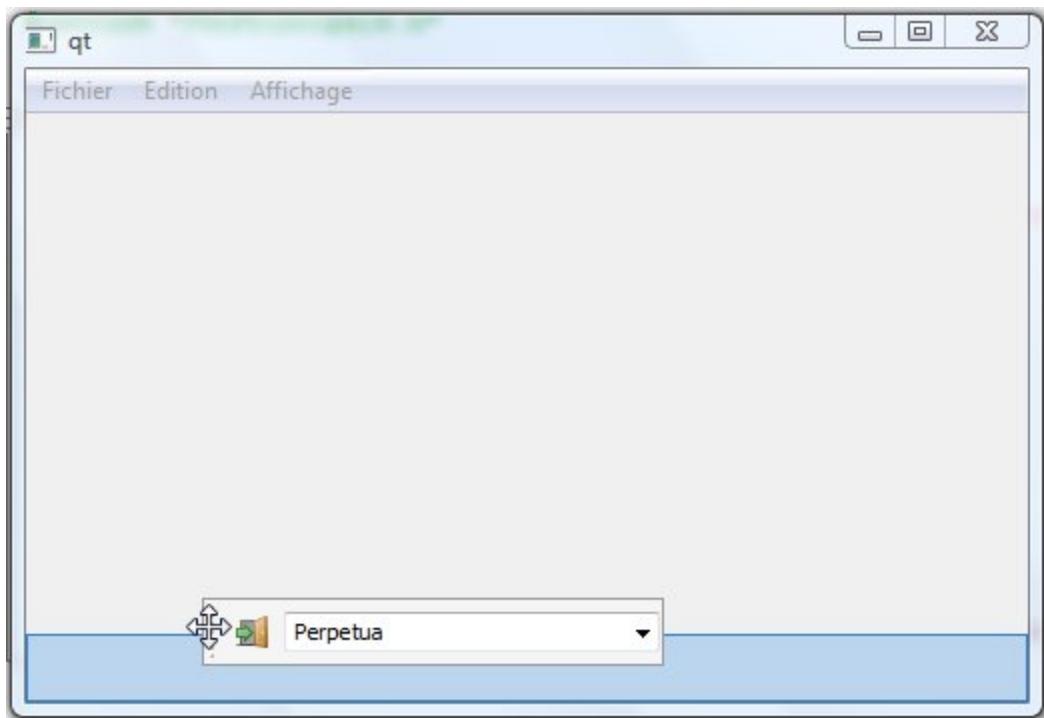


Notez le séparateur entre le bouton Quitter et la liste déroulante

Vous pouvez aussi insérer un séparateur à une position précise grâce à `insertSeparator()`.

Plus d'options pour la barre d'outils

Par défaut, la barre d'outils est déplaçable. Vous pouvez la placer en haut, sur les côtés ou en bas, et vous pouvez même en faire une mini-fenêtre indépendante :



Si vous souhaitez éviter que la barre d'outils soit déplaçable, modifiez sa propriété `movable`.

Si vous souhaitez juste éviter qu'on puisse créer une mini-fenêtre indépendante, modifiez sa propriété `floatable`.

Les docks

Les docks sont des mini-fenêtres que l'on peut généralement déplacer à notre guise dans la fenêtre principale. Vous avez l'habitude d'en voir dans des programmes complexes comme Photoshop ou votre IDE (Code::Blocks en utilise un pour afficher la liste des fichiers du projet par exemple).

Créer un QDockWidget

Créez un [QDockWidget](#) et affectez-lui un titre pour commencer. Indiquez que la fenêtre principale (this) est son widget parent en second paramètre :

Code : C++ - [Sélectionner](#)

```
QDockWidget *dock = new QDockWidget("Palette", this);
```

Puis, placez ce dock sur la fenêtre principale à l'aide de la méthode addDockWidget() :

Code : C++ - [Sélectionner](#)

```
addDockWidget(Qt::LeftDockWidgetArea, dock);
```

Comme vous pouvez le voir, le premier paramètre permet d'indiquer à quel endroit le dock doit être placé. On peut le mettre en haut, en bas, à gauche ou à droite. On peut même faire flotter le dock comme une mini-fenêtre.

Par défaut, l'utilisateur a le droit de déplacer le dock. Si vous ne souhaitez pas qu'il puisse le faire, faites appel à la méthode setFeatures() du QDockWidget. Vous pouvez tout personnaliser à partir de là.

De manière générale, évitez d'empêcher à l'utilisateur de réorganiser les docks. La puissance des docks c'est justement que l'on peut les réorganiser selon ses préférences. Ce serait dommage de perdre en fonctionnalités.

Peupler le QDockWidget

Notre QDockWidget est créé mais il ne contient rien.

Pour le peupler en widgets, il faut d'abord créer un widget conteneur et indiquer que ce widget gère le contenu du QDockWidget.

Code : C++ - [Sélectionner](#)

```
QWidget *contenuDock = new QWidget;
dock->setWidget(contenuDock);
```

Après, il n'y a plus qu'à placer des widgets et / ou des layouts dans contenuDock, comme si c'était une fenêtre. Je mets quelques widgets au pif, ne faites pas spécialement attention au détail de ce code :

Code : C++ - [Sélectionner](#)

```
QPushButton *crayon = new QPushButton("Crayon");
QPushButton *pinceau = new QPushButton("Pinceau");
QPushButton *feutre = new QPushButton("Feutre");
QLabel *labelEpaisseur = new QLabel("Epaisseur :");
QSpinBox *epaisseur = new QSpinBox;

QVBoxLayout *dockLayout = new QVBoxLayout;
dockLayout->addWidget(crayon);
dockLayout->addWidget(pinceau);
dockLayout->addWidget(feutre);
dockLayout->addWidget(labelEpaisseur);
dockLayout->addWidget(epaisseur);

contenuDock->setLayout(dockLayout);
```

Code complet

Voici un code complet qui utilise des menus, une barre d'outils et un dock. Il est un peu long mais ne vous laissez pas impressionner,

il n'y a rien de nouveau ni de compliqué.

Code : C++ - Sélectionner

```
#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
    // Cr ation des menus
    QMenu *menuFichier = menuBar()->addMenu( "&Fichier" );
    QAction *actionQuitter = menuFichier->addAction( "&Quitter" );
    actionQuitter->setShortcut(QKeySequence( "Ctrl+Q" ));
    actionQuitter->setIcon(QIcon( "quitter.png" ));

    QMenu *menuEdition = menuBar()->addMenu( "&Edition" );
    QMenu *menuAffichage = menuBar()->addMenu( "&Affichage" );

    // Cr ation de la barre d'outils
    QToolBar *toolBarFichier = addToolBar( "Fichier" );
    toolBarFichier->addAction(actionQuitter);
    toolBarFichier->addSeparator();
    QFontComboBox *choixPolice = new QFontComboBox;
    toolBarFichier->addWidget(choixPolice);

    // Cr ation des docks
    QDockWidget *dock = new QDockWidget( "Palette" , this );
    addDockWidget(Qt::LeftDockWidgetArea, dock);

    QWidget *contenuDock = new QWidget;
    dock->setWidget( contenuDock );

    QPushButton *crayon = new QPushButton( "Crayon" );
    QPushButton *pinceau = new QPushButton( "Pinceau" );
    QPushButton *feutre = new QPushButton( "Feutre" );
    QLabel *labelEpaisseur = new QLabel( "Epaisseur :" );
    QSpinBox *epaisseur = new QSpinBox;

    QVBoxLayout *dockLayout = new QVBoxLayout;
    dockLayout->addWidget(crayon);
    dockLayout->addWidget(pinceau);
    dockLayout->addWidget(feutre);
    dockLayout->addWidget(labelEpaisseur);
    dockLayout->addWidget(epaisseur);

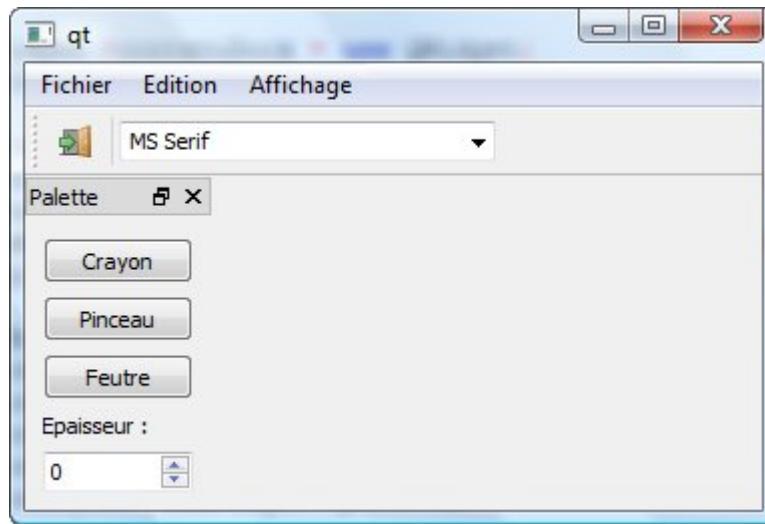
    contenuDock->setLayout(dockLayout);

    // Cr ation de la zone centrale
    QWidget *zoneCentrale = new QWidget;
    setCentralWidget(zoneCentrale);

    connect(actionQuitter, SIGNAL(triggered()), qApp, SLOT(quit()));
}
```

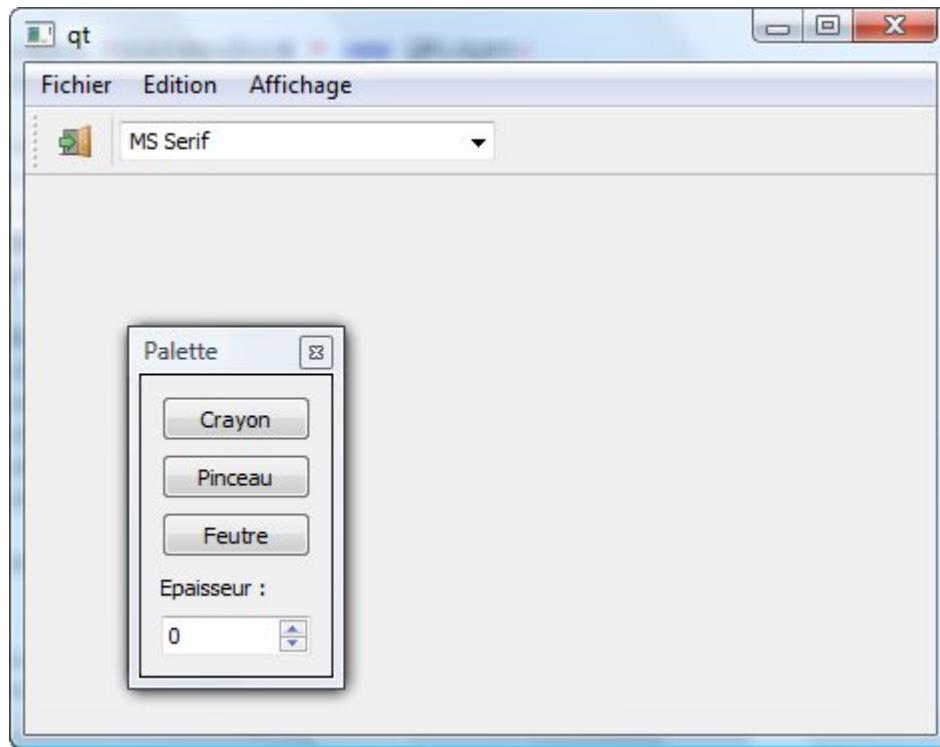
J'ai volontairement mis des commentaires pour séparer les différentes sections de la génération de la fenêtre.

Voil  le r sultat :



Le dock est placé à gauche comme nous l'avons demandé, mais nous pouvons le changer de côté.

D'autre part, nous pouvons faire sortir le dock et nous en servir comme d'une mini-fenêtre indépendante :



Bien sûr, cet exemple est à améliorer : il vaudrait mieux afficher des icônes sur les boutons plutôt que du texte, c'est plus intuitif et plus habituel. 😊

La barre d'état

La barre d'état est une petite barre affichée en bas de la fenêtre. Elle indique ce qu'est en train de faire l'application.

Par exemple, un navigateur web comme Firefox affiche le message "Terminé" lorsque la page web a été chargée. Lorsque la page est en cours de chargement, une barre de progression apparaît à cet emplacement.

La barre d'état est automatiquement créée et renournée par la méthode statusBar() de la QMainWindow. Celle-ci renvoie un pointeur vers une QStatusBar que vous devez conserver :

Code : C++ - Sélectionner

```
QStatusBar *barreEtat = statusBar();
```

Notre barre d'état est créée ! 😊

Maintenant, que peut-on faire dedans ?

Il faut savoir qu'une barre d'état peut afficher 3 types de messages différents :

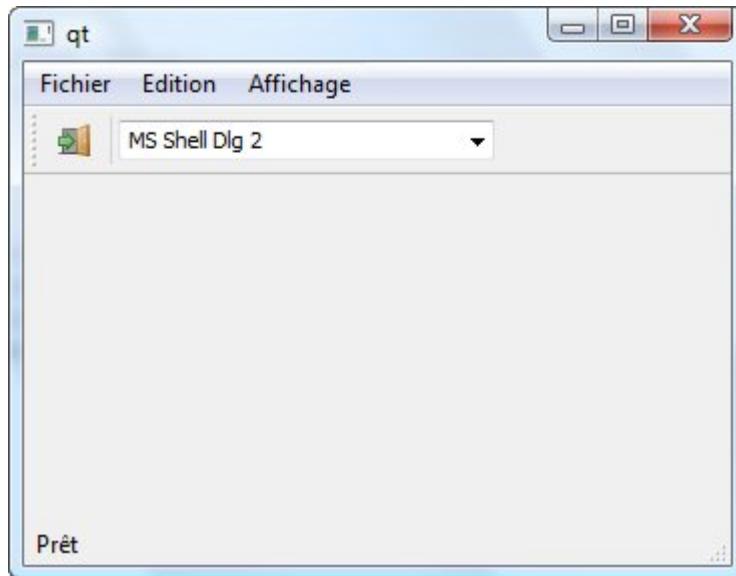
- Un message temporaire : il est affiché brièvement par-dessus tous les messages normaux.
- Un message normal : il est affiché tout le temps, sauf quand un message temporaire est affiché.
- Un message permanent : il est affiché tout le temps, même quand un message temporaire est affiché.

Les messages temporaires

C'est le plus simple : appelez la méthode showMessage() et indiquez le message à afficher. Par exemple :

Code : C++ - Sélectionner

```
barreEtat->showMessage( "Prêt" );
```



Vous pouvez indiquer en second paramètre la durée d'affichage du message en millisecondes, avant qu'il disparaisse. Exemple :

Code : C++ - Sélectionner

```
barreEtat->showMessage( "Le fichier a été sauvegardé" , 2000 );
```

Ce message restera affiché 2 secondes.

Les messages normaux et permanents

Pour les messages normaux et permanents, c'est un peu plus compliqué : il faut utiliser des widgets. Vous pouvez placer a priori n'importe quel widget dans cette zone, les plus courants étant les QLabel et les QProgressBar.

Utilisez :

- `addWidget()` : pour afficher un widget normal.
- `addPermanentWidget()` : pour afficher un widget permanent.

Exemple de widget normal affiché en barre d'état :

Code : C++ - [Sélectionner](#)

```
QProgressBar *progression = new QProgressBar;  
barreEtat->addWidget(progression);
```

Voilà, avec ça vous pouvez faire tout ce que vous voulez. 😊

Les status tips des QAction

Les QAction disposent d'une propriété dont je ne vous ai pas parlé jusqu'ici : `statusTip`.

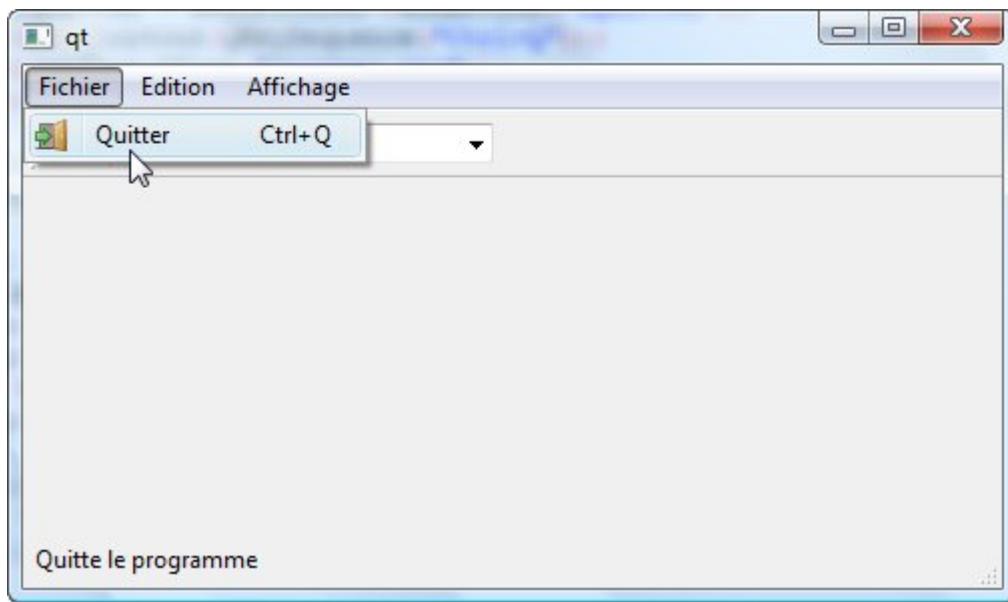
Elle permet d'indiquer un message qui doit s'afficher dans la barre d'état lorsqu'on pointe sur une action. Cela permet de donner de plus amples indications sur le rôle d'un élément de menu par exemple.

Mettons en place un `statusTip` sur l'action "Quitter" :

Code : C++ - [Sélectionner](#)

```
actionQuitter->setStatusTip("Quitte le programme");
```

Et voici ce que fait le `statusTip`, lorsqu'on pointe sur l'élément de menu :



Vous pouvez voir que la barre d'état affiche le `statusTip` lorsque la souris pointe sur l'action. 😊

Pfiou ! Ce n'est pas la fenêtre principale pour rien : c'est fou tout ce qu'on peut mettre dedans. A tel point qu'on se demande des fois s'il reste de la place pour afficher le widget central. 😮

La QMainWindow offre beaucoup de possibilités, mais ne vous sentez pas obligés de toutes les utiliser. Au contraire, ne faites appel qu'à ce qui vous sert, et ne surchargez pas inutilement la fenêtre principale. Sinon, vos utilisateurs mettront du temps avant d'arriver à la maîtriser, ce qui n'est jamais très bon.

Exercice : créer un éditeur de texte

Je crois que l'exercice de ce chapitre est tout trouvé : je vous propose de faire un éditeur de texte (en mode MDI, of course ). En effet, un éditeur de texte contient un peu tout ce qu'on vient d'apprendre :

- Des menus
- Une barre d'outils
- Une barre d'état
- Un mode multi-documents (MDI)

Et vous pouvez même ajouter des docks si vous leur trouvez une utilité. 

Bien que classique, c'est un très bon exercice. Il faudra bien vous organiser. Pour ce qui est de l'écriture des fichiers, utilisez [QFile](#). Bon courage !

Traduire son programme avec Qt Linguist

Si vous avez de l'ambition pour vos programmes, vous aurez peut-être envie un jour de les traduire dans d'autres langues. En effet, ce serait dommage de limiter votre programme seulement aux francophones, il y a certainement de nombreuses autres personnes qui aimeraient pouvoir en profiter !

La traduction d'applications n'est normalement pas une chose facile. D'ailleurs, il ne s'agit pas seulement de traduire des mots ou des phrases. Il ne suffit pas toujours de dire "Ouvrir" = "Open".

En effet, on parle des milliers de langues et dialectes différents sur notre bonne vieille planète. Certaines ressemblent au français, certaines écrivent de droite à gauche (l'arabe par exemple), d'autres ont des accents très particuliers que nous n'avons l'habitude d'utiliser (le ñ espagnol par exemple). Et je vous parle même pas des caractères hébreïques et japonais. 

On ne parle donc pas seulement de traduction mais de localisation. Il faut que notre programme puisse s'adapter avec les habitudes de chaque langue. Il faut que les phrases soient écrites de droite à gauche si nécessaire.

C'est là que Qt excelle et vous simplifie littéralement la tâche. Tout est prévu. Tous les outils pour traduire au mieux vos applications sont installés de base.

Comment ça fonctionne ? Nous allons voir ça. 

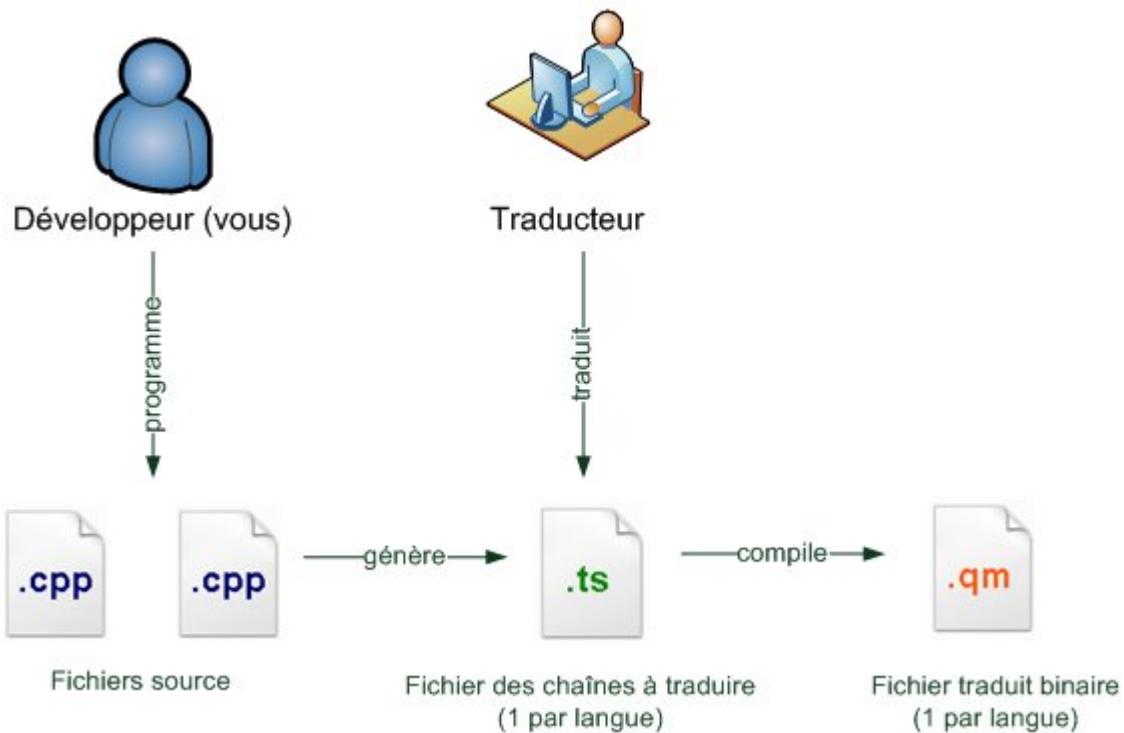
Les étapes de la traduction

La traduction de programmes Qt est un processus bien pensé... mais encore faut-il comprendre comment ça fonctionne. 

Qt suppose que les développeurs (vous) ne sont pas des traducteurs. Il suppose donc que ce sont 2 personnes différentes. Tout a été fait pour que les traducteurs, même si ce ne sont pas des informaticiens, soient capables de traduire votre programme.

Dans la pratique, si c'est un petit projet personnel, vous serez peut-être aussi le traducteur de votre programme. Mais nous allons supposer ici que le traducteur est une autre personne.

Je vous propose de regarder ce schéma de mon crû qui résume grosso modo les étapes de la traduction :



1. Tout d'abord, il y a le développeur. C'est vous. Vous écrivez normalement votre programme, en rédigeant les messages dans le code source dans votre langue maternelle (le français). Bref, rien ne change, à part un ou deux petits détails dont on reparlera dans la prochaine sous-partie.
2. Puis, vous générez un fichier contenant les chaînes à traduire. Un programme livré avec Qt le fait automatiquement pour vous. Ce fichier porte l'extension .ts, et est généralement de la forme : nomduprogramme_langue.ts. Par exemple, pour le ZeroClassGenerator, ça donne quelque chose comme zeroclassname_en.ts pour la traduction anglaise, zeroclassname_es.ts pour la traduction espagnole, etc. Il faut connaître le symbole à 2 lettres de la langue de destination pour donner un nom correct au fichier .ts. Généralement c'est la même chose que les extensions des noms de domaine : fr (français), pl (polonais), ru (russe)...
3. Le traducteur récupère le ou les fichiers .ts à traduire (un par langue). Il les traduit via le programme Qt Linguist qu'on découvrira dans quelques minutes.
4. Une fois que le traducteur a fini, il retourne les fichiers .ts traduits au développeur, qui les "compile" en fichiers .qm binaires. La différence entre un .ts et un .qm, c'est un peu comme la différence entre un .cpp (la source) et un .exe (le programme binaire final). Le .qm contenant les traductions au format binaire, Qt pourra le charger et le lire très rapidement lors de l'exécution du programme, ce qui fait qu'on ne sentira pas de ralentissement si on charge une version traduite du programme.

Je vous propose de découvrir pas à pas chacune de ces étapes dans ce chapitre. ☺

Nous allons commencer par vous, le développeur. Que faut-il faire de spécial lorsque vous écrivez le code source du programme ?

Préparer son code à une traduction

La toute première étape de la traduction consiste à écrire son code de manière adaptée, afin que des traducteurs puissent ensuite récupérer tous les messages à traduire.

Utilisez `QString` pour manipuler des chaînes de caractères

Comme vous le savez déjà, Qt utilise exclusivement sa classe `QString` pour gérer les chaînes de caractères. Cette classe, très complète, gère nativement l'Unicode.

L'Unicode est une norme qui indique comment sont gérés les caractères à l'intérieur de l'ordinateur. Elle permet à un ordinateur d'afficher sans problème tous types de caractères, en particulier les caractères étrangers.

[En savoir plus sur Unicode.](#)

QString n'a donc aucun problème pour gérer des alphabets cyrilliques ou arabes.

C'est pourquoi il est recommandé, si votre application est susceptible d'être traduite, d'utiliser toujours des QString pour manipuler des chaînes de caractères.

Code : C++ - [Sélectionner](#)

```
QString chaine = "Bonjour"; // Bon : adapté pour la traduction  
char chaine[] = "Bonjour"; // Mauvais : inadapté pour la traduction
```

Voilà, c'est juste un conseil que je vous donne là : de manière générale, utilisez autant que possible des QString. Evitez les tableaux de char.

Faites passer les chaînes à traduire par la méthode tr()

Utilisation basique

La méthode tr() permet d'indiquer qu'une chaîne devra être traduite. Par exemple, avant vous faisiez :

Code : C++ - [Sélectionner](#)

```
quitter = new QPushButton("&Quitter");
```

Cela ne permettra pas de traduire le texte du bouton. En revanche, si vous faites d'abord appel à la méthode tr() :

Code : C++ - [Sélectionner](#)

```
quitter = new QPushButton(tr("&Quitter"));
```

... alors le texte pourra être traduit ensuite. 😊

Vous rédigez donc les textes de votre programme dans votre langue maternelle (ici le français) en les entourant d'un tr().

La méthode tr() est définie dans QObject. C'est donc une méthode statique dont héritent toutes les classes de Qt, puisqu'elles dérivent de QObject. Dans la plupart des cas, écrire tr() devrait donc fonctionner. Si toutefois vous n'êtes pas dans une classe qui hérite de QObject, il faudra faire précéder tr() de QObject::, comme ceci :

```
quitter = new QPushButton(QObject::tr("&Quitter"));
```

Facultatif : ajouter un message de contexte

Parfois, il arrivera que le texte à traduire ne soit pas suffisamment explicite à lui tout seul, ce qui rendra difficile sa traduction pour le traducteur qui ne le verra pas dans son contexte.

Vous pouvez ajouter en second paramètre un message pour expliquer le contexte au traducteur.

Code : C++ - [Sélectionner](#)

```
quitter = new QPushButton(tr("&Quitter", "Utilisé pour le bouton de fermeture"));
```

Ce message ne sera pas affiché dans votre programme : il aidera juste le traducteur à comprendre ce qu'il doit traduire. En effet, dans certaines langues "Quitter" se traduit peut-être de plusieurs manières différentes. Avec le message de contexte, le traducteur saura comment bien traduire le mot.

En général, le message de contexte n'est pas obligatoire.

Parfois cependant, il devient vraiment indispensable. Par exemple quand on doit traduire un raccourci clavier (eh oui !) :

Code : C++ - [Sélectionner](#)

```
actionQuitter->setShortcut(QKeySequence(tr("Ctrl+Q"), "Raccourci clavier pour quitter"));
```

Le traducteur pourra ainsi traduire la chaîne en "Ctrl+S" si c'est le raccourci adapté dans la langue de destination.

Facultatif : gestion des pluriels

Parfois, une chaîne doit être écrite différemment selon le nombre d'éléments.

Imaginons un programme qui lit le nombre de fichiers dans un répertoire. Il affiche le message "Il y a X fichier(s)". Comment traduire ça correctement ?

En fait, ça dépend vraiment des langues. Le pluriel est géré différemment en anglais et en français par exemple :

Nombre	Français	Anglais
0	Il y a 0 fichier.	There are 0 files.
1	Il y a 1 fichier.	There is 1 file.
2	Il y a 2 fichiers.	There are 2 files.

Comme vous pouvez le voir, en français on dit "Il y a 0 fichier.", et en anglais "There are 0 files.". Les anglais mettent du pluriel quand le nombre d'éléments est à 0 !

Et encore, je vous parle pas des russes, qui ont un pluriel pour quand il y a 2 éléments et un autre pluriel pour quand il y en a 3 ! (je simplifie parce qu'en fait c'est même un peu plus compliqué que ça encore)



J'avais jamais m'en sortir avec tous ces cas à gérer ! 😱

Eh bien si, assurez-vous. Qt est capable de gérer tous les pluriels pour chacune des langues.

Ce sera bien entendu le rôle du traducteur ensuite de traduire ces pluriels correctement.

Comment faire ? Utilisez le 3ème paramètre facultatif qui indique la cardinalité (le nombre d'éléments).

Exemple :

Code : C++ - [Sélectionner](#)

```
tr("Il y a %n fichier(s)", "", nombreFichiers);
```

Si on ne veut pas indiquer de contexte comme moi dans ce cas, on est quand même obligé d'envoyer une chaîne vide pour utiliser le 3ème paramètre (c'est la règle des paramètres facultatifs en C++).

Qt utilisera automatiquement la bonne version du texte traduit selon la langue de destination et le nombre d'éléments. Par ailleurs, le %n sera remplacé par le nombre indiqué en 3ème paramètre.

Bon, avec tout ça, votre programme est codé de manière à pouvoir être traduit.
Maintenant, comment se passe l'étape de la traduction ?

Créer les fichiers de traduction .ts

Nous avons maintenant un programme qui fait appel à la méthode tr() pour désigner toutes les chaînes de caractères qui doivent

être traduites.

On va prendre l'exemple de notre TP ZeroClassGenerator. Je l'ai adapté pour qu'il utilise des `tr()`.
On souhaite que ZeroClassGenerator soit traduit dans les langues suivantes :

- Anglais
- Espagnol

Nous devons générer 2 fichiers de traduction :

- `zeroclassnamesgenerator_en.ts` pour l'anglais
- `zeroclassnamesgenerator_es.ts` pour l'espagnol

Il va falloir éditer le fichier `.pro`. Celui-ci se trouve dans le dossier de votre projet et a normalement été généré automatiquement lorsque vous avez fait `qmake -project`.
Ouvrez ce fichier (dans mon cas `ZeroClassGenerator.pro`) avec un éditeur de texte comme Bloc-Notes, ou Notepad++, ou ce que vous voulez.

Pour le moment il devrait contenir quelque chose comme ça :

Code : Autre - [Sélectionner](#)

```
#####
# Automatically generated by qmake (2.01a) ven. 23. mai 16:31:10 2008
#####

TEMPLATE = app
TARGET =
DEPENDPATH += .
INCLUDEPATH += .

# Input
HEADERS += FenCodeGenere.h FenPrincipale.h
SOURCES += FenCodeGenere.cpp FenPrincipale.cpp main.cpp
```

Rajoutez à la fin une directive `TRANSLATIONS` en indiquant les noms des fichiers de traduction à générer. Ici, nous rajoutons un fichier pour la traduction anglaise, et un autre pour la traduction espagnole :

Code : Autre - [Sélectionner](#)

```
#####
# Automatically generated by qmake (2.01a) ven. 23. mai 16:31:10 2008
#####

TEMPLATE = app
TARGET =
DEPENDPATH += .
INCLUDEPATH += .

# Input
HEADERS += FenCodeGenere.h FenPrincipale.h
SOURCES += FenCodeGenere.cpp FenPrincipale.cpp main.cpp
TRANSLATIONS = zeroclassnamesgenerator_en.ts zeroclassnamesgenerator_es.ts
```

Bien. Maintenant, nous allons faire appel à un programme en console de Qt qui permet de générer automatiquement les fichiers `.ts`.

Ouvrez une console (sous Windows, utilisez le raccourci Qt Command Prompt que vous utilisez normalement d'habitude pour compiler). Allez dans le dossier de votre projet.

Tapez :

Code : Console - [Sélectionner](#)

```
lupdate NomDuProjet.pro
```

lupdate est un programme qui va mettre à jour les fichiers de traduction .ts, ou les créer si ceux-ci n'existent pas.

Essayons d'exécuter lupdate sur ZeroClassGenerator.pro :

Code : Console - [Sélectionner](#)

```
C:\Users\Mateo\Projets\ZeroClassGenerator>lupdate ZeroClassGenerator.pro
Updating 'zeroclassgenerator_en.ts'...
    Found 17 source text(s) (17 new and 0 already existing)
Updating 'zeroclassgenerator_es.ts'...
    Found 17 source text(s) (17 new and 0 already existing)
```

Le programme lupdate a trouvé dans mon code source 17 chaînes à traduire. Il a vérifié si les .ts existaient (ce qui n'était pas le cas) et les a donc créé.

Ce programme est intelligent puisque, si vous l'exécutez une seconde fois, il ne mettra à jour que les chaînes qui ont changé. C'est très pratique, puisque cela permet d'avoir à faire traduire au traducteur seulement ce qui a changé par rapport à la version précédente de votre programme !

Vous devriez maintenant avoir 2 fichiers supplémentaires dans le dossier de votre projet : zeroclassgenerator_en.ts et zeroclassgenerator_es.ts.

Envoyez-les à votre traducteur (s'il sait parler anglais et espagnol 😊). Bien entendu, le fait qu'on ait 2 fichiers distincts nous permet d'envoyer le premier à un traducteur anglais, et le second à un traducteur espagnol.

Traduire l'application sous Qt Linguist

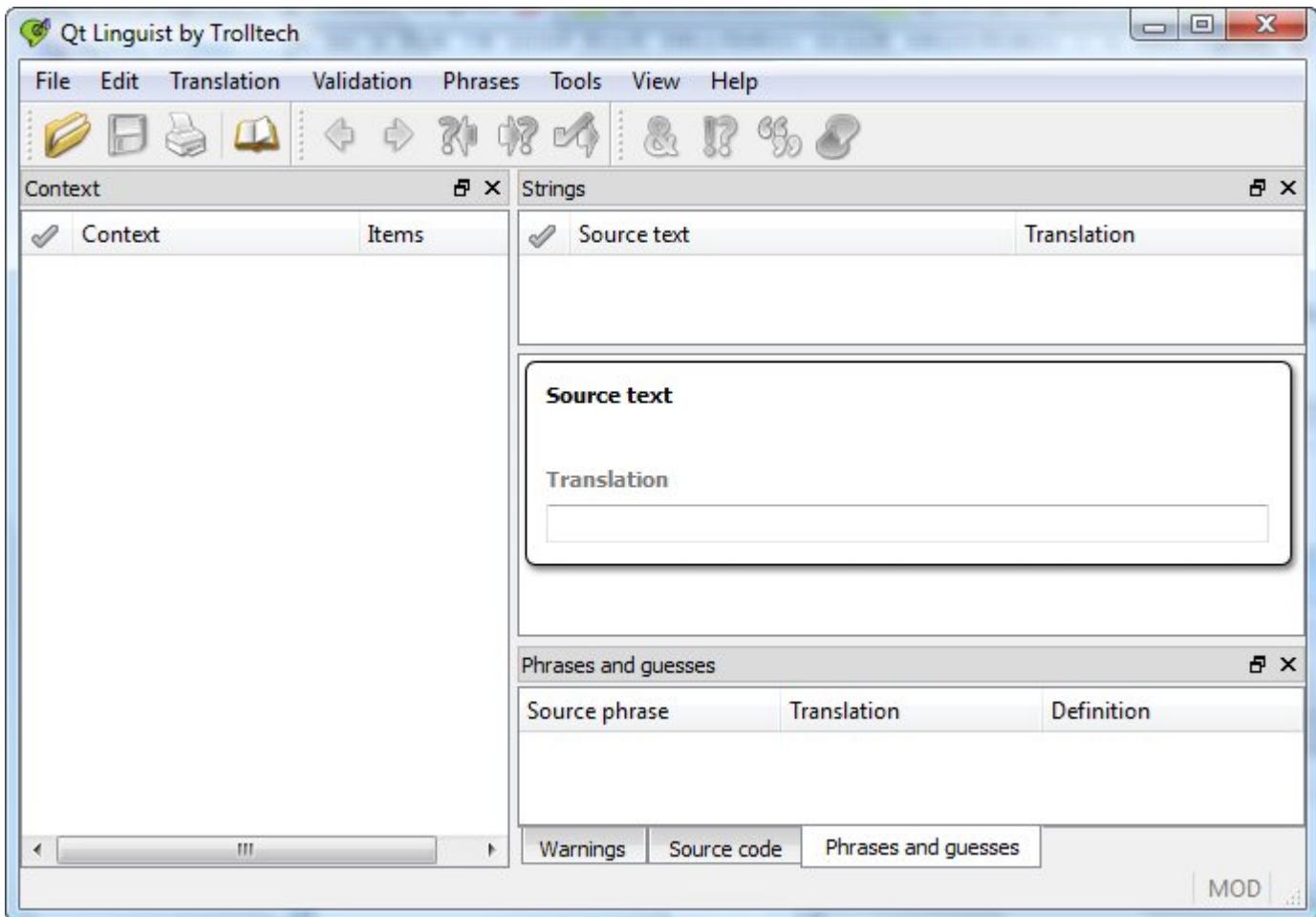


Qt a installé plusieurs programmes, vous vous souvenez ?
L'un d'eux va nous être sacrément utile maintenant : c'est Qt Linguist.

Votre traducteur a besoin de 2 choses :

- Du fichier .ts à traduire
- Et de Qt Linguist pour pouvoir le traduire !

Votre traducteur lance donc Qt Linguist. Il devrait voir quelque chose comme ça :



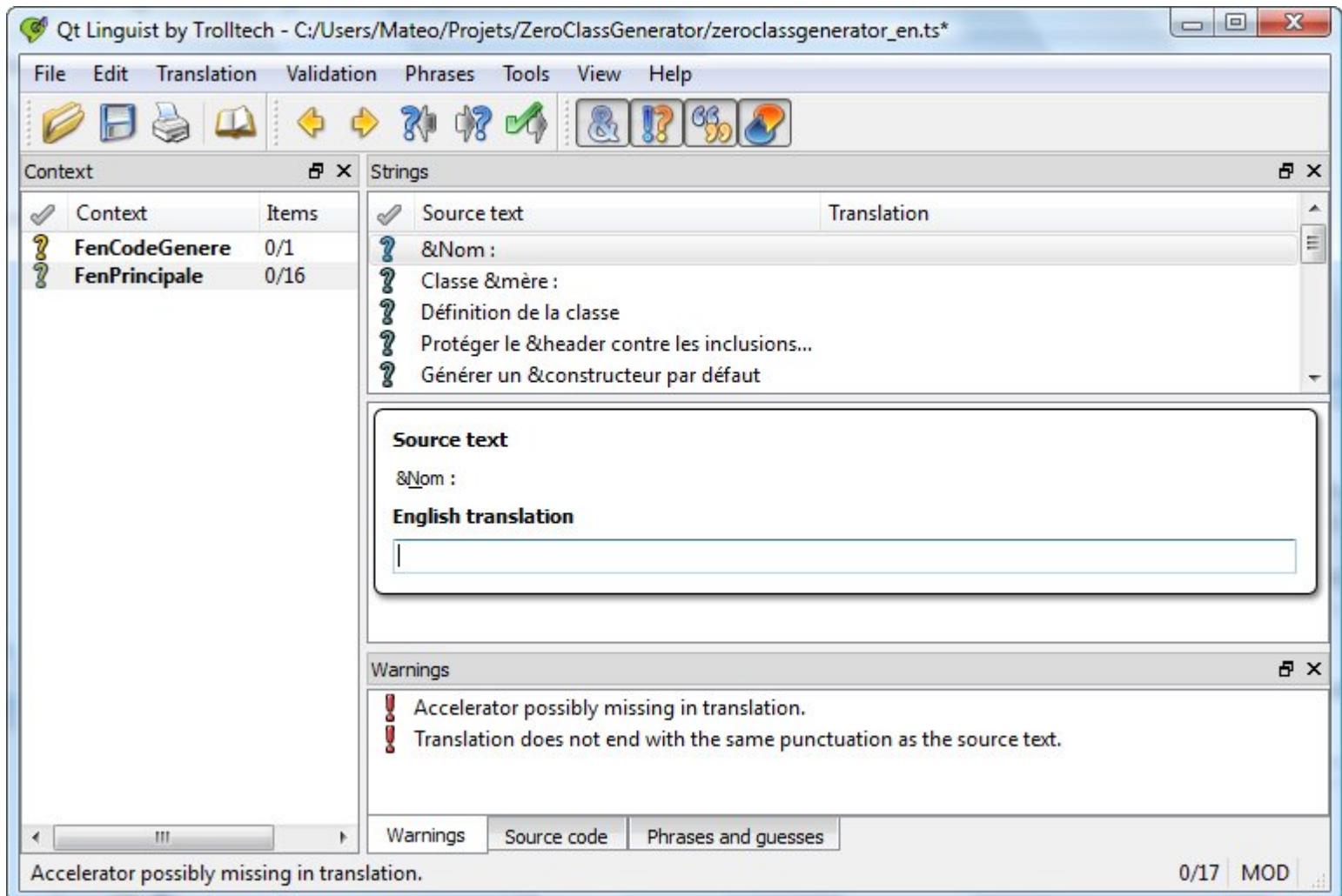
Ca a l'air un petit peu compliqué (et encore, vous avez pas vu Qt Designer 😊).

Vous reconnaissiez d'ailleurs sûrement une QMainWindow, avec une barre de menus, une barre d'outils, des docks, et un widget central (bah oui, les programmes livrés avec Qt sont faits avec Qt 😊).

En fait, les docks prennent tellement de place qu'on a du mal à savoir où est le widget central. Pour vous aider, c'est l'espèce de bulle ronde au milieu à droite, avec "Source text" et "Translation". C'est justement là qu'on traduira les chaînes de caractères.

Comme vous le savez déjà, les docks peuvent être déplacés. N'hésitez pas à arranger la fenêtre à votre guise. Vous pouvez même faire sortir les docks de la fenêtre principale pour en faire des mini-fenêtres flottantes.

Ouvrez un des fichiers .ts avec Qt Linguist, par exemple zero.classgenerator_en.ts. La fenêtre se remplit :



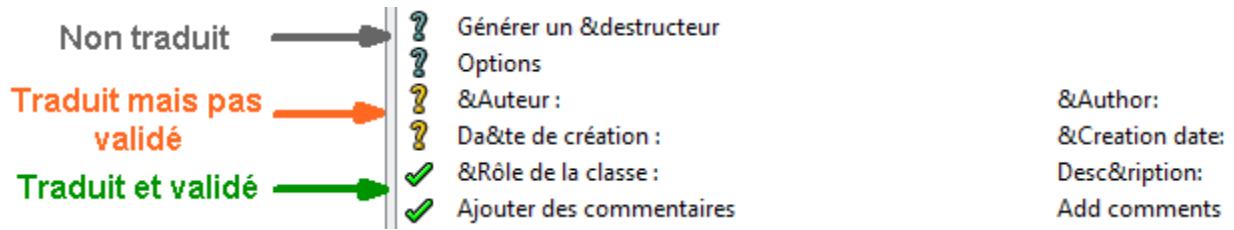
Détaillons un peu chaque section de la fenêtre :

- Context (à gauche) : affiche la liste des fichiers source qui contiennent des chaînes à traduire. Vous reconnaissiez vos fenêtres FenCodeGenere et FenPrincipale. Le nombre de chaînes traduites est indiqué à droite.
- Strings (en haut) : c'est la liste des chaînes à traduire pour le fichier sélectionné. Ces chaînes ont été extraites grâce à la présence de la méthode tr().
- Au milieu : vous avez la version française de la chaîne, et on vous demande d'écrire la version anglaise. Notez que Qt a automatiquement détecté que vous alliez traduire en anglais, grâce au nom du fichier qui contient "en". Si la chaîne à traduire peut être mise au pluriel, Qt Linguist vous demandera 2 traductions : une au singulier ("There is %n file") et une au pluriel ("There are %n files").
- Warnings (en bas) : affiche des avertissements bien utiles , comme "Vous avez oublié de mettre un & pour faire un raccourci", ou "La chaîne traduite ne se termine pas par le même signe de ponctuation" (ici un deux-points). Cette zone peut afficher aussi la chaîne à traduire dans son contexte du code source.

C'est maintenant au traducteur de traduire tout ça ! 😊

Lorsqu'il est sûr de sa traduction, il doit marquer la chaîne comme étant validée (en cliquant sur le petit "?" ou en faisant Ctrl + Entrée). Un petit symbole coché vert doit apparaître, et le dock context doit afficher que toutes les chaînes ont bien été traduites (16/16 par exemple).

Voici les 3 états que peut avoir chaque message :



On procède donc en 2 temps : d'abord on traduit, puis ensuite on se relit et on valide. Lorsque toutes les chaînes sont validées (en vert), le traducteur vous rend le fichier .ts.

Il ne nous reste plus qu'une étape : compiler ce .ts en un .qm, et adapter notre programme pour qu'il charge automatiquement le programme dans la bonne langue.

Lancer l'application traduite

Dernière ligne droite !

Nous avons le .ts entièrement traduit par notre traducteur adoré, il ne nous reste plus qu'à le compiler dans le format final binaire .qm, et à le charger dans l'application.

Compiler le .ts en .qm

Pour effectuer cette compilation, nous devons utiliser un autre programme de Qt : lrelease.

Ouvrez donc une console Qt, rendez-vous dans le dossier de votre projet, et tapez :

Code : Console - Sélectionner

```
lrelease nomDuFichier.ts
```

... pour compiler le fichier .ts indiqué.

Vous pouvez aussi faire :

Code : Console - Sélectionner

```
lrelease nomDuProjet.pro
```

... pour compiler tous les fichiers .ts du projet.

Comme je viens de terminer la traduction anglaise, je vais compiler le fichier .ts anglais :

Code : Console - Sélectionner

```
C:\Users\Mateo\Projets\ZeroClassGenerator>lrelease zeroclassnamesgenerator_en.ts
Updating 'zeroclassnamesgenerator_en.qm'...
Generated 17 translation(s) (17 finished and 0 unfinished)
```

Vous pouvez voir que lrelease ne compile que les chaînes marquées comme terminées (celles qui ont le symbole vert dans Qt Linguist). Si certaines ne sont pas marquées comme terminées, elles ne seront pas compilées dans le .qm.

Les chaînes non traduites ou non validées n'apparaîtront donc pas dans le programme. Dans ce cas, c'est la chaîne par défaut écrite dans le code (ici en français) qui sera affichée à la place.

D'autre part, notez que vous pouvez aussi faire la même chose directement dans Qt Linguist, en allant dans le menu File / Release.

Nous avons maintenant un fichier zeroclassnamesgenerator_en.qm dans le dossier de notre projet. Cool.

Si on le chargeait dans notre programme maintenant ? 😊

Charger un fichier de langue .qm dans l'application

Le chargement d'un fichier de langue s'effectue au début de la fonction main(). Pour le moment, votre fonction main() devrait ressembler à quelque chose comme ceci :

Code : C++ - Sélectionner

```
int main(int argc, char* argv[])
{
    QApplication app(argc, argv);

    FenPrincipale fenetre;
    fenetre.show();

    return app.exec();
}
```

Juste après la création de l'objet de type QApplication, nous allons rajouter les lignes suivantes :

Code : C++ - Sélectionner

```
int main(int argc, char* argv[])
{
    QApplication app(argc, argv);

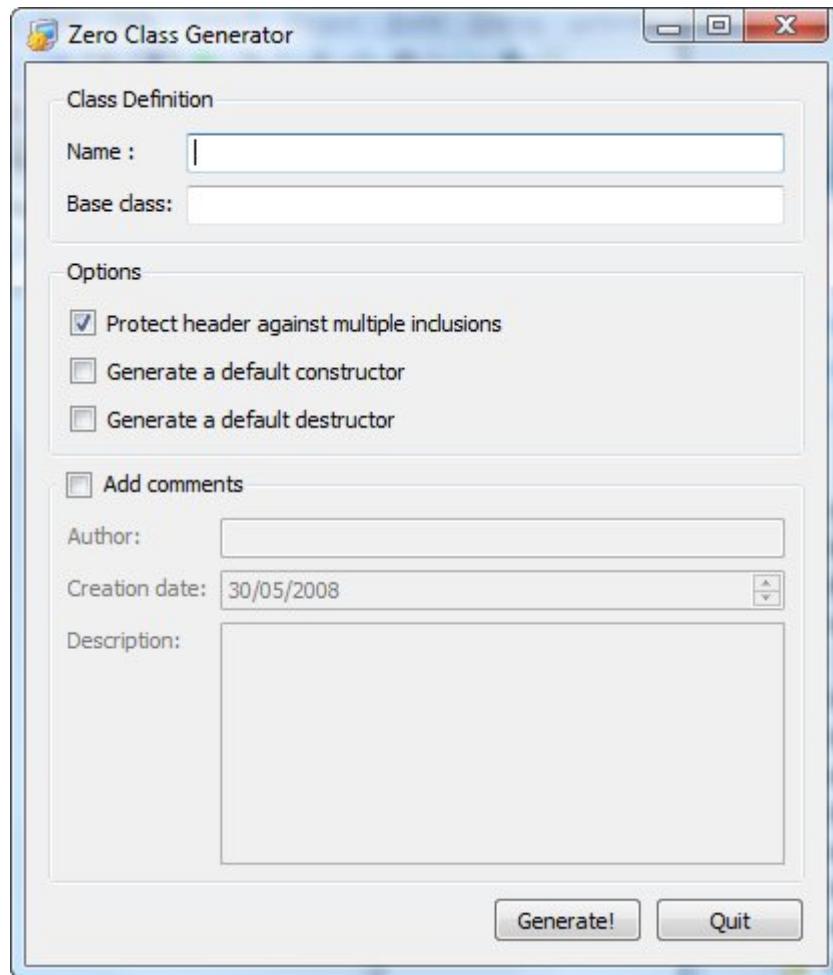
    QTranslator translator;
    translator.load("zeroclassnamesgenerator_en");
    app.installTranslator(&translator);

    FenPrincipale fenetre;
    fenetre.show();

    return app.exec();
}
```

Vérifiez bien que le fichier .qm se trouve dans le même dossier que l'exécutable, sinon la traduction ne sera pas chargée et vous aurez toujours l'application en français !

Si tout va bien, bravo, vous avez traduit votre application ! 😊



Euh, c'est bien mais c'est pas pratique. Là, mon application se chargera toujours en anglais. Il n'y a pas moyen qu'elle s'adapte à la langue de l'utilisateur ? 😞

Si, bien sûr, c'est faisable. C'est même ce qu'on fera dans 99% des cas.
Dans ce cas, on peut procéder comme ceci :

Code : C++ - Sélectionner

```
int main(int argc, char* argv[])
{
    QApplication app(argc, argv);

    QString locale = QLocale::system().name().section('_', 0, 0);

    QTranslator translator;
    translator.load(QString("zeroclassnamegenerator_") + locale);
    app.installTranslator(&translator);

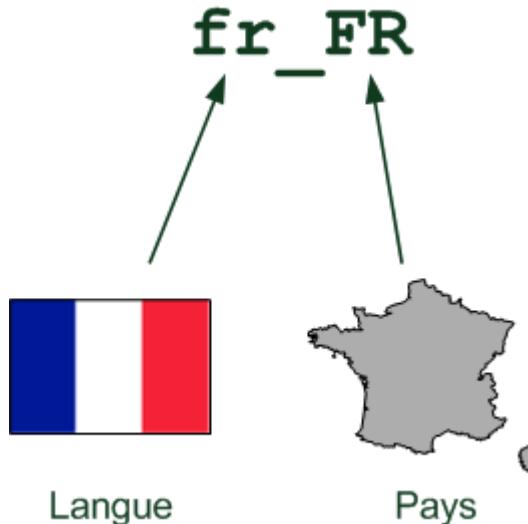
    FenPrincipale fenetre;
    fenetre.show();

    return app.exec();
}
```

Explication : on veut récupérer le code à 2 lettres de la langue du PC de l'utilisateur. On utilise une méthode statique de [QLocale](#) pour récupérer des informations sur le système d'exploitation sur lequel le programme a été lancé.

La méthode `QLocale::system().name()` renvoie un code ressemblant à ceci : "fr_FR", où "fr" est la langue (français) et "FR" le pays (France).

Si vous êtes québécois, vous aurez par exemple "fr_CA" (français au Canada).



On veut juste récupérer les 2 premières lettres. On utilise la méthode `section()` pour couper le chaîne en deux autour de l'underscore `_`. Les 2 autres paramètres permettent d'indiquer qu'on veut le premier mot à gauche de l'underscore, à savoir le "fr".

Au final, notre variable locale contiendra juste ce qu'on veut : la langue de l'utilisateur (par exemple "fr").
On combine cette variable avec le début du nom du fichier de traduction, comme ceci :

Code : C++ - [Sélectionner](#)

```
QString( "zeroclassgenerator_" ) + locale
```

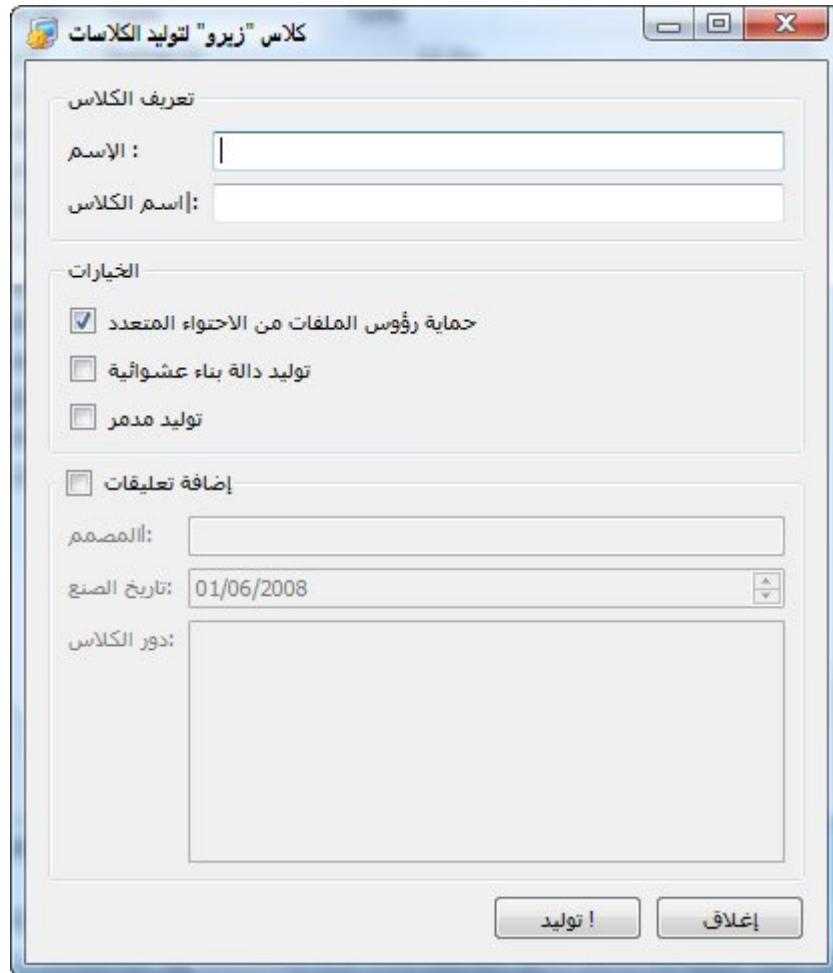
Si `locale` vaut "fr", le fichier de traduction chargé sera "zeroclassgenerator_fr".
Si `locale` vaut "en", le fichier de traduction chargé sera "zeroclassgenerator_en".

C'est compris ? 😊

Grâce à ça, notre programme ira chercher le fichier de traduction correspondant à la langue de l'utilisateur. Au pire des cas, si le fichier de traduction n'existe pas car vous n'avez pas fait de traduction dans cette langue, c'est la langue française qui sera utilisée.

Vous voilà maintenant aptes à traduire dans n'importe quelle langue ! 😊

Pour information, voilà ce que donne le ZeroClassGenerator traduit en arabe (merci à zoro_2009 pour la traduction !) :



Voilà donc la preuve que Qt peut vraiment gérer tous les caractères de la planète grâce à son support de l'Unicode. 😊

Comme vous avez pu le constater, la traduction d'applications Qt est un processus bien rôdé : tout est prévu ! 😊

Vous avez maintenant tous les outils en main pour diffuser votre programme partout dans le monde, même au Japon ! Encore faut-il trouver un traducteur japonais...

... et pour ça, désolé les amis, mais je ne pourrai vraiment pas vous aider. 🍃

Modéliser ses fenêtres avec Qt Designer

A force d'écrire le code de vos fenêtres, vous devez peut-être commencer à trouver ça long et répétitif. C'est amusant au début, mais au bout d'un moment on en a un peu marre d'écrire des constructeurs de 3 kilomètres de long juste pour placer les widgets sur la fenêtre.

C'est là que Qt Designer vient vous sauver la vie. Il s'agit d'un programme livré avec Qt (vous l'avez donc déjà installé) qui permet de dessiner vos fenêtres visuellement. Mais plus encore, Qt Designer vous permet aussi de modifier les propriétés des widgets, d'utiliser des layouts, et d'effectuer la connexion entre signaux et slots.

Qt Designer n'est pas un programme magique qui va réfléchir à votre place. Il vous permet juste de gagner du temps et d'éviter les tâches répétitives d'écriture du code de génération de la fenêtre.

N'utilisez PAS Qt Designer et ne lisez PAS ce chapitre si vous ne savez pas coder vos fenêtres à la main. En clair, si vous avez voulu sauter les chapitres précédents et juste lire celui-ci parce que vous le trouvez attrant, vous allez vous planter. C'est dit. 🍃

Nous commencerons par apprendre à manipuler Qt Designer lui-même. Vous verrez que c'est un outil complexe mais qu'on s'y fait vite car il est assez intuitif.

Ensuite, nous apprendrons à utiliser les fenêtres générées avec Qt Designer dans notre code source. Comme vous le verrez, il y a plusieurs façons de faire en fonction de vos besoins.

C'est parti ! 😊

Présentation de Qt Designer

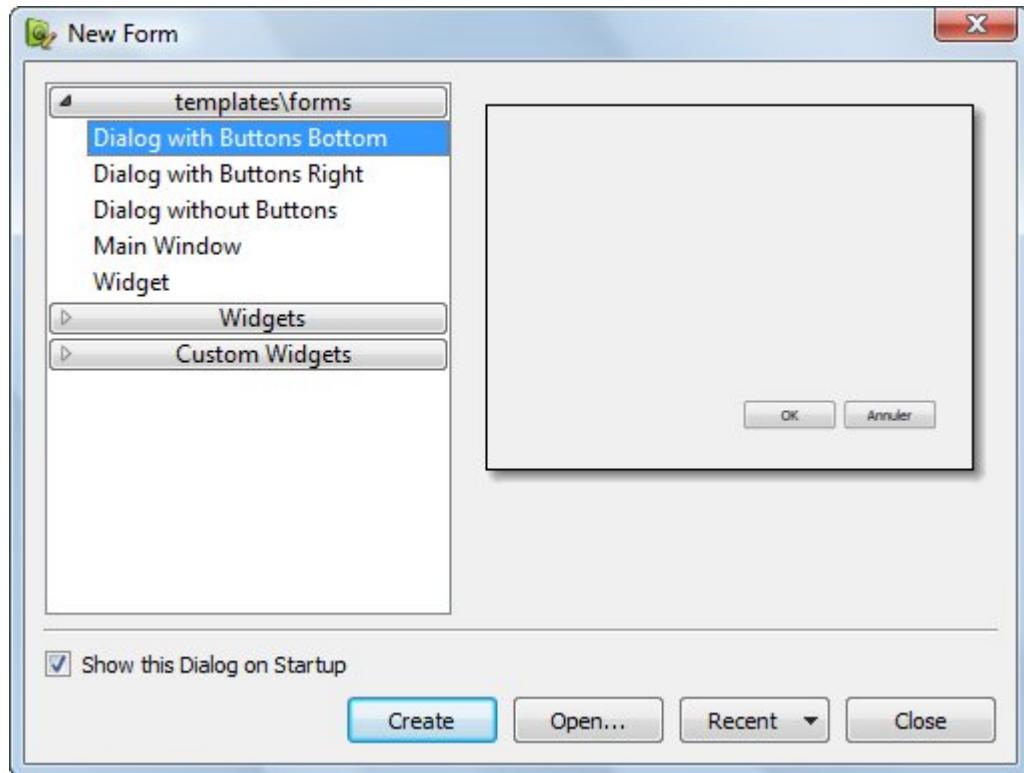


Nous allons commencer par démarrer directement Qt Designer.
Normalement, un raccourci a déjà été créé sur votre système (cf icône ci-contre).

Attention : si vous utilisez un thème personnalisé sur votre ordinateur (par exemple un thème Windows XP téléchargé sur internet), il se pourrait que Qt Designer rencontre des bugs d'affichage. Essayez de désactiver le thème personnalisé et de revenir au thème par défaut avant d'exécuter Qt Designer.

Choix du type de fenêtre à créer

Lorsque vous lancez Qt Designer, il vous propose de créer un nouveau projet. Vous avez le choix entre plusieurs types de fenêtres :



Les 3 premiers choix correspondent à des QDialog.

Vous pouvez aussi créer une QMainWindow si vous avez besoin de gérer des menus et des barres d'outils.

Enfin, le dernier choix correspond à une simple fenêtre de type QWidget.

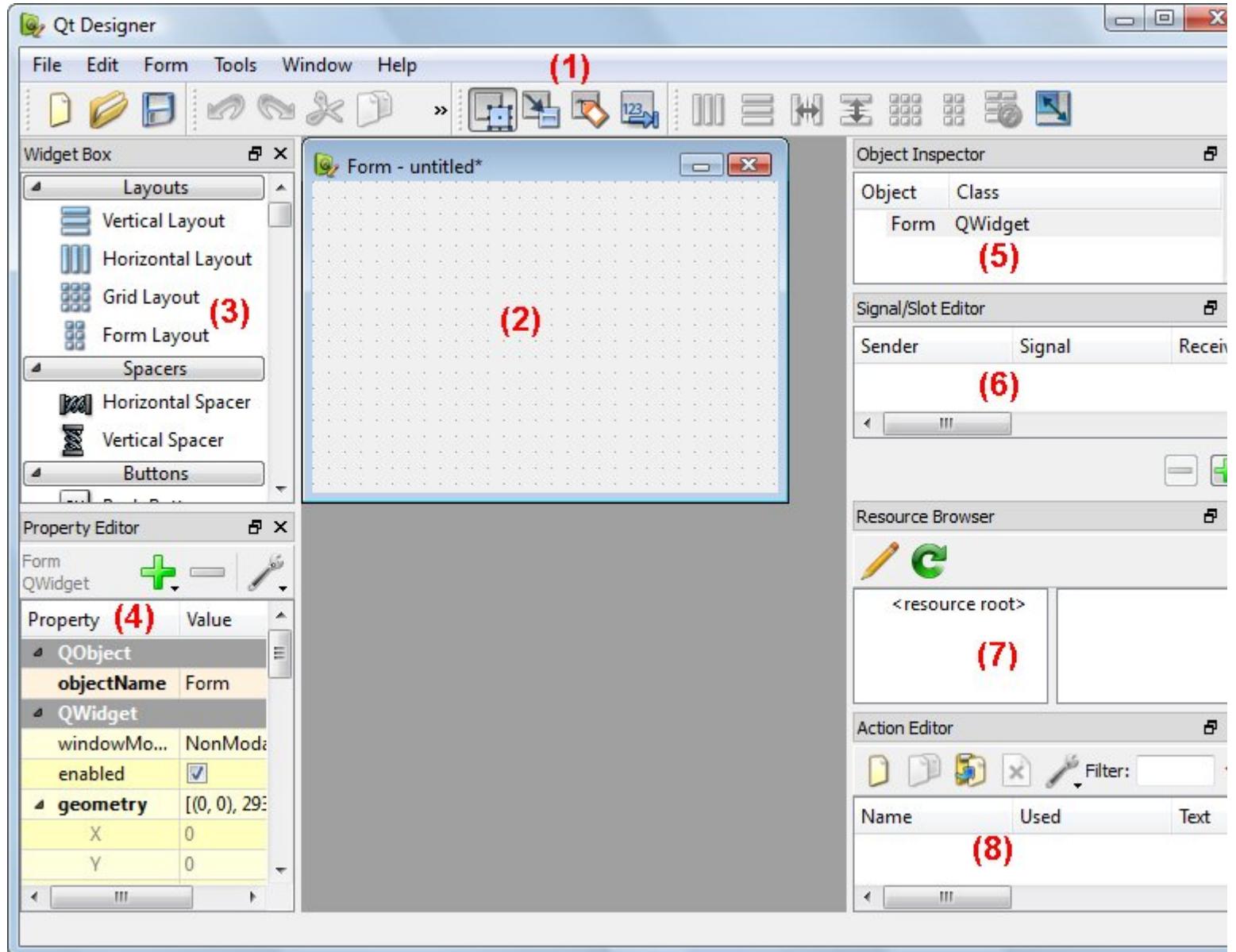
Pour nos exemples, nous allons choisir de créer une fenêtre simple de type QWidget. Sélectionnez donc le choix Widget.

Il y a d'autres choix que je ne détaillerai pas ici, dans les sous-catégories "Widgets" et "Custom Widgets". Par exemple, on peut créer une fenêtre-QGroupBox.

Vous utiliserez très rarement ces choix.

Analyse de la fenêtre de Qt Designer

Lorsque vous avez créé un nouveau projet, la fenêtre de Qt Designer commence à s'animer et... comme vous pouvez le voir, c'est assez complet :



Wow ! Mais comment je vais faire pour m'y retrouver avec tous ces boutons ? 😱

En y allant méthodiquement. 😊

Détaillons chacune des zones importantes dans l'ordre :

1. Sur la barre d'outils de Qt Designer, au moins 4 boutons méritent votre attention. Ce sont les 4 boutons situés sous la marque "(1)" rouge que j'ai placée sur la capture d'écran.



Ils permettent de passer d'un mode d'édition à un autre. Qt Designer propose 4 modes d'édition :

- o Edit Widgets : le mode par défaut, que vous utiliserez le plus souvent. Il permet d'insérer des widgets sur la fenêtre et de modifier leurs propriétés.
- o Edit Signals/Slots : permet de créer des connexions entre les signaux et les slots de vos widgets.

- Edit Buddies : permet d'associer des QLabel avec leurs champs respectifs. Lorsque vous faites un layout de type QFormLayout, ces associations sont automatiquement créées.
- Edit Tab Order : permet de modifier l'ordre de tabulation entre les champs de la fenêtre, pour ceux qui naviguent au clavier et passent d'un champ à l'autre en appuyant sur la touche "Tab".

Nous ne verrons dans ce chapitre que les 2 premiers modes (Edit Widgets et Edit Signals/Slots). Les autres modes sont peu importants et je vous laisse les découvrir par vous-mêmes.

2. Au centre de Qt Designer, vous avez la fenêtre que vous êtes en train de dessiner. Pour le moment celle-ci est vide. Si vous créez une QMainWindow, vous aurez en plus une barre de menus et une barre d'outils. Leur édition se fait à la souris, c'est très intuitif.
Si vous créez une QDialog, vous aurez probablement des boutons "OK" et "Annuler" déjà disposés.
3. Widget Box : ce dock vous donne la possibilité de sélectionner un widget à placer sur la fenêtre. Vous pouvez constater qu'il y a un assez large choix ! Heureusement, ceux-ci sont organisés par groupes pour y voir plus clair.
Pour placer un de ces widgets sur la fenêtre, il suffit de faire un glisser-déplacer. Simple et intuitif.

Les widgets en bas de la liste sont soit d'anciens widgets, soit des widgets modifiés non standards. Vous ne devriez pas avoir besoin d'y toucher.

4. Property Editor : lorsqu'un widget est sélectionné sur la fenêtre principale, vous pouvez éditer ses propriétés. Vous noterez que les widgets possèdent en général beaucoup de propriétés, et que celles-ci sont organisées en fonction de la classe dans laquelle elles ont été définies. On peut ainsi modifier toutes les propriétés dont un widget hérite, en plus des propriétés qui lui sont propres.

Comme toutes les classes héritent de QObject, vous aurez toujours la propriété objectName. C'est le nom de l'objet qui sera créé. N'hésitez pas à le personnaliser, afin d'y voir plus clair tout à l'heure dans votre code source (sinon vous aurez par exemple des boutons appelés pushButton, pushButton_2, pushButton_3, ce qui n'est pas très clair).

Si aucun widget n'est sélectionné, ce sont les propriétés de la fenêtre que vous éditez. Vous pourrez donc par exemple modifier son titre avec la propriété windowTitle, son icône avec windowIcon, etc.

5. Object Inspector : affiche la liste des widgets placés sur la fenêtre, en fonction de leur relation de parenté, sous forme d'arbre. Ca peut être pratique si vous avez une fenêtre complexe et que vous commencez à vous perdre dedans. Vous pouvez ainsi y voir par exemple que votre fenêtre contient un QGroupBox qui contient 3 cases à cocher.
6. Signal / slot editor : si vous avez associé des signaux et des slots, les connexions du widget sélectionné apparaissent ici. Nous verrons comment réaliser des connexions dans Qt Designer tout à l'heure.
7. Resource Browser : un petit utilitaire qui vous permet de naviguer à travers les fichiers de ressources de votre application. Ces fichiers de ressources rappellent un peu ceux de Windows (on en a brièvement parlé dans l'annexe du cours de C, à propos de l'ajout d'icône à un programme sous Windows).
Ici, les fichiers de ressources portent l'extension .qrc et ont l'avantage d'être compatibles avec tous les OS.

Les fichiers de ressources servent empaqueter des fichiers (images, sons, texte...) au sein même de votre exécutable. Cela permet d'éviter d'avoir à placer ces fichiers dans le même dossier que votre programme, et cela évite donc le risque de les perdre (puisque'ils se trouveront toujours dans votre exécutable).

C'est un peu hors-sujet, donc je n'en parlerai pas plus ici. Consultez la doc à propos des ressources si vous voulez en savoir plus.

8. Action Editor : permet de créer des QAction. C'est donc utile lorsque vous créez une QMainWindow avec des menus et une barre d'outils.

Voilà qui devrait suffire pour une présentation générale de Qt Designer. Maintenant, pratiquons un peu. 😊

Placer des widgets sur la fenêtre

Placer des widgets sur la fenêtre est en fait très simple : vous prenez le widget que vous voulez dans la liste à gauche, et vous le faites glisser où vous voulez sur la fenêtre.

Ce qui est très important à savoir, c'est qu'on peut placer ses widgets de 2 manières différentes :

- De manière absolue : vos widgets seront disposés au pixel près sur la fenêtre. C'est la méthode par défaut, la plus précise, mais la moins flexible aussi. Je vous avais parlé de ses défauts dans le chapitre sur les layouts.
- Avec des layouts (recommandé pour les fenêtres complexes) : vous pouvez utiliser tous les layouts que vous connaissez. Verticaux, horizontaux, en grille, en formulaire... Grâce à cette technique, les widgets s'adapteront automatiquement à la taille de votre fenêtre.

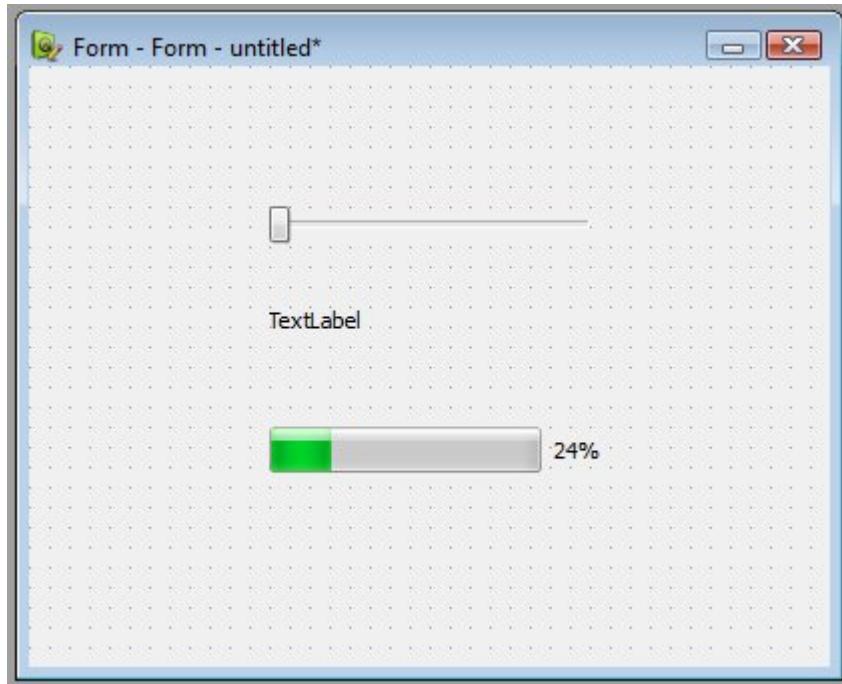
Commençons par les placer de manière absolue, puis nous verrons comment utiliser les layouts dans Qt Designer.

Placer les widgets de manière absolue

Je vous propose pour vous entraîner de faire une petite fenêtre simple composée de 3 widgets :

- QSlider
- QLabel
- QProgressBar

Votre fenêtre devrait à peu près ressembler à ceci maintenant :



Vous pouvez déplacer ces widgets comme bon vous semble sur la fenêtre.
Vous pouvez les agrandir ou les rétrécir.

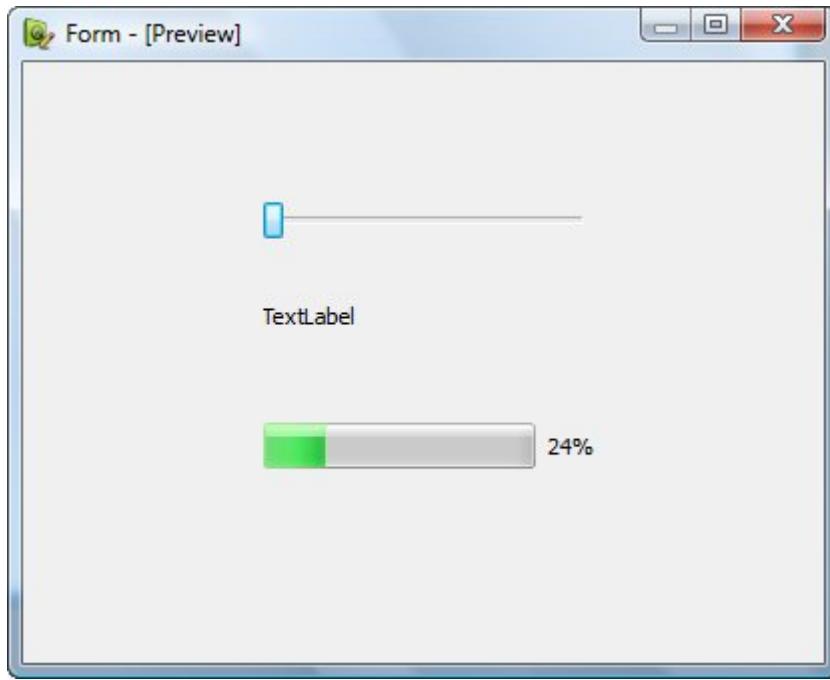
Quelques raccourcis à connaître :

- En maintenant la touche Shift appuyée, vous pouvez sélectionner plusieurs widgets en même temps.
 - Faites Suppr pour supprimer les widgets sélectionnés.
 - Si vous maintenez la touche Ctrl enfonce lorsque vous déplacez un widget, celui-ci sera copié.
 - Vous pouvez double-cliquer sur un widget pour modifier son nom (il vaut mieux donner un nom personnalisé plutôt que laisser le nom par défaut).
- Sur certains widgets complexes, comme la QComboBox (liste déroulante), le double clic a pour effet de vous permettre d'édition la liste des éléments contenus dans la liste déroulante.
- Pensez aussi à faire un clic droit sur les widgets pour modifier certaines propriétés, comme la bulle d'aide (toolTip).

Vous pouvez prévisualiser la fenêtre en faisant Ctrl + R, ou encore en allant dans le menu "Form / Preview".

Le menu "Form / Preview in..." vous permet de prévisualiser la fenêtre dans un autre style (Windows, Windows XP, Windows Vista, Plastique, CDE...)

Voici notre fenêtre en mode "Preview" :



Ce mode nous permet de tester la fenêtre telle qu'elle apparaîtra à la fin, de manipuler les widgets, etc. Sortez du mode Preview et revenez à l'édition, nous avons encore des choses à voir.

Utiliser les layouts

Pour le moment, nous n'utilisons aucun layout. Si vous essayez de redimensionner la fenêtre, vous verrez que les widgets ne s'adaptent pas à la nouvelle taille et qu'ils peuvent même disparaître si on réduit trop la taille de la fenêtre !

Il y a 2 façons d'utiliser des layouts :

- Utiliser la barre d'outils en haut.
- Glisser-déplacer des layouts depuis le dock de sélection de widgets ("Widget Box").

Pour une fenêtre simple comme celle-là, nous n'aurons besoin que d'un layout principal.

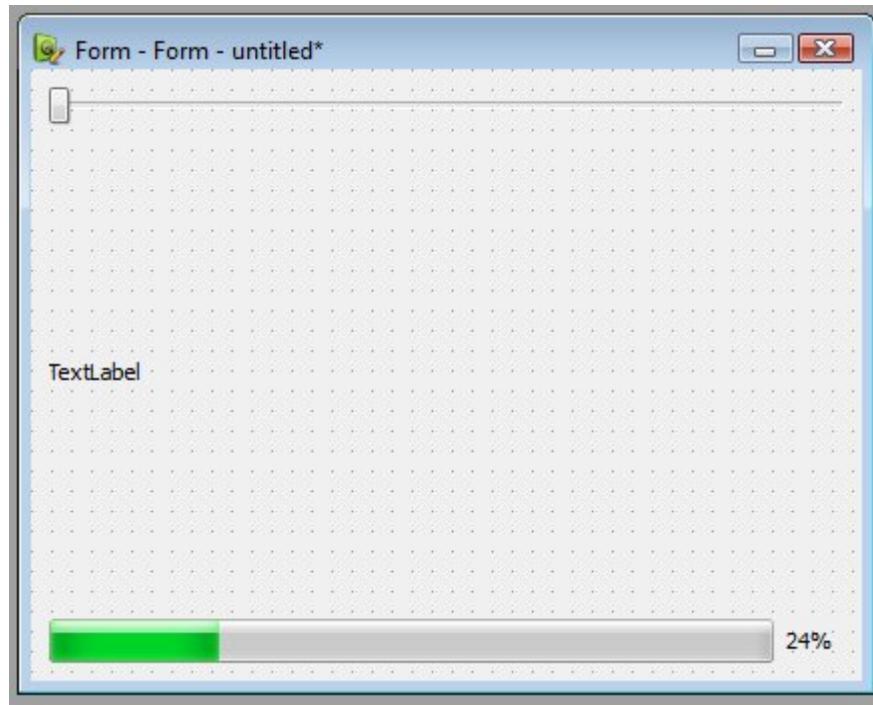
Pour définir ce layout principal, le mieux est de passer par la barre d'outils :



Cliquez sur une zone vide de la fenêtre (en clair, il faut que ce soit la fenêtre qui soit sélectionnée et non un de ses widgets). Vous devriez alors voir les boutons de la barre d'outils des layouts s'activer, comme sur l'image ci-dessus.

Cliquez sur le bouton correspondant au layout vertical (le second) pour organiser automatiquement la fenêtre selon un layout vertical. ☺

Vous devriez alors voir vos widgets s'organiser comme ceci :



C'est le layout vertical qui les place comme ça afin qu'ils occupent toute la taille de la fenêtre. Bien sûr, vous pouvez réduire la taille de la fenêtre si vous le désirez.

Vous pouvez aussi demander à ce que la fenêtre soit réduite à la taille minimale acceptable, en cliquant sur le bouton tout à droite de la barre d'outils, intitulé "Adjust Size".

Maintenant que vous avez défini le layout principal de la fenêtre, sachez que vous pouvez insérer un sous-layout en plaçant par exemple un des layouts proposés dans la Widget Box.

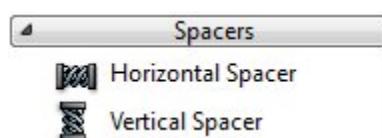
Insérer des spacers

Vous trouvez que la fenêtre est un peu moche si on l'agrandit trop ?

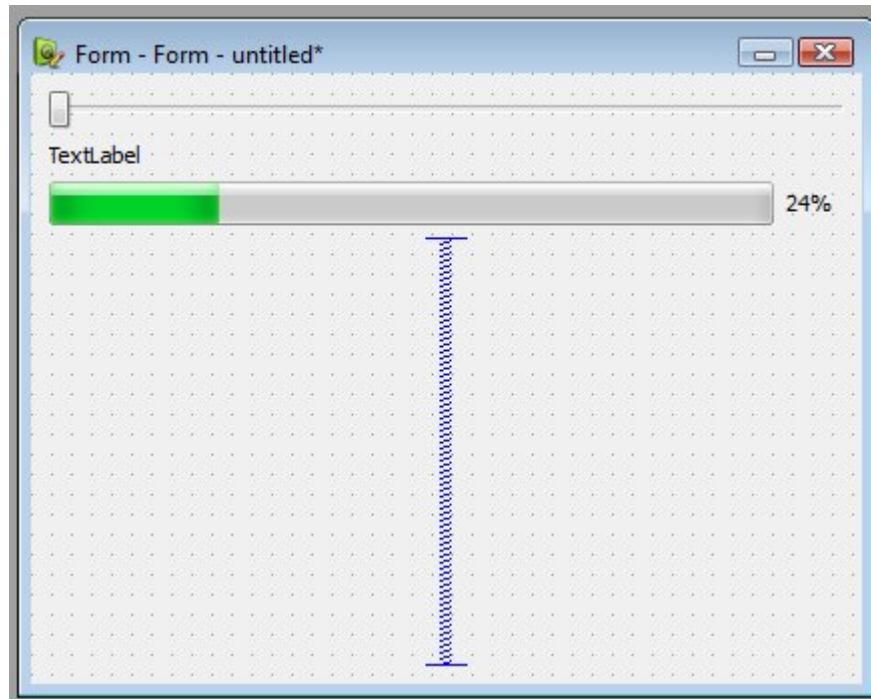
Moi aussi. Les widgets sont trop espacés, ça ne me convient pas.

Pour changer la position des widgets tout en conservant le layout, on peut insérer un spacer. Il s'agit d'un widget invisible qui sert à créer de l'espace sur la fenêtre.

Le mieux est encore d'essayer pour comprendre ce que ça fait. Dans la Widget Box, vous devriez avoir une section "Spacers" :



Prenez un "Vertical Spacer", et insérez-le tout en bas de la fenêtre. Vous devriez alors voir ceci :



Le spacer va forcer les autres widgets à se coller tout en haut. Ils sont toujours organisés selon un layout, mais au moins maintenant nos widgets sont plus rapprochés les uns des autres.

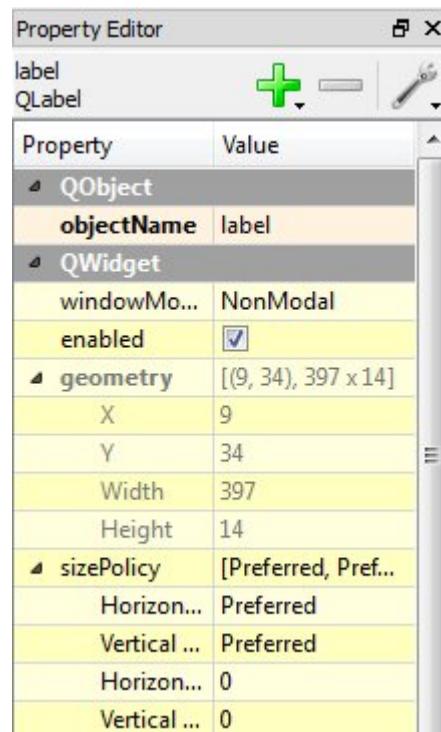
Essayez de déplacer le spacer sur la fenêtre pour voir. Placez-le entre le libellé et la barre de progression. Vous devriez voir que la barre de progression se colle maintenant tout en bas.

Le comportement du spacer est assez logique, mais il faut l'essayer pour bien comprendre. 😊

Editer les propriétés des widgets

Il nous reste une chose très importante à voir : l'édition des propriétés des widgets.

Sélectionnez par exemple le libellé (QLabel). Regardez le dock intitulé "Property Editor". Il affiche maintenant les propriétés du QLabel :



Ces propriétés sont organisées en fonction de la classe dans laquelle elles ont été définies, et c'est une bonne chose. Je m'explique. Vous savez peut-être qu'un QLabel hérite de QFrame, qui hérite de QWidget, qui hérite lui-même de QObject ?

Chacune de ces classes définit des propriétés. QLabel hérite donc des propriétés de QFrame, QWidget et QObject, mais a aussi des propriétés qui lui sont propres.

Sur ma capture d'écran ci-dessus, on peut voir une propriété de QObject : `objectName`. C'est le nom de l'objet qui sera créé dans le code. Je vous conseille de le personnaliser pour que vous puissiez vous y retrouver dans le code source ensuite.

La plupart du temps, on peut éditer le nom d'un widget en double-cliquant dessus sur la fenêtre.

Si vous descendez un peu plus bas dans la liste, vous devriez vous rendre compte qu'un grand nombre de propriétés sont proposées par QWidget (notamment la police, le style de curseur de la souris, etc.). Descendez encore plus bas. Vous devriez arriver sur les propriétés héritées de QFrame, puis celles propres à QLabel :

styleSheet	
► locale	French, France
▢ QFrame	
frameShape	NoFrame
frameShadow	Plain
lineWidth	1
midLineWid...	0
▢ QLabel	
► text	TextLabel
textFormat	AutoText
pixmap	
scaledCont...	□
► alignment	AlignLeft, Align...
wordWrap	□
margin	0
indent	-1
openExtern...	□
► textInteracti...	LinksAccessible...
buddy	

Comme vous pouvez le voir, ces propriétés ont été mises en valeur : elles sont en vert.

Je trouve que c'est très bien d'avoir organisé les propriétés comme ça. Ainsi, on voit bien où elles sont définies.

Vous devriez modifier la propriété `text`, pour changer le texte affiché dans le QLabel. Mettez par exemple "0". Amusez-vous à changer la police (propriété `font` issue de QWidget) ou encore à mettre une bordure (propriété `frameShape` issue de QFrame).

Vous remarquerez que lorsque vous éditez une propriété, son nom s'affiche en gras pour être mis en valeur. Cela vous permet par la suite de repérer du premier coup d'œil les propriétés que vous avez modifiées.

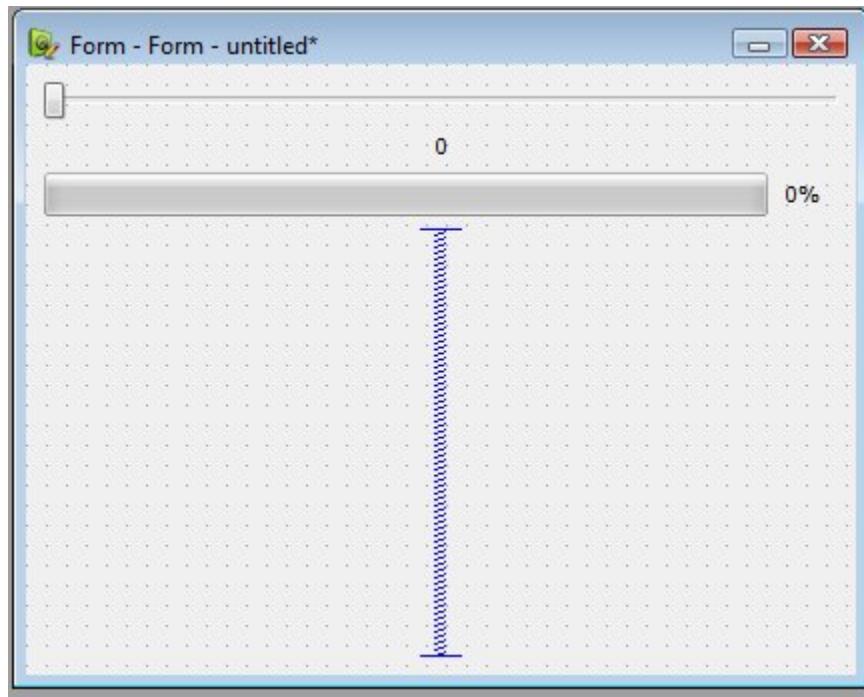
Certaines propriétés, comme alignement de QLabel, possèdent des sous-propriétés. Cliquez sur la petite flèche à gauche pour afficher et modifier ces sous-propriétés. Essayez de faire en sorte que le texte de notre libellé soit centré horizontalement par exemple.

Modifiez aussi les propriétés de la QProgressBar pour qu'elle affiche 0% pour défaut (propriété `value`).

Vous pouvez aussi modifier les propriétés de la fenêtre. Cliquez sur une zone vide de la fenêtre afin qu'aucun widget ne soit sélectionné. Le dock "Property Editor" vous affichera alors les propriétés de la fenêtre (ici, notre fenêtre est un QWidget, donc vous aurez juste les propriétés de QWidget).

Astuce : si vous ne comprenez pas à quoi sert une propriété, cliquez dessus puis appuyez sur la touche F1. Qt Designer lancera automatiquement Qt Assistant pour afficher l'aide sur la propriété sélectionnée.

Essayez d'avoir une fenêtre qui ressemble au final grossomodo à la mienne :



Le libellé et la barre de progression doivent afficher 0 par défaut.

Bravo, vous savez maintenant insérer des widgets, les organiser selon un layout et personnaliser leurs propriétés dans Qt Designer !



Nous n'avons utilisé pour le moment que le mode "Edit Widgets". Il nous reste à étudier le mode "Edit Signals/Slots"...

Configurer les signaux et les slots

Passez en mode "Edit Signals/Slots" en cliquant sur le second bouton de la barre d'outils :

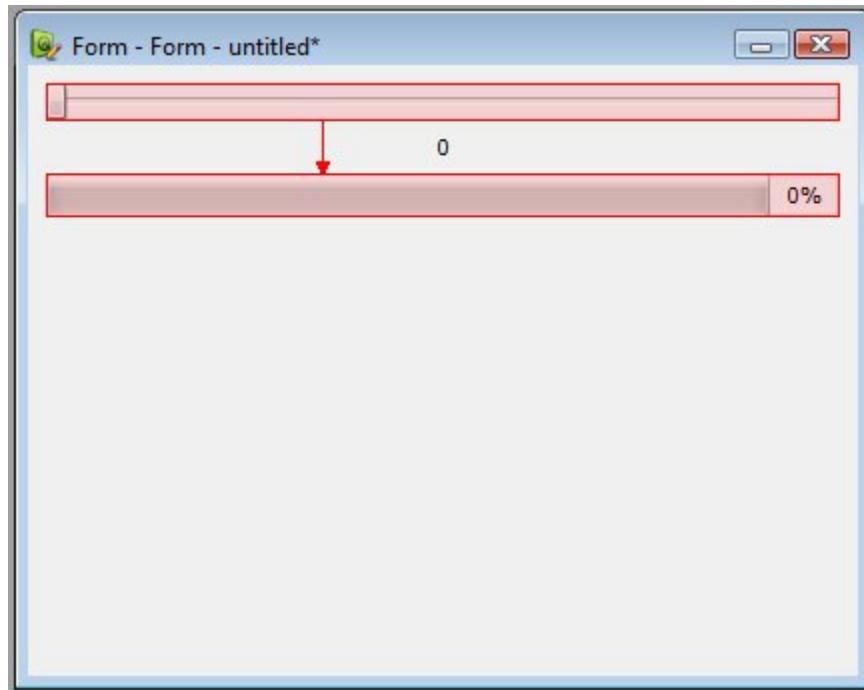


Vous pouvez aussi appuyer sur la touche F4. Vous pourrez faire F3 pour revenir au mode d'édition des widgets.

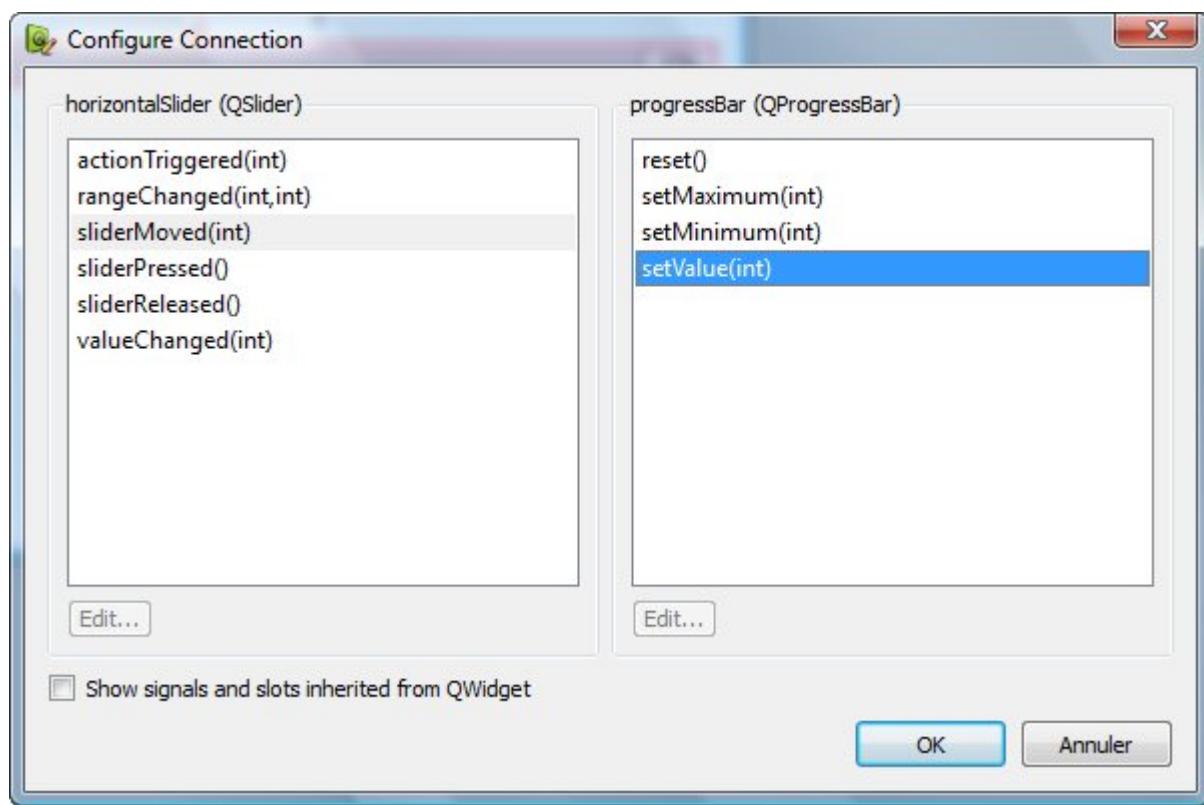
Dans ce mode, on ne peut pas ajouter, modifier, supprimer, ni déplacer de widgets. Par contre, si vous pointez sur les widgets de votre fenêtre, vous devriez voir un cadre rouge autour d'eux.

Vous pouvez, de manière très intuitive, associer les widgets entre eux pour créer des connexions simples entre leurs signaux et slots. Je vous propose par exemple d'associer le QSlider avec notre QProgressBar.

Pour cela, cliquez sur le QSlider et maintenez le bouton gauche de la souris enfoncé. Pointez sur la QProgressBar et relâchez le bouton. La connexion que vous allez faire devrait ressembler à ceci :



Une fenêtre apparaît alors pour que vous puissiez choisir le signal et le slot à connecter :



A gauche : les signaux disponibles dans le QSlider.

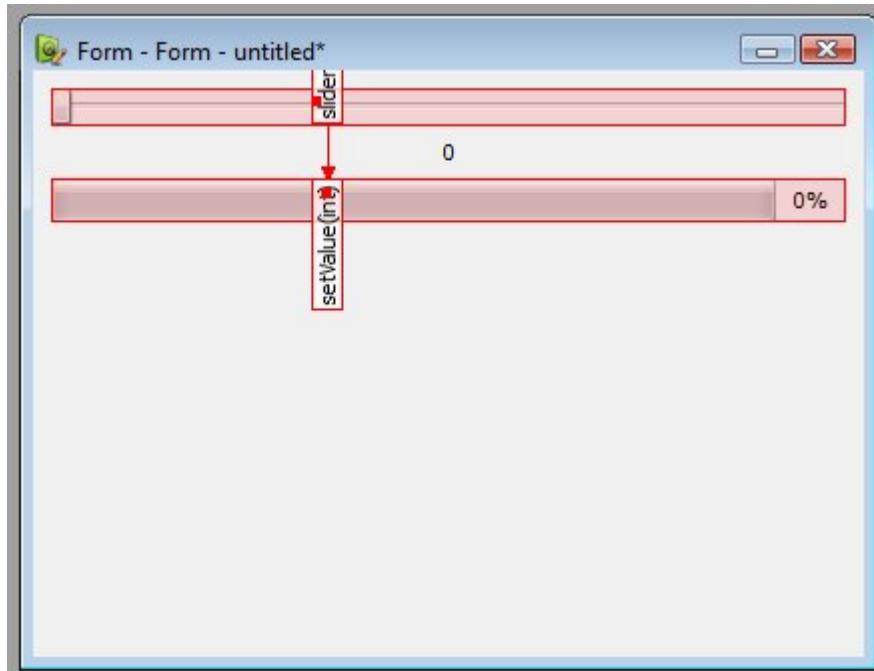
A droite : les slots compatibles disponibles dans la QProgressBar.

Sélectionnez un signal à gauche, par exemple `sliderMoved(int)`. Ce signal est envoyé dès que l'on déplace un peu le slider. Vous verrez que la liste des slots compatibles apparaît à droite.

En fonction du signal choisi, Qt Designer ne vous affiche que les slots de destination compatibles. Par exemple, `sliderMoved(int)` s'accorde bien avec `setValue(int)`. On peut aussi le connecter à `reset()`, dans ce cas le nombre envoyé en paramètre sera perdu. Par contre, on ne peut pas connecter le signal `sliderMoved(int)` au slot `setRange(int, int)` car le signal n'envoie pas assez de paramètres. D'ailleurs, vous ne devriez pas voir ce slot disponible dans la liste des slots si vous avez choisi le signal `sliderMoved(int)`, ce qui vous empêche de créer une connexion incompatible.

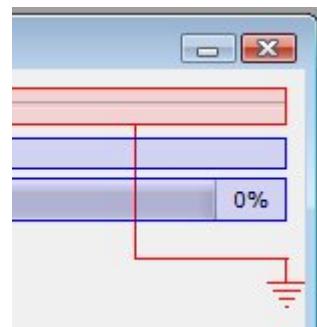
Nous allons connecter sliderMoved(int) du QSlider avec setValue(int) de la QProgressBar.

Faites OK pour valider une fois le signal et le slot choisis. C'est bon, la connexion est créée. 😊



Faites de même pour associer sliderMoved(int) du QSlider à setNum(int) du QLabel.

Notez que vous pouvez aussi connecter un widget à la fenêtre. Dans ce cas, visez une zone vide de la fenêtre. La flèche devrait se transformer en symbole de masse (bien connu par ceux qui font de l'électricité ou de l'électronique) :



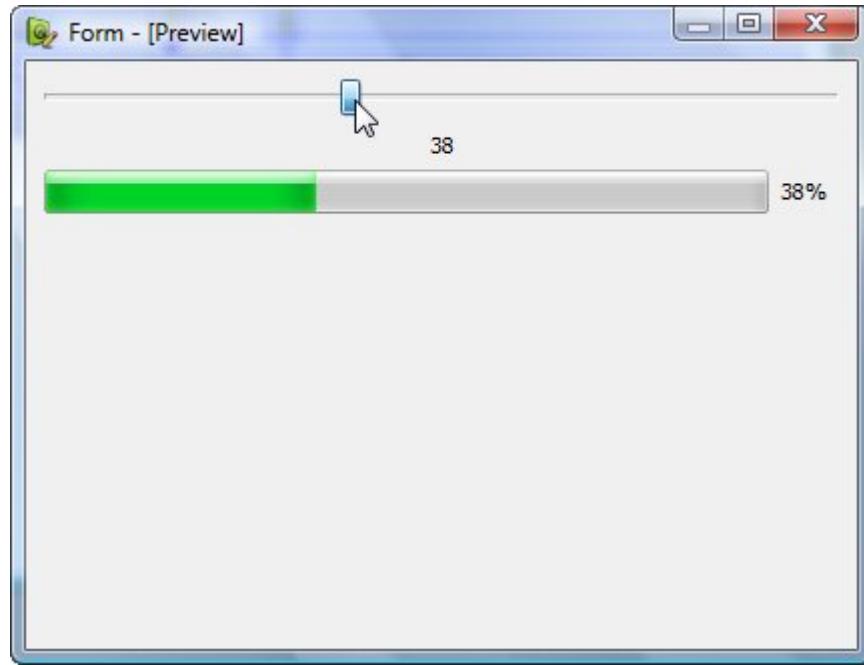
Cela vous permet d'associer un signal du widget à un slot de la fenêtre, ce qui peut vous être utile si vous voulez créer un bouton "Fermer la fenêtre" par exemple.

Attention : si dans la fenêtre du choix du signal et du slot vous ne voyez aucun slot s'afficher pour la fenêtre, c'est normal. Qt les masque par défaut car ils sont nombreux. Si on les affichait pour chaque connexion entre 2 widgets, on en aurait beaucoup trop (puisque tous les widgets héritent de QWidget).

Pour afficher quand même les signaux et slots issus de QWidget, cochez la case "Show signals and slots inherited from QWidget".

Passez maintenant en mode preview (Ctrl + R) pour tester vos connexions.

Essayez de déplacer le slider. Si vous avez fait les choses correctement, vous devriez voir le libellé et la barre de progression changer de valeur en même temps ! 😊



Pour des connexions simples entre les signaux et les slots des widgets, Qt Designer est donc très intuitif et convient parfaitement.

Eh, mais si je veux créer un slot personnalisé pour faire des manipulations un peu plus complexes, comment je fais ?

Qt Designer ne peut pas vous aider pour ça. Si vous voulez créer un signal ou un slot personnalisé, il faudra le faire tout à l'heure dans le code source.

Comme vous pourrez le voir néanmoins, c'est très simple à faire.

En y réfléchissant bien, c'est même d'ailleurs la seule chose que vous aurez à coder ! En effet, tout le reste est automatiquement géré par Qt Designer. Vous n'avez plus qu'à vous concentrer sur la partie "réflexion" de votre code source.

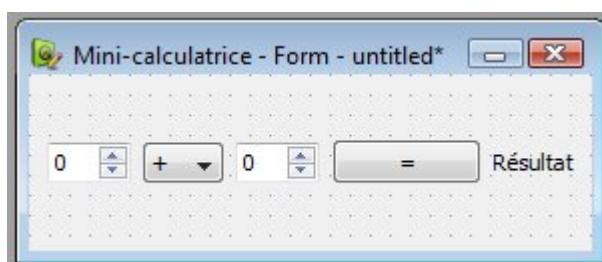
Qt Designer vous permet donc de gagner du temps en vous épargnant les tâches répétitives et basiques qu'on fait à chaque fois que l'on crée une fenêtre.

Utiliser la fenêtre dans votre application

Il reste une dernière étape, et pas des moindres : apprendre à utiliser la fenêtre ainsi créée dans votre application.

Notre nouvel exemple

Je vous propose de créer une nouvelle fenêtre (parce que l'exemple de tout à l'heure était bien joli, mais pas très intéressant à part pour tester les signaux et slots 😊). On va créer une mini-calculatrice :



Essayez de reproduire à peu près la même fenêtre que moi.
Un layout principal horizontal suffira à organiser les widgets.

La fenêtre est constituée des widgets suivants, de gauche à droite :

Widget	Nom de l'objet
QSpinBox	nombre1
QComboBox	operation
QSpinBox	nombre2
QPushButton	boutonEgal
QLabel	resultat

Pensez à bien renommer les widgets afin que vous puissiez vous y retrouver dans votre code source ensuite. 😊
Pensez aussi à donner un nom à la fenêtre. Je l'ai appelée "FenCalculatrice".

Pour la liste déroulante du choix de l'opération, je l'ai déjà pré-remplie avec 4 valeurs : +, -, * et /. Double-cliquez sur la liste déroulante pour ajouter / supprimer des valeurs.

Enregistrez cette fenêtre dans le dossier de votre projet. Je lui ai donné le nom calculatrice.ui.
Tous les fichiers de fenêtres créés avec Qt Designer portent l'extension .ui (comme User Interface, "Interface Utilisateur" en français).

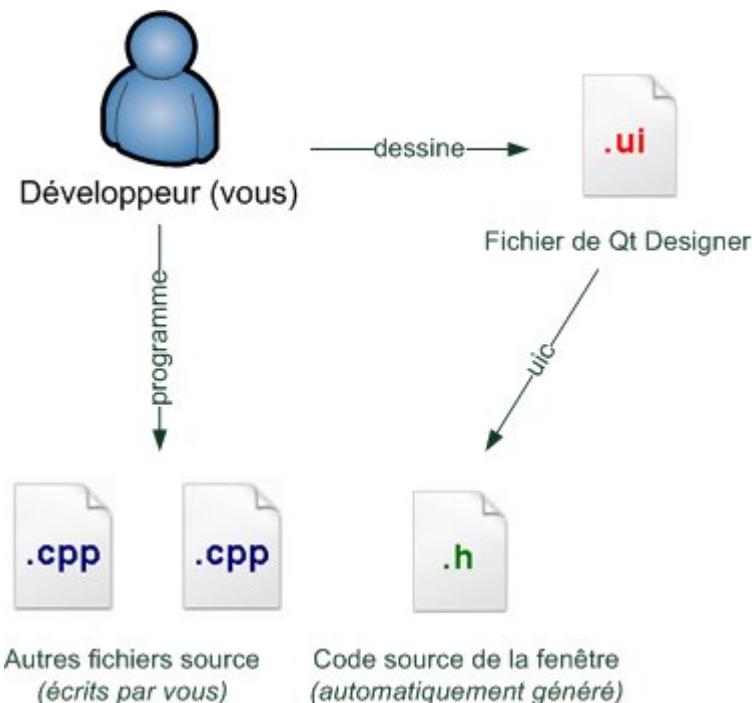
Le principe de la génération du code source

Essayons maintenant de récupérer le code de la fenêtre dans notre application et d'ouvrir cette fenêtre.

Le code ? Quel code ? Je ne vois pas de code moi ?
Qt Designer est censé générer un code source ?

Non, Qt Designer ne fait que produire un fichier .ui. C'est le petit programme uic qui se charge de transformer le .ui en code source C++.

Voilà ce que ça donne schématiquement :



Vous dessinez la fenêtre avec Qt Designer qui produit un fichier .ui.

Ce fichier est transformé automatiquement en code source par le petit programme en ligne de commande uic. Celui-ci générera un

fichier ui_nomDeVotreFenetre.h. Qt met tout le code dans le fichier .h, ne vous étonnez donc pas s'il n'y a pas de .cpp correspondant.

Vous continuez à programmer vos autres fichiers source comme avant (.cpp et .h).

A la compilation, le fichier ui_nomDeVotreFenetre.h sera compilé avec vos autres fichiers source !

Pour information : ce n'est pas obligatoire, mais si vous voulez voir le code source qui sera généré pour votre fenêtre, vous pouvez à tout moment aller dans le menu "Form / Preview Code..." de Qt Designer.

Regardez ce code mais ne l'enregistrez pas. C'est vraiment juste pour avoir une idée de ce à quoi ressemblera le code généré.

Nous allons procéder en 2 étapes :

1. Nous allons mettre à jour le fichier .pro de notre projet, afin que Qt sache que nous avons un fichier .ui et qu'il faudra générer le code source correspondant. Vous n'appellerez pas uic directement, c'est Qt qui le fera pour vous avant la compilation.
2. Puis, nous adapterons le code source dans notre application afin d'ouvrir la fenêtre générée avec Qt Designer.

Préparer notre projet à générer le code

La première étape consiste à mettre à jour le fichier .pro de votre application pour lui faire comprendre que votre programme utilise désormais un .ui créé avec Qt Designer.

Le fichier .pro a normalement été automatiquement généré par `qmake -project`.

Vous avez 2 possibilités :

- Soit vous retapez `qmake -project` dans la console. Dans ce cas, Qt découvrira automatiquement la présence d'un fichier .ui dans le dossier du projet, et modifiera le fichier .pro en conséquence.
- Soit vous éditez à la main le fichier .pro avec un éditeur de texte, et vous rajoutez la ligne surlignée dans le code ci-dessous :

Code : Autre - [Sélectionner](#)

```
#####
# Automatically generated by qmake (2.01a) lun. 2. juin 12:00:20 2008
#####

TEMPLATE = app
TARGET =
DEPENDPATH += .
INCLUDEPATH += .

# Input
FORMS += calculatrice.ui
SOURCES += main.cpp
```

Comme vous pouvez le voir, j'ai ajouté la ligne FORMS. Elle donne la liste des fichiers .ui utilisés par votre application.

Voilà, c'est bon. 😊

Maintenant, faites un `qmake`. Cela aura pour effet de préparer votre code à la compilation en générant un makefile, comme je vous l'avais expliqué au tout début de la partie sur Qt.

Le fichier ui_calculatrice.h sera généré par uic au moment de la compilation, lorsque vous taperez `make` (ne le faites pas maintenant, nous le ferons plus tard).

Vous n'avez rien d'autre à faire, vous n'avez pas besoin d'appeler uic vous-mêmes. Le programme uic sera automatiquement appelé juste avant la compilation.

Utiliser la fenêtre dans notre application

Il nous reste une importante étape : modifier le code source de notre application pour ouvrir la fenêtre créée sous Qt Designer. Et là, nous avons le choix. Nous pouvons utiliser la fenêtre de 3 manières différentes, de la plus simple à la plus compliquée (la plus compliquée étant la meilleure bien sûr 😊) :

- Utilisation directe
- Utilisation avec un héritage simple
- Utilisation avec un héritage multiple

Je vais vous décrire chacune de ces 3 méthodes. 😊

Vous verrez que la dernière, bien que plus complexe, est la plus pratique et la plus souple.

Utilisation directe

Avantages : technique très simple à mettre en oeuvre, à peine quelques lignes à écrire.

Défauts : pas de possibilité de personnaliser la fenêtre, ni d'écrire des slots personnalisés. La fenêtre est "figée".

La technique la plus simple, mais la moins puissante, consiste à utiliser directement la fenêtre générée. On va supposer que votre programme n'est constitué que d'un main(). Ajoutez les lignes surlignées :

Code : C++ - Sélectionner

```
#include <QApplication>
#include <QtGui>
#include "ui_calculatrice.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QWidget *fenetre = new QWidget;
    Ui::FenCalculatrice ui;
    ui.setupUi(fenetre);

    fenetre->show();

    return app.exec();
}
```

Dans un premier temps, on inclut le fichier ui_calculatrice.h qui sera généré par uic juste avant la compilation.

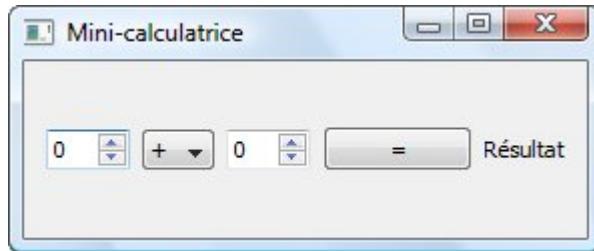
Ensuite, on fait comme si on créait une nouvelle fenêtre en créant un nouvel objet de type QWidget (ligne 9).

Au lieu d'afficher cette fenêtre directement, on la précharge avec le contenu que l'on a dessiné dans Qt Designer. Pour cela, on crée un objet de type Ui::FenCalculatrice (où "FenCalculatrice" est le nom que vous avez donné à votre fenêtre dans Qt Designer). On appelle setupUi(fenetre) pour dessiner le contenu de la fenêtre avec l'interface réalisée sous Qt Designer.

On peut ensuite ouvrir la fenêtre avec fenetre->show(); comme d'habitude. 😊

Compilez maintenant avec make en ligne de commande. Le programme uic sera appelé en arrière-plan pour que le fichier ui_calculatrice.h soit généré. Puis, l'ensemble des fichiers .cpp et .h seront compilés comme d'habitude.

Admirez ensuite le programme ainsi généré :



Ca marche ! 😊

Vous noterez toutefois qu'il y a un défaut : notre fenêtre s'affiche, c'est bien beau, mais elle ne réagit pas au clic sur le bouton "=".

En effet, la méthode que nous venons de voir est très simple, mais elle a un énorme défaut : nous ne pouvons pas créer nos propres slots pour personnaliser un peu le code de la fenêtre.

Les techniques suivantes que nous allons voir nous permettent de le faire, et sont donc bien plus souples.

Utilisation avec un héritage simple

Avantages : on peut personnaliser la fenêtre et écrire nos propres slots.

Défauts : il faut utiliser le préfixe "ui" devant les noms de tous les widgets pour pouvoir les utiliser.

Nous allons hériter de la fenêtre créée avec Qt Designer. Pour faire cela, nous allons créer une nouvelle classe dans notre projet intitulée "FenCalculatrice" (du même nom que la fenêtre créée sous Qt Designer, oui oui). Créez un .cpp et un .h.

Au final, votre projet devrait comporter les fichiers suivants :

- main.cpp
- FenCalculatrice.h
- FenCalculatrice.cpp

Refaites un `qmake -project` pour mettre à jour le fichier .pro de votre projet, afin de vous assurer que les nouveaux fichiers FenCalculatrice.h et FenCalculatrice.cpp seront bien compilés.

Définissez le fichier FenCalculatrice.h comme ceci :

Code : C++ - [Sélectionner](#)

```
#ifndef HEADER_FENCALCULATRICE
#define HEADER_FENCALCULATRICE

#include <QtGui>
#include "ui_calculatrice.h"

class FenCalculatrice : public QWidget
{
    Q_OBJECT

public:
    FenCalculatrice(QWidget *parent = 0);

private slots:
    /* Insérez les prototypes de vos slots personnalisés ici */

private:
    Ui::FenCalculatrice ui;
};

#endif
```

On inclut "ui_calculatrice.h" pour pouvoir utiliser la fenêtre créée avec Qt Designer.

On crée une classe FenCalculatrice héritant de QWidget. Ehhh oui, il faut hériter du même type que la fenêtre créée sous Qt Designer (qui était un QWidget si vous vous souvenez bien).

On crée un constructeur classique.

On déclare un objet "ui" de type Ui::FenCalculatrice. Ca c'est la particularité. L'objet ui contiendra tous les widgets de la fenêtre, vous allez voir.

Maintenant, on va implémenter le constructeur dans FenCalculatrice.cpp. Là, vous allez voir, c'est ultra-simple :

Code : C++ - Sélectionner

```
#include "FenCalculatrice.h"

FenCalculatrice::FenCalculatrice(QWidget *parent) : QWidget(parent)
{
    ui.setupUi(this); // A faire en premier

    /*
    Personnalisez vos widgets ici si nécessaire
    Réalisez des connexions supplémentaires entre signaux et slots
    */
}
```

Tout ce que vous avez à faire, c'est un `ui.setupUi(this)` pour créer le contenu de la fenêtre.

Il faut faire cela en premier dans le constructeur. Ensuite, libre à vous de personnaliser les widgets et de créer des connexions supplémentaires entre des signaux et des slots.

Particularité : tous les widgets sont accessibles en faisant `ui.nomDuWidget`.

Par exemple, on peut changer le texte du bouton comme ceci :

Code : C++ - Sélectionner

```
#include "FenCalculatrice.h"

FenCalculatrice::FenCalculatrice(QWidget *parent) : QWidget(parent)
{
    ui.setupUi(this);

    ui.boutonEgal->setText("Egal");
}
```

Le nom du bouton "boutonEgal", nous l'avons défini dans Qt Designer tout à l'heure (propriété objectName de QObject). Retournez voir le petit tableau un peu plus haut pour vous souvenir de la liste des noms des widgets de la fenêtre.

Bon en général vous n'aurez pas besoin de personnaliser vos widgets, vu que vous avez tout fait sous Qt Designer. Mais si vous avez besoin d'adapter leur contenu à l'exécution (pour afficher le nom de l'utilisateur par exemple), il faudra passer par là.

Maintenant ce qui est intéressant surtout, c'est d'effectuer une connexion :

Code : C++ - Sélectionner

```
#include "FenCalculatrice.h"

FenCalculatrice::FenCalculatrice(QWidget *parent) : QWidget(parent)
{
    ui.setupUi(this);

    connect(ui.boutonEgal, SIGNAL(clicked()), this, SLOT(calculerOperation()));
}
```

N'oubliez pas à chaque fois de mettre le préfixe "ui" devant chaque nom de widget !

Ce code nous permet de faire en sorte que le slot calculerOperation() de la fenêtre soit appelé à chaque fois que l'on clique sur le bouton.

Il ne vous reste plus qu'à adapter votre main pour appeler la fenêtre comme une fenêtre classique :

Code : C++ - Sélectionner

```
#include <QApplication>
#include <QtGui>
#include "FenCalculatrice.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    FenCalculatrice fenetre;
    fenetre.show();

    return app.exec();
}
```

La méthode que nous venons de voir est très pratique et on peut faire tout ce qu'on veut avec, mais il faut écrire le préfixe "ui" devant le nom du widget à chaque fois.

Si vous voulez évitez d'avoir à écrire "ui", il va falloir faire un héritage multiple...

Utilisation avec un héritage multiple

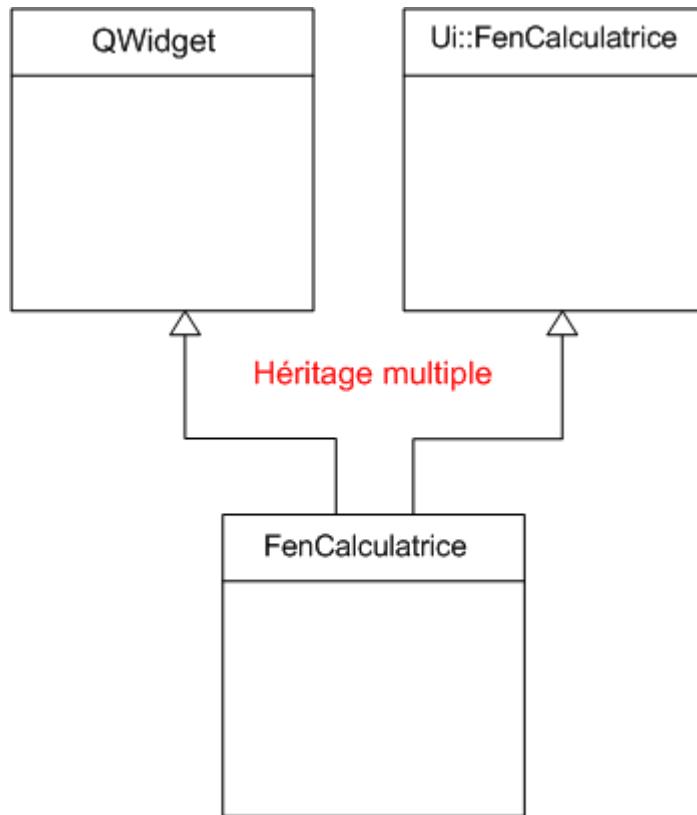
Avantages : on peut personnaliser la fenêtre, écrire nos propres slots, et on n'a pas besoin de mettre le préfixe "ui" devant chaque nom de widget.

Défauts : il faut faire un héritage multiple, une technique un peu plus complexe que l'héritage classique.

L'héritage multiple est une technique complexe du C++ que je ne vous ai pas enseignée jusqu'alors dans le cours. Il faut dire qu'on l'utilise rarement et, bien que cette technique soit puissante, elle est considérée comme trop complexes par certains nouveaux langages (Java, Ruby...) qui ont décidé de ne pas gérer l'héritage multiple.

Bon, le principe c'est quoi ? A priori c'est tout bête : c'est une classe qui hérite de 2 classes (ou plus !).

Dans notre cas, il faut que l'on hérite à la fois de QWidget (le type de la fenêtre) et de Ui::FenCalculatrice (la fenêtre créée sous Qt Designer) !



En pratique, dans la déclaration de la classe (fichier FenCalculatrice.h), ce qui change c'est la ligne 7 :

Code : C++ - [Sélectionner](#)

```

#ifndef HEADER_FENCALCULATRICE
#define HEADER_FENCALCULATRICE

#include <QtGui>
#include "ui_calculatrice.h"

class FenCalculatrice : public QWidget, private Ui::FenCalculatrice
{
    Q_OBJECT

public:
    FenCalculatrice(QWidget *parent = 0);

private slots:
    /* Insérez les prototypes de vos slots personnalisés ici */
};

#endif

```

La seule ligne qui change a été soulignée, c'est celle de déclaration de la classe. On hérite de QWidget et de Ui::FenCalculatrice à la fois.

Vous noterez qu'on n'a plus besoin de définir un objet "ui" de type Ui::FenCalculatrice cette fois.

Dans le fichier .cpp, c'est pareil sauf que maintenant vous pouvez enlever tous les préfixes "ui" :

Code : C++ - [Sélectionner](#)

```

#include "FenCalculatrice.h"

FenCalculatrice::FenCalculatrice(QWidget *parent) : QWidget(parent)
{
    setupUi(this);

    connect(boutonEgal, SIGNAL(clicked()), this, SLOT(calculerOperation()));
}

```

Personnaliser le code et utiliser les Auto-Connect

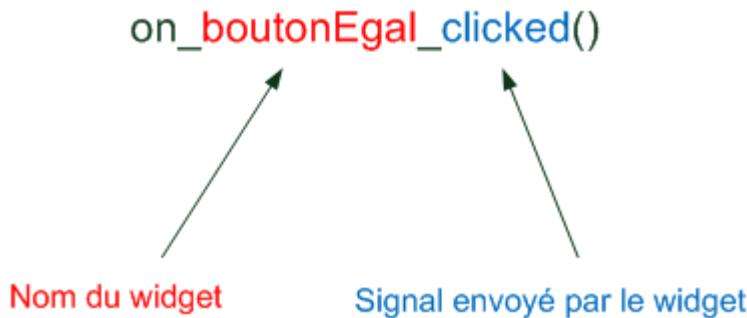
Les fenêtres créées avec Qt Designer bénéficient du système "Auto-Connect" de Qt. C'est un système qui crée les connexions tout seul.

Par quelle magie ?

Il vous suffit en fait de créer des slots en leur donnant un nom qui respecte une convention.

Prenons le widget `boutonEgal` et son signal `clicked()`. Si vous créez un slot appelé `on_boutonEgal_clicked()` dans votre fenêtre, ce slot sera automatiquement appelé lors d'un clic sur le bouton.

La convention à respecter est représentée sur le schéma ci-dessous :



Essayons d'utiliser l'Auto-Connect dans notre programme. Je me base ici sur un héritage multiple.

Voici le .h :

Code : C++ - [Sélectionner](#)

```

#ifndef HEADER_FENCALCULATRICE
#define HEADER_FENCALCULATRICE

#include <QtGui>
#include "ui_calculatrice.h"

class FenCalculatrice : public QWidget, private Ui::FenCalculatrice
{
    Q_OBJECT

public:
    FenCalculatrice(QWidget *parent = 0);

private slots:
    void on_boutonEgal_clicked();
};

#endif

```

... et le .cpp :

Code : C++ - Sélectionner

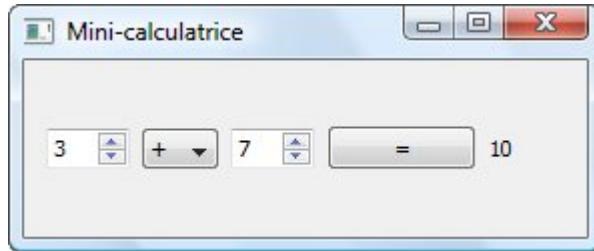
```
#include "FenCalculatrice.h"

FenCalculatrice::FenCalculatrice(QWidget *parent) : QWidget(parent)
{
    setupUi(this);
}

void FenCalculatrice::on_boutonEgal_clicked()
{
    int somme = nombre1->value() + nombre2->value();
    resultat->setNum(somme);
}
```

Vous noterez qu'on n'a plus besoin de faire de connexion dans le constructeur. Ben oui, c'est le principe de l'Auto-Connect. 🍪 Comme vous le voyez, il suffit de créer un slot avec un nom particulier, et tout roule comme sur des roulettes !

Vous pouvez tester le programme, ça marche !



Bon, j'avoue, je n'ai géré ici que l'addition. Mais je vais pas tout vous faire non plus hein. 🍪

Exercice (me dites pas que vous l'avez pas vu venir 🍪) : complétez le code de la calculatrice pour effectuer la bonne opération en fonction de l'élément sélectionné dans la liste déroulante.

L'Auto-Connect est activé par défaut dans les fenêtres créées avec Qt Designer, mais vous pouvez aussi vous en servir dans vos autres fenêtres "faites main".

Il suffira d'ajouter la ligne suivante dans le constructeur de la fenêtre pour bénéficier de toute la puissance de l'Auto-Connect : `QMetaObject::connectSlotsByName(this);`

Ceux qui croyaient que Qt Designer était un "programme magique qui allait réaliser des fenêtres tout seul sans avoir besoin de coder" en ont été pour leurs frais ! 🤡

Pourtant, comme avec Qt Linguist, le processus de création de fenêtres de Qt Designer a été très bien pensé. Tout est logique et s'enchaîne de bout en bout, mais encore faut-il comprendre cette logique. J'espère vous y avoir aidé à travers ce chapitre.

J'ai profité de ce chapitre pour survoler la notion d'héritage multiple. Il y aurait à redire sur cette notion, mais c'est un peu complexe et on peut se contenter de retenir ce qui a été dit dans ce chapitre si on a juste besoin d'utiliser des fenêtres créées avec Qt Designer.

Si vous voulez en savoir plus (et je vous y encourage), il vous faudra chercher d'autres cours sur le web traitant de l'héritage multiple en C++.

Entraînez-vous à utiliser quelques fenêtres créées avec Qt Designer, et en particulier à créer des slots personnalisés. Tant qu'à faire, je vous conseille de vous servir de l'Auto-Connect. Une fois qu'on y a goûté on ne peut plus s'en passer. 🍪

TP : zNavigo, le navigateur web des Zéros !

Depuis le temps que vous pratiquez Qt, vous avez acquis sans vraiment le savoir les capacités de base pour réaliser des programmes

complexes. Le but d'un TP comme celui-ci, c'est de vous montrer justement que vous êtes capables de mener à bien des projets qui auraient pu vous sembler complètement fous il y a quelques temps.

Vous ne rêvez pas : le but de ce TP sera de... réaliser un navigateur web !



Quoi ? Je suis capable de faire ça moi ? 😊

Oui, et nous allons voir comment dans ce chapitre !

Nous allons commencer dans un premier temps par découvrir la notion de moteur web, pour bien comprendre comment fonctionnent les autres navigateurs. Puis, nous mettrons en place le plan du développement de notre programme afin de nous assurer que nous partons dans la bonne direction et que nous n'oublions rien.

Firefox n'a qu'à bien se tenir. 😈

Les navigateurs et les moteurs web

Comme toujours, il faut d'abord prendre le temps de réfléchir à son programme avant de foncer le coder tête baissée. C'est ce qu'on appelle la phase de conception.

Je sais, je me répète à chaque fois, mais c'est vraiment parce que c'est très important. Si je vous dis "faites-moi un navigateur web" et que vous créez de suite un nouveau projet en vous demandant ce que vous allez bien pouvoir mettre dans le main, ben... c'est le ramassage assuré. 🍄

Pour moi, la conception est l'étape la plus difficile du projet. Plus difficile même que le code. En effet, si vous concevez bien votre programme, si vous réfléchissez bien à la façon dont il doit fonctionner, vous aurez simplifié à l'avance votre projet et vous n'aurez pas à écrire des lignes de code difficiles inutilement.

Dans un premier temps, je vais vous expliquer comment fonctionne un navigateur web. Un peu de culture générale à ce sujet vous permettra de mieux comprendre ce que vous avez à faire (et ce que vous n'avez pas à faire).

Je vous donnerai ensuite quelques conseils pour organiser votre code : quelles classes créer, par quoi commencer, etc.

Les principaux navigateurs

Commençons par le commencement : vous savez ce qu'est un navigateur web ?

Bon, je ne me moque pas de vous, mais il vaut mieux être sûr de ne perdre personne. 😊

Un navigateur web est un programme qui permet de consulter des sites web.

Parmi les plus connus d'entre eux, citons Internet Explorer, Mozilla Firefox ou encore Safari. Mais il y en a aussi beaucoup d'autres, certes moins utilisés, comme Opera, Konqueror, Epiphany, Maxthon, Lynx...

Je vous rassure, il n'est pas nécessaire de tous les connaître pour pouvoir prétendre en créer un.
Par contre, ce qu'il faut que vous sachiez, c'est que chacun de ces navigateurs est constitué de ce qu'on appelle un moteur web.
Qu'est-ce que c'est que cette bête-là ?

Le moteur web

Tous les sites web sont écrits en langage HTML (ou XHTML). Voici un exemple de code HTML permettant de créer une page très simple :

Code : HTML - Sélectionner

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="fr" >
  <head>
    <title>Bienvenue sur mon site !</title>
    <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
  </head>
  <body>
    </body>

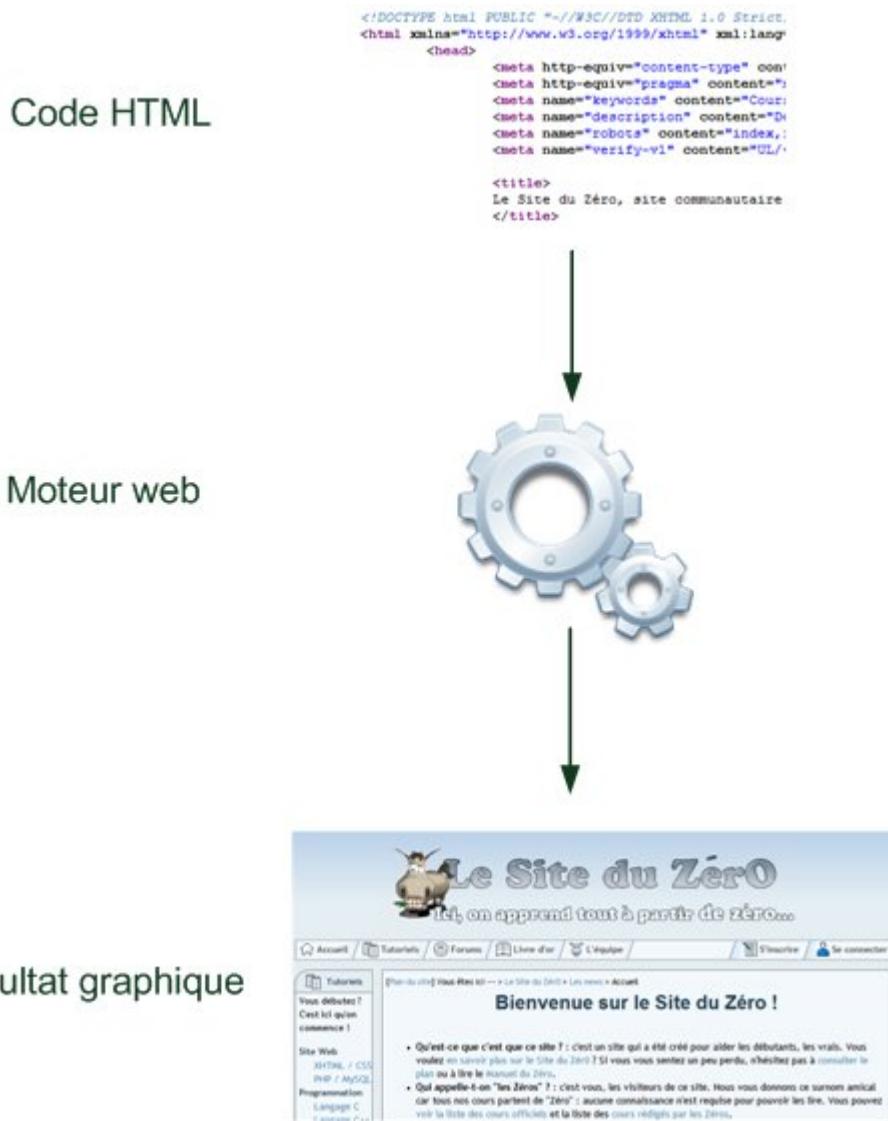
```

3

4

C'est bien joli tout ce code, mais ça ne ressemble pas au résultat visuel qu'on a l'habitude de voir lorsqu'on navigue sur le web.

L'objectif est justement de transformer ce code en un résultat visuel : le site web. C'est le rôle du moteur web. Voici son fonctionnement, résumé dans un schéma très simple :



Ca n'a l'air de rien, mais c'est un travail difficile : réaliser un moteur web est très délicat. C'est généralement le fruit des efforts de nombreux programmeurs experts (et encore, ils avouent avoir du mal 😊). Certains moteurs sont meilleurs que d'autres, mais aucun n'est parfait ni complet. Comme le web est en perpétuelle évolution, il est peu probable qu'un moteur parfait sorte un jour.

Quand on programme un navigateur, on utilise généralement le moteur web sous forme de bibliothèque. Le moteur web n'est donc pas un programme, mais il est utilisé par des programmes.



Ce sera plus clair avec un schéma. 😊

Regardons comment est constitué Firefox par exemple :



On voit que le navigateur (en vert) "contient" le moteur web (en jaune au centre).

La partie en vert est habituellement appelée le "chrome", pour désigner l'interface.

Mais c'est nul ! Alors le navigateur web c'est juste les 2-3 boutons en haut et c'est tout ?

Oh non ! Loin de là.

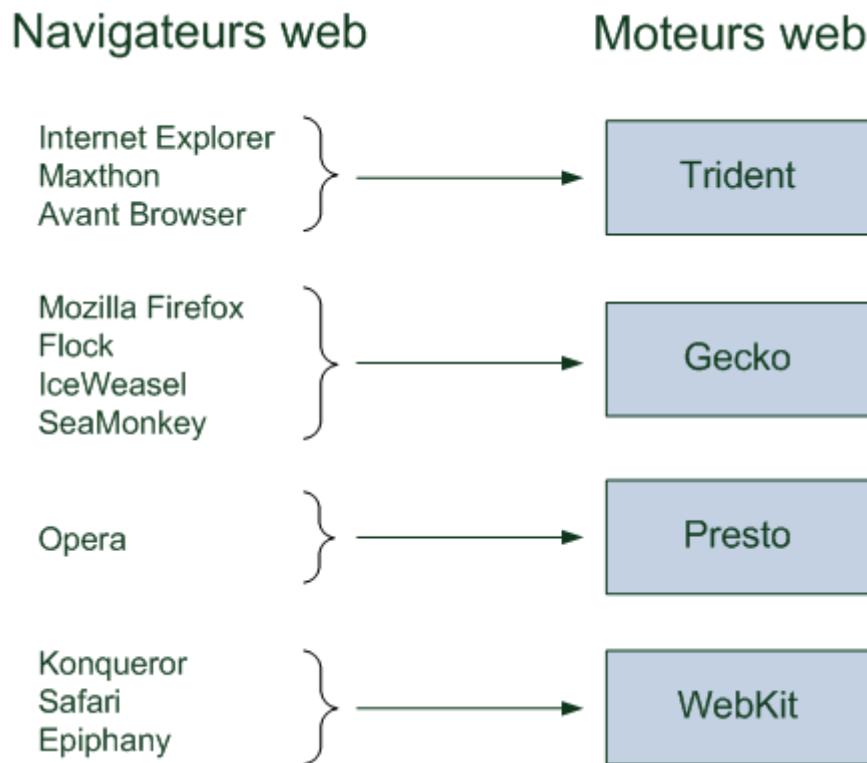
Le navigateur ne se contente pas de gérer les boutons "Précédente", "Suivante", "Actualiser", etc. C'est aussi lui qui gère les marque-pages (favoris), le système d'onglets, les options d'affichage, la barre de recherche, etc.

Tout cela représente déjà un énorme travail ! En fait, les développeurs de Firefox ne sont pas les mêmes que ceux qui développent son moteur web. Il y a des équipes séparées, tellement chacun de ces éléments représente du travail.

Les principaux navigateurs et leurs moteurs

Un grand nombre de navigateurs ne s'occupent pas du moteur web. Ils en utilisent un "tout prêt".

De nombreux navigateurs sont basés sur le même moteur web. Voici un petit schéma de mon crû qui vous permet de vous donner une idée un peu de "qui utilise quoi" :



Les noms des moteurs web ne sont pas connus du grand public. D'ailleurs, il est probable que vous n'ayez entendu parler d'aucun d'eux jusqu'à aujourd'hui. Ce qui est connu, c'est le navigateur, alors que c'est le moteur web qui se tape tout le sale boulot. 😊

Je n'ai pas mis tous les navigateurs et moteurs web existants, mais cela permet déjà d'avoir une bonne idée de ce qui se passe. Comme vous le voyez, rares sont les navigateurs à avoir leur propre moteur web. On peut noter l'exception d'Opera (et encore, le moteur a été revendu à Adobe qui ne voulait pas en coder un pour son logiciel Dreamweaver).

Tout ça pour dire quoi ? Eh bien déjà que créer un moteur web n'est ni de votre niveau, ni du mien. Comme de nombreux navigateurs, nous en utiliserons un déjà existant.

Lequel ? Eh bien il se trouve que Qt (oui, parce qu'on parle de Qt ici, j'espère que vous n'avez pas oublié 😊), Qt donc vous propose depuis peu d'utiliser le moteur WebKit dans vos programmes. C'est donc ce moteur-là que nous allons utiliser pour créer notre navigateur.

Configurer son projet pour utiliser WebKit

WebKit est un des nombreux modules de Qt. Il ne fait pas partie du module "GUI", dédié à la création de fenêtres, il s'agit d'un module à part.

Pour pouvoir l'utiliser, il faudra modifier le fichier .pro du projet pour que Qt sache qu'il a besoin de charger WebKit. Voici un exemple de fichier .pro qui indique que le projet utilise WebKit :

Code : Autre - [Sélectionner](#)

```
#####
# Automatically generated by qmake (2.01a) mer. 18. juin 11:49:49 2008
#####

TEMPLATE = app
QT += webkit
TARGET =
DEPENDPATH += .
INCLUDEPATH += .

# Input
HEADERS += FenPrincipale.h
SOURCES += FenPrincipale.cpp main.cpp
```

Le plus simple est de faire un `qmake -project` d'abord pour générer le fichier `.pro`, puis de rajouter la ligne que j'ai surlignée : `QT += webkit`

D'autre part, vous devrez rajouter l'include suivant dans les fichiers de votre code source faisant appel à WebKit :

Code : C++ - [Sélectionner](#)

```
#include <QtWebKit>
```

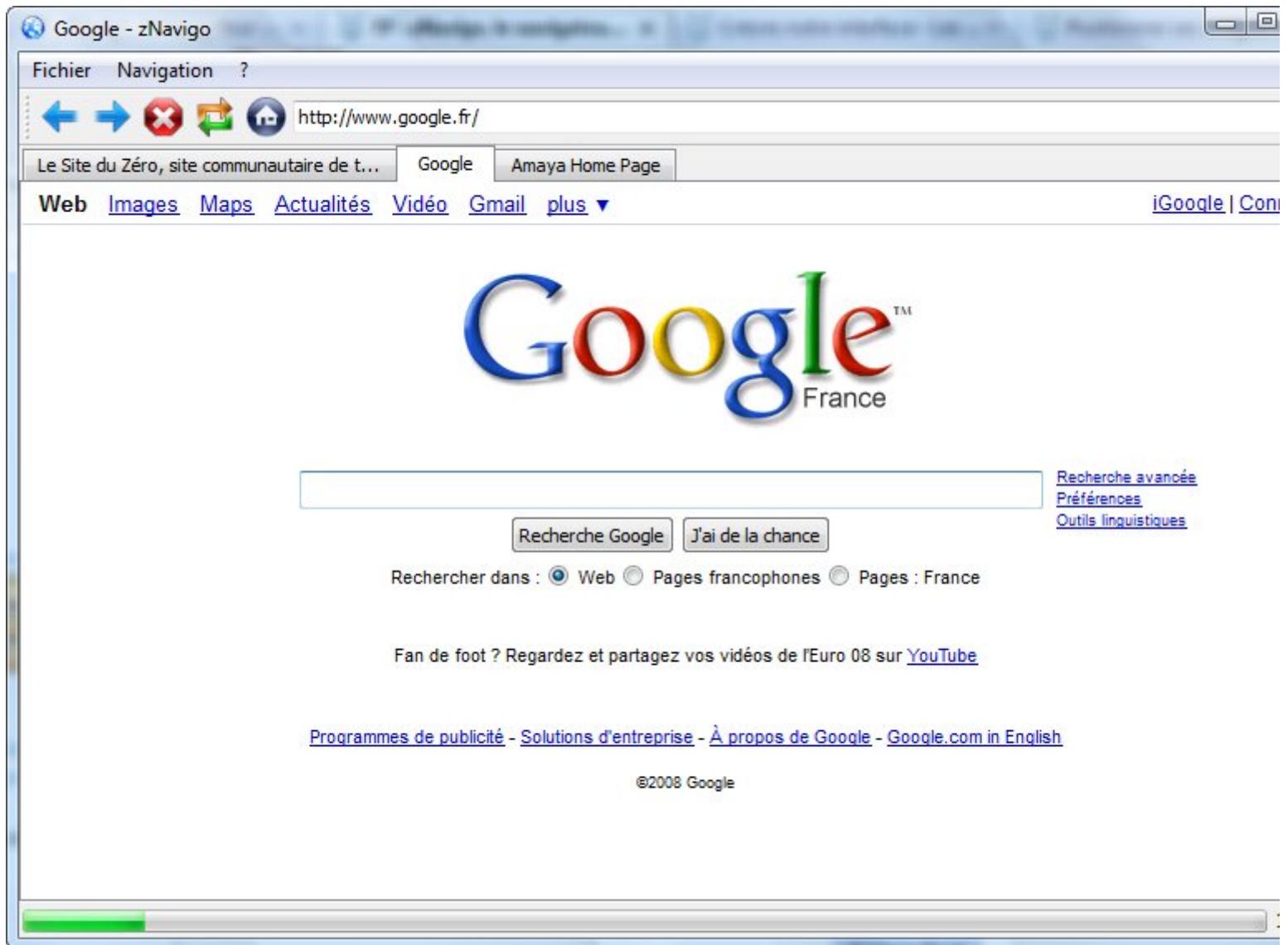
Enfin, il faudra joindre 2 nouvelles DLL à votre programme pour qu'il fonctionne : `QtWebKit4.dll` et `QtNetwork4.dll`.

Ouf, tout est prêt. 😊

Organisation du projet

Objectif

Avant d'aller plus loin, il me semble indispensable de vous montrer à quoi doit ressembler le navigateur une fois terminé. Votre objectif est de réaliser un navigateur web semblable à celui-ci :



Parmi les fonctionnalités de ce super navigateur, affectueusement nommé "zNigo", on compte :

- Accès aux pages précédentes et suivantes
- Arrêter le chargement de la page
- Actualiser la page
- Retour à la page d'accueil
- Saisie d'une adresse
- Navigation par onglets
- Affichage du pourcentage de chargement dans la barre d'état

Le menu "Fichier" propose d'ouvrir et de fermer un onglet, ainsi que de quitter le programme.

Le menu "Navigation" reprend le contenu de la barre d'outils (ce qui est très facile à faire grâce aux QAction je vous le rappelle). Le menu "?" (aide) propose d'afficher les fenêtres "A propos..." et "A propos de Qt..." qui donnent des informations respectivement sur notre programme et sur Qt.

Ca n'a l'air de rien comme ça, mais ça représente déjà un sacré boulot !

Si vous avez du mal dans un premier temps, vous pouvez vous épargner la gestion des onglets... mais moi j'ai trouvé que c'était un peu trop simple sans les onglets, alors j'ai choisi de vous faire jouer avec, histoire de corser le tout. 

Les fichiers du projet

J'ai l'habitude de faire une classe par fenêtre. Comme notre projet ne sera constitué (au moins dans un premier temps) que d'une

seule fenêtre, nous aurons donc les fichiers suivants :

- main.cpp
- FenPrincipale.h
- FenPrincipale.cpp

Si vous voulez utiliser les mêmes icônes que moi, les voici :



Notez que la dernière est l'icône du programme (affichée en haut à gauche).

Toutes ces icônes sont sous licence LGPL et proviennent du site <http://www.veraldo.com>

Utiliser QWebView pour afficher une page web

Le [QWebView](#) est le principal nouveau widget que vous aurez besoin d'utiliser dans ce chapitre. Il permet d'afficher une page web. C'est lui le moteur web.

Vous ne savez pas vous en servir, mais vous savez maintenant [lire la doc](#). Vous allez voir, ce n'est pas bien difficile !

Regardez en particulier les signaux et slots proposés par le QWebView. Il y a tout ce qu'il faut savoir pour, par exemple, connaître le pourcentage de chargement de la page pour le répercuter sur la barre de progression de la barre d'état (signal `loadProgress (int)`). 😊

Comme l'indique la doc, pour créer le widget et charger une page, c'est très simple :

Code : C++ - [Sélectionner](#)

```
QWebView *pageWeb = new QWebView;
pageWeb->load(QUrl("http://www.siteduzero.com/"));
```

Voilà c'est tout ce que je vous expliquerai sur QWebView, pour le reste lisez la doc. 🍪

La navigation par onglets

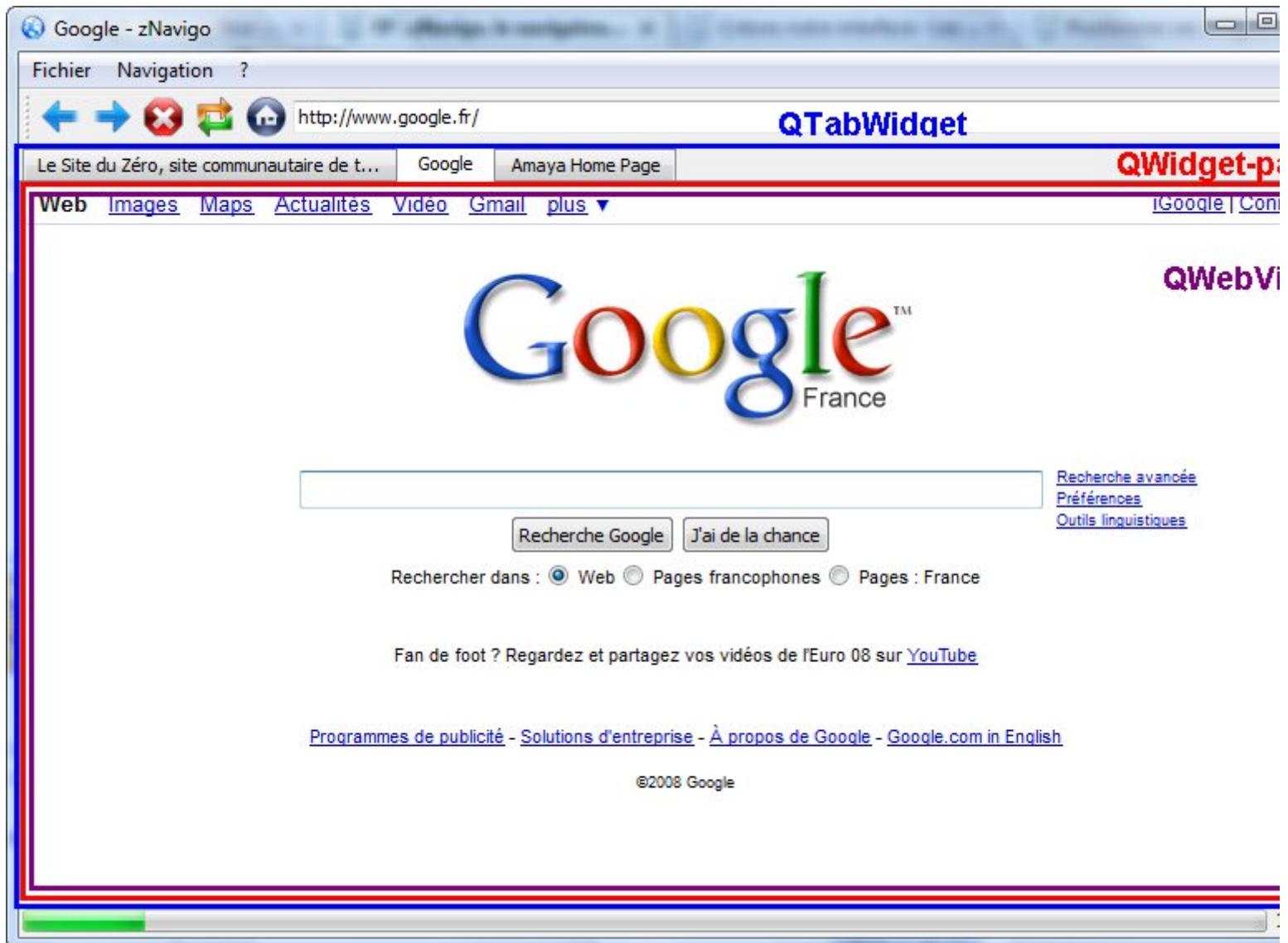
Le problème de QWebView, c'est qu'il ne permet d'afficher qu'une seule page web à la fois. Il ne gère pas la navigation par onglets. Il va falloir implémenter le système d'onglets nous-mêmes.

Vous n'avez jamais entendu parler de QTabWidget ? Si si, souvenez-vous, nous l'avons découvert dans un des chapitres précédents. Ce widget-conteneur est capable d'accueillir n'importe quels widgets... comme un QWebView !

En combinant un QTabWidget et des QWebView (un par onglet), vous pourrez reconstituer un véritable navigateur par onglets !

Une petite astuce toutefois, qui pourra vous être bien utile : savoir retrouver un widget contenu dans un widget parent. Comme vous le savez, le QTabWidget utilise des sous-widgets pour gérer chacune des pages. Ces sous-widgets sont généralement des QWidget génériques (invisibles), qui servent à contenir d'autres widgets.

Dans notre cas : [QTabWidget](#) contient des [QWidget](#) (pages d'onglet) qui eux-mêmes contiennent chacun un [QWebView](#) (la page web).



La méthode `findChild` (définie dans `QObject`) permet de retrouver le widget enfant contenu dans le widget parent.

Par exemple, si je connais le QWidget "pageOnglet", je peux retrouver le QWebView qu'il contient comme ceci :

Code : C++ - [Sélectionner](#)

```
QWebView *pageWeb = pageOnglet->findChild<QWebView *>();
```

Mieux encore, je vous donne la méthode toute faite qui permet de retrouver le QWebView actuellement visualisé par l'utilisateur :

Code : C++ - [Sélectionner](#)

```
QWebView *FenPrincipale::pageActuelle()
{
    return onglets->currentWidget()->findChild<QWebView *>();
}
```

```
onglets->currentWidget()->findChild<QWebView *>();
```



"onglets" correspond au **QTabWidget**.

Sa méthode **currentWidget()** permet d'obtenir un pointeur vers le **QWidget** qui sert de page pour la page actuellement affichée. On demande ensuite à retrouver le **QWebView** que le **QWidget** contient à l'aide de la méthode **findChild()**. Cette méthode utilise les templates C++ (avec **<QWebView *>**). Je ne vais pas rentrer dans les détails ici car ce serait un peu trop long, mais en gros cela permet de faire en sorte que la méthode retourne bien un **QWebView *** (sinon elle n'aurait pas su quoi renvoyer).

J'admet, c'est un petit peu compliqué, mais au moins ça pourra vous aider. 😊

Let's go !

Voilà, vous savez déjà tout ce qu'il faut pour vous en sortir.

Notez que ce TP fait la part belle à la **QMainWindow**, n'hésitez donc pas à relire ce chapitre dans un premier temps pour bien vous remémorer son fonctionnement.

Pour ma part, j'ai choisi de coder la fenêtre "à la main" (pas de **Qt Designer** donc) car celle-ci est un peu complexe.

Comme il y a beaucoup d'initialisations à faire dans le constructeur, je vous conseille de les placer dans des méthodes que vous appellerez depuis le constructeur pour améliorer la lisibilité :

Code : C++ - [Sélectionner](#)

```
FenPrincipale::FenPrincipale()
{
    creerActions();
    creerMenus();
    creerBarresOutils();

    /* Autres initialisations */

}
```

Bon courage ! 😊

Génération de la fenêtre principale

Commençons par les choses simples (et un peu répétitives).

main.cpp

Tout d'abord le **main.cpp**, qui ne devrait pas vous perturber :

Code : C++ - [Sélectionner](#)

```
#include <QApplication>
#include <QTranslator>
#include <QLocale>
#include <QLibraryInfo>
#include "FenPrincipale.h"

int main(int argc, char* argv[])
{
    QApplication app(argc, argv);

    // Traduction des chaînes prédéfinies par Qt dans notre langue
    QString locale = QLocale::system().name();
    QTranslator translator;
    translator.load(QString("qt_") + locale, QLibraryInfo::location(QLibraryInfo::TranslationsPath));
    app.installTranslator(&translator);

    // Ouverture de la fenêtre principale du navigateur
    FenPrincipale principale;
    principale.show();

    return app.exec();
```

3

1

5

4

6

Je me contente d'ouvrir la fenêtre principale. Les quelques lignes de code au début sont celles que je vous avais données il y a quelques chapitres, pour faire en sorte que les chaînes de caractères de base (Yes / No) soient traduites en français dans l'application.

Maintenant, créons la classe FenPrincipale, notre plus gros morceau.

FenPrincipale.h (première version)

Dans un premier temps, je ne crée que le squelette de la classe et ses premières méthodes, j'en rajouterai d'autres au fur et à mesure si besoin est.

Code : C++ - [Sélectionner](#)

```

#ifndef HEADER_FENPRINCIPALE
#define HEADER_FENPRINCIPALE

#include <QtGui>
#include <QtWebKit>

class FenPrincipale : public QMainWindow
{
    Q_OBJECT

public:
    FenPrincipale();

private:
    void creerActions();
    void creerMenus();
    void creerBarresOutils();
    void creerBarreEtat();

private slots:

private:
    QTabWidget *onglets;

    QAction *actionNouvelOnglet;
    QAction *actionFermerOnglet;
    QAction *actionQuitter;
    QAction *actionAPropos;
    QAction *actionAProposQt;
    QAction *actionPrecedente;
    QAction *actionSuivante;
    QAction *actionStop;
    QAction *actionActualiser;
    QAction *actionAccueil;
    QAction *actionGo;

    QLineEdit *champAdresse;
    QProgressBar *progression;
};

#endif

```

La classe hérite de QMainWindow comme prévu. J'ai inclus QtGui et QtWebKit pour pouvoir utiliser le module GUI et le module WebKit (moteur web).

Mon idée c'est, comme je vous l'avais dit, de couper le constructeur en plusieurs sous-méthodes qui s'occupent chacune de créer une section différente de la QMainWindow : actions, menus, barre d'outils, barre d'état...

J'ai prévu une section pour les slots personnalisés mais je n'ai encore rien mis, je verrai au fur et à mesure.

Enfin, j'ai préparé les principaux attributs de la classe. En fin de compte, à part de nombreuses QAction, il n'y en a pas beaucoup. Je n'ai même pas eu besoin de mettre des objets de type QWebView : ceux-ci seront créés à la volée au cours du programme et on pourra les retrouver grâce à la méthode pageActuelle() que je vous ai donnée un peu plus tôt.

Voyons voir l'implémentation du constructeur et de ses sous-méthodes qui génèrent le contenu de la fenêtre.

Construction de la fenêtre

Direction FenPrincipale.cpp, on commence par le constructeur :

Code : C++ - [Sélectionner](#)

```
#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
    // Génération des widgets de la fenêtre principale
    creerActions();
    creerMenus();
    creerBarresOutils();
    creerBarreEtat();

    // Génération des onglets et chargement de la page d'accueil
    onglets = new QTabWidget;
    onglets->addTab(creerOngletPageWeb(tr("http://www.siteduzero.com")), tr("(Nouvelle pa
    connect(onglets, SIGNAL(currentChanged(int)), this, SLOT(changementOnglet(int)));
    setCentralWidget(onglets);

    // Définition de quelques propriétés de la fenêtre
    setMinimumSize(500, 350);
    setWindowIcon(QIcon("images/znavigo.png"));
    setWindowTitle(tr("zNavigo"));
}

1
3
4
5
6
```

Nous allons voir juste après le code des méthodes `creerActions()`, `creerMenus()`, etc. Ce code est un peu long et répétitif, pas très intéressant mais il fallait le faire.

Par contre, ce qui est intéressant ensuite dans le constructeur, c'est que l'on crée le `QTabWidget` et on lui ajoute un premier onglet. Pour la création d'un onglet, on va faire appel à une méthode "maison" `creerOngletPageWeb()` qui va se charger de créer le QWidget-page de l'onglet, ainsi que de créer un `QWebView` et de lui faire charger la page web envoyée en paramètre ("`http://www.siteduzero.com`" sera donc la page d'accueil par défaut .

Vous noterez que l'on utilise la fonction de `tr()` partout, au cas où on voudrait traduire le programme par la suite. C'est une bonne habitude à prendre, même si on n'a pas forcément l'intention de traduire le programme au début (on peut toujours changer d'avoir après).

On connecte enfin et surtout le signal `currentChanged()` du `QTabWidget` à un slot personnalisé `changementOnglet()` que l'on va devoir écrire. Ce slot sera appelé à chaque fois que l'utilisateur change d'onglet, pour, par exemple, mettre à jour l'URL dans la barre d'adresse ainsi que le titre de la page affiché en haut de la fenêtre.

Bon, il faut maintenant écrire les méthodes de génération des actions, des menus, etc. C'était un peu long et fastidieux mais je suis arrivé jusqu'au bout. 

Ne vous laissez pas impressionner par la taille du code, je n'ai pas tout écrit d'un coup, j'y suis allé petit à petit.

Code : C++ - [Sélectionner](#)

```

void FenPrincipale::creerActions()
{
    actionNouvelOnglet = new QAction(tr("&Nouvel onglet"), this);
    actionNouvelOnglet->setShortcut(tr("Ctrl+T"));
    connect(actionNouvelOnglet, SIGNAL(triggered()), this, SLOT(nouvelOnglet()));
    actionFermerOnglet = new QAction(tr("&Fermer l'onglet"), this);
    actionFermerOnglet->setShortcut(tr("Ctrl+W"));
    connect(actionFermerOnglet, SIGNAL(triggered()), this, SLOT(fermerOnglet()));
    actionQuitter = new QAction(tr("&Quitter"), this);
    actionQuitter->setShortcut(tr("Ctrl+Q"));
    connect(actionQuitter, SIGNAL(triggered()), qApp, SLOT(quit()));

    actionPrecedente = new QAction(QIcon("images/precedente.png"), tr("&Precedente"), this);
    actionPrecedente->setShortcut(QKeySequence::Back);
    connect(actionPrecedente, SIGNAL(triggered()), this, SLOT(precedente()));
    actionSuivante = new QAction(QIcon("images/suivante.png"), tr("&Suivante"), this);
    actionSuivante->setShortcut(QKeySequence::Forward);
    connect(actionSuivante, SIGNAL(triggered()), this, SLOT(suivante()));
    actionStop = new QAction(QIcon("images/stop.png"), tr("S&top"), this);
    connect(actionStop, SIGNAL(triggered()), this, SLOT(stop()));
    actionActualiser = new QAction(QIcon("images/actualiser.png"), tr("&Actualiser"), this);
    actionActualiser->setShortcut(QKeySequence::Refresh);
    connect(actionActualiser, SIGNAL(triggered()), this, SLOT(actualiser()));
    actionAccueil = new QAction(QIcon("images/accueil.png"), tr("A&ccueil"), this);
    connect(actionAccueil, SIGNAL(triggered()), this, SLOT(accueil()));
    actionGo = new QAction(QIcon("images/go.png"), tr("A&ller à"), this);
    connect(actionGo, SIGNAL(triggered()), this, SLOT(chargerPage()));

    actionAPropos = new QAction(tr("&A propos..."), this);
    connect(actionAPropos, SIGNAL(triggered()), this, SLOT(aPropos()));
    actionAProposQt = new QAction(tr("A propos de &Qt..."), this);
    connect(actionAProposQt, SIGNAL(triggered()), qApp, SLOT(aboutQt()));
}

void FenPrincipale::creerMenus()
{
    QMenu *menuFichier = menuBar()->addMenu(tr("&Fichier"));

    menuFichier->addAction(actionNouvelOnglet);
    menuFichier->addAction(actionFermerOnglet);
    menuFichier->addSeparator();
    menuFichier->addAction(actionQuitter);

    QMenu *menuNavigation = menuBar()->addMenu(tr("&Navigation"));

    menuNavigation->addAction(actionPrecedente);
    menuNavigation->addAction(actionSuivante);
    menuNavigation->addAction(actionStop);
    menuNavigation->addAction(actionActualiser);
    menuNavigation->addAction(actionAccueil);

    QMenu *menuAide = menuBar()->addMenu(tr("&?"));

    menuAide->addAction(actionAPropos);
    menuAide->addAction(actionAProposQt);
}

void FenPrincipale::creerBarresOutils()
{
    champAdresse = new QLineEdit;
    connect(champAdresse, SIGNAL(returnPressed()), this, SLOT(chargerPage()));

    QToolBar *toolBarNavigation = addToolBar(tr("Navigation"));

    toolBarNavigation->addAction(actionPrecedente);
    toolBarNavigation->addAction(actionSuivante);
}

```

Ce code ne fait rien d'extraordinaire nouveau, je ne vois pas trop ce que je pourrais commenter. Il y a beaucoup de connexions à des slots personnalisés que l'on devra écrire.

C'était la partie "longue" du code, mais certainement pas la plus complexe.

Voyons maintenant quelques méthodes qui s'occupent de gérer les onglets...

Méthodes de gestion des onglets

En fait, il n'y a que 2 méthodes dans cette catégorie :

- `creerOngletPageWeb()` : je vous en ai parlé dans le constructeur, elle se charge de créer un QWidget-page ainsi qu'un QWebView à l'intérieur, et de retourner ce QWidget-page à l'appelant pour qu'il puisse créer le nouvel onglet.
- `pageActuelle()` : une méthode bien pratique que je vous ai donnée un peu plus tôt, qui permet à tout moment d'obtenir un pointeur vers le QWebView de l'onglet actuellement sélectionné.

Voici ces méthodes :

Code : C++ - Sélectionner

```
QWidget *FenPrincipale::creerOngletPageWeb(QString url)
{
    QWidget *pageOnglet = new QWidget;
    QWebView *pageWeb = new QWebView;

    QVBoxLayout *layout = new QVBoxLayout;
    layout->setContentsMargins(0,0,0,0);
    layout->addWidget(pageWeb);
    pageOnglet->setLayout(layout);

    if (url.isEmpty())
    {
        pageWeb->load(QUrl(tr("html/page_blanche.html")));
    }
    else
    {
        if (url.left(7) != "http://")
        {
            url = "http://" + url;
        }
        pageWeb->load(QUrl(url));
    }

    // Gestion des signaux envoyés par la page web
    connect(pageWeb, SIGNAL(titleChanged(QString)), this, SLOT(changementTitre(QString)));
    connect(pageWeb, SIGNAL(urlChanged(QUrl)), this, SLOT(changementUrl(QUrl)));
    connect(pageWeb, SIGNAL(loadStarted()), this, SLOT(changementDebut()));
    connect(pageWeb, SIGNAL(loadProgress(int)), this, SLOT(changementEnCours(int)));
    connect(pageWeb, SIGNAL(loadFinished(bool)), this, SLOT(changementTermine(bool)));

    return pageOnglet;
}

QWebView *FenPrincipale::pageActuelle()
{
    return onglets->currentWidget()->findChild<QWebView *>();
}
```

Je ne commente pas pageActuelle(), je l'ai déjà fait auparavant.

Pour ce qui est de creerOngletPageWeb(), elle crée comme prévu un QWidget et elle place un nouveau QWebView à l'intérieur. La page web charge l'URL indiquée en paramètre, et rajoute le "http://" en préfixe si celui-ci a été oublié.

Si aucune URL n'a été spécifiée, on charge une page blanche. J'ai pour l'occasion créé un fichier HTML vide, placé dans un sous-dossier "html" du programme.

On connecte plusieurs signaux intéressants envoyés par le QWebView, qui, à mon avis, parlent d'eux-mêmes : "Le titre a changé", "L'URL a changé", "Début du chargement", "Chargement en cours", "Chargement terminé".

Bref, rien de sorcier, mais ça fait encore tout plein de slots personnalisés à écrire tout ça ! 

Les slots personnalisés

Bon, il y a de quoi faire. Reprenons notre FenPrincipale.h, que voici maintenant en version complète avec toutes les méthodes et tous les slots.

FenPrincipale.h (version complète)

Code : C++ - [Sélectionner](#)

```

#ifndef HEADER_FENPRINCIPALE
#define HEADER_FENPRINCIPALE

#include <QtGui>
#include <QtWebKit>

class FenPrincipale : public QMainWindow
{
    Q_OBJECT

public:
    FenPrincipale();

private:
    void creerActions();
    void creerMenus();
    void creerBarresOutils();
    void creerBarreEtat();
    QWidget *creerOngletPageWeb(QString url = "");
    QWebView *pageActuelle();

private slots:
void precedente();
void suivante();
void accueil();
void stop();
void actualiser();

void aPropos();
void nouvelOnglet();
void fermerOnglet();
void chargerPage();
void changementOnglet(int index);

void changementTitre(const QString & titreComplet);
void changementUrl(const QUrl & url);
void chargementDebut();
void chargementEnCours(int pourcentage);
void chargementTermine(bool ok);

private:
    QTabWidget *onglets;

    QAction *actionNouvelOnglet;
    QAction *actionFermerOnglet;
    QAction *actionQuitter;
    QAction *actionAPropos;
    QAction *actionAProposQt;
    QAction *actionPrecedente;
    QAction *actionSuivante;
    QAction *actionStop;
    QAction *actionActualiser;
    QAction *actionAccueil;
    QAction *actionGo;

    QLineEdit *champAdresse;
    QProgressBar *progression;
};

#endif

```

Les lignes ajoutées par rapport à la fois précédente ont été surlignées.

Maintenant implémentons tout ça. 😊

Implémentation des slots

Slots appelés par les actions de la barre d'outils

Commençons par les actions de la barre d'outils :

Code : C++ - [Sélectionner](#)

```
void FenPrincipale::precedente()
{
    pageActuelle()->back();
}

void FenPrincipale::suivante()
{
    pageActuelle()->forward();
}

void FenPrincipale::accueil()
{
    pageActuelle()->load(QUrl(tr("http://www.siteduzero.com")));
}

void FenPrincipale::stop()
{
    pageActuelle()->stop();
}

void FenPrincipale::actualiser()
{
    pageActuelle()->reload();
}
```

On utilise la (très) pratique fonction `pageActuelle()` pour obtenir un pointeur vers le `QWebView` que l'utilisateur est en train de regarder (histoire d'affecter la page web de l'onglet en cours, et pas les autres).

Toutes ces méthodes, comme `back()` et `forward()`, sont des slots. On les appelle ici comme si c'étaient de simples méthodes.

Pourquoi ne pas avoir connecté directement les signaux envoyés par les `QAction` aux slots du `QWebView` ?

On aurait pu s'il n'y avait pas eu d'onglets. Le problème justement ici, c'est qu'on gère plusieurs onglets différents.

Par exemple, on ne pouvait pas connecter lors de sa création la `QAction "actualiser"` au `QWebView...` parce que le `QWebView` à actualiser dépend de l'onglet actuellement sélectionné !

Voilà donc pourquoi on passe par un petit slot maison qui va d'abord chercher à savoir quel est le `QWebView` que l'on est en train de visualiser pour être sûr qu'on recharge la bonne page. 😊

Slots appelés par d'autres actions des menus

Voici les slots appelés par les actions des menus suivants :

- Nouvel onglet
- Fermer l'onglet
- A propos...

Code : C++ - [Sélectionner](#)

```

void FenPrincipale::aPropos()
{
    QMessageBox::information(this, tr("A propos..."), tr("zNavigo est un projet réalisé par"));
}

void FenPrincipale::nouvelOnglet()
{
    int indexNouvelOnglet = onglets->addTab(creerOngletPageWeb(), tr("(Nouvelle page)"));
    onglets->setcurrentIndex(indexNouvelOnglet);

    champAdresse->setText("");
    champAdresse->setFocus(Qt::OtherFocusReason);
}

void FenPrincipale::fermerOnglet()
{
    // On ne doit pas fermer le dernier onglet, sinon le QTabWidget ne marche plus
    if (onglets->count() > 1)
    {
        onglets->removeTab(onglets->currentIndex());
    }
    else
    {
        QMessageBox::critical(this, tr("Erreur"), tr("Il faut au moins un onglet !"));
    }
}

```

3

4

5

6

Le slot aPropos() se contente d'afficher une boîte de dialogue.

nouvelOnglet() rajoute un nouvel onglet à l'aide de la méthode addTab() du QTabWidget, comme on l'avait fait dans le constructeur. Pour que le nouvel onglet s'affiche immédiatement, on force son affichage avec setCurrentIndex() qui se sert de l'index (numéro) de l'onglet que l'on vient de créer.

On vide la barre d'adresse et on lui donne le focus, c'est-à-dire que le curseur est directement placé dedans pour que l'utilisateur puisse écrire une URL.

L'action "Nouvel onglet" a comme raccourci "Ctrl+T", ce qui permet d'ouvrir un onglet à tout moment à l'aide du raccourci clavier correspondant.

Vous pouvez aussi ajouter un bouton dans la barre d'outils pour ouvrir un nouvel onglet ou, encore mieux, rajouter un mini-bouton dans un des coins du QTabWidget. Regardez du côté de la méthode [setCornerWidget\(\)](#).

fermerOnglet() supprime l'onglet actuellement sélectionné. Il vérifie au préalable que l'on n'est pas en train d'essayer de supprimer le dernier onglet, auquel cas le QTabWidget n'aurait plus lieu d'exister. Un système à onglets sans onglets, ça fait désordre. 😊

SLOTS de chargement d'une page et de changement d'onglet

Ces slots sont appelés respectivement lorsqu'on demande à charger une page (appui sur la touche Entrée après avoir écrit une URL, ou clic sur le bouton tout à droite de la barre d'outils) et lorsqu'on change d'onglet.

Code : C++ - [Sélectionner](#)

```

void FenPrincipale::chargerPage()
{
    QString url = champAdresse->text();

    // On rajoute le "http://" s'il n'est pas déjà dans l'adresse
    if (url.left(7) != "http://")
    {
        url = "http://" + url;
        champAdresse->setText(url);
    }

    pageActuelle()->load(QUrl(url));
}

void FenPrincipale::changementOnglet(int index)
{
    changementTitre(pageActuelle()->title());
    changementUrl(pageActuelle()->url());
}

```

On vérifie au préalable que l'utilisateur a mis le préfixe `http://`, et si ce n'est pas le cas on le rajoute (sinon l'adresse n'est pas valide).

Lorsque l'utilisateur change d'onglet, on met à jour 2 choses sur la fenêtre : le titre de la page, affiché tout en haut de la fenêtre et sur un onglet, et l'URL inscrite dans la barre d'adresse.

`changementTitre()` et `changementUrl()` sont des slots personnalisés, que l'on se permet d'appeler comme n'importe quelle méthode. Ces slots sont aussi automatiquement appelés lorsque le QWebView envoie les signaux correspondants.

Voyons voir comment implémenter ces slots...

Slots appelés lorsqu'un signal est envoyé par le QWebView

Lorsque le QWebView s'active, il va envoyer des signaux. Ceux-ci sont connectés à des slots personnalisés de notre fenêtre. Les voici :

Code : C++ - [Sélectionner](#)

```

void FenPrincipale::changementTitre(const QString & titreComplet)
{
    QString titreCourt = titreComplet;

    // On tronque le titre pour éviter des onglets trop larges
    if (titreComplet.size() > 40)
    {
        titreCourt = titreComplet.left(40) + "...";
    }

    setWindowTitle(titreCourt + " - " + tr("zNavigo"));
    onglets->setTabText(onglets->currentIndex(), titreCourt);
}

void FenPrincipale::changementUrl(const QUrl & url)
{
    if (url.toString() != tr("html/page_blanche.html"))
    {
        champAdresse->setText(url.toString());
    }
}

void FenPrincipale::chargementDebut()
{
    progression->setVisible(true);
}

void FenPrincipale::chargementEnCours(int pourcentage)
{
    progression->setValue(pourcentage);
}

void FenPrincipale::chargementTermine(bool ok)
{
    progression->setVisible(false);
    statusBar()->showMessage(tr("Prêt"), 2000);
}

```

Ces slots ne sont pas très complexes. Ils mettent à jour la fenêtre (par exemple la barre de progression en bas) lorsqu'il y a lieu.

Certains sont très utiles, comme `changementUrl()`. En effet, lorsque l'utilisateur clique sur un lien sur la page, l'URL change et il faut par conséquent mettre à jour le champ d'adresse.

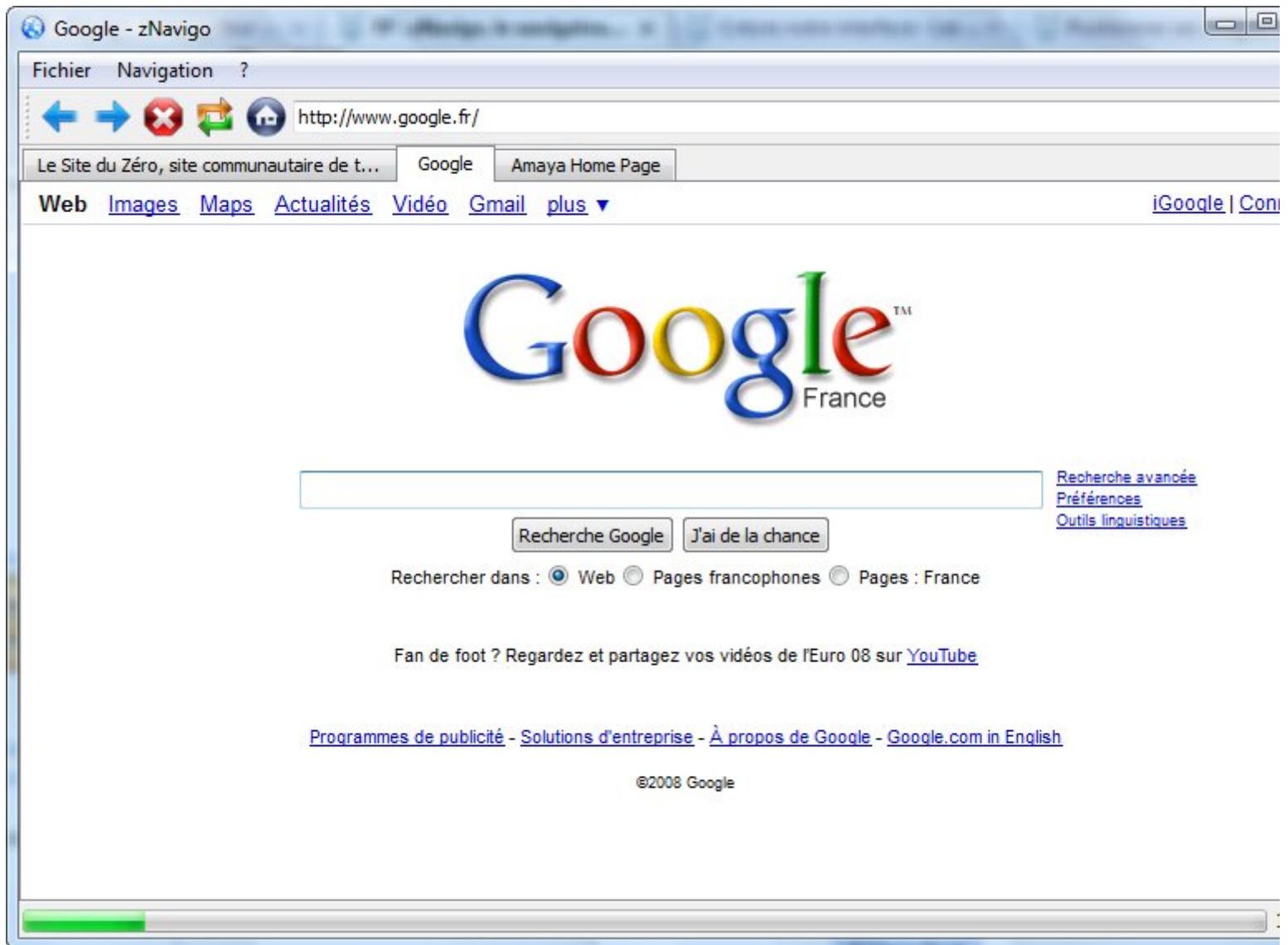
Vous noterez que je tronque le titre de la page à 40 caractères si celui-ci est trop long. Cela permet d'éviter d'avoir des onglets qui font 4km de large. 😊

Conclusion et améliorations possibles

Ouf ! 😊

Pas fâché d'en avoir terminé, mais le plus beau dans tout ça, c'est qu'on a un navigateur parfaitement fonctionnel !

Je remets ici le screenshot par plaisir. 😊



A l'heure actuelle, un bug dans QtWebKit dans la gestion des cookies ne permet pas de se connecter au Site du Zéro. C'est un peu dommage, mais comme le module est assez récent il devrait s'améliorer et être corrigé à l'avenir. Ne soyez donc pas surpris si vous ne pouvez pas vous connecter au site avec.

Télécharger le code source et l'exécutable

Je vous propose de télécharger le code source ainsi que l'exécutable Windows du projet :

[Télécharger le code source et l'exécutable Windows \(39 Ko\)](#)

Pensez à ajouter les DLL nécessaires dans le même dossier que l'exécutable si vous voulez que celui-ci fonctionne. Cette fois, comme je vous l'avais dit, il faut 2 nouvelles DLL : QtWebKit4.dll et QtNetwork4.dll.

Améliorations possibles

Améliorer le navigateur, c'est possible ?

Certainement ! Il fonctionne, mais il est encore loin d'être parfait, et j'ai des tonnes d'idées pour l'améliorer. Bon ces idées sont repompées des navigateurs qui existent déjà, mais rien ne vous empêche d'en inventer de nouvelles super-révolutionnaires bien sûr.



- Afficher l'historique dans un menu : il existe une classe [QWebHistory](#) qui permet de récupérer l'historique de toutes les pages visitées via un QWebView. Pour obtenir un objet de type QWebHistory du QWebView de l'onglet en cours, utilisez `pageActuelle() -> page() -> history()`. Renseignez-vous ensuite sur la doc de QWebHistory pour essayer de trouver comment récupérer la liste des pages visitées. Vous pourriez, par exemple, rajouter un menu "Historique" qui afficherait l'historique des pages vues sur l'onglet en cours.
- Gestion des adresses HTTPS : certains sites sont sécurisés (comme Paypal et Adsense par exemple). Ils utilisent des adresses https:// au lieu de http://. A cause de la vérification que l'on a faite, notre navigateur n'accepte que les adresses http://. Modifiez-le pour qu'il puisse gérer aussi bien les http:// que les https://.
- Zone de recherche Google : vous pourriez rajouter une zone de recherche google en haut à droite, qui appelle automatiquement Google avec les mots-clés sélectionnés. Je vous laisse vous renseigner sur le fonctionnement de Google. Vous pouvez imiter un autre navigateur comme Firefox par exemple, pour voir l'URL qu'il charge lorsqu'on fait une recherche Google.
- Recherche dans la page : rajoutez la possibilité de faire une recherche dans le texte de la page web affichée. Indice : QWebView dispose d'une méthode `findText()` !
- Disparition des onglets s'il ne reste plus qu'une seule page : voilà une amélioration délicate. Un QTabWidget sans onglets ne peut exister. Mais s'il ne reste qu'une seule page affichée, ça ne sert à rien d'utiliser un système à onglets. Essayer de gérer le cas où il ne reste plus qu'une seule page affichée, afin que le QTabWidget disparaisse (au moins jusqu'à ce qu'on demande à ouvrir un nouvel onglet).
- Fenêtre d'options : vous pourriez créer une nouvelle fenêtre d'options qui permet de définir la taille de police par défaut, l'URL de la page d'accueil, etc. Pour modifier la taille de la police par défaut, regardez du côté de [QWebSettings](#).

Pour enregistrer les options, vous pouvez passer par la classe QFile pour écrire dans un fichier. Mais j'ai mieux : utilisez la classe [QSettings](#) qui est spécialement faite pour enregistrer des options. En général, les options sont enregistrées dans un fichier (.ini, .conf...), mais on peut aussi enregistrer les options dans la base de registres sous Windows. Prenez bien le temps de lire la description de cette classe, c'est un peu long mais c'est vraiment très intéressant.

- Gestion des marque-pages (favoris) : voilà une fonctionnalité très répandue sur la plupart des navigateurs. L'utilisateur aime bien pouvoir enregistrer les adresses de ses sites web préférés. Là encore, pour l'enregistrement, je vous recommande chaudement de passer par un [QSettings](#).

Vous pourrez ensuite afficher la liste des sites favoris dans un menu, ou encore dans une nouvelle barre d'outils comme le fait Firefox.

- Impression d'une page web : pourquoi ne pas faire s'amuser avec l'imprimante ? Les QWebFrame contiennent une méthode [print\(\)](#) qui prend en paramètre une imprimante (QPrinter).

Le QWebView est constitué d'une QWebPage qui est elle-même constituée d'un ou plusieurs QWebFrame (généralement un seul). Je vous laisse découvrir comment obtenir un pointeur vers le QWebFrame. Je vous laisse aussi découvrir comment manipuler les [QPrinter](#), qui permettent de faire appel à l'imprimante. Et je vous invite aussi à jeter un œil à la classe [QPrintDialog](#) qui permet d'afficher une boîte de dialogue générique d'impression.

Ah, que de choses à découvrir !

- Sauvegarde de l'état de la fenêtre à la clôture : c'est peut-être une des fonctionnalités les plus appréciées des navigateurs actuels. Lorsque l'utilisateur veut quitter le programme, enregistrez (toujours avec QSettings) la liste des onglets ouverts avec leurs URL. Vous pourrez ainsi les réouvrir automatiquement lors du prochain chargement du programme.

Voilà, avec tout ce que je vous ai donné à faire, je crois que j'ai le temps d'aller à la nage à Hawaï siroter une Piña Colada dans un bar en bord de mer.

Et peut-être même que j'ai le temps de revenir d'ailleurs. 😊

L'architecture MVC avec les widgets complexes

Nous attaquons maintenant un des chapitres les plus intéressants de ce cours sur Qt, mais aussi un des plus difficiles.

Dans ce chapitre, nous apprendrons à manipuler 3 widgets complexes :

- QListWidget : une liste d'éléments à un seul niveau.
- QTreeWidget : une liste d'éléments à plusieurs niveaux, organisée en arbre.
- QTableWidget : un tableau.

On ne peut pas utiliser ces widgets sans un minimum de théorie. Et c'est justement cette partie théorique qui me fait dire que ce chapitre sera l'un des plus intéressants : nous allons découvrir l'architecture MVC, une façon de programmer (on parle de design pattern) très puissante qui va nous donner énormément de flexibilité.

Présentation de l'architecture MVC

Avant de commencer à manipuler les 3 widgets complexes dont je vous ai parlé en introduction, il est indispensable que je vous présente l'architecture MVC.

Qu'est-ce que l'architecture MVC ? A quoi ça sert ? Quel rapport avec la création de GUI ?

MVC est l'abréviation de Model-View-Controller, ce qui signifie en français : "Modèle-Vue-Contrôleur".

...

... ça ne vous avance pas trop, j'ai l'impression. 🤪

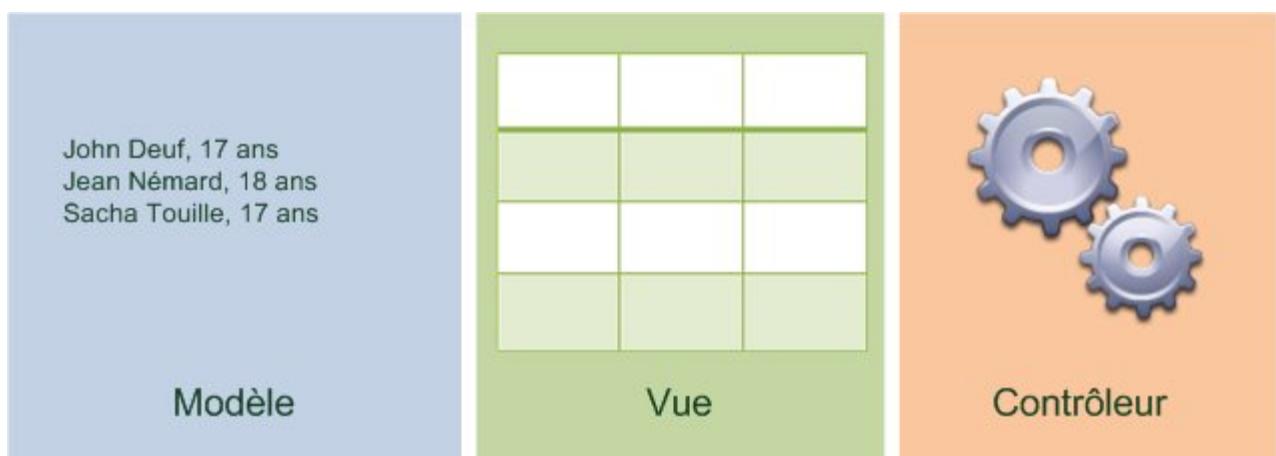
Il s'agit d'un design pattern, une technique de programmation. C'est une "façon de programmer et d'organiser son code" bien pensée. Vous n'êtes pas obligés de programmer de cette manière-là, mais si vous le faites votre code sera plus lisible, plus clair et plus souple.

L'architecture MVC vous propose de séparer les éléments de votre programme en 3 parties :

- Le modèle : c'est la partie qui contient les données. Le modèle peut par exemple contenir la liste des élèves d'une classe, avec leurs noms, prénoms, âges...
- La vue : c'est la partie qui s'occupe de l'affichage. Elle affiche ce que contient le modèle.
Par exemple, la vue pourrait être un tableau. Ce tableau affichera la liste des élèves si c'est ce que contient le modèle.
- Le contrôleur : c'est la partie "réflexion" du programme. Lorsque l'utilisateur sélectionne 3 élèves dans le tableau et appuie sur la touche "Supprimer", le contrôleur est appelé et se charge de supprimer les 3 élèves du modèle.

C'est dur à imaginer au début. Mais assurez-vous, vous allez comprendre au fur et à mesure de ce chapitre l'intérêt de séparer le code en 3 parties et le rôle de chacune de ces parties. Ce n'est pas grave si vous ne voyez pas de suite comment ces parties interagissent entre elles.

Commençons par un schéma, visuel et simple à retenir, qui présente le rôle de chacune de ces parties :



Comme on peut le voir sur ce schéma :

- Le modèle est la partie qui contient les données (comment, on verra ça après). Les données sont généralement récupérées en lisant un fichier ou une base de données.
 - La vue est juste la partie qui affiche le modèle, ce sera donc un widget dans notre cas.
Si un élément est ajouté au modèle (par exemple un nouvel élève apparaît) la vue se met à jour automatiquement pour afficher le nouveau modèle.
 - Le contrôleur est la partie la plus algorithmique, c'est-à-dire le cerveau de votre programme. S'il y a des calculs à faire, c'est là qu'ils sont faits.
-
-

L'architecture simplifiée modèle/vue de Qt

En fait (vous allez me tuer je le sens 😊), Qt n'utilise pas vraiment MVC mais une version simplifiée de ce système : l'architecture modèle/vue.

Et le contrôleur on en fait quoi ? Poubelle ? Notre programme ne réfléchit pas ?

Si si, je vous rassure. En fait, le contrôleur est intégré à la vue avec Qt.

Grâce à ça, les données sont toujours séparées de leur affichage, mais on diminue un peu la complexité du modèle MVC en évitant au programmeur d'avoir à gérer les 3 parties.

On s'éloigne donc un petit peu de la théorie pure sur MVC ici, pour s'intéresser à la façon dont Qt utilise ce principe en pratique. Vous venez donc d'apprendre que Qt adaptait ce principe à sa manière, pour garder les bonnes idées principales sans pour autant vous obliger à "trop" découper votre code ce qui aurait pu être un peu trop complexe et répétitif à la longue.

Nous n'allons donc plus parler ici que de modèle et de vue. Comment sont gérés chacun de ces éléments avec Qt ?
Cette question... avec des classes, comme d'habitude ! 🧐

Les classes gérant le modèle

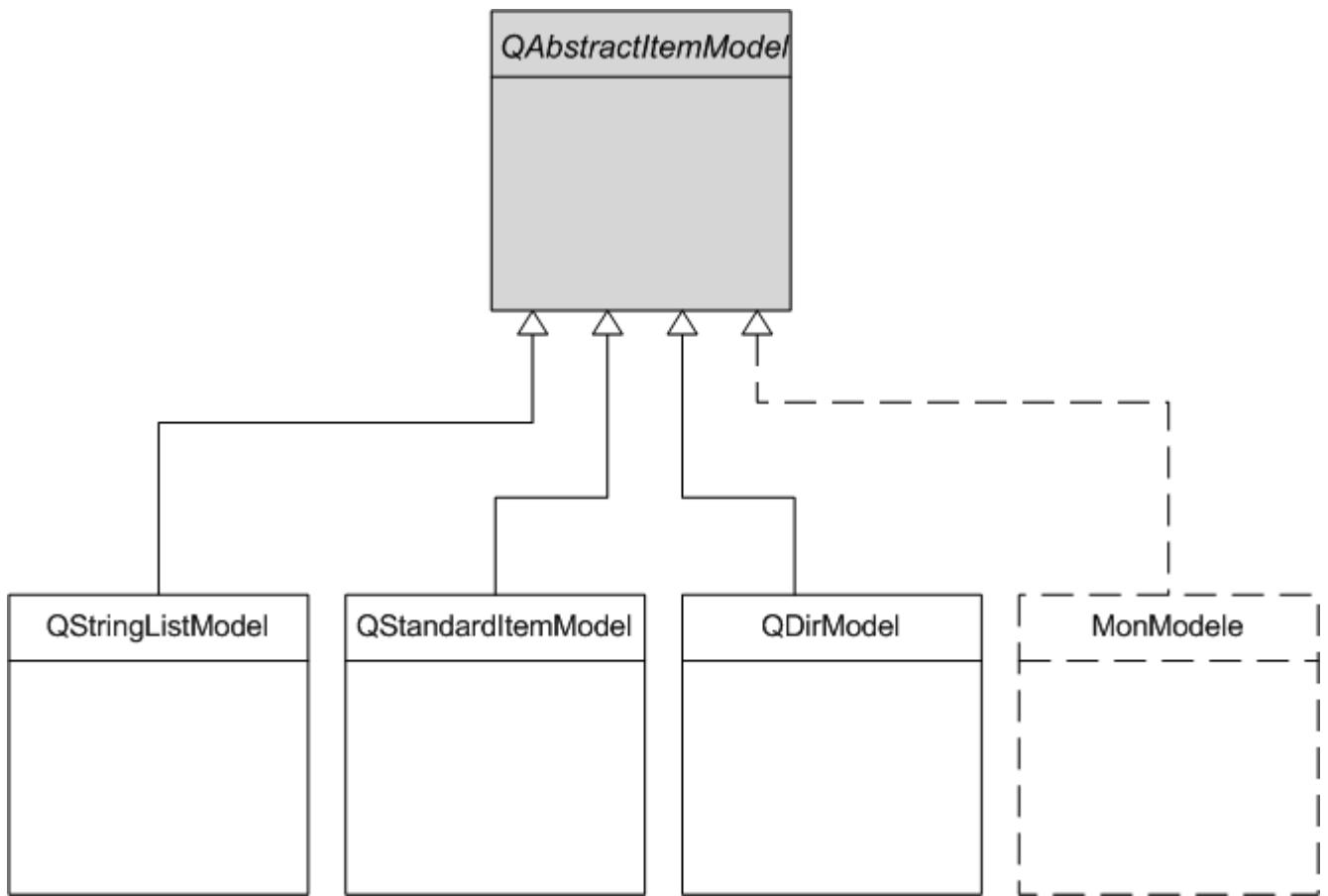
Il y a plusieurs types de modèles différents. En effet : on ne stocke pas de la même manière une liste d'élèves qu'une liste de villes !

Vous avez 2 possibilités :

- Soit vous créez votre propre classe de modèle. Il faut créer une classe héritant de [QAbstractItemModel](#). C'est la solution la plus flexible mais aussi la plus complexe, nous ne la verrons pas ici.
 - Soit vous utilisez une des classes génériques toutes prêtes offertes par Qt :
 - QStringListModel : une liste de chaînes de caractères, de type QString. Très simple, très basique. Ca peut suffire pour les cas les plus simples.
 - QStandardItemModel : une liste d'éléments organisés sous forme d'arbre (chaque élément peut contenir des sous-éléments). Ce type de modèle est plus complexe que le précédent, car il gère plusieurs niveaux d'éléments. Avec QStringListModel, c'est un des modèles les plus utilisés.
 - QDirModel : la liste des fichiers et dossiers stockés sur votre ordinateur. Ce modèle va analyser en arrière-plan votre disque dur, et restitue la liste de vos fichiers sous la forme d'un modèle prêt à l'emploi.
 - QSqlQueryModel, QSqlTableModel et QSqlRelationalTableModel : données issues d'une base de données. On peut s'en servir pour accéder à une base de données (ceux qui ont déjà utilisé MySQL, Oracle ou un autre système du genre seront probablement intéressés).
- Je ne vais pas rentrer dans les détails de la connexion à une base de données dans ce chapitre, ce serait un peu hors-sujet.

Toutes ces classes proposent donc des modèles prêts à l'emploi, qui héritent de QAbstractItemModel.

Si aucune de ces classes ne vous convient, vous devrez créer votre propre classe en héritant de `QAbstractItemModel`.



Notez que je n'ai pas représenté toutes les classes de modèles ici.

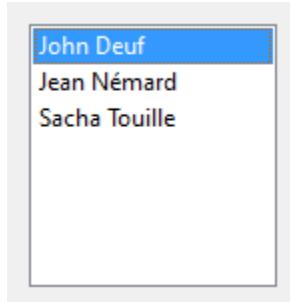
Les classes gérant la vue

Pour afficher les données issues du modèle, il nous faut une vue. Avec Qt, la vue est un widget, puisqu'il s'agit d'un affichage dans une fenêtre.

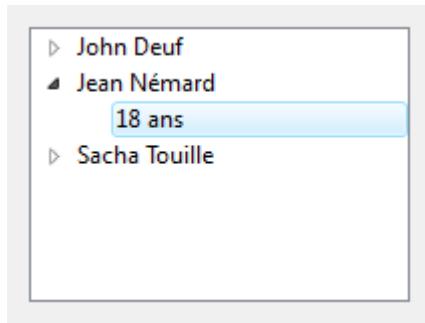
Tous les widgets de Qt ne sont pas bâtis autour de l'architecture modèle/vue, loin de là (et ça n'aurait pas d'intérêt pour les plus simples d'entre eux que nous avons vu jusqu'à présent).

On compte 3 widgets adaptés pour la vue avec Qt :

- [QListView](#) : une liste d'éléments.



- [QTreeView](#) : un arbre d'éléments, où chaque élément peut avoir des éléments enfants.



- [QTableView](#) : un tableau.

John	Deuf	17 ans
Jean	Némard	18 ans
Sacha	Touille	17 ans

Voilà donc les fameux "widgets" complexes que je vais vous présenter dans ce chapitre. Mais pour pouvoir les utiliser et les peupler de données, il faut d'abord créer un modèle !

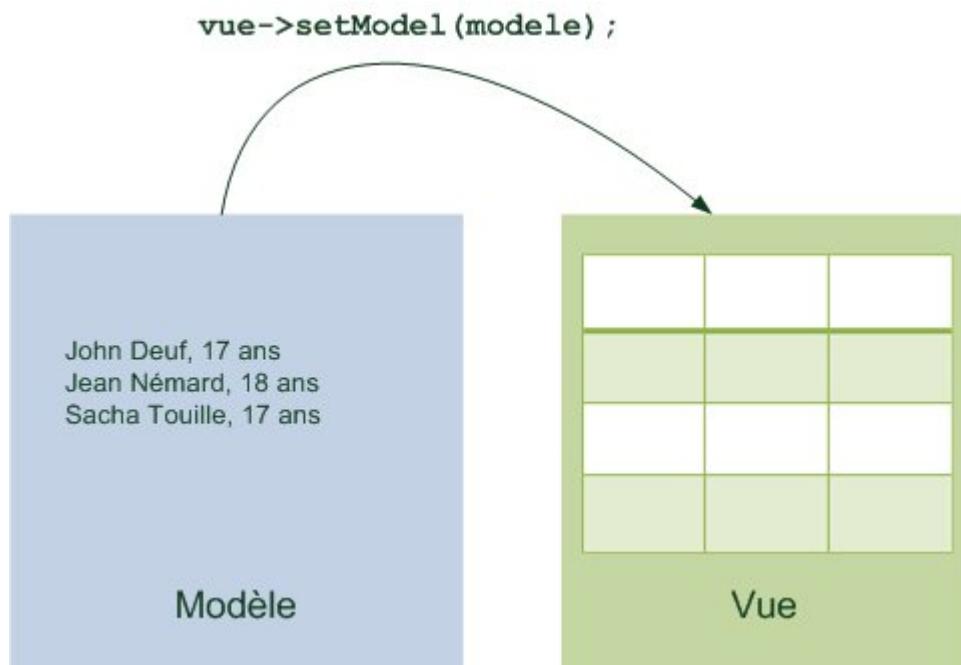
Appliquer un modèle à la vue

Lorsqu'on utilise l'architecture modèle/vue avec Qt, cela se passe toujours en 3 temps. Il faut :

1. Créer le modèle
2. Créer la vue
3. Associer la vue et le modèle

La dernière étape est essentielle. Cela revient en quelque sorte à "connecter" notre modèle à notre vue. Si on ne donne pas de modèle à la vue, elle ne saura pas quoi afficher, donc elle n'affichera rien. 

La connexion se fait toujours avec la méthode `setModel()` de la vue :



Le contrôleur n'a pas été représenté sur ce schéma car, comme je vous l'ai dit, Qt utilise une architecture modèle/vue simplifiée et se charge de gérer la partie contrôleur pour vous.

Voilà donc comment on connecte un modèle à une vue en pratique. 😊

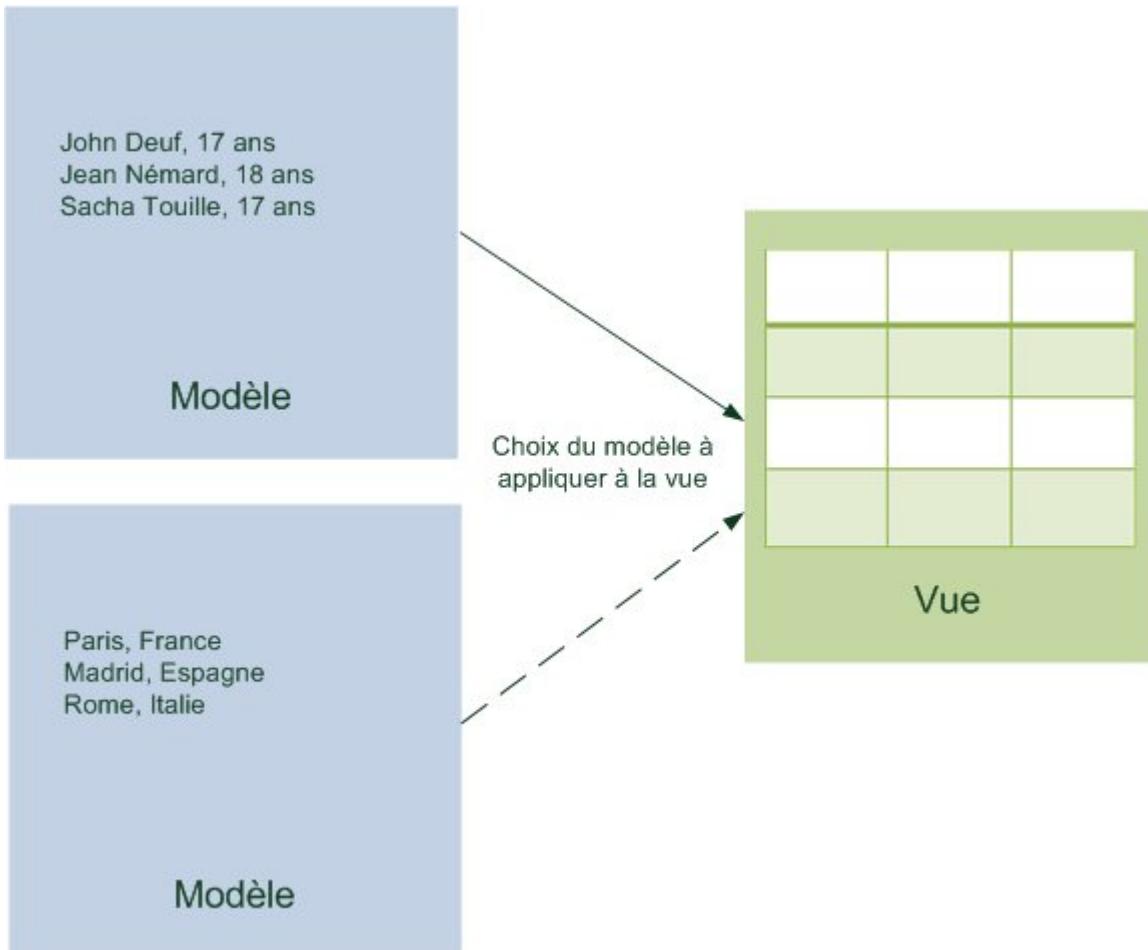
L'avantage de ce système, c'est qu'il est flexible. On peut avoir 2 modèles et appliquer soit l'un soit l'autre à la vue en fonction des besoins. Un gros intérêt est que dès que l'on modifie le modèle, la vue affiche instantanément les nouveaux éléments !

Plusieurs modèles ou plusieurs vues

On peut pousser le principe un peu plus loin. Essayons d'imaginer que l'on a plusieurs modèles ou plusieurs vues.

Plusieurs modèles et une vue

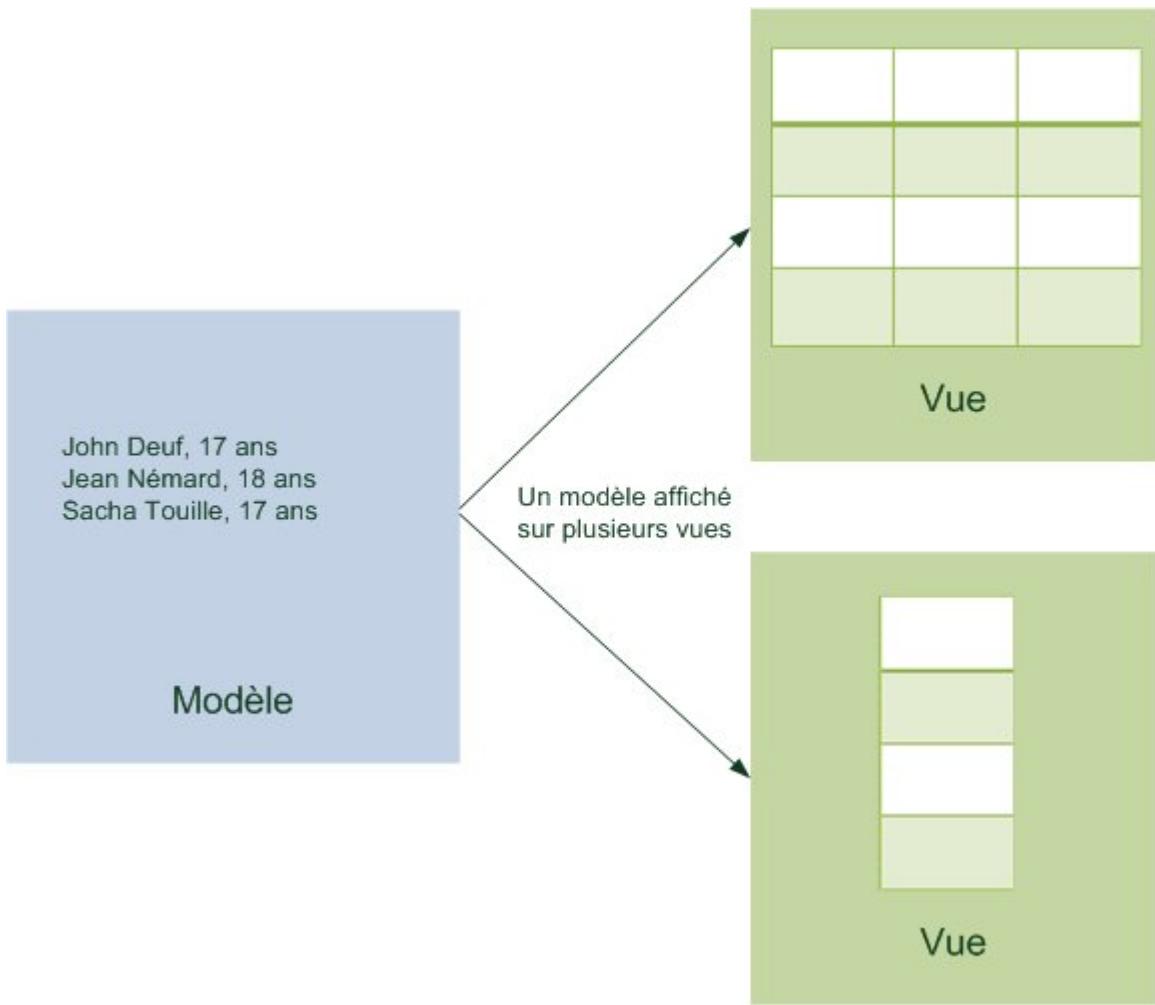
Imaginons que l'on ait 2 modèles : un qui contient une liste d'élèves, un autre qui contient une liste de capitales avec leur pays. Notre vue peut afficher soit l'un, soit l'autre :



Il faut bien sûr faire un choix : une vue ne peut afficher qu'un seul modèle à la fois !
L'avantage, c'est qu'au besoin on peut changer le modèle affiché par la vue en cours d'exécution, en appelant juste la méthode `setModel()` !

Un modèle pour plusieurs vues

Imaginons le cas inverse. On a un modèle, mais plusieurs vues. Cette fois, rien ne nous empêche d'appliquer ce modèle à 2 vues en même temps !



On peut ainsi visualiser le même modèle de 2 façons différentes (ici sous forme de tableau ou de liste dans mon schéma). Comme le même modèle est associé à 2 vues différentes, si le modèle change alors les 2 vues changent en même temps ! Par exemple, si je modifie l'âge d'un des élèves dans une cellule du tableau, l'autre vue (la liste) est automatiquement mise à jour sans avoir besoin d'écrire la moindre ligne de code !

Utilisation d'un modèle simple

Pour découvrir en douceur l'architecture modèle/vue de Qt, je vais vous proposer d'utiliser un modèle tout fait : [QDirModel](#).

Sa particularité, c'est qu'il est très simple à utiliser. Il analyse votre disque dur et génère le modèle correspondant. Pour créer ce modèle, c'est tout bête :

Code : C++ - [Sélectionner](#)

```
QDirModel *modele = new QDirModel;
```

On possède désormais un modèle qui représente notre disque. On va l'appliquer à une vue.

Mais quelle vue utiliser ? Une liste, un arbre, un tableau ? Les modèles sont-ils compatibles avec toutes les vues ?

Oui, toutes les vues peuvent afficher n'importe quel modèle. C'est toujours compatible.

Par contre, même si ça marche avec toutes les vues, vous allez vous rendre compte que certaines sont plus adaptées que d'autres en fonction du modèle que vous utilisez.

Par exemple, pour un QDirModel, la vue la plus adaptée est sans aucun doute l'arbre (QTreeView). Nous essaierons toutefois toutes

les vues avec ce modèle pour comparer le fonctionnement.

Le modèle appliqué à un QTreeView

Code : C++ - [Sélectionner](#)

```
#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
    QVBoxLayout *layout = new QVBoxLayout;

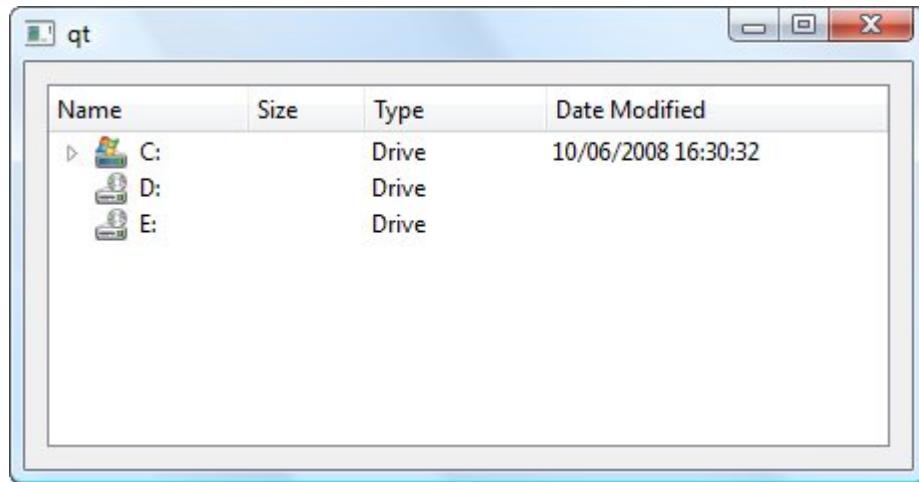
    QDirModel *modele = new QDirModel;
    QTreeView *vue = new QTreeView;
    vue->setModel(modele);

    layout->addWidget(vue);

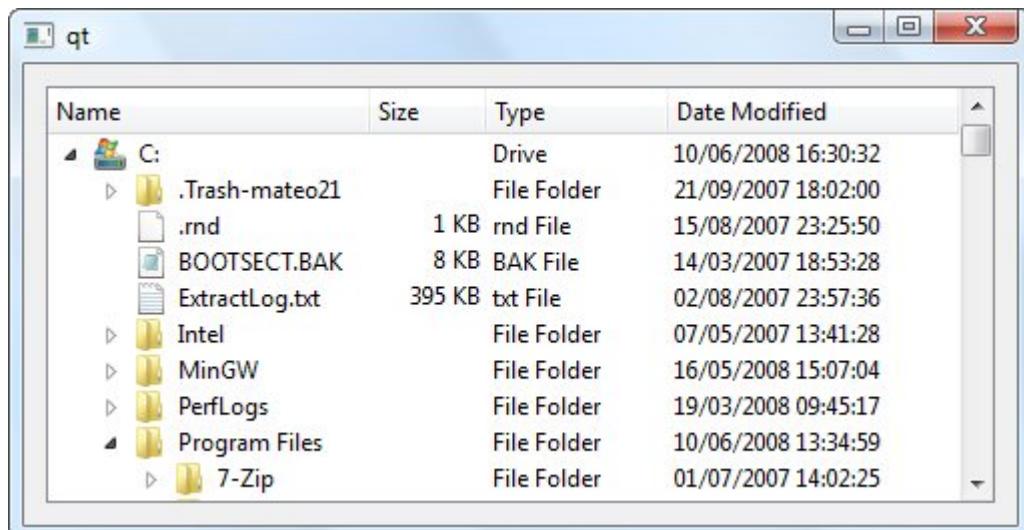
    setLayout(layout);
}
```

On crée le modèle, puis la vue, et on dit à la vue "Utilise ce modèle pour savoir quoi afficher" (ligne 9).

Le résultat est le suivant :



Une vue en forme d'arbre affiche le modèle de notre disque. Chaque élément peut avoir des sous-éléments dans un QTreeView, essayez de naviguer dedans :



Voilà un bel exemple d'arbre en action ! 😊

Le modèle appliqué à un QListview

Maintenant, essayons de faire la même chose, mais avec une liste (QListView). On garde le même modèle, mais on l'applique à une vue différente :

Code : C++ - [Sélectionner](#)

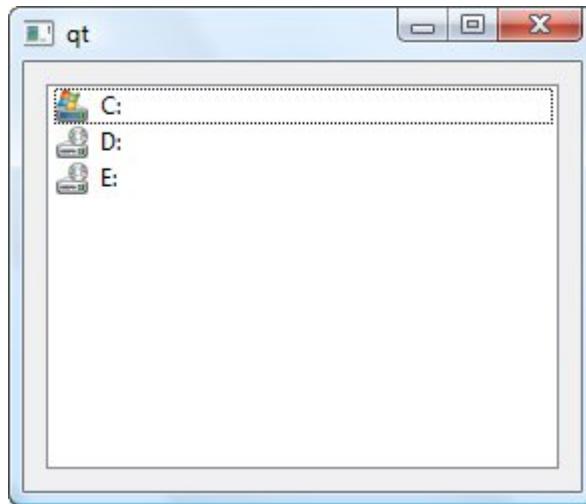
```
#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
    QVBoxLayout *layout = new QVBoxLayout;

    QDirModel *modele = new QDirModel;
    QListview *vue = new QListview;
    vue->setModel(modele);

    layout->addWidget(vue);

    setLayout(layout);
}
```



Ce type de vue ne peut afficher qu'un seul niveau d'éléments à la fois. Cela explique pourquoi je vois uniquement la liste de mes disques... et pourquoi je ne peux pas afficher leur contenu !

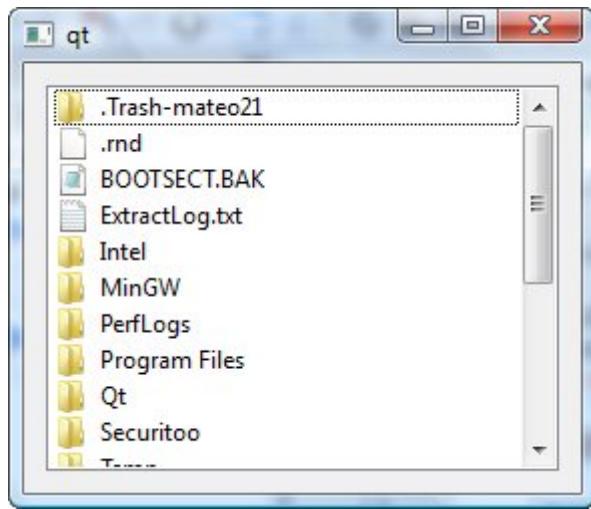
La vue affiche bêtement ce qu'elle est capable d'afficher, c'est-à-dire le premier niveau d'éléments.

Voilà la preuve qu'un même modèle marche sur plusieurs vues différentes, mais que certaines vues sont plus adaptées que d'autres !

Vous pouvez modifier la racine utilisée par la vue en vous inspirant du code suivant, que je ne détaillerai pas pour le moment :

Code : C++ - [Sélectionner](#)

```
vue->setRootIndex(modele->index("C:"));
```



Le modèle appliquée à un QTableView

Un tableau ne peut pas afficher plusieurs niveaux d'éléments (seul l'arbre QTreeView peut le faire). Par contre, il peut afficher plusieurs colonnes :

Code : C++ - [Sélectionner](#)

```
#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
    QVBoxLayout *layout = new QVBoxLayout;

    QDirModel *modele = new QDirModel;
    QTableView *vue = new QTableView;
    vue->setModel(modele);

    layout->addWidget(vue);

    setLayout(layout);
}
```

A screenshot of a Windows-style application window titled "qt". Inside, there is a QTableView component displaying a table with four columns: "Name", "Size", "Type", and "Date Modified". The table has three rows, indexed 1, 2, and 3. Row 1 contains the entry "C:" with a small icon. Row 2 contains "D:". Row 3 contains "E:". All entries are labeled as "Drive" under the "Type" column and show the date "10/06/2008 16:3..." under "Date Modified". The "Name" column includes row numbers (1, 2, 3) and icons representing drive types.

Comme précédemment, on peut appeler `setRootIndex()` pour modifier la racine des éléments affichés par la vue.

Utilisation de modèles personnalisables

Le modèle QDirModel que nous venons de voir était très simple à utiliser. Rien à paramétrer, rien à configurer, il analyse automatiquement votre disque dur pour construire son contenu.

C'était une bonne introduction pour découvrir les vues avec un modèle simple. Cependant, dans la plupart des cas, vous voudrez utiliser vos propres données, votre propre modèle. C'est ce que nous allons voir ici, à travers 2 nouveaux modèles :

- [QStringListModel](#) : une liste simple d'éléments de type texte, à un seul niveau.
- [QStandardItemModel](#) : une liste plus complexe à plusieurs niveaux et plusieurs colonnes, qui peut convenir dans la plupart des cas.

Pour les cas simples, nous utiliserons donc QStringListModel, mais nous découvrirons aussi QStandardItemModel qui nous donne plus de flexibilité.

QStringListModel : une liste de chaînes de caractères QString

Ce modèle, très simple, vous permet de gérer une liste de chaînes de caractères. Par exemple, si l'utilisateur doit choisir son pays parmi une liste :

```
France  
Espagne  
Italie  
Portugal  
Suisse
```

Pour construire ce modèle, il faut procéder en 2 temps :

- Construire un objet de type QStringList, qui contiendra la liste des chaînes.
- Créer un objet de type QStringListModel et envoyer à son constructeur le QStringList que vous venez de créer pour l'initialiser.

QStringList surcharge l'opérateur "<<" pour vous permettre d'ajouter des éléments à l'intérieur simplement.
Un exemple de code sera sûrement plus parlant :

Code : C++ - Sélectionner

```
#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
    QVBoxLayout *layout = new QVBoxLayout;

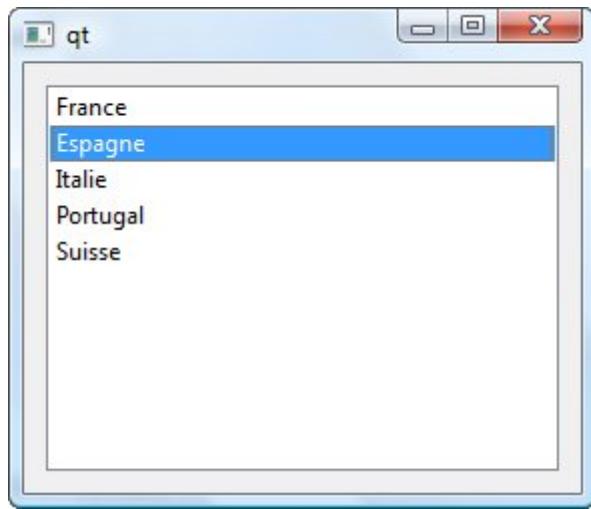
    QStringList listePays;
    listePays << "France" << "Espagne" << "Italie" << "Portugal" << "Suisse";
    QStringListModel *modele = new QStringListModel(listePays);

    QListWidget *vue = new QListWidget;
    vue->setModel(modele);

    layout->addWidget(vue);

    setLayout(layout);
}
```

Notre vue affiche maintenant la liste des pays qui se trouvent dans le modèle :



La surcharge de l'opérateur "<<" est très pratique comme vous pouvez le voir. Sachez toutefois qu'il est aussi possible d'utiliser la méthode append() :

Code : C++ - [Sélectionner](#)

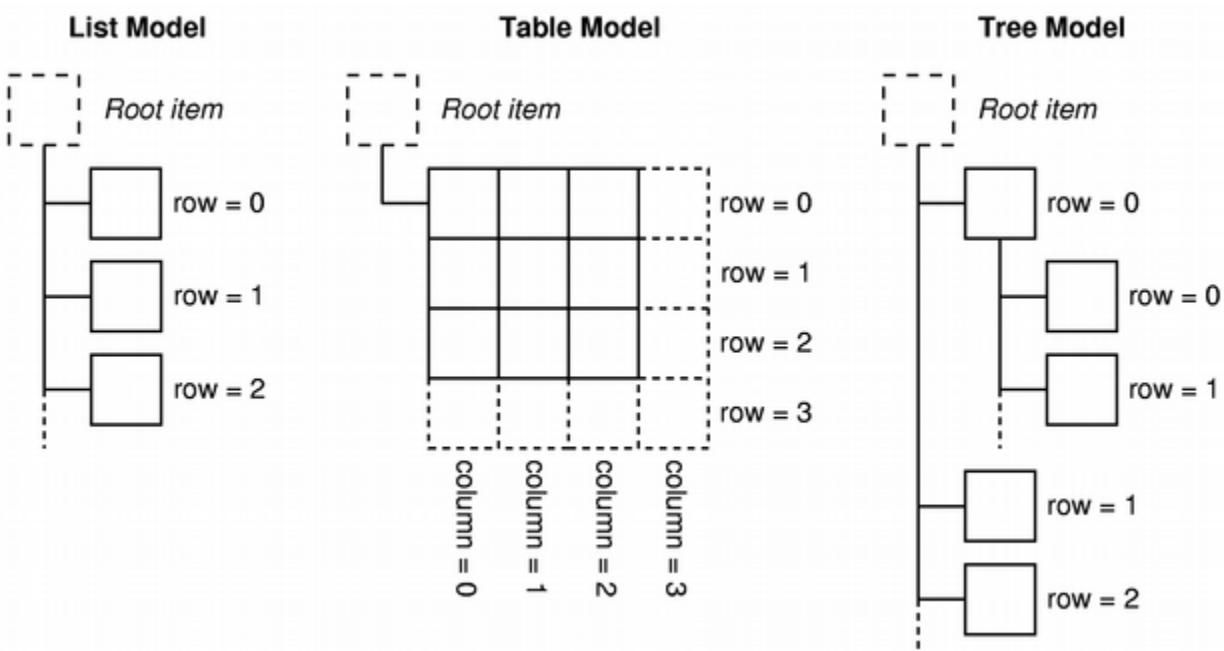
```
listePays.append( "Belgique" );
```

Si, au cours de l'exécution du programme, un pays est ajouté, supprimé ou modifié, la vue (la liste) affichera automatiquement les modifications. Nous testerons cela en pratique un peu plus loin dans le chapitre.

QStandardItemModel : une liste à plusieurs niveaux et plusieurs colonnes

Ce type de modèle est beaucoup plus complet (et donc complexe 😊) que le précédent. Il permet de créer tous les types de modèles possibles et imaginables.

Pour bien visualiser les différents types de modèles que l'on peut concevoir avec un QStandardItemModel, voici un super schéma offert par la doc de Qt :



- List Model : c'est un modèle avec une seule colonne et pas de sous-éléments. C'est le modèle utilisé par QStringList, mais QStandardItemModel peut aussi le faire (qui peut le plus peut le moins !).

Ce type de modèle est en général adapté à un QList View.

- Table Model : les éléments sont organisés avec plusieurs lignes et colonnes.

Ce type de modèle est en général adapté à un QTableView.

- Tree Model : les éléments ont des sous-éléments, ils sont organisés en plusieurs niveaux. Ce n'est pas représenté sur le schéma ci-dessus, mais rien n'interdit de mettre [plusieurs colonnes dans un modèle en arbre](#) aussi !

Ce type de modèle est en général adapté à un QTreeView.

Gérer plusieurs lignes et colonnes

Pour construire un QStandardItemModel, on doit indiquer en paramètres le nombre de lignes et de colonnes qu'il doit gérer. Des lignes et des colonnes supplémentaires peuvent toujours être ajoutées par la suite au besoin.

Code : C++ - [Sélectionner](#)

```
#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
    QVBoxLayout *layout = new QVBoxLayout;

    QStandardItemModel *modele = new QStandardItemModel(5, 3);

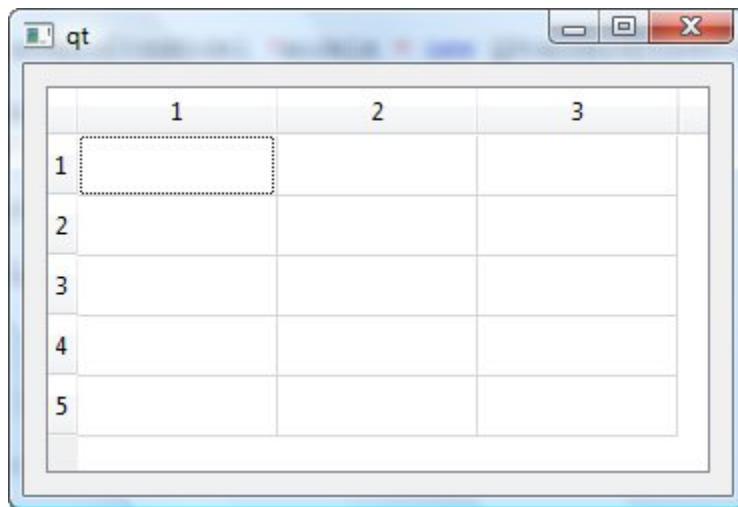
    QTableView *vue = new QTableView;
    vue->setModel(modele);

    layout->addWidget(vue);

    setLayout(layout);
}
```

Si on ne demande qu'une seule colonne, cela reviendra à créer un modèle de type "List Model".

Ici, un modèle à 5 lignes et 3 colonnes sera créé. Les éléments sont vides au départ, mais on a déjà un tableau :



On peut aussi appeler le constructeur par défaut (sans paramètres) si on ne connaît pas du tout la taille du tableau à l'avance. Il faudra appeler appendRow() et appendColumn() pour ajouter respectivement une nouvelle ligne ou une nouvelle colonne.

Chaque élément est représenté par un objet de type QStandardItem.

Pour définir un élément, on utilise la méthode `setItem()` du modèle. Donnez-lui respectivement le numéro de ligne, de colonne, et un QStandardItem à afficher. Attention : la numérotation commence à 0.

Code : C++ - Sélectionner

```
#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
    QVBoxLayout *layout = new QVBoxLayout;

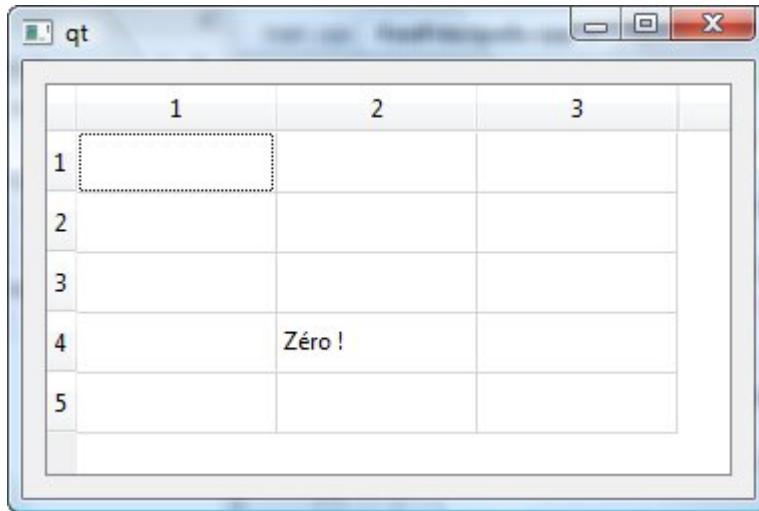
    QStandardItemModel *modele = new QStandardItemModel(5, 3);
    modele->setItem(3, 1, new QStandardItem("Zéro !"));

    QTableView *vue = new QTableView;
    vue->setModel(modele);

    layout->addWidget(vue);

    setLayout(layout);
}
```

Ici, je crée un nouvel élément contenant le texte "Zéro !" à la 4ème ligne, 2nde colonne :



Voilà comment on peut peupler le modèle d'éléments. 😊

Ajouter des éléments enfants

Essayons maintenant d'ajouter des éléments enfants.

Pour pouvoir voir les éléments enfants, on va devoir changer de vue et passer par un QTreeView.

Il faut procéder dans l'ordre :

1. Créer un élément (par exemple "item"), de type QStandardItem. Ligne 9
2. Ajouter cet élément au modèle avec appendRow(). Ligne 10
3. Ajouter un sous-élément à "item" avec appendRow(). Ligne 11

Code : C++ - Sélectionner

```

#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
    QVBoxLayout *layout = new QVBoxLayout;

    QStandardItemModel *modele = new QStandardItemModel;

    QStandardItem *item = new QStandardItem("John Deuf");
    modele->appendRow(item);
    item->appendRow(new QStandardItem("17 ans"));

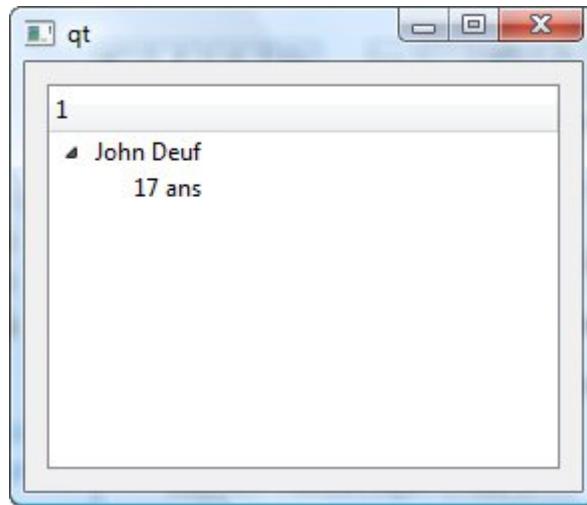
    QTreeView *vue = new QTreeView;
    vue->setModel(modele);

    layout->addWidget(vue);

    setLayout(layout);
}

```

Résultat, on a créé un élément "John Deuf" qui contient un élément enfant "17 ans" :



Entraînez-vous à créer plusieurs éléments et des sous-éléments enfants, ce n'est pas compliqué si on est bien organisé mais il faut pratiquer. 😊

Pour supprimer l'en-tête (marqué "1" et inutile), vous pouvez appeler : `vue->header()->hide();`

Gestion des sélections

Nous avons découvert comment associer un modèle à une vue, et comment manipuler plusieurs modèles : QDirModel, QStringListModel et QStandardItemModel.

Il nous reste à voir comment on peut récupérer le ou les éléments sélectionnés dans la vue, pour savoir quel est le choix de l'utilisateur.

Nous entrons dans une partie vraiment complexe où de nombreuses classes se mêlent les unes aux autres. L'architecture modèle/vue de Qt est extrêmement flexible (on peut faire ce qu'on veut avec), mais en contrepartie il est beaucoup plus délicat de s'en servir car il y a plusieurs étapes à suivre dans un ordre précis.

Par conséquent, et afin d'éviter de faire un chapitre beaucoup trop long et surtout trop complexe, j'ai volontairement décidé de limiter mes exemples ici aux sélections d'un QListWidget. Je vous laisserai le soin d'adapter ces exemples aux autres vues, en sachant que c'est relativement similaire à chaque fois (les principes sont les mêmes).

Nous allons rajouter un bouton "Afficher la sélection" à notre fenêtre. Elle va ressembler à ceci :



Lorsqu'on cliquera sur le bouton, il ouvrira une boîte de dialogue (QMessageBox) qui affichera le nom de l'élément sélectionné.

Nous allons apprendre à gérer 2 cas :

- Lorsqu'on ne peut sélectionner qu'un seul élément à la fois.
- Lorsqu'on peut sélectionner plusieurs éléments à la fois.

Une sélection unique

Nous allons devoir créer une connexion entre un signal et un slot pour que le clic sur le bouton fonctionne.

Modifions donc pour commencer le .h de la fenêtre :

Code : C++ - Sélectionner

```
#ifndef HEADER_FENPRINCIPALE
#define HEADER_FENPRINCIPALE

#include <QtGui>

class FenPrincipale : public QWidget
{
    Q_OBJECT

public:
    FenPrincipale();

private:
    QStringListModel *modele;
    QListView *vue;
    QPushButton *bouton;

private slots:
    void clicSelection();
};

#endif
```

J'ai rajouté la macro Q_OBJECT, mis quelques éléments de la fenêtre en attributs (pour pouvoir y accéder dans le slot), et ajouté le slot clicSelection() qui sera appelé après un clic sur le bouton.

Maintenant retour au .cpp, où je fais la connexion et où j'écris le contenu du slot :

Code : C++ - Sélectionner

```
#include "FenPrincipale.h"

FenPrincipale::FenPrincipale()
{
    QVBoxLayout *layout = new QVBoxLayout;

    QStringList listePays;
    listePays << "France" << "Espagne" << "Italie" << "Portugal" << "Suisse";
    modele = new QStringListModel(listePays);

    vue = new QListView ;
    vue->setModel(modele);

    bouton = new QPushButton("Afficher la sélection");

    layout->addWidget(vue);
    layout->addWidget(bouton);

    setLayout(layout);

    connect(bouton, SIGNAL(clicked()), this, SLOT(clicSelection()));
}

void FenPrincipale::clicSelection()
{
    QItemSelectionModel *selection = vue->selectionModel();
    QModelIndex indexElementSelectionne = selection->currentIndex();
    QVariant elementSelectionne = modele->data(indexElementSelectionne, Qt::DisplayRole);
    QMessageBox::information(this, "Elément sélectionné", elementSelectionne.toString());
}
```

Analysons le contenu du slot, qui contient les nouveautés (lignes 26 à 29). Voici ce que nous faisons ligne par ligne :

1. On récupère un objet QItemSelectionModel qui contient des informations sur ce qui est sélectionné sur la vue. C'est la vue qui nous donne un pointeur vers cet objet grâce à vue->selectionModel().
2. On appelle la méthode currentIndex() de l'objet qui contient des informations sur la sélection. Cela renvoie un index, c'est-à-dire en gros le numéro de l'élément sélectionné sur la vue.
3. Maintenant qu'on connaît le numéro de l'élément sélectionné, on veut retrouver son texte. On appelle la méthode data() du modèle, et on lui donne l'index qu'on a récupéré (c'est-à-dire le numéro de l'élément sélectionné). On récupère le résultat dans un QVariant, qui est une classe qui peut aussi bien stocker des int que des chaînes de caractères.
4. On n'a plus qu'à afficher l'élément sélectionné récupéré. Pour extraire la chaîne du QVariant, on appelle toString().

Ouf ! Ce n'est pas simple je le reconnais, il y a plusieurs étapes.

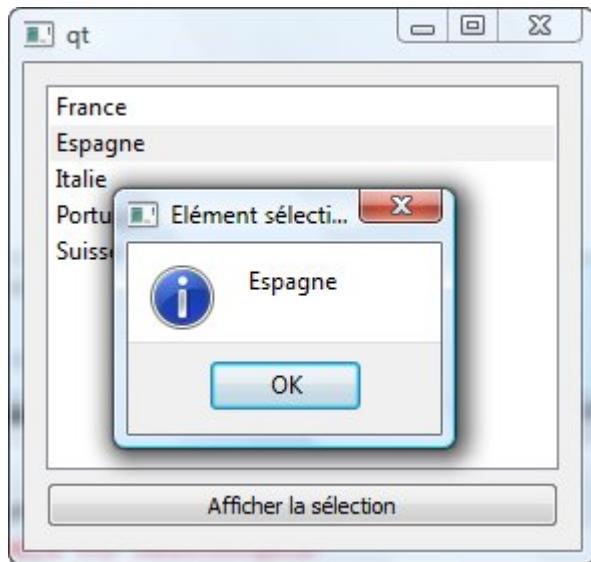
D'abord on récupère l'objet qui contient des informations sur les éléments sélectionnés sur la vue.

Ensuite on demande à cet objet quel est l'indice (numéro) de l'élément actuellement sélectionné.

On peut alors récupérer le texte contenu dans le modèle à cet indice.

On affiche ce texte avec la méthode toString() de l'objet de type QVariant.

Désormais, un clic sur le bouton vous indique quel élément est sélectionné :



Une sélection multiple

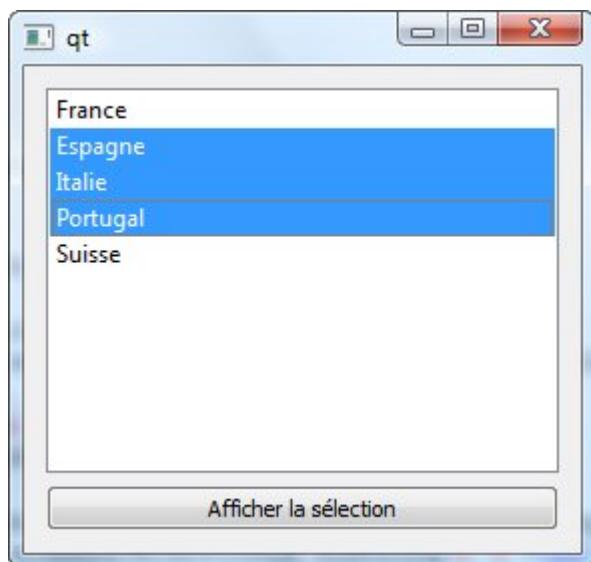
Par défaut, on ne peut sélectionner qu'un seul élément à la fois sur une liste. Pour changer ce comportement et autoriser la sélection multiple, rajoutez ceci dans le constructeur :

Code : C++ - [Sélectionner](#)

```
vue->setSelectionMode(QAbstractItemView::ExtendedSelection);
```

D'autres modes de sélection sont disponibles, mais je vous laisse aller voir la doc de QAbstractItemView pour en savoir plus. 😊
Avec ce mode, on peut sélectionner n'importe quels éléments. On peut utiliser la touche Shift du clavier pour faire une sélection continue, ou Ctrl pour une sélection discontinue (avec des trous).

Voici un exemple de sélection continue, désormais possible :



Pour récupérer la liste des éléments sélectionnés, c'est un peu plus compliqué ici parce qu'il y en a plusieurs. On ne peut plus utiliser la méthode `currentIndex()`, il va falloir utiliser `selectedIndexes()`.

Je vous donne le nouveau code du slot, et on l'analyse ensuite ensemble. 😊

Code : C++ - [Sélectionner](#)

```

void FenPrincipale::clicSelection()
{
    QItemSelectionModel *selection = vue->selectionModel();
    QModelIndexList listeSelections = selection->selectedIndexes();
    QString elementsSelectionnes;

    for (int i = 0 ; i < listeSelections.size() ; i++)
    {
        QVariant elementSelectionne = modele->data(listeSelections[i], Qt::DisplayRole);
        elementsSelectionnes += elementSelectionne.toString() + "<br />";
    }

    QMessageBox::information(this, "Eléments sélectionnés", elementsSelectionnes);
}

```

C'est un peu plus gros bien sûr. 😊

1. La première ligne ne change pas : on récupère l'objet qui contient des informations sur les éléments sélectionnés.
2. Ensuite, au lieu d'appeler currentIndex(), on demande selectedIndexes() parce qu'il peut y en avoir plusieurs.
3. On crée un QString vide dans lequel on stockera la liste des pays pour l'afficher ensuite dans la boîte de dialogue.
4. Vient ensuite une boucle. En effet, l'objet listeSelections récupéré est un tableau (en fait c'est un objet de type QList, mais on peut faire comme si c'était un tableau). On parcourt donc ce tableau ligne par ligne, et on récupère à chaque fois le texte correspondant.
5. On stocke ce texte à la suite du QString, qui se remplit au fur et à mesure.
6. Une fois la boucle terminée, on affiche le QString qui contient la liste des pays sélectionnés. Et voilà le travail !



Ici, je me contente d'afficher la liste des éléments sélectionnés dans une boîte de dialogue, mais en pratique vous ferez sûrement quelque chose de beaucoup plus intelligent avec ça. 😊

En ce qui me concerne je vous ai donné la base pour démarrer, mais je serais bien incapable de vous montrer toutes les utilisations possibles de l'architecture modèle/vue de Qt. Nous ne pouvons pas tout voir, ce serait bien trop vaste.

A vous de voir, maintenant que vous commencez à connaître le principe, de quels outils vous avez besoin. Entraînez-vous avec les autres vues (arbre, tableau) et essayez d'en faire une utilisation plus poussée. Vous pouvez refaire un TP précédent et y intégrer un widget basé sur l'architecture modèle/vue pour vous entraîner.

Ce sera probablement difficile, mais bon, il faut bien des chapitres difficiles dans le cours sinon on va croire que c'est un site pour les débutants ici. 😊

N'oubliez pas de lire la doc surtout, elle contient toutes les informations dont vous avez besoin !

Au fait, il existe des classes "simplifiées" des vues que l'on vient de voir. Ces classes s'appellent QListWidget, QTreeWidget, et

QTableWidget. Elles ne nécessitent pas la création d'un modèle à part, mais sont du coup moins flexibles.
Utilisez-les lorsque vous avez une application très simple et que vous ne voulez pas manipuler l'architecture modèle/vue.

Communiquer en réseau avec son programme

Ah... Le réseau...

C'est un peu le fantasme de la plupart des nouveaux programmeurs : arriver à faire en sorte que son programme puisse communiquer à travers le réseau, que ce soit en local (entre 2 PC chez vous) ou sur internet.

Pourtant, c'est un sujet complexe parce que... il ne suffit pas seulement de savoir programmer en C++, il faut aussi beaucoup de connaissances théoriques sur le fonctionnement du réseau. Les couches d'abstraction, TCP/IP, UDP, Sockets... peut-être avez-vous entendu ces mots-là mais sauriez-vous vraiment les définir ?

En fait, pour que les choses soient claires, il faut savoir que je n'avais pas prévu de rédiger ce chapitre à la base. Tout d'abord parce que ce n'est plus vraiment du GUI (création de fenêtre), donc c'est un peu hors-sujet vis à vis des chapitres précédents. D'autre part, comme je vous l'ai dit, c'est un sujet complexe et il faudrait un tutoriel entier sur plusieurs chapitres pour bien vous expliquer la théorie sur les réseaux... chose que je ne peux pas faire sauf si vous me payez la greffe d'un troisième bras. 

Cependant, vous êtes nombreux à m'avoir demandé de faire un chapitre traitant du réseau. Face à la demande, j'ai finalement accepté de faire une exception.

Exceptionnellement donc, nous n'allons pas vraiment parler que de GUI, nous allons aussi parler de réseau.

Seulement voilà, comme je vous l'ai dit, pour bien faire il faudrait un tutoriel complet que je n'ai ni le temps ni les moyens de rédiger. Du coup, j'ai finalement trouvé un compromis : on va faire une sorte de chapitre-TP. Il y aura de la théorie et de la pratique à la fois.

Nous ne verrons pas tout, nous nous concentrerons sur l'architecture réseau la plus classique (client / serveur). Cela vous donnera les bases pour comprendre un peu comment ça marche, et puis après il ne tiendra plus qu'à vous d'adapter ces exemples à vos programmes.

C'est un chapitre-TP ? Mais alors, quel est le sujet du TP ?

Le sujet du "chapitre-TP" sera la réalisation d'un logiciel de Chat en réseau. Vous pourrez aussi bien communiquer en réseau local (entre vos PC à la maison) qu'entre plusieurs PC via internet.

On y va ? 

On va devoir commencer par un petit cours théorique, absolument indispensable pour comprendre la suite de ce chapitre !

Comment communique-t-on en réseau ?

Voilà une bien bonne question !

A laquelle... je pourrais vous répondre par une encyclopédie en 12 volumes, et encore je n'aurais pas tout expliqué. 

Nous allons donc voir les notions théoriques de base sur le réseau de façon light et ludique. A partir de là, nous pourrons voir comment on utilise ces connaissances en pratique avec Qt pour réaliser un Chat en réseau.

Pour nos exemples, nous allons imaginer 2 utilisateurs. Appelons-les... par exemple Patrice et Ludovic. Patrice et Ludovic ont chacun un ordinateur et ils voudraient communiquer entre eux.



Comment faire ? Comment communiquer, sachant qu'il y a des centaines, des milliers d'autres ordinateurs sur le réseau ? Et comment peuvent-ils se faire comprendre entre eux, faut-il qu'ils parlent le même langage ?

Pour que vous puissiez avoir 2 programmes qui communiquent entre eux via le réseau, il vous faut 3 choses :

1. Connaître l'adresse IP identifiant l'autre ordinateur.
2. Utiliser un port libre et ouvert.
3. Utiliser le même protocole de transmission des données.

Si tous ces éléments sont réunis, c'est bon. 😊

Voyons voir comment faire pour avoir tout ça...

1/ L'adresse IP : identification des machines sur le réseau

La première chose qui devrait vous préoccuper, c'est de savoir comment les ordinateurs font pour se reconnaître entre eux sur un réseau.

Comment fait Patrice pour envoyer un message à Ludovic et seulement à lui ?

Qu'est-ce qu'une IP ?

Il faut savoir que chaque ordinateur est identifié sur le réseau par ce qu'on appelle une adresse IP. C'est une série de nombres, par exemple :

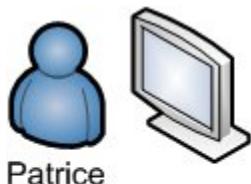
85.215.27.118

Cette adresse représente un ordinateur. Lorsque vous connaissez l'adresse IP de la personne avec qui vous voulez communiquer, vous savez déjà au moins vers qui vous vous dirigez. 🍒

Mais voilà, le problème, parce que sinon ça serait trop simple, c'est qu'un ordinateur peut avoir non pas une mais plusieurs IP. En général aujourd'hui, on peut considérer qu'un ordinateur a en moyenne 3 IP :

- Une IP interne : c'est le localhost, aussi appelé loopback. C'est une IP qui sert pour communiquer à soi-même. Pas très utile vu qu'on n'emprunte pas le réseau du coup, mais ça nous sera très pratique pour les tests vous verrez.
Exemple : 127.0.0.1
- Une IP du réseau local : si vous avez plusieurs ordinateurs en réseau chez vous, ils peuvent communiquer entre eux sans passer par internet grâce à ces IP. Elles sont propres au réseau de votre maison.
Exemple : 192.168.0.3
- Une IP internet : c'est l'IP utilisée pour communiquer avec tous les autres ordinateurs de la planète qui sont connectés à internet.
Exemple : 86.79.12.105

Patrice et Ludovic ont donc plusieurs IP, selon le niveau auquel on se place :



Patrice

127.0.0.1

192.168.0.3

86.76.12.105

Adresse localhost

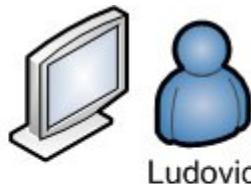
Adresse réseau local

Adresse internet

127.0.0.1

192.168.0.7

90.41.29.47



Ludovic

Si je vous raconte ça, c'est parce que nous aurons besoin d'utiliser l'une ou l'autre de ces IP en fonction de la distance qui sépare Patrice de Ludovic.

Si Patrice et Ludovic sont dans une même maison, reliés par un réseau local, nous utiliserons une IP du réseau local (en rouge sur mon schéma).

Si Patrice et Ludovic sont reliés par internet, nous utiliserons leur adresse internet (en vert).

Pour ce qui est de l'adresse localhost, elle peut nous servir pour "simuler" le fonctionnement du réseau. Si Patrice envoie un message à 127.0.0.1, celui-ci va immédiatement lui revenir. Cela peut nous être utile si on ne veut pas déranger notre ami Ludovic toutes les 5 minutes pour tester la dernière version de notre programme. 🤖

Retrouver son adresse IP

Comment je connais mon IP ? Ou plutôt mes IP ?

Et comment je sais laquelle correspond au réseau local, et laquelle correspond à celle d'internet ?

La méthode dépend de l'IP que vous recherchez.

- Pour l'IP interne : pas besoin d'aller chercher plus loin, à coup sûr c'est 127.0.0.1 (ou son équivalent texte "localhost").
- Pour l'IP locale : pour la retrouver tout dépend de votre système d'exploitation.
 - Sous Windows, ouvrez une invite de commande (par exemple celle que vous utilisez avec Qt pour compiler) et tapez `ipconfig`
Il est possible que vous ayez plusieurs réponses, en fonction des moyens de connexion disponibles (câble ethernet, wifi...). En tout cas, l'une des IP que l'on vous donne est la bonne (à la ligne "Adresse IPv4").
 - Sous Linux ou Mac OS, c'est le même principe dans une console mais pas la même commande : `ifconfig`
L'adresse est en général de la forme "192.168.XXX.XXX", mais cela peut être parfois différent.
- Pour l'IP internet : le plus simple est probablement d'aller sur un site web qui est capable de vous la donner, comme par exemple www.whatismyip.com !

Maintenant que vous connaissez l'adresse IP de votre interlocuteur, alors vous allez pouvoir communiquer avec lui... ou presque. Le problème, c'est qu'il y a plusieurs portes d'entrée sur chaque ordinateur. C'est ce qu'on appelle les ports.

2/ Les ports : différents moyens d'accès à un même ordinateur

Un ordinateur connecté à un réseau reçoit beaucoup de messages en même temps.

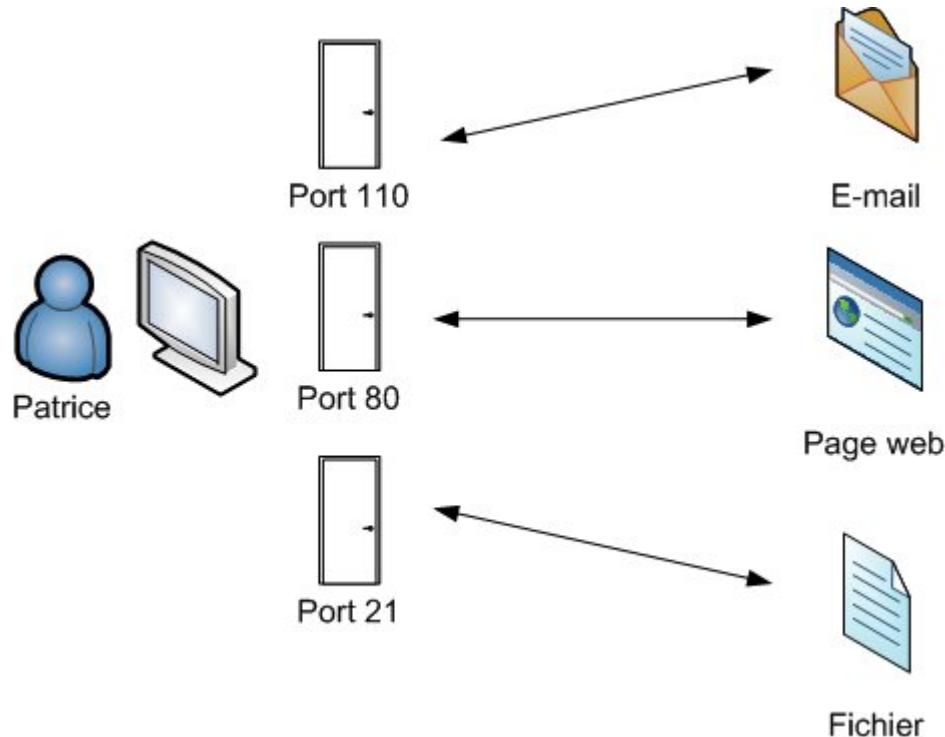
Par exemple, si vous allez sur un site web en même temps que vous récupérez vos mails, des données différentes vont vous arriver simultanément.

Pour ne pas confondre ces données et organiser tout ce bazar, on a inventé le concept de port.

Un port est un nombre compris entre 1 et 65 536. Voici quelques ports célèbres :

- 21 : utilisé par les logiciels FTP pour envoyer et recevoir des fichiers.
- 80 : utilisé pour naviguer sur le web par votre navigateur (par exemple Firefox, ou plutôt zNavigo ☺).
- 110 : utilisé pour la réception de mails.

Imaginez que ces ports sont autant de portes d'entrée à votre ordinateur :



Si on veut faire un programme qui communique avec Ludovic, il va falloir choisir un port qui ne soit pas déjà utilisé par un autre programme.

La plupart des ports dont les numéros sont inférieurs à 1 024 sont déjà réservés par votre machine. Nous ferons donc en sorte de préférence dans notre programme d'utiliser un numéro de port compris entre 1 024 et 65 536.

Pour éviter que n'importe quel programme puisse communiquer sur le réseau et accéder à une machine sans autorisation, on a inventé les firewalls (pare-feu). Leur rôle est de bloquer tous les ports d'une machine, et d'en autoriser seulement certains qui sont considérés comme "sûrs" (comme les ports 21, 80, 110...).

Il faudra bien vérifier la configuration de votre firewall si vous en avez un (il y en a un activé sous Windows par défaut depuis Windows XP SP2), car celui-ci pourrait tout simplement bloquer les communications de notre programme !

3/ Le protocole : transmettre des données avec le même "langage"

Bon, nous savons désormais 2 choses :

- Chaque ordinateur est identifié par une adresse IP.
- On peut accéder à une IP via des milliers de ports différents.

L'IP, vous savez la retrouver. Le port, il faudra en choisir un qui soit libre (nous verrons comment en pratique plus tard). Vous êtes donc maintenant en mesure d'établir une connexion avec un ordinateur distant, car vous avez les 2 éléments nécessaires : une IP et un port.

Il reste maintenant à envoyer des données à l'ordinateur distant pour que les 2 programmes puissent "parler" entre eux. Et ça mine de rien, ce n'est pas simple. En effet, il faut que les 2 programmes parlent la même langue, le même protocole. Il faut qu'ils communiquent de la même façon.

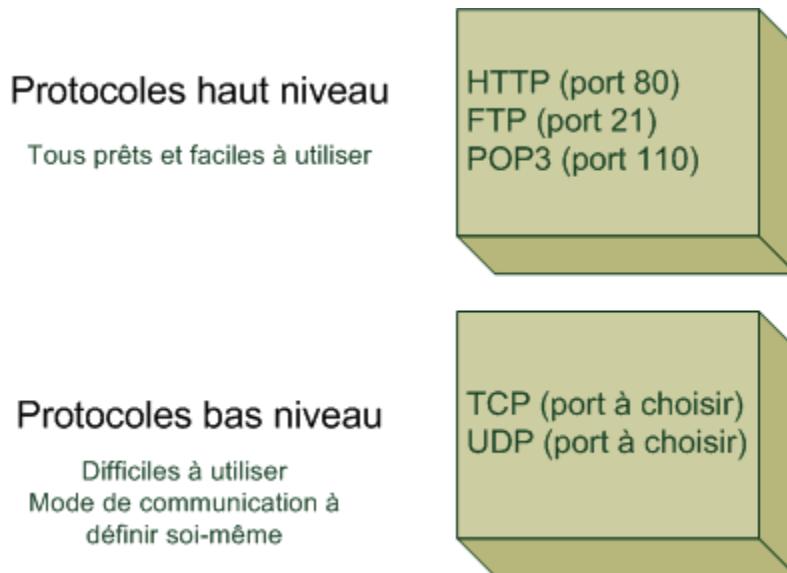
Définition : un protocole est un ensemble de règles qui permettent à 2 ordinateurs de communiquer. Il faut impérativement que les 2 ordinateurs parlent le même protocole pour que l'échange de données puisse fonctionner.

Exemple de la vie courante : vous dites "Bonjour" lorsque vous commencez à parler à quelqu'un, et "Au revoir" lorsque vous partez. Eh bien pour les ordinateurs c'est pareil !

Les différents niveaux des protocoles de communication

Il existe des centaines de protocoles de communication différents. Ceux-ci peuvent être très simples comme très complexes, selon si vous discutez à un "haut niveau" ou à un "bas niveau". On peut donc les ranger dans 2 catégories :

- Protocoles de haut niveau : par exemple le protocole FTP, qui utilise le port 21 pour envoyer et recevoir des fichiers, est un système d'échange de données de haut niveau. Son mode de fonctionnement est déjà écrit et documenté. Il est donc assez facile à utiliser, mais on ne peut pas lui rajouter des possibilités.
- Protocoles de bas niveau : par exemple le protocole TCP. Il est utilisé par les programmes pour lesquels aucun protocole de haut niveau ne convient. Vous devrez manipuler les données qui transitent sur le réseau octet par octet. C'est plus difficile, mais vous pouvez faire tout ce que vous voulez.



Pour ceux qui veulent aller plus loin, renseignez-vous sur le [modèle OSI](#). C'est un modèle d'organisation des données sur le réseau qui vous explique les différents niveaux de communication (on parle de "couches").

Ici j'ai beaucoup (énormément) simplifié le schéma, mais sinon on ne s'en sortait pas en un seul chapitre. 😊

Les protocoles de haut niveau utilisent des ports bien connus et déjà définis.

Les protocoles de bas niveau peuvent emprunter n'importe quel port, sont beaucoup plus flexibles, mais le problème c'est qu'il faut définir tout leur fonctionnement.

En fait, tous les protocoles de haut niveau utilisent des protocoles de bas niveau pour leur fonctionnement interne. Les protocoles de bas niveau sont "la base", on les utilise pour construire des protocoles de plus haut niveau.

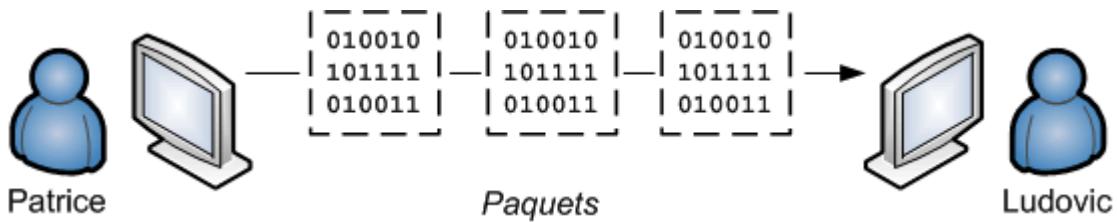
Nous n'allons pas créer un logiciel de mails, ni un client FTP. Nous allons inventer notre propre technique de discussion pour notre programme, notre propre protocole basé sur un protocole de bas niveau... Nous allons donc travailler à bas niveau.

Mauvaise nouvelle : ça va être plus difficile. 😞

Bonne nouvelle : ça va être intéressant techniquement.

Les protocoles de bas niveau TCP et UDP

Il faut savoir que les données s'envoient sur le réseau par petits bouts. On parle de paquets, qui peuvent être chacun découpés en sous-paquets :



Par exemple, imaginons que Patrice envoie à Ludovic le message : "Salut Ludovic, comment ça va ?". Le message ne sera peut-être pas envoyé d'un seul coup, il sera probablement découpé en plus petits paquets. Par exemple, on peut imaginer qu'il y aura 4 sous-paquets (j'invente, car le découpage sera peut-être différent) :

1. Sous-paquet 1 : "Salut Ludov"
2. Sous-paquet 2 : "ic, co"
3. Sous-paquet 3 : "mment ça v"
4. Sous-paquet 4 : "a ?"

Ce n'est pas vous qui gérez le découpage en sous-paquets, c'est le protocole de bas niveau qui s'en occupe. Il est donc impossible de connaître à l'avance la taille des paquets ou même leur nombre.

Il est cependant important de savoir que ça fonctionne comme ça pour la suite.

On peut envoyer ces paquets de plusieurs façons différentes, tout dépend du protocole de bas niveau que l'on utilise :

- Protocole TCP : le plus classique. Il nécessite d'établir une connexion au préalable entre les ordinateurs. Il y a un système de contrôle qui permet de demander à renvoyer un paquet au cas où l'un d'entre eux se serait perdu sur le réseau (ça arrive ). Par conséquent, avec TCP on est sûr que tous les paquets arrivent à destination, et dans le bon ordre. En contrepartie de ces contrôles sécurisants, l'envoi des données est plus lent qu'avec UDP.
- Protocole UDP : il ne nécessite pas d'établir de connexion au préalable et il est très rapide. En revanche, il n'y a aucun contrôle ce qui fait qu'un paquet de données peut très bien se perdre sans qu'on en soit informé, ou les paquets peuvent arriver dans le désordre !

Il va falloir choisir l'un de ces 2 protocoles.

Pour moi, le choix est tout fait : ce sera TCP. En effet, nous allons réaliser un Chat et nous ne pouvons pas nous permettre que des messages (ou des bouts de messages) n'arrivent pas à destination, sinon la conversation pourrait devenir difficile à suivre et on risquerait de recevoir des messages comme : "Salut Ludovmment ça va ?" 

Mais alors, du coup tout le monde utilise TCP pour être sûr que le paquet arrive à destination non ? Qui peut bien être assez fou pour utiliser UDP ?

Certaines applications complexes qui utilisent beaucoup le réseau peuvent être amenées à utiliser UDP. Je pense par exemple aux jeux vidéo.

Prenez un jeu de stratégie comme Starcraft, ou un FPS comme Quake par exemple : il peut y avoir des dizaines d'unités qui se déplacent sur la carte en même temps. Il faut en continu envoyer la nouvelle position des unités qui se déplacent à tous les ordinateurs de la partie. On a donc besoin d'un protocole rapide comme UDP, et si un paquet se perd ce n'est pas grave : vu que la position des joueurs est rafraîchie plusieurs fois par seconde, ça ne se verra pas.

L'architecture du projet de Chat avec Qt

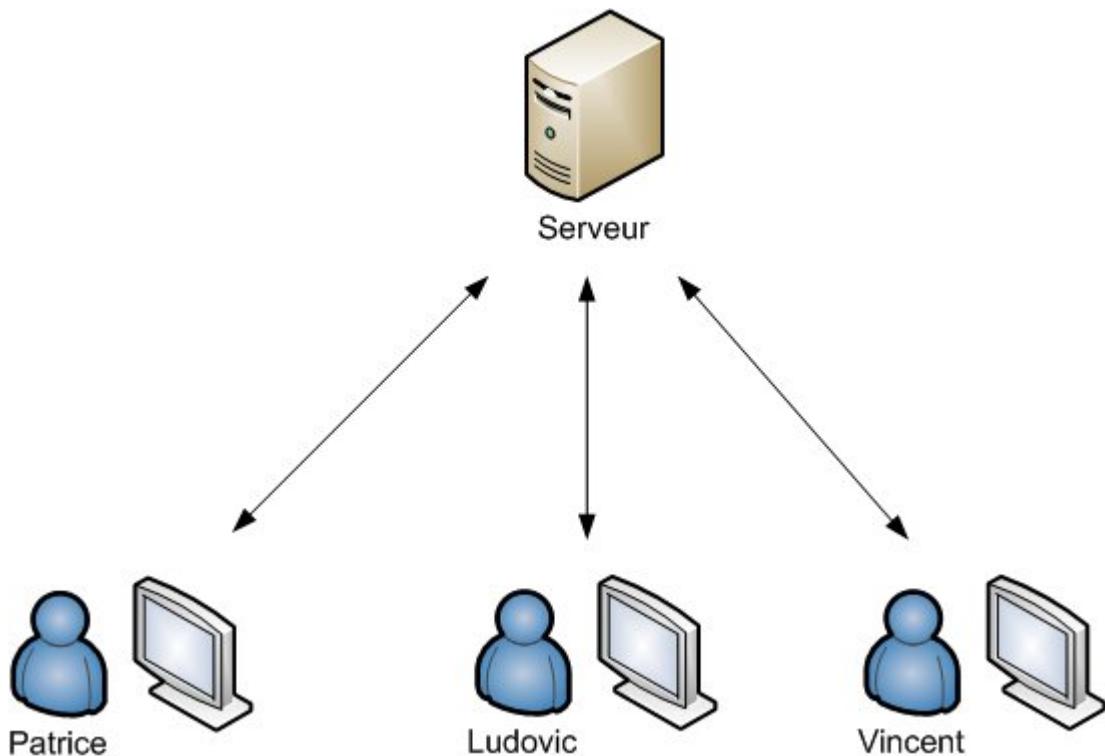
Nous venons de voir quelques petites notions théoriques sur le réseau, mais il va encore falloir préciser quelle est l'architecture réseau de notre programme de Chat.

Jusqu'ici, nous avons supposé un cas très simple : il n'y avait que 2 ordinateurs (celui de Patrice et celui de Ludovic). Le problème, c'est que notre programme de Chat doit permettre à plus de 2 personnes de discuter en même temps. Imaginons qu'une troisième personne appelée Vincent arrive sur le Chat. Vous le placez où sur le schéma ? Au milieu entre les 2 autres compères ? 😊

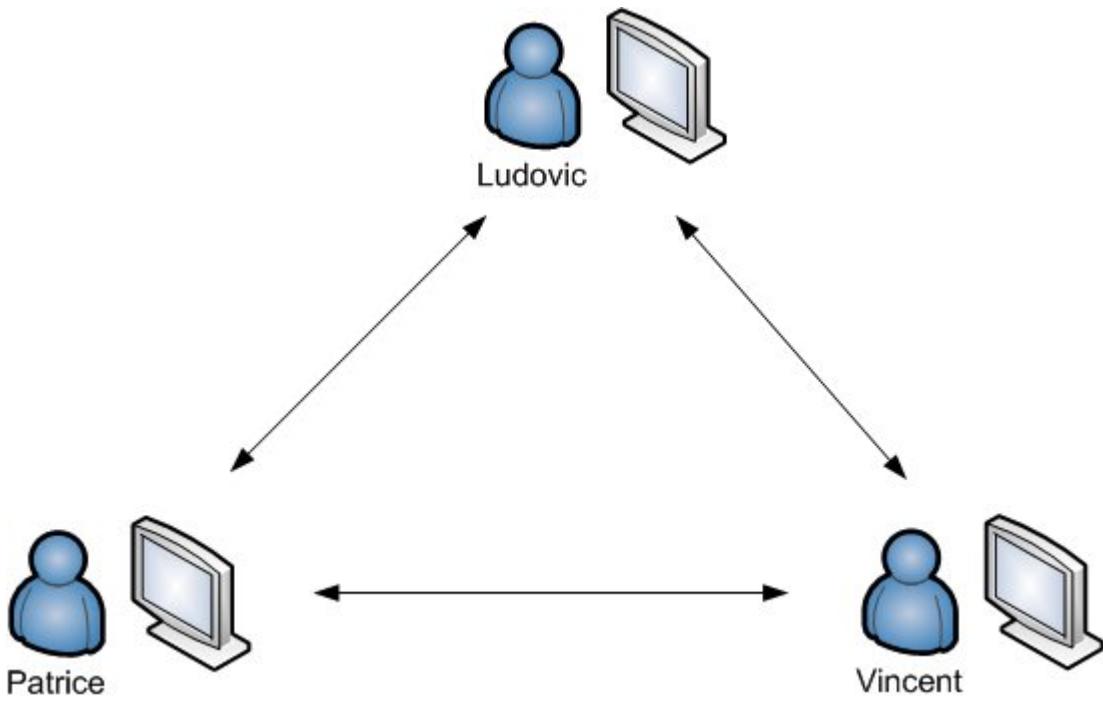
Les architectures réseau

Pour faire simple, on a 2 architectures possibles pour résoudre le problème :

- Une architecture client / serveur : c'est l'architecture réseau la plus classique et la plus simple à mettre en oeuvre. Les machines des utilisateurs (Patrice, Ludovic, Vincent...) sont appelées des "clients". En plus de ces machines, on utilise un autre ordinateur (appelé "serveur") qui va se charger de répartir les communications entre les clients.



- Une architecture Peer-To-Peer (P2P) : ce mode plus complexe est dit décentralisé, car il n'y a pas de serveur. Chaque client peut communiquer directement avec un autre client. C'est plus direct, ça évite d'encombrer un serveur, mais c'est plus délicat à mettre en place.



Nous, nous allons utiliser une architecture client / serveur, la plus simple.

Il va en fait falloir faire non pas un mais deux projets :

- Un projet "serveur" : pour créer le programme qui va répartir les messages entre les clients.
- Un projet "client" : pour chaque client qui participera au Chat.

Vous n'êtes pas obligés d'utiliser une machine spécialement pour faire serveur. L'une des machines des clients peut aussi faire office de serveur. Il suffira de faire tourner un programme "serveur" en même temps qu'un programme "client", c'est tout à fait possible.

En pratique donc, une seule personne lancera le programme "serveur" et le programme "client" à la fois, et toutes les autres lanceront uniquement le programme "client".

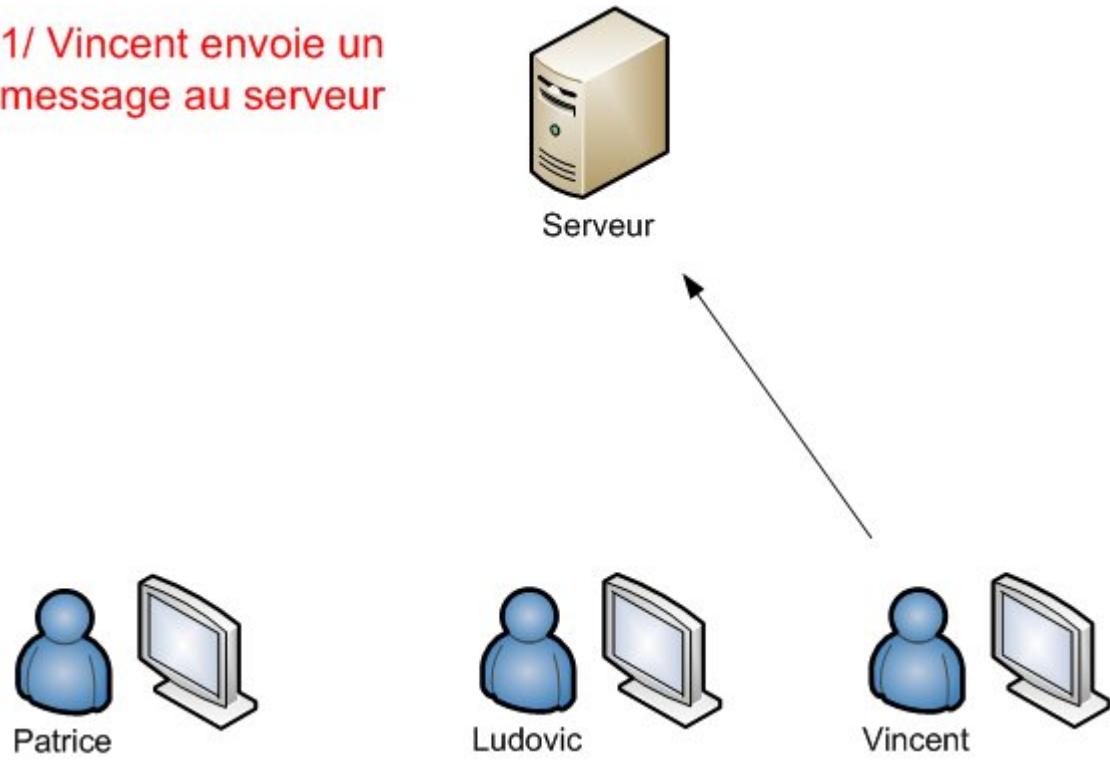
Principe de fonctionnement du Chat

Le principe du Chat est simple : une personne écrit un message, et tout le monde reçoit ce message sur son écran.

Les choses se passent en 2 temps :

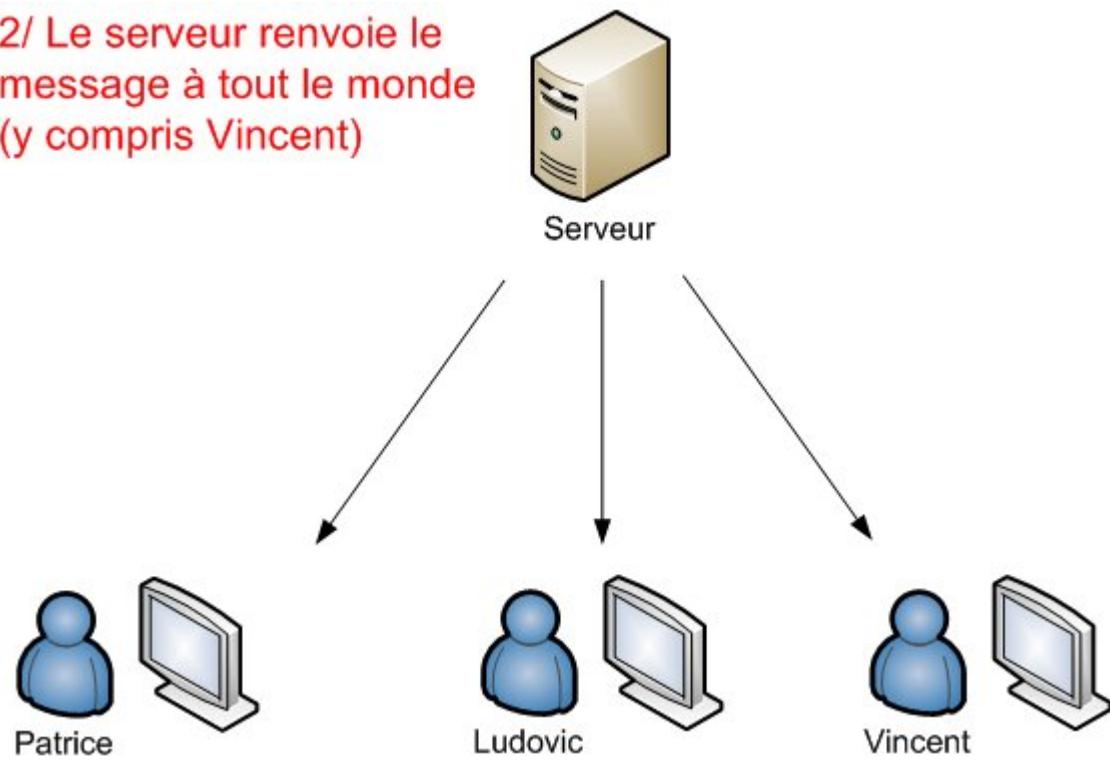
- Un client envoie un message au serveur.

1/ Vincent envoie un message au serveur



- Le serveur renvoie ce message à tous les clients pour qu'il s'affiche sur leur fenêtre.

2/ Le serveur renvoie le message à tout le monde (y compris Vincent)



Pourquoi le serveur renverrait-il le message à Vincent, vu que c'est lui qui l'a envoyé ?

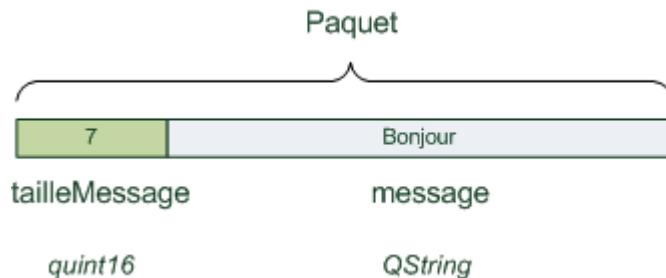
On peut gérer les choses de plusieurs manières. On pourrait s'arranger pour que le serveur n'envoie pas le message à Vincent pour éviter un trafic réseau inutile, mais cela compliquerait un petit peu le programme.

Il est plus simple de faire en sorte que le serveur renvoie le message à tout le monde sans distinction. Vincent verra donc son message s'afficher sur son écran de discussion uniquement quand le serveur l'aura reçu et le lui aura renvoyé. Cela permet de vérifier en outre que la communication sur le réseau fonctionne correctement.

Structure des paquets

Les messages qui circuleront sur le réseau seront placés dans des paquets. C'est à nous de définir la structure des paquets que l'on veut envoyer.

Par exemple, quand Vincent va envoyer un message, un paquet va être créé avant d'être envoyé sur le réseau. Voici la structure de paquet que je propose pour notre programme de Chat :

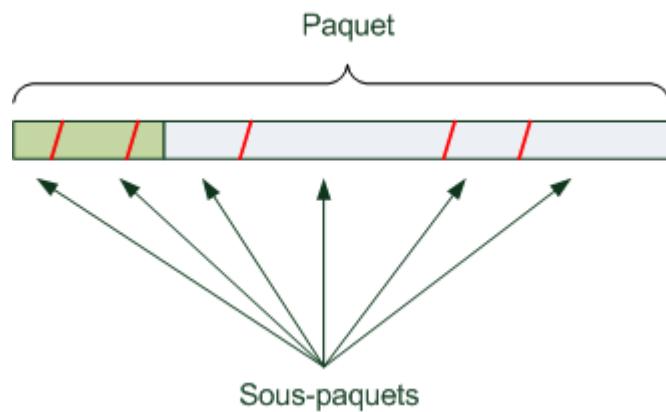


Le paquet est constitué de 2 parties :

- **tailleMessage** : un nombre entier qui sert à indiquer la taille du message qui suit. Cela permet au serveur de connaître la taille totale du message envoyé, pour qu'il puisse savoir quand il a reçu le message en entier.
Ce nombre ne sera pas de type int comme on aurait pu s'y attendre mais de type quint16. En effet, le type int peut avoir une taille différente selon les machines sur le réseau (un int peut prendre 16 bits de mémoire sur une machine, et 8 bits sur une autre).
Pour résoudre le problème, on utilise un type spécial de Qt, le quint16, qui correspond à un nombre entier prenant 16 bits de mémoire quelle que soit la machine (quand je vous avais dit que c'était bas niveau 😊).
quint16 signifie : "Qt Unsigned Int 16", soit "Entier non signé codé sur 16 bits".
- **message** : c'est le message envoyé par le client. Ce message sera de type QString (ça c'est simple, vous connaissez !).

Pourquoi envoie-t-on la taille du message en premier ? On ne pourrait pas envoyer le message tout court ?

Il faut savoir que le protocole TCP va découper le paquet en sous-paquets avant de l'envoyer sur le réseau. Il n'enverra peut-être pas tout d'un coup. Par exemple, notre paquet pourrait être découpé comme ceci :



On n'a aucun contrôle sur la taille de ces sous-paquets, et il n'y a aucun moyen de savoir à l'avance comment ça va être découpé. Le problème, c'est que le serveur va recevoir ces paquets petit à petit, et non pas tout d'un coup. Il ne peut pas savoir quand la totalité du message a été reçue.

Le protocole TCP ne permet pas de contrôler la taille des sous-paquets ni leur nombre, par contre il s'arrange pour que les paquets arrivent à destination dans le bon ordre (ce qui est pratique, parce que sinon ça aurait été encore plus compliqué à remettre en ordre 😊).

Pour information, le protocole UDP, qui est plus rapide, ne fait aucun contrôle sur l'ordre des paquets envoyés !

Bon, il faut qu'on arrive à savoir quand on a reçu le message en entier, et donc quand ce n'est plus la peine d'attendre de nouveaux sous-paquets.

Pour résoudre ce problème, on envoie la taille du message dans un premier temps. Lorsque la taille du message a été reçue, on va attendre que le message soit au complet. On se base sur `tailleMessage` pour savoir combien d'octets il nous reste à recevoir.

Lorsqu'on a récupéré tous les octets restants du paquet, on sait que le paquet est au complet, et cela veut dire qu'on a donc reçu le message entier.

Bon, c'est pas simple, mais je vous avais prévenu hein ! 😊

Réalisation du serveur

Comme je vous l'ai dit, nous allons devoir réaliser 2 projets :

- Un projet "client"
- Un projet "serveur"

Nous commençons par le serveur.

Création du projet

Créez un nouveau projet constitué de 3 fichiers :

- `main.cpp`
- `FenServeur.cpp`
- `FenServeur.h`

Faites un `qmake -project` pour générer le `.pro`.

Editez ensuite ce fichier `.pro` pour demander à Qt de rajouter la gestion du réseau :

Code : Autre - [Sélectionner](#)

```
#####
# Automatically generated by qmake (2.01a) mer. 25. juin 14:54:55 2008
#####

TEMPLATE = app
QT += network
TARGET =
DEPENDPATH += .
INCLUDEPATH += .

# Input
HEADERS += FenServeur.h
SOURCES += FenServeur.cpp main.cpp
```

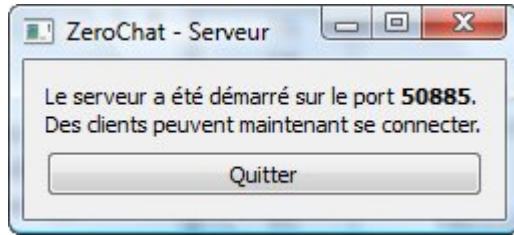
Avec `QT += network`, Qt sait que le projet va utiliser le réseau et peut préparer un makefile approprié.

La fenêtre du serveur

Le serveur est une application qui tourne en tâche de fond. Normalement, rien ne nous oblige à créer une fenêtre pour ce projet,

mais on va quand même en faire une pour que l'utilisateur puisse arrêter le serveur en fermant la fenêtre.

Notre fenêtre sera toute simple, elle affichera le texte "Le serveur a été lancé sur le port XXXX" et un bouton "Quitter".



Construire la fenêtre sera donc très simple, la vraie difficulté sera de faire toute la gestion du réseau derrière.

main.cpp

Les main sont toujours très simples et classiques avec Qt :

Code : C++ - [Sélectionner](#)

```
#include <QApplication>
#include "FenServeur.h"

int main(int argc, char* argv[])
{
    QApplication app(argc, argv);

    FenServeur fenetre;
    fenetre.show();

    return app.exec();
}
```

FenServeur.h

Voici maintenant le header de la fenêtre du serveur :

Code : C++ - [Sélectionner](#)

```

#ifndef HEADER_FENSERVEUR
#define HEADER_FENSERVEUR

#include <QtGui>
#include <QtNetwork>

class FenServeur : public QWidget
{
    Q_OBJECT

public:
    FenServeur();
    void envoyerATous(const QString &message);

private slots:
    void nouvelleConnexion();
    void donneesRecues();
    void deconnexionClient();

private:
    QLabel *etatServeur;
    QPushButton *boutonQuitter;

    QTcpServer *serveur;
    QList<QTcpSocket *> clients;
    quint16 tailleMessage;
};

#endif

```

Notre fenêtre hérite de QWidget, ce qui nous permet de créer une fenêtre simple. Elle est constituée comme vous le voyez d'un QLabel et d'un QPushButton comme prévu.

En plus de ça, j'ai rajouté d'autres attributs spécifiques à la gestion du réseau :

- QTcpServer *serveur : c'est l'objet qui représente le serveur sur le réseau.
- QList<QTcpSocket *> clients : c'est un tableau qui contient la liste des clients connectés. On aurait pu utiliser un tableau classique, mais on va passer par une QList, un tableau de taille dynamique. En effet, on ne connaît pas à l'avance le nombre de clients qui se connecteront. Chaque QTcpSocket de ce tableau représentera une connexion à un client.
Je ne vais pas m'étendre dans ce chapitre sur les [QList](#). Considérez juste que c'est une classe qui permet de gérer un tableau de taille variable, ce qui est très pratique quand on ne sait pas comme ici le nombre de clients qui vont se connecter.
Pour plus d'infos pour apprendre à vous en servir, lisez la doc. 😊
- quint16 tailleMessage : ce quint16 sera utilisé dans le code pour se "souvenir" de la taille du message que le serveur est en train de recevoir. Nous en avons déjà parlé et nous en reparlerons plus loin.

Voilà, à part ces attributs, on note que la classe est constituée de plusieurs méthodes (dont des slots) :

- Le constructeur : il initialise les widgets sur la fenêtre et initialise aussi le serveur (QTcpServer) pour qu'il démarre.
- envoyerATous() : une méthode à nous qui se charge d'envoyer à tous les clients connectés le message passé en paramètre.
- Slot nouvelleConnexion() : appelé lorsqu'un nouveau client se connecte.
- Slot donneesRecues() : appelé lorsque le serveur reçoit des données. Attention, c'est là que c'est délicat, car ce slot est appelé à chaque sous-paquet reçu. Il faudra "attendre" d'avoir reçu le nombre d'octets indiqués dans tailleMessage avant de pouvoir considérer qu'on a reçu le message entier.
- Slot deconnexionClient() : appelé lorsqu'un client se déconnecte.

Implémentons ces méthodes en cœur, dans la joie et la bonne humeur ! 😊

FenServeur.cpp

Le constructeur

Le constructeur se charge de placer les widgets sur la fenêtre et de faire démarrer le serveur via le QTcpServer :

Code : C++ - [Sélectionner](#)

```
FenServeur::FenServeur()
{
    // Cr ation et disposition des widgets de la fen tre
    etatServeur = new QLabel;
    boutonQuitter = new QPushButton(tr("Quitter"));
    connect(boutonQuitter, SIGNAL(clicked()), qApp, SLOT(quit()));

    QVBoxLayout *layout = new QVBoxLayout;
    layout->addWidget(etatServeur);
    layout->addWidget(boutonQuitter);
    setLayout(layout);

    setWindowTitle(tr("ZeroChat - Serveur"));

    // Gestion du serveur
    serveur = new QTcpServer(this);
    if (!serveur->listen(QHostAddress::Any, 50885)) // D marrage du serveur sur toutes les interfaces
    {
        // Si le serveur n'a pas  t  d marr  correctement
        etatServeur->setText(tr("Le serveur n'a pas pu  tre d marr . Raison :<br />") + serveur->errorString());
    }
    else
    {
        // Si le serveur a  t  d marr  correctement
        etatServeur->setText(tr("Le serveur a  t  d marr  sur le port <strong>") + QString::number(serveur->serverPort()) + "</strong>.");
        connect(serveur, SIGNAL(newConnection()), this, SLOT(nouvelleConnexion()));
    }

    tailleMessage = 0;
}
```

3

5
6

J'ai fait en sorte de bien commenter mes codes sources pour vous aider du mieux possible   comprendre ce qui se passe. Vous voyez bien une premi re  tape o  on dispose les widgets sur la fen tre (classique, rien de nouveau) et une seconde  tape o  on d marre le serveur.

Quelques pr cisions sur la seconde  tape, la plus int ressante pour ce chapitre. On cr e un nouvel objet de type QTcpServer dans un premier temps (ligne 16). On lui passe en param tre this, un pointeur vers la fen tre, pour faire en sorte que la fen tre soit le parent du QTcpServer. Cela permet de faire en sorte que le serveur soit automatiquement d truit lorsqu'on quitte la fen tre.

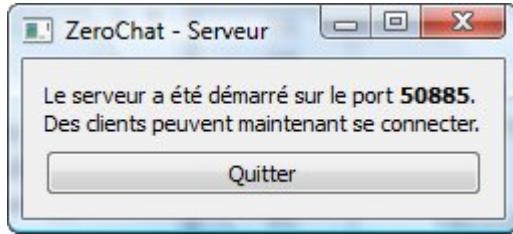
Ensuite, on essaie de d marrer le serveur gr ce   serveur->listen(QHostAddress::Any, 50885) . Il y a 2 param tres :

- L'IP : c'est l'IP sur laquelle le serveur " cout " si de nouveaux clients arrivent. Comme je vous l'avais dit, un ordinateur peut avoir plusieurs IP : une IP interne (127.0.0.1), une IP pour le r seau local, une IP sur internet, etc. La mention QHostAddress::Any autorise toutes les connexions : internes (clients connect s sur la m me machine), locales (clients connect s sur le m me r seau local) et externes (clients connect s via internet).
- Le port : c'est le num ro du port sur lequel on souhaite lancer le serveur. J'ai choisi un num ro au hasard, compris entre 1 024 et 65 536. J'aurais aussi pu omettre ce param tre, dans ce cas le serveur aurait choisi un port libre au hasard. N'h sitez pas   changer la valeur si le port n'est pas libre chez vous.

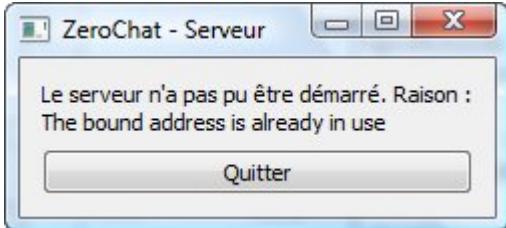
La m thode listen() renvoie un bool en : vrai si le serveur a bien pu se lancer, faux s'il y a eu un probl me. On affiche un message en cons quence sur la fen tre du serveur.

Si le d marrage du serveur a fonctionn , on connecte le signal newConnection() vers notre slot personnalis  nouvelleConnexion() pour traiter l'arriv e d'un nouveau client sur le serveur.

Si tout va bien, la fen tre suivante devrait donc s'ouvrir :



S'il y a une erreur, vous aurez un message d'erreur adapté. Par exemple, essayez de lancer une seconde fois le serveur alors qu'un autre serveur tourne déjà :



Ici, comme le port 50885 est déjà utilisé par un 1er serveur, notre 2nd serveur n'a pas le droit de démarrer sur ce port. D'où l'erreur.



Slot nouvelleConnexion()

Ce slot est appelé dès qu'un nouveau client se connecte au serveur :

Code : C++ - [Sélectionner](#)

```
void FenServeur::nouvelleConnexion()
{
    envoyerATous(tr("Un nouveau client vient de se connecter"));

    QTcpSocket *nouveauClient = serveur->nextPendingConnection();
    clients << nouveauClient;

    connect(nouveauClient, SIGNAL(readyRead()), this, SLOT(donneesRecues()));
    connect(nouveauClient, SIGNAL(disconnected()), this, SLOT(deconnexionClient()));
}
```

On envoie à tous les clients déjà connectés un message comme quoi un nouveau client vient de se connecter. On verra le contenu de la méthode envoyerATous() un peu plus loin.

Chaque client est représenté par un QTcpSocket. Pour récupérer la socket correspondant au nouveau client qui vient de se connecter, on appelle la méthode nextPendingConnection() du QTcpServer. Cette méthode retourne la QTcpSocket du nouveau client.

Comme je vous l'ai dit, on conserve la liste des clients connectés dans un tableau, appelé clients.

Ce tableau est géré par la classe QList qui est très simple d'utilisation. On ajoute le nouveau client à la fin du tableau très facilement, comme ceci :

Code : C++ - [Sélectionner](#)

```
clients << nouveauClient;
```

(vive la surcharge de l'opérateur << 😊)

On connecte ensuite les signaux que peut envoyer le client à des slots. On va gérer 2 signaux :

- `readyRead()` : signale que le client a envoyé des données. Ce signal est émis pour chaque sous-paquet reçu. Lorsqu'un client enverra un message, ce signal pourra donc être émis plusieurs fois jusqu'à ce que tous les sous-paquets soient arrivés. C'est notre slot personnalisé `donneesRecues()` (qui sera coton à écrire 😊) qui traitera les sous-paquets.
- `disconnected()` : signale que le client s'est déconnecté. Notre slot se chargera d'informer les autres clients de son départ et de supprimer la `QTcpSocket` correspondante dans la liste des clients connectés.

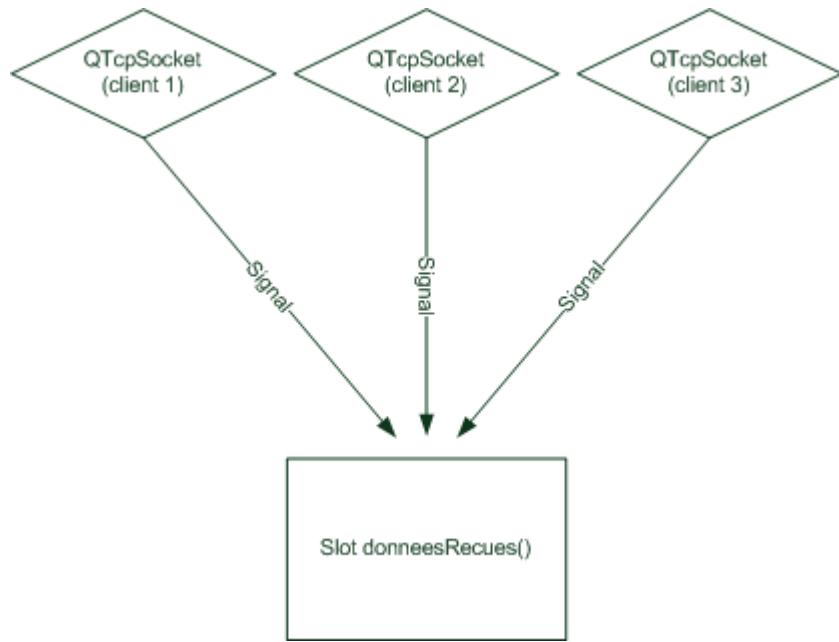
Slot `donneesRecues()`

Voilà sans aucun doute LE point le plus délicat de ce chapitre. C'est un slot qui va être appelé à chaque fois qu'on reçoit un sous-paquet d'un des clients.

On a au moins 2 problèmes pas évidents à résoudre :

- Comme on va recevoir plusieurs sous-paquets, il va falloir "attendre" d'avoir tout reçu avant de pouvoir dire qu'on a reçu le message en entier.
- C'est le même slot qui est appelé quel que soit le client qui a envoyé un message. Du coup, comment savoir quel est le client à l'origine du message pour récupérer les données ?

Il faut utiliser l'objet `QTcpSocket` du client pour récupérer les sous-paquets qui ont transité par le réseau. Le problème, c'est qu'on a connecté les signaux de tous les clients à un même slot :



Comment le slot sait-il dans quelle `QTcpSocket` lire les données ?

Vous ne pouviez pas trop le deviner, et à vrai dire je ne savais pas moi-même comment faire avant d'écrire ce chapitre 😊 .

Il se trouve que j'ai découvert qu'on pouvait appeler la méthode [sender\(\) de QObject](#) dans le slot pour retrouver un pointeur vers l'objet à l'origine du message. Très pratique ! 😊

Nouveau problème : cette méthode renvoie systématiquement un `QObject` (classe générique de Qt) car elle ne sait pas à l'avance de quel type sera l'objet. Notre objet `QTcpSocket` sera donc représenté par un `QObject`.

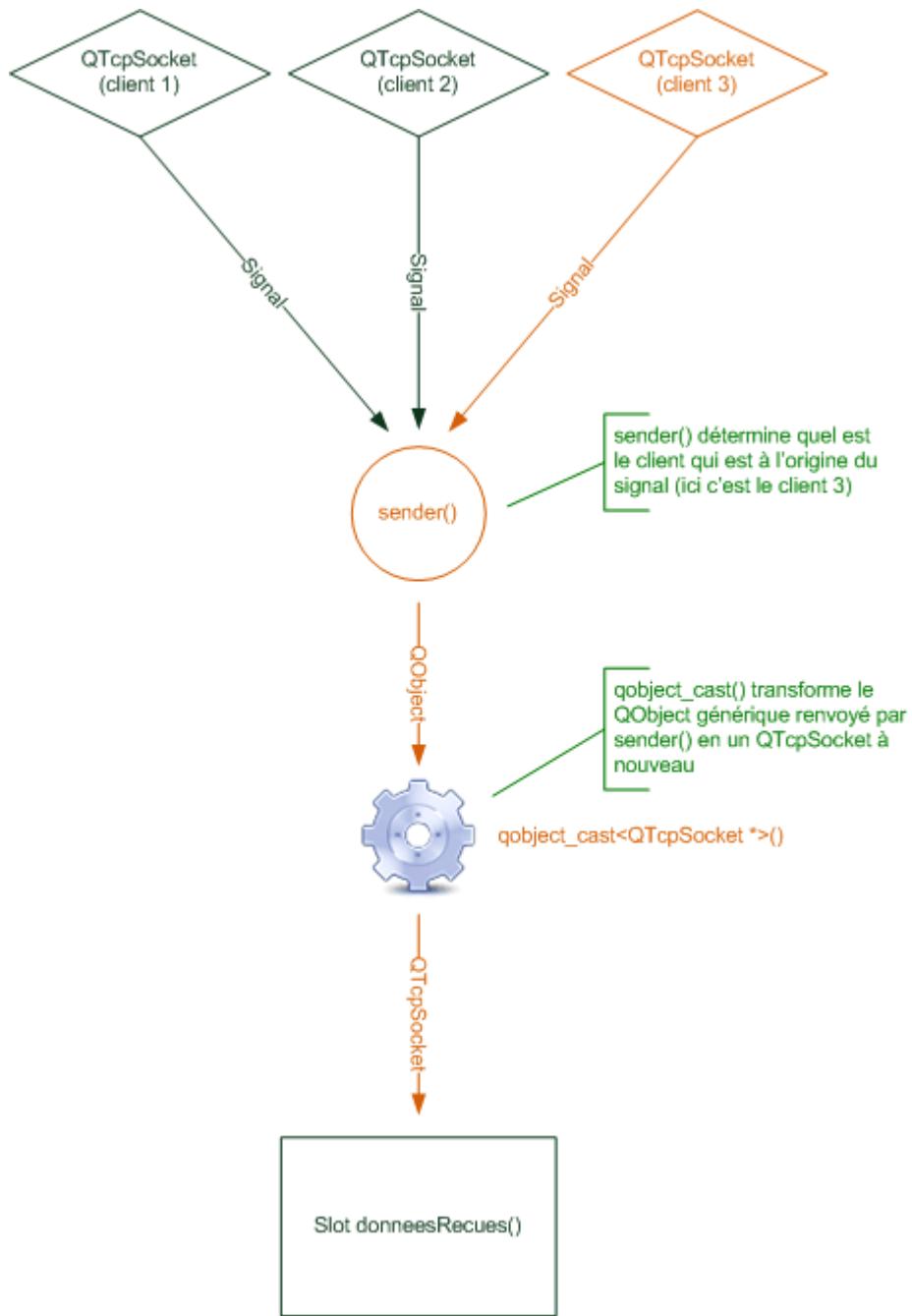
Pour le transformer à nouveau en `QTcpSocket`, il faudra forcer sa conversion à l'aide de la méthode `qobject_cast()`.

En résumé, pour obtenir un pointeur vers la bonne `QTcpSocket` à l'origine du signal, il faudra écrire :

Code : C++ - [Sélectionner](#)

```
QTcpSocket *socket = qobject_cast<QTcpSocket *>(sender());
```

Ce qui, schématiquement, revient à faire ceci :



1. On utilise `sender()` pour déterminer l'objet à l'origine du signal.
2. Comme `sender()` renvoie systématiquement un `QObject`, il faut le transformer à nouveau en `QTcpSocket`. Pour cela, on passe l'objet en paramètre à la méthode `qobject_cast()`, on indiquant entre les chevrons le type de retour que l'on souhaite obtenir : `<QTcpSocket *>`.

La méthode `qobject_cast()` est similaire au `dynamic_cast()` de la bibliothèque standard du C++. Son rôle est de forcer la transformation d'un objet d'un type vers un autre.

Il se peut que le `qobject_cast()` n'ait pas fonctionné (par exemple parce que l'objet n'était pas de type `QTcpSocket` contrairement à ce qu'on attendait). Dans ce cas, il renvoie 0. Il faut que l'on teste si le `qobject_cast()` a fonctionné avant d'aller plus loin. On va faire un return qui va arrêter la méthode s'il y a eu un problème :

Code : C++ - [Sélectionner](#)

```
QTcpSocket *socket = qobject_cast<QTcpSocket *>(sender());  
if (socket == 0) // Si par hasard on n'a pas trouvé le client à l'origine du signal, on arrête.
```

3

5
6

4

On peut ensuite travailler à récupérer les données. On commence par créer un flux de données pour lire ce que contient la socket :

Code : C++ - [Sélectionner](#)

```
QDataStream in(socket);
```

Notre objet "in" va nous permettre de lire le contenu du sous-paquet que vient de recevoir la socket du client.

C'est maintenant que l'on va utiliser l'entier tailleMessage défini en tant qu'attribut de la classe. Si lors de l'appel au slot ce tailleMessage vaut 0, cela signifie qu'on est en train de recevoir le début d'un nouveau message. On demande à la socket combien d'octets ont été reçus dans le sous-paquet grâce à la méthode bytesAvailable(). Si on a reçu moins d'octets que la taille d'un quint16, on arrête la méthode de suite. On attendra le prochain appel de la méthode pour vérifier à nouveau si on a reçu assez d'octets pour récupérer la taille du message.

Code : C++ - [Sélectionner](#)

```
if (tailleMessage == 0) // Si on ne connaît pas encore la taille du message, on essaie de la déterminer.  
{  
    if (socket->bytesAvailable() < (int)sizeof(quint16)) // On n'a pas reçu la taille du message.  
        return;  
  
    in >> tailleMessage; // Si on a reçu la taille du message en entier, on la récupère.
```

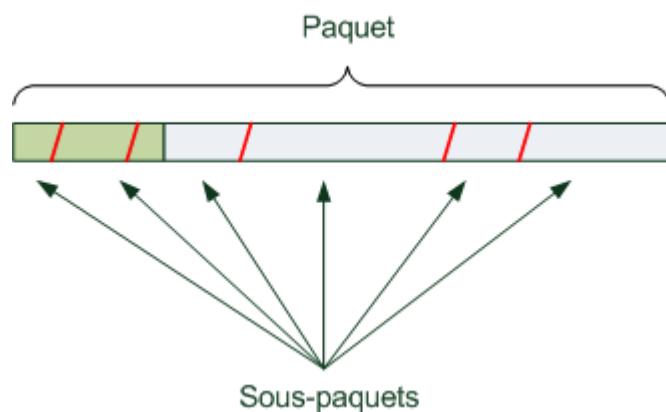
3

5
6

4

La ligne 6 est exécutée uniquement si on a reçu assez d'octets. En effet, le return a arrêté la méthode avant si ce n'était pas le cas. On récupère donc la taille du message et on la stocke. On la "retient" pour la suite des opérations.

Pour bien comprendre ce code, il faut se rappeler que le paquet est découpé en sous-paquets :



Notre slot est appelé à chaque fois qu'un sous-paquet a été reçu.

On vérifie si on a reçu assez d'octets pour récupérer la taille du message (première section en gris foncé). La taille de la première section "tailleMessage" peut être facilement retrouvée grâce à l'opérateur sizeof() que vous avez probablement déjà utilisé. Si on n'a pas reçu assez d'octets, on arrête la méthode (return). On attendra que le slot soit à nouveau appelé et on vérifiera alors cette fois si on a reçu assez d'octets.

Maintenant la suite des opérations. On a reçu la taille du message. On va maintenant essayer de récupérer le message lui-même :

Code : C++ - [Sélectionner](#)

3 4 // Si on connaît la taille du message, on vérifie si on a reçu le message en entier
if (socket->bytesAvailable() < tailleMessage) // Si on n'a pas encore tout reçu, on arrête
 return;

Le principe est le même. On regarde le nombre d'octets reçus, et si on en a moins que la taille annoncée du message, on arrête (return).

Si tout va bien, on peut passer à la suite de la méthode. Si ces lignes s'exécutent, c'est qu'on a reçu le message en entier, donc qu'on peut le récupérer dans une QString :

Code : C++ - [Sélectionner](#)

```
// Si ces lignes s'exécutent, c'est qu'on a reçu tout le message : on peut le récupérer !
QString message;
in >> message;
```

Notre QString "message" contient maintenant le message envoyé par le client !

Ouf ! Le serveur a reçu le message du client !

Mais ce n'est pas fini : il faut maintenant renvoyer le message à tous les clients comme je vous l'avais expliqué. Pour reprendre notre exemple, Vincent vient d'envoyer un message au serveur, celui-ci l'a récupéré et s'apprête à le renvoyer à tout le monde.

L'envoi du message à tout le monde se fait via la méthode envoyerATous dont je vous ai déjà parlé et qu'il va falloir écrire.

Code : C++ - [Sélectionner](#)

```
// 2 : on renvoie le message à tous les clients
envoyerATous(message);
```

On a presque fini. Il manque juste une petite chose : remettre tailleMessage à 0 pour que l'on puisse recevoir de futurs messages d'autres clients :

Code : C++ - [Sélectionner](#)

```
// 3 : remise de la taille du message à 0 pour permettre la réception des futurs messages
tailleMessage = 0;
```

Si on n'avait pas fait ça, le serveur aurait cru lors du prochain sous-paquet reçu que le nouveau message est de la même longueur que le précédent, ce qui n'est certainement pas le cas. 😊

Bon, résumons le slot en entier :

Code : C++ - [Sélectionner](#)

```

void FenServeur::donneesRecues()
{
    // 1 : on reçoit un paquet (ou un sous-paquet) d'un des clients

    // On détermine quel client envoie le message (recherche du QTcpSocket du client)
    QTcpSocket *socket = qobject_cast<QTcpSocket *>(sender());
    if (socket == 0) // Si par hasard on n'a pas trouvé le client à l'origine du signal,
        return;

    // Si tout va bien, on continue : on récupère le message
    QDataStream in(socket);

    if (tailleMessage == 0) // Si on ne connaît pas encore la taille du message, on essaie
    {
        if (socket->bytesAvailable() < (int)sizeof(quint16)) // On n'a pas reçu la taille
            return;

        in >> tailleMessage; // Si on a reçu la taille du message en entier, on la récupère
    }

    // Si on connaît la taille du message, on vérifie si on a reçu le message en entier
    if (socket->bytesAvailable() < tailleMessage) // Si on n'a pas encore tout reçu, on arrête
        return;

    // Si ces lignes s'exécutent, c'est qu'on a reçu tout le message : on peut le récupérer
    QString message;
    in >> message;

    // 2 : on renvoie le message à tous les clients
    envoyerATous(message);

    // 3 : remise de la taille du message à 0 pour permettre la réception des futurs messages
    tailleMessage = 0;
}

```

3

6

J'espère avoir été clair, car ce slot n'est pas simple et pas très facile à lire je dois bien avouer. La clé, le truc à comprendre, c'est que chaque return arrête la méthode. Le slot sera à nouveau appelé au prochain sous-paquet reçu, donc ces instructions s'exécuteront probablement plusieurs fois pour un message.

Si la méthode arrive à s'exécuter jusqu'au bout, c'est qu'on a reçu le message en entier. 😊

Slot deconnexionClient()

Ce slot est appelé lorsqu'un client se déconnecte.

On va envoyer un message à tous les clients encore connectés pour qu'ils sachent qu'un client vient de partir. Puis, on supprime la QTcpSocket correspondant au client dans notre tableau QList. Ainsi, le serveur "oublie" ce client, il ne considère plus qu'il fait partie des connectés.

Voici le slot en entier :

Code : C++ - [Sélectionner](#)

```

void FenServeur::deconnexionClient()
{
    envoyerATous(tr("<em>Un client vient de se déconnecter</em>"));

    // On détermine quel client se déconnecte
    QTcpSocket *socket = qobject_cast<QTcpSocket *>(sender());
    if (socket == 0) // Si par hasard on n'a pas trouvé le client à l'origine du signal,
        return;

    clients.removeOne(socket);

    socket->deleteLater();

```

3

4

5

6

Comme plusieurs signaux sont connectés à ce slot, on ne sait pas quel est le client à l'origine de la déconnexion. Pour le retrouver, on utilise la même technique que pour le slot donneesRecues(), je ne la réexplique donc pas.

La méthode removeOne() de QList permet de supprimer le pointeur vers l'objet dans le tableau. Notre liste des clients est maintenant à jour.

Il ne reste plus qu'à finir de supprimer l'objet lui-même (nous venons seulement de supprimer le pointeur de la QList là). Pour supprimer l'objet, il faudrait faire un **delete** client;. Petit problème : si on supprime l'objet à l'origine du signal, on risque de faire bugger Qt. Heureusement tout a été prévu : on a juste à appeler deleteLater() (qui signifie "supprimer plus tard") et Qt se chargera de faire le delete lui-même un peu plus tard, lorsque notre slot aura fini de s'exécuter.

Méthode envoyerATous()

Ah, cette fois ce n'est pas un slot. 😊

C'est juste une méthode que j'ai décidé d'écrire dans la classe pour bien séparer le code, et aussi parce qu'on en a besoin plusieurs fois (vous avez remarqué que j'ai appelé cette méthode plusieurs fois dans les codes précédents non ?).

Dans le slot donneesRecues, nous recevions un message. Là, nous voulons au contraire en envoyer un, et ce à tous les clients connectés (tous les clients présents dans la QList).

Code : C++ - [Sélectionner](#)

```

void FenServeur::envoyerATous(const QString &message)
{
    // Préparation du paquet
    QByteArray paquet;
    QDataStream out(&paquet, QIODevice::WriteOnly);

    out << (quint16) 0; // On écrit 0 au début du paquet pour réservé la place pour écrire
    out << message; // On ajoute le message à la suite
    out.device()->seek(0); // On se replace au début du paquet
    out << (quint16) (paquet.size() - sizeof(quint16)); // On écrase le 0 qu'on avait réservé

    // Envoi du paquet préparé à tous les clients connectés au serveur
    for (int i = 0; i < clients.size(); i++)
    {
        clients[i]->write(paquet);
    }

```

3

4

5

6

Quelques explications bien sûr. 😊

On crée un QByteArray "paquet" qui va contenir le paquet à envoyer sur le réseau. La classe QByteArray représente une suite d'octets quelconque.

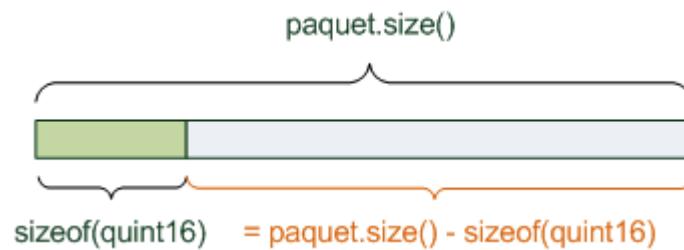
On utilise un QDataStream comme tout à l'heure pour écrire dans le QByteArray facilement. Cela va nous permettre d'utiliser

l'opérateur "<<".

Ce qui est particulier, c'est qu'on écrit d'abord le message (QString) et ensuite on calcule sa taille qu'on écrit au début du message.

Voilà ce qu'on fait sur le paquet dans l'ordre :

1. On écrit le nombre 0 de type quint16 pour "réserver" de la place.
2. On écrit à la suite le message, de type QString. Le message a été reçu en paramètre de la méthode envoyerATous().
3. On se replace au début du paquet (comme si on remettait le curseur au début d'un texte dans un traitement de texte).
4. On écrase le 0 qu'on avait écrit pour réservé de la place par la bonne taille du message. Cette taille est calculée via une simple soustraction : la taille du message est égale à la taille du paquet moins la taille réservée pour le quint16.



Notre paquet est prêt. Nous allons l'envoyer à tous les clients grâce à la méthode write() du socket. Pour cela, on fait une boucle sur la QList, et on envoie le message à chaque client.

Et voilà, le message est parti ! 😊

FenServeur.cpp en entier

Voici le contenu du fichier FenServeur.cpp que je viens de décortiquer en entier :

Code : C++ - [Sélectionner](#)

```

#include "FenServeur.h"

FenServeur::FenServeur()
{
    // Création et disposition des widgets de la fenêtre
    etatServeur = new QLabel;
    boutonQuitter = new QPushButton(tr("Quitter"));
    connect(boutonQuitter, SIGNAL(clicked()), qApp, SLOT(quit()));

    QVBoxLayout *layout = new QVBoxLayout;
    layout->addWidget(etatServeur);
    layout->addWidget(boutonQuitter);
    setLayout(layout);

    setWindowTitle(tr("ZeroChat - Serveur"));

    // Gestion du serveur
    serveur = new QTcpServer(this);
    if (!serveur->listen(QHostAddress::Any, 50885)) // Démarrage du serveur sur toutes les interfaces
    {
        // Si le serveur n'a pas été démarré correctement
        etatServeur->setText(tr("Le serveur n'a pas pu être démarré. Raison :<br />") +
    }
    else
    {
        // Si le serveur a été démarré correctement
        etatServeur->setText(tr("Le serveur a été démarré sur le port <strong>") + QString::number(serveur->serverPort()) + "</strong>");
        connect(serveur, SIGNAL(newConnection()), this, SLOT(nouvelleConnexion()));
    }

    tailleMessage = 0;
}

void FenServeur::nouvelleConnexion()
{
    envoyerATous(tr("<em>Un nouveau client vient de se connecter</em>"));

    QTcpSocket *nouveauClient = serveur->nextPendingConnection();
    clients << nouveauClient;

    connect(nouveauClient, SIGNAL(readyRead()), this, SLOT(donneesRecues()));
    connect(nouveauClient, SIGNAL(disconnected()), this, SLOT(deconnexionClient()));
}

void FenServeur::donneesRecues()
{
    // 1 : on reçoit un paquet (ou un sous-paquet) d'un des clients

    // On détermine quel client envoie le message (recherche du QTcpSocket du client)
    QTcpSocket *socket = qobject_cast<QTcpSocket *>(sender());
    if (socket == 0) // Si par hasard on n'a pas trouvé le client à l'origine du signal
        return;

    // Si tout va bien, on continue : on récupère le message
    QDataStream in(socket);

    if (tailleMessage == 0) // Si on ne connaît pas encore la taille du message, on essaie de la lire
    {
        if (socket->bytesAvailable() < (int)sizeof(quint16)) // On n'a pas reçu la taille
            return;

        in >> tailleMessage; // Si on a reçu la taille du message en entier, on la récupère
    }

    // Si on connaît la taille du message, on vérifie si on a reçu le message en entier
    if (socket->bytesAvailable() < tailleMessage) // Si on n'a pas encore tout reçu, on retourne
        return;
}

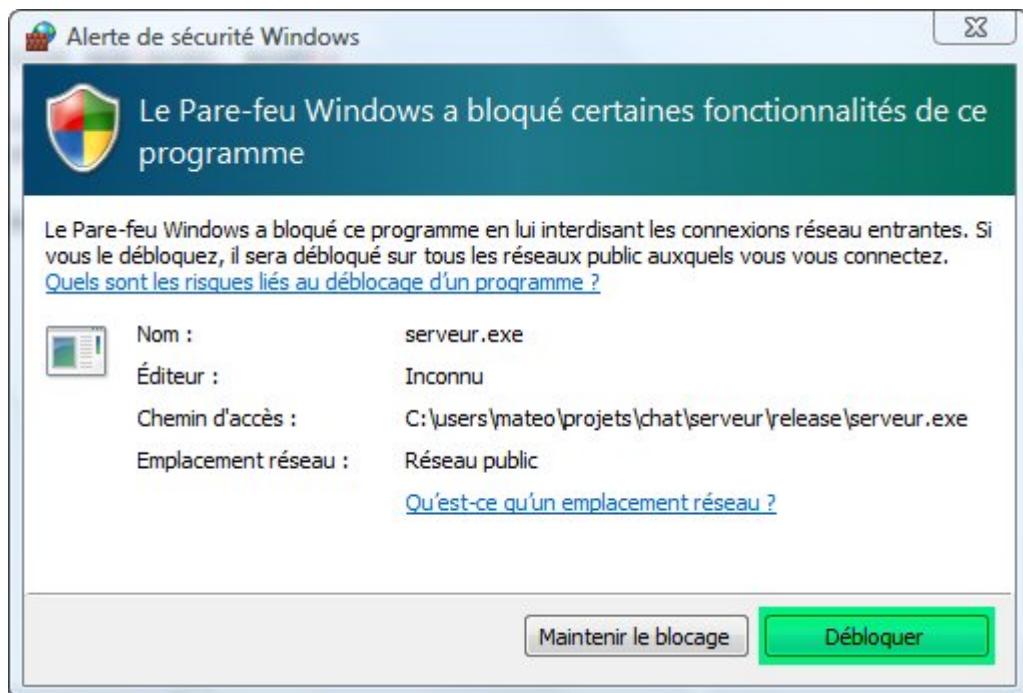
```

Lancement du serveur

Bonne nouvelle, devinez quoi : notre projet "serveur" est terminé !

Nous avons fait le plus dur, l'implémentation du serveur dans FenServeur.cpp. Compilez, et lancez le serveur ainsi créé.

Vous risquez d'avoir une alerte de votre pare-feu (firewall). Par exemple sous Windows :

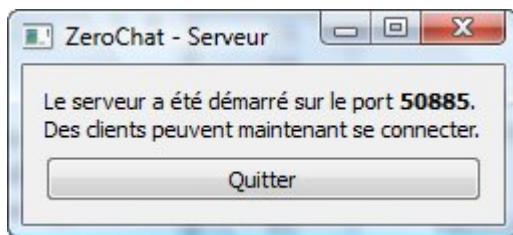


En effet, notre programme va communiquer sur le réseau. Le pare-feu nous demande si nous voulons autoriser notre programme à le faire : répondez oui en cliquant sur "Débloquer".

Dans cet exemple, j'ai considéré que le pare-feu de votre système d'exploitation était votre seul pare-feu. Si vous comptez utiliser le Chat sur internet et que vous êtes derrière un routeur, vérifiez la configuration du routeur et ouvrez le port 50885 pour les données TCP.

Si le Chat n'a pas l'air de fonctionner, c'est très probablement à cause d'un pare-feu quelque part qui bloque le port.

Notre serveur est maintenant lancé :



Bravo ! 😊

Laissez ce programme tourner en fond sur votre ordinateur (vous pouvez réduire la fenêtre). Il va servir à faire la communication entre les différents clients.

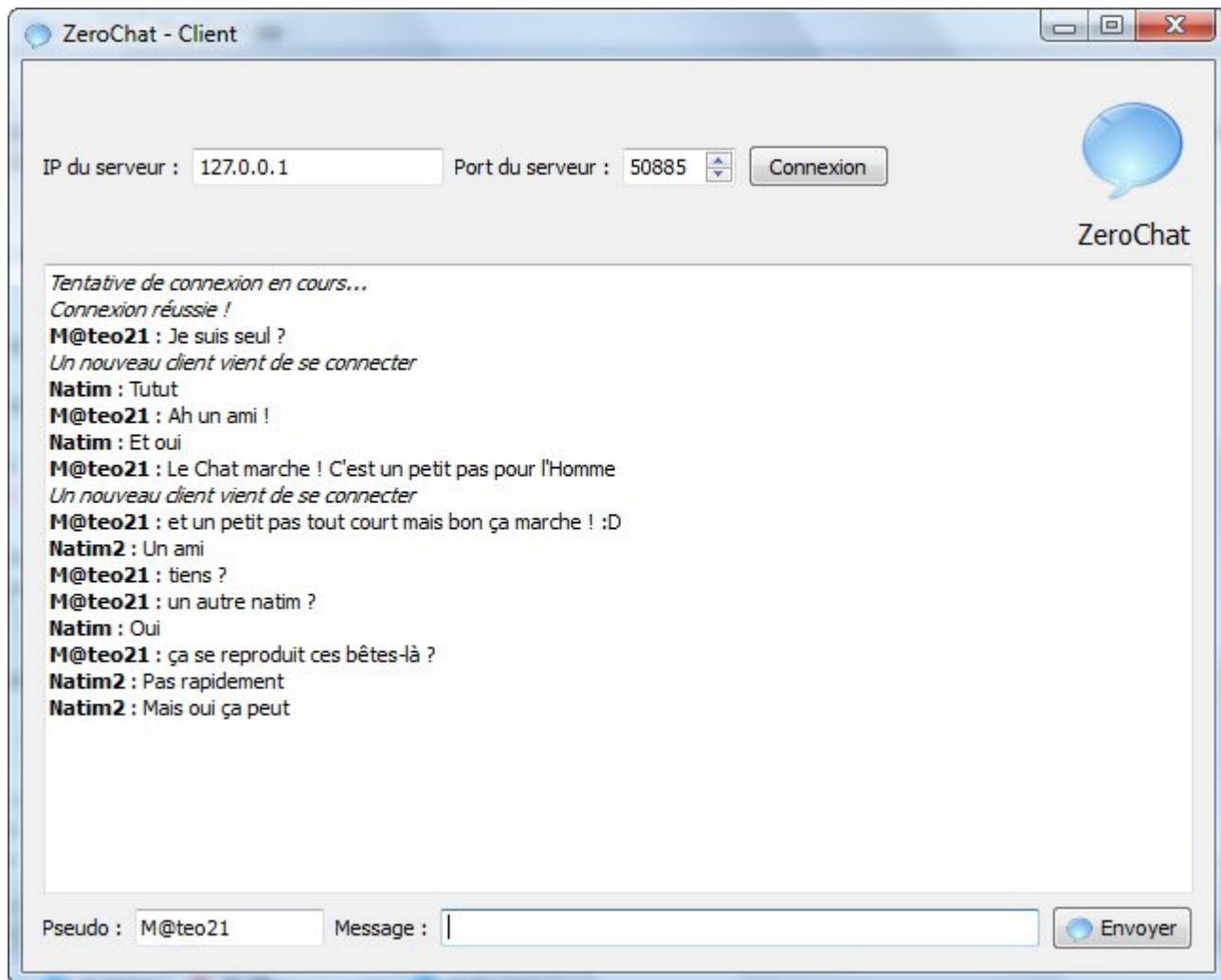
Bon, mauvaise nouvelle chers auditeurs : nous avons beaucoup sué, mais nous avons fait seulement 50% du travail ! Il faut maintenant s'attaquer au projet "client" pour réaliser le programme qui sera utilisé par tous les clients pour chatter. Heureusement, nous avons déjà fait le plus dur en analysant le slot donnéesRecues, on devrait donc aller un peu plus vite. 😊

Réalisation du client

Si la fenêtre du serveur était toute simple, il en va autrement pour la fenêtre du client.

En effet, autant créer une fenêtre pour le serveur était facultatif, autant pour le client il faut bien qu'il ait une fenêtre pour écrire ses messages. 😊

Voici la fenêtre de client que l'on veut coder :



Nous aurons 3 fichiers à nouveau :

- main.cpp
- FenClient.h
- FenClient.cpp

Dessin de la fenêtre avec Qt Designer

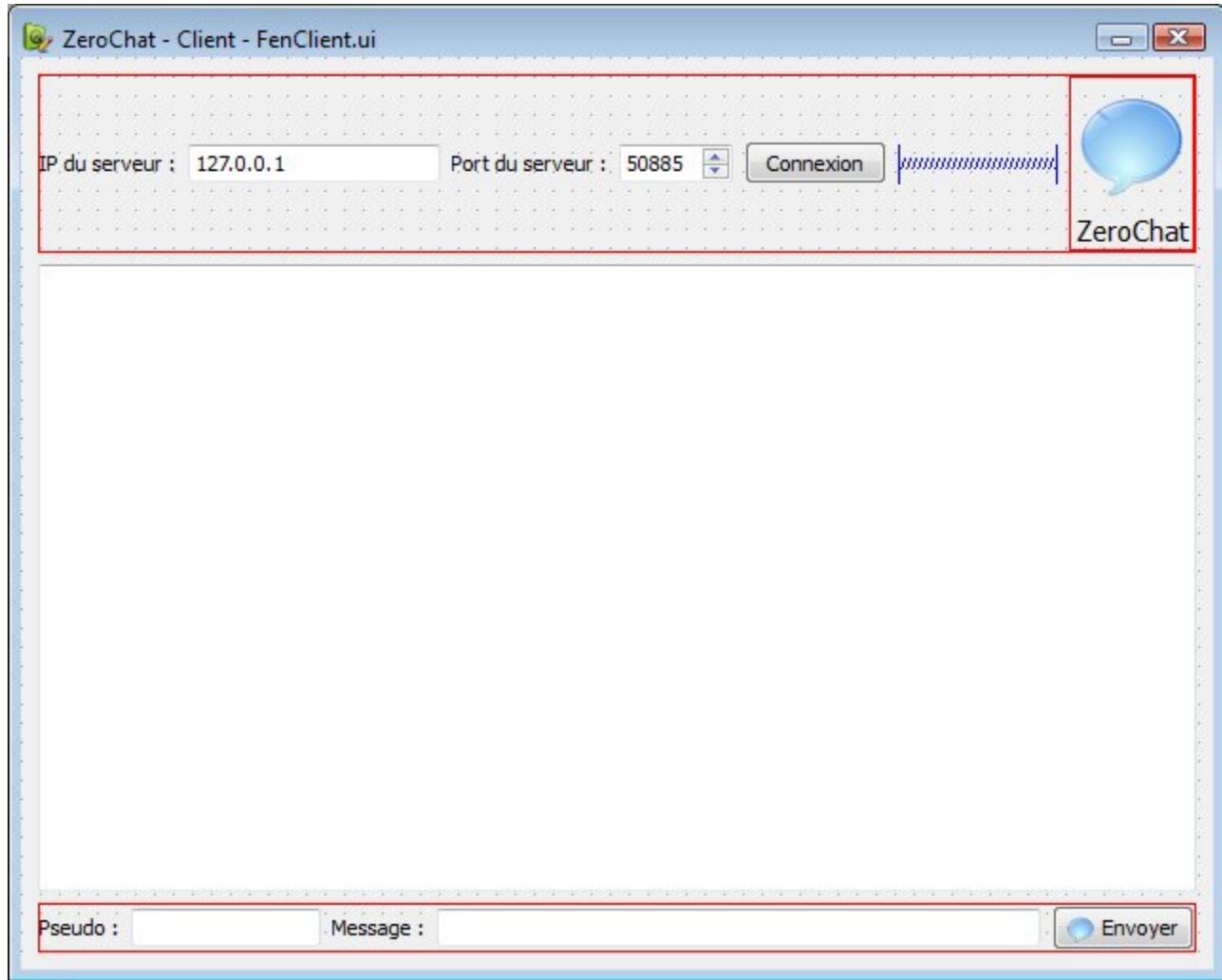
Bon, la réalisation de cette fenêtre ne nous intéresse pas vraiment. C'est une fenêtre tout ce qu'il y a de plus classique.

Pour gagner du temps, je vous propose de créer l'interface de la fenêtre du client via Qt Designer. Ce programme a justement été conçu pour gagner du temps pour ceux qui savent déjà coder la fenêtre à la main, ce qui est notre cas.

J'aurais pu utiliser Qt Designer pour le TP zNavigo aussi, mais la fenêtre était beaucoup plus complexe et le nombre d'onglets variable. J'avais donc décidé de l'écrire à la main.

Pour une fenêtre assez simple comme celle-là, l'utilisation de Qt Designer permet au contraire de gagner du temps, pour peu qu'on sache déjà coder la fenêtre à la main.

J'ouvre donc Qt Designer et je dessine la fenêtre suivante :



Je prends soin à bien donner un nom correct à chacun des widgets via la propriété `objectName` (sauf pour les `QLabel`, car on n'aura pas à les réutiliser ceux-là donc on s'en moque un peu 😊).

Je veille aussi à utiliser des layouts partout sur ma fenêtre pour la rendre redimensionnable sans problème. Je sélectionne plusieurs widgets à la fois et je clique sur un des boutons de la barre d'outils en haut pour les assembler selon un layout (notez que je n'utilise jamais les layouts de la widget box à gauche).

Pour vous faire gagner du temps si vous ne voulez pas redessiner la fenêtre sous Qt Designer, voici le fichier `FenClient.ui` que j'ai généré :

[Télécharger FenClient.ui](#)
(faites clic droit / enregistrer sous)

Client.pro

J'ai mis à jour le fichier `.pro` pour indiquer qu'il y avait un UI dans le projet et qu'on utilisait le module `network` de Qt. Vérifiez donc que votre `.pro` ressemble au mien :

Code : Autre - [Sélectionner](#)

```
#####
# Automatically generated by qmake (2.01a) mer. 25. juin 17:13:47 2008
#####

TEMPLATE = app
QT += network
TARGET =
DEPENDPATH += .
INCLUDEPATH += .

# Input
HEADERS += FenClient.h
FORMS += FenClient.ui
SOURCES += FenClient.cpp main.cpp
```

main.cpp

Revenons au code. Le main est toujours très simple et sans originalité : il ouvre la fenêtre.

Code : C++ - [Sélectionner](#)

```
#include <QApplication>
#include "FenClient.h"

int main(int argc, char* argv[])
{
    QApplication app(argc, argv);

    FenClient fenetre;
    fenetre.show();

    return app.exec();
}
```

FenClient.h

Notre fenêtre utilise un fichier généré avec Qt Designer. Direction le chapitre sur Qt Designer si vous avez oublié comment se servir d'une fenêtre générée dans son code.

Nous allons ici utiliser un héritage multiple afin de simplifier au maximum le code par la suite :

Code : C++ - [Sélectionner](#)

```

#ifndef HEADER_FENCLIENT
#define HEADER_FENCLIENT

#include <QtGui>
#include <QtNetwork>
#include "ui_FenClient.h"

class FenClient : public QWidget, private Ui::FenClient
{
    Q_OBJECT

public:
    FenClient();

private slots:
    void on_boutonConnexion_clicked();
    void on_boutonEnvoyer_clicked();
    void on_message_returnPressed();
    void donneesRecues();
    void connecte();
    void deconnecte();
    void erreurSocket(QAbstractSocket::SocketError erreur);

private:
    QTcpSocket *socket; // Représente le serveur
    quint16 tailleMessage;
};

#endif

```

Notez qu'on inclut ui_FenClient.h pour réutiliser la fenêtre générée.

Notre fenêtre comporte pas mal de slots qu'il va falloir implémenter, heureusement ils seront assez simples. On utilise les autoconnect pour les 3 premiers d'entre eux pour gérer les évènements de la fenêtre :

- on_boutonConnexion_clicked() : appelé lorsqu'on clique sur le bouton "Connexion" et qu'on souhaite donc se connecter au serveur.
- on_boutonEnvoyer_clicked() : appelé lorsqu'on clique sur "Envoyer" pour envoyer un message dans le Chat.
- on_message_returnPressed() : appelé lorsqu'on appuie sur la touche "Entrée" lorsqu'on rédige un message. Comme cela revient au même que on_boutonEnvoyer_clicked(), on appellera cette méthode directement pour éviter d'avoir à écrire 2 fois le même code.
- donneesRecues() : appelé lorsqu'on reçoit un sous-paquet du serveur. Ce slot sera très similaire à celui du serveur qui possède le même nom.
- connecte() : appelé lorsqu'on vient de réussir à se connecter au serveur.
- deconnecte() : appelé lorsqu'on vient de se déconnecter du serveur.
- erreurSocket(QAbstractSocket::SocketError erreur) : appelé lorsqu'il y a eu une erreur sur le réseau (connexion au serveur impossible par exemple).

En plus de ça, on a 2 attributs à manipuler :

- QTcpSocket *socket : une socket qui représentera la connexion au serveur. On utilisera cette socket pour envoyer des paquets au serveur, par exemple lorsque l'utilisateur veut envoyer un message.
- quint16 tailleMessage : permet à l'objet de se "souvenir" de la taille du message qu'il est en train de recevoir dans son slot donneesRecues(). Il a la même utilité que sur le serveur.

FenClient.cpp

Maintenant implémentons tout ce beau monde !

Courage, après ça c'est fini vous allez pouvoir savourer votre Chat ! 😊

Le constructeur

Notre constructeur se doit d'appeler setupUi() dès le début pour mettre en place les widgets sur la fenêtre. C'est justement là qu'on gagne du temps grâce à Qt Designer : on n'a pas à coder le placement des widgets sur la fenêtre. 😊

Code : C++ - [Sélectionner](#)

```
FenClient::FenClient()
{
    setupUi(this);

    socket = new QTcpSocket(this);
    connect(socket, SIGNAL(readyRead()), this, SLOT(donneesRecues()));
    connect(socket, SIGNAL.connected(), this, SLOT(connecte()));
    connect(socket, SIGNAL(disconnected()), this, SLOT(deconnecte()));
    connect(socket, SIGNAL(error(QAbstractSocket::SocketError)), this, SLOT(erreurSocket

    tailleMessage = 0;
}
1
3
4
5
6
```

En plus de setupUi(), on fait quelques initialisations supplémentaires indispensables :

- On crée l'objet de type QTcpSocket qui va représenter la connexion au serveur.
- On connecte les signaux qu'il est susceptible d'envoyer à nos slots personnalisés.
- On met tailleMessage à 0 pour permettre la réception de nouveaux messages.

Notez qu'on ne se connecte pas au serveur dans le constructeur. On prépare juste la socket, mais on ne fera la connexion que lorsque le client aura cliqué sur le bouton "Connexion" (il faut bien lui laisser le temps de rentrer l'adresse IP du serveur !).

Slot on_boutonConnexion_clicked()

Ce slot se fait appeler dès que l'on a cliqué sur le bouton "Connexion" en haut de la fenêtre.

Code : C++ - [Sélectionner](#)

```
// Tentative de connexion au serveur
void FenClient::on_boutonConnexion_clicked()
{
    // On annonce sur la fenêtre qu'on est en train de se connecter
    listeMessages->append(tr("Tentative de connexion en cours..."));
    boutonConnexion->setEnabled(false);

    socket->abort(); // On désactive les connexions précédentes s'il y en a
    socket->connectToHost(serveurIP->text(), serveurPort->value()); // On se connecte au
}
1
3
4
5
6
```

1. Dans un premier temps, on affiche sur la zone de messages "listeMessages" au centre de la fenêtre que l'on est en train d'essayer de se connecter.
2. On désactive temporairement le bouton "Connexion" pour empêcher au client de retenter une connexion alors qu'une tentative de connexion est déjà en cours.
3. Si la socket est déjà connectée à un serveur, on coupe la connexion avec abort(). Si on n'était pas connecté, cela n'aura aucun effet, mais c'est par sécurité pour que l'on ne soit pas connecté à 2 serveurs à la fois.
4. On se connecte enfin au serveur avec la méthode connectToHost(). On utilise l'IP et le port demandés par l'utilisateur dans les champs de texte en haut de la fenêtre.

Slot on_boutonEnvoyer_clicked()

Ce slot est appelé lorsqu'on essaie d'envoyer un message (le bouton "Envoyer" en bas à droite a été cliqué).

Code : C++ - [Sélectionner](#)

```
// Envoi d'un message au serveur
void FenClient::on_boutonEnvoyer_clicked()
{
    QByteArray paquet;
    QDataStream out(&paquet, QIODevice::WriteOnly);

    // On prépare le paquet à envoyer
    QString messageAEnvoyer = tr("<strong>") + pseudo->text() + tr("</strong> : ") + message;
    out << (quint16) 0;
    out << messageAEnvoyer;
    out.device()->seek(0);
    out << (quint16) (paquet.size() - sizeof(quint16));

    socket->write(paquet); // On envoie le paquet

    message->clear(); // On vide la zone d'écriture du message
    message->setFocus(); // Et on remet le curseur à l'intérieur
```

3

5

6

4

Ce code est similaire à celui de la méthode `envoyerATous()` du serveur. Il s'agit d'un envoi de données sur le réseau.

1. On prépare un `QByteArray` dans lequel on va écrire le paquet qu'on veut envoyer.
2. On construit ensuite la `QString` contenant le message à envoyer. Vous noterez qu'on met le nom de l'auteur et son texte directement dans la même `QString`. Idéalement, il vaudrait mieux séparer les deux pour avoir un code plus logique et plus modulable, mais cela aurait compliqué le code de ce chapitre bien délicat, donc ça sera dans les améliorations à faire à la fin. 😊
3. On calcule la taille du message.
4. On envoie le paquet ainsi créé au serveur en utilisant la socket qui le représente et sa méthode `write()`.
5. On efface automatiquement la zone d'écriture des messages en bas pour qu'on puisse en écrire un nouveau et on donne le focus à cette zone immédiatement pour que le curseur soit placé dans le bon widget.

Slot `on_message_returnPressed()`

Ce slot est appelé lorsqu'on a appuyé sur la touche "Entrée" après avoir rédigé un message.

Cela a le même effet qu'un clic sur le bouton "Envoyer", nous appelons donc le slot que nous venons d'écrire :

Code : C++ - [Sélectionner](#)

```
// Appuyer sur la touche Entrée a le même effet que cliquer sur le bouton "Envoyer"
void FenClient::on_message_returnPressed()
{
    on_boutonEnvoyer_clicked();
}
```

Slot `donneesRecues()`

Voilà à nouveau le slot de vos pires cauchemars. 😱

Il est quasiment identique à celui du serveur (la réception de données fonctionne de la même manière) je ne le réexplique donc pas :

Code : C++ - [Sélectionner](#)

```

// On a reçu un paquet (ou un sous-paquet)
void FenClient::donneesRecues()
{
    /* Même principe que lorsque le serveur reçoit un paquet :
    On essaie de récupérer la taille du message
    Une fois qu'on l'a, on attend d'avoir reçu le message entier (en se basant sur la ta-
    */
    QDataStream in(socket);

    if (tailleMessage == 0)
    {
        if (socket->bytesAvailable() < (int)sizeof(quint16))
            return;

        in >> tailleMessage;
    }

    if (socket->bytesAvailable() < tailleMessage)
        return;

    // Si on arrive jusqu'à cette ligne, on peut récupérer le message entier
    QString messageReçu;
    in >> messageReçu;

    // On affiche le message sur la zone de Chat
    listeMessages->append(messageReçu);

    // On remet la taille du message à 0 pour pouvoir recevoir de futurs messages
    tailleMessage = 0;

```

3

6

La seule différence ici en fait, c'est qu'on affiche le message reçu dans la zone de Chat à la fin : `listeMessages->append(messageReçu);`

Slot connecte()

Ce slot est appelé lorsqu'on a réussi à se connecter au serveur.

Code : C++ - [Sélectionner](#)

```

// Ce slot est appelé lorsque la connexion au serveur a réussi
void FenClient::connecte()
{
    listeMessages->append(tr(" <em>Connexion réussie !</em> "));
    boutonConnexion->setEnabled(true);
}

```

Tout bêtement, on se contente d'afficher "Connexion réussie" dans la zone de Chat pour que le client sache qu'il est bien connecté au serveur.

On réactive aussi le bouton "Connexion" qu'on avait désactivé, pour permettre une nouvelle connexion à un autre serveur.

Slot deconnecte()

Ce slot est appelé lorsqu'on est déconnecté du serveur.

Code : C++ - [Sélectionner](#)

```
// Ce slot est appelé lorsqu'on est déconnecté du serveur
void FenClient::deconnecte()
{
    listeMessages->append(tr("Déconnecté du serveur"));
}
```

On affiche juste un message sur la zone de texte pour que le client soit au courant.

Slot erreurSocket()

Ce slot est appelé lorsque la socket a rencontré une erreur.

Code : C++ - [Sélectionner](#)

```
// Ce slot est appelé lorsqu'il y a une erreur
void FenClient::erreurSocket(QAbstractSocket::SocketError erreur)
{
    switch(erreur) // On affiche un message différent selon l'erreur qu'on nous indique
    {
        case QAbstractSocket::HostNotFoundError:
            listeMessages->append(tr("ERREUR : le serveur n'a pas pu être trouvé. Vérifiez les informations d'entrée."));
            break;
        case QAbstractSocket::ConnectionRefusedError:
            listeMessages->append(tr("ERREUR : le serveur a refusé la connexion. Vérifiez les informations d'entrée."));
            break;
        case QAbstractSocket::RemoteHostClosedError:
            listeMessages->append(tr("ERREUR : le serveur a coupé la connexion."));
            break;
        default:
            listeMessages->append(tr("ERREUR : ") + socket->errorString() + tr("</em>"));
    }

    boutonConnexion->setEnabled(true);
}
```

La raison de l'erreur est passée en paramètre. Elle est de type [QAbstractSocket::SocketError](#) (c'est une énumération).

On fait un switch pour afficher un message différent en fonction de l'erreur. Je n'ai pas traité toutes les erreurs possibles, lisez la doc pour connaître les autres raisons d'erreurs que l'on peut gérer.

La plupart des erreurs que je gère ici sont liées à la connexion au serveur. J'affiche un message intelligible en français pour que l'on comprenne la raison de l'erreur.

Le cas "default" est appelé pour les erreurs que je n'ai pas gérées. J'affiche le message d'erreur envoyé par la socket (qui sera peut-être en anglais mais bon c'est mieux que rien).

FenClient.cpp en entier

C'est fini ! 😊

Bon, ça n'a pas été trop long ni trop difficile après avoir fait le serveur, avouez. 😊

Voici le code complet de FenClient.cpp :

Code : C++ - [Sélectionner](#)

```

#include "FenClient.h"

FenClient::FenClient()
{
    setupUi(this);

    socket = new QTcpSocket(this);
    connect(socket, SIGNAL(readyRead()), this, SLOT(donneesRecues()));
    connect(socket, SIGNAL.connected(), this, SLOT(connecte()));
    connect(socket, SIGNAL(disconnected()), this, SLOT(deconnecte()));
    connect(socket, SIGNAL(error(QAbstractSocket::SocketError)), this, SLOT(erreurSocket()));

    tailleMessage = 0;
}

// Tentative de connexion au serveur
void FenClient::on_boutonConnexion_clicked()
{
    // On annonce sur la fenêtre qu'on est en train de se connecter
    listeMessages->append(tr("Tentative de connexion en cours..."));
    boutonConnexion->setEnabled(false);

    socket->abort(); // On désactive les connexions précédentes s'il y en a
    socket->connectToHost(serveurIP->text(), serveurPort->value()); // On se connecte au serveur
}

// Envoi d'un message au serveur
void FenClient::on_boutonEnvoyer_clicked()
{
    QByteArray paquet;
    QDataStream out(&paquet, QIODevice::WriteOnly);

    // On prépare le paquet à envoyer
    QString messageAEnvoyer = tr("<strong>") + pseudo->text() + tr("</strong> : ") + message->text();
    out << (quint16) 0;
    out << messageAEnvoyer;
    out.device()->seek(0);
    out << (quint16) (paquet.size() - sizeof(quint16));

    socket->write(paquet); // On envoie le paquet

    message->clear(); // On vide la zone d'écriture du message
    message->setFocus(); // Et on remet le curseur à l'intérieur
}

// Appuyer sur la touche Entrée a le même effet que cliquer sur le bouton "Envoyer"
void FenClient::on_message_returnPressed()
{
    on_boutonEnvoyer_clicked();
}

// On a reçu un paquet (ou un sous-paquet)
void FenClient::donneesRecues()
{
    * Même principe que lorsque le serveur reçoit un paquet :
    On essaie de récupérer la taille du message
    Une fois qu'on l'a, on attend d'avoir reçu le message entier (en se basant sur la taille)
    */
    QDataStream in(socket);

    if (tailleMessage == 0)
    {
        if (socket->bytesAvailable() < (int)sizeof(quint16))
            return;

        in >> tailleMessage;
    }
}

```

Test du Chat et améliorations

Notre projet Nos projets sont terminés !
Nous avons fait le client et le serveur !

Je vous propose de tester le bon fonctionnement du Chat dans un premier temps, et éventuellement de télécharger les projets tous prêts.

Nous verrons ensuite comment vous pouvez améliorer tout cela. 😊

Tester le Chat

Avant toute chose, vous voudrez peut-être récupérer le projet tout prêt et zippé pour partir sur la même base que moi.

[Télécharger Chat.zip \(60 Ko\)](#)

Le zip contient un sous-dossier par projet : serveur et client.

Vous pouvez exécuter directement les programmes serveur.exe et client.exe si vous êtes sous Windows (en n'oubliant pas de mettre les DLL de Qt dans le même répertoire). Si vous utilisez un autre OS, vous devrez recompiler le projet (faites un qmake et un make).

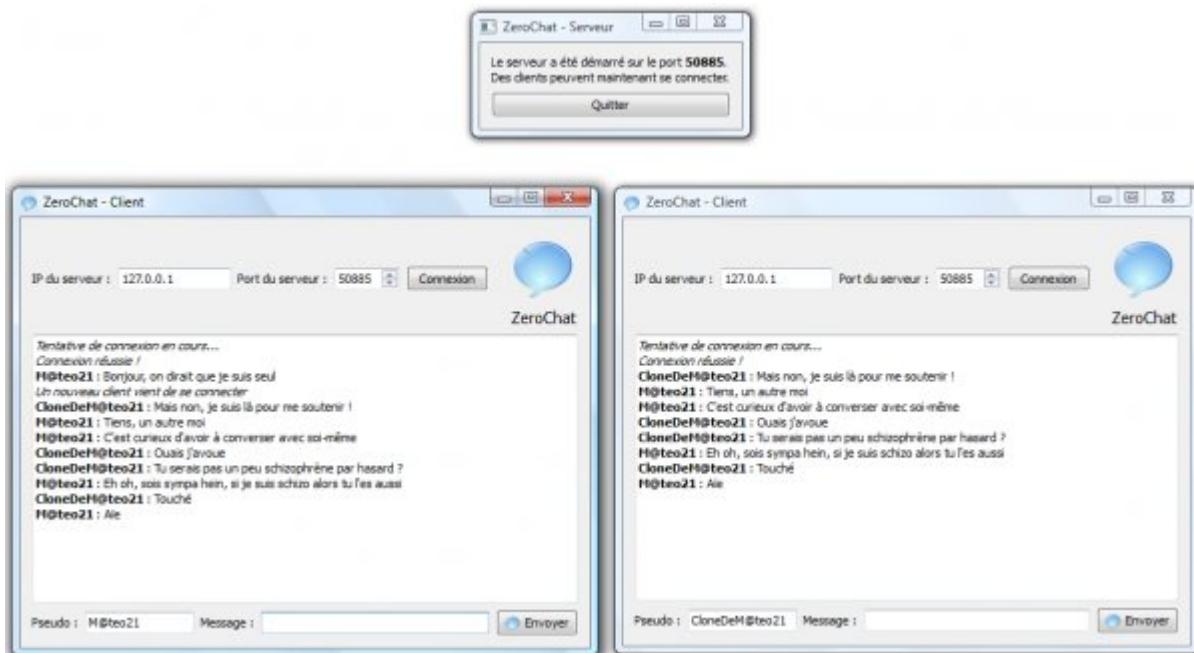
Vous pouvez dans un premier temps tester le Chat en interne sur votre propre ordinateur. Je vous propose de lancer :

- Un serveur
- Deux clients

Cela va nous permettre de simuler une conversation en interne sur notre ordinateur. Cela utilisera le réseau, mais à l'intérieur de votre propre machine. 😊

J'avoue que c'est un peu curieux, mais si ça fonctionne en interne, ça fonctionnera en réseau local et sur internet sans problème (pour peu que le port soit ouvert). C'est donc une bonne idée de faire ses tests en interne dans un premier temps.

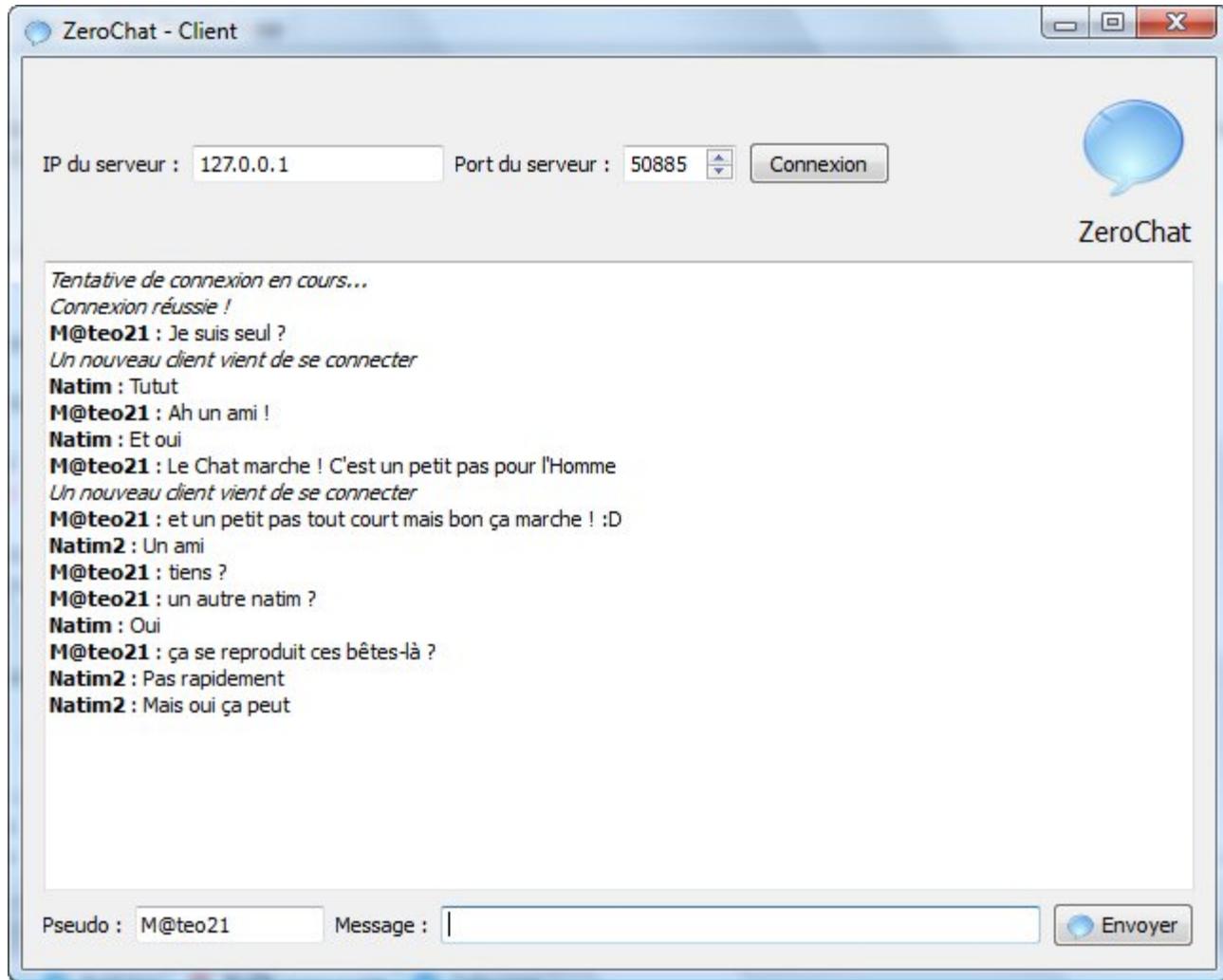
Voici ce que ça donne quand je me parle à moi-même :



Comme vous pouvez le voir, tout fonctionne (sauf peut-être mon cerveau 😊).

Amusez-vous ensuite à tester le programme en réseau local ou sur internet avec des amis. Pensez à chaque fois à vérifier si le port est ouvert. Si vous avez un problème avec le programme, il y a 99% de chances que ça vienne du port.

Voici une petite conversation en réseau local :



Je ne l'ai pas testé sur internet mais je sais pertinemment que ça fonctionne. Le principe du réseau est le même partout, que ce soit en interne, en local ou via internet. C'est juste l'IP qui change à chaque fois.

Améliorations à réaliser

Bon, le moins qu'on puisse dire c'est que j'ai bien travaillé dans ce chapitre, maintenant à votre tour de bosser un peu. 😊 Voici quelques suggestions d'améliorations que vous pouvez réaliser sur le Chat, plus ou moins difficiles selon le cas :

- Vous pouvez griser la zone d'envoi des messages ainsi que le bouton "Envoyer" lorsque le client n'est pas connecté.
- Sur la fenêtre du serveur, il devrait être assez facile d'afficher le nombre de clients qui sont connectés.
- Plus délicat car ça demande un peu de réorganisation du code : au lieu d'avoir une QList de QTcpSocket, faites une QList d'objets de type Client.
Il faudra créer une nouvelle classe Client qui va représenter un client. Elle aura des attributs comme : sa QTcpSocket, le pseudo (QString), pourquoi pas l'avatar (QPixmap), etc. A partir de là vous aurez alors beaucoup plus de souplesse dans votre Chat !
- Vous pourriez alors facilement afficher le pseudo du membre qui vient se connecter. Pour le moment, on a juste "Un client vient de se connecter".
- Plutôt graphique mais sympa : vous pourriez gérer la mise en forme des messages (gras, rouge...) ainsi que des smilies. Bon après il s'agit pas de recréer MSN (quoique, le principe est tout à fait le même 😊) donc n'allez pas trop loin dans ce genre

de fonctionnalités quand même.

- Plus délicat, mais très intéressant : essayez d'afficher la liste des clients connectés sur la fenêtre des clients. Vous devriez rajouter un widget QListView pour afficher cette liste, ça vous ferait travailler MVC en plus. 😊
Le plus délicat est de gérer la liste des connectés, car pour le moment le pseudo est directement intégré aux messages qui sont envoyés. Il faudrait essayer de gérer le contenu des paquets un peu différemment, à vous de voir.
- Actuellement, le serveur est un peu minimal et ne gère pas tous les cas. Par exemple, si 2 clients envoient un message en même temps, il n'y a qu'une seule variable tailleMessage pour 2 messages en cours de réception. Je vous recommande de gérer plutôt 1 tailleMessage par client (vous n'avez qu'à mettre tailleMessage dans la classe Client).

Je m'arrête là pour les suggestions, il y a déjà du travail !

On pourrait aussi imaginer de permettre un Chat en privé entre certains clients, ou encore d'autoriser l'envoi de fichier sur le réseau (le tout étant de récupérer le fichier à envoyer sous forme de QByteArray).

Enfin, n'oubliez pas que le réseau ne se limite pas au Chat. Si vous faites un jeu en réseau par exemple, il faudra non pas envoyer des messages texte, mais plutôt les actions des autres joueurs. Dans ce cas, le schéma des paquets envoyés deviendra un peu plus complexe, mais c'est nécessaire.

A vous d'adapter un peu mon code, vous êtes grands maintenant, au boulot ! 😊

Bon, je crois que c'était mon plus gros chapitre. Il était temps que je m'arrête. 😊

J'espère que vous avez apprécié cette partie sur Qt, nous avons vu beaucoup de choses (un peu trop même) et pourtant nous n'avons pas tout vu. 😊

Je vous laisse vous entraîner avec le réseau, les widgets, et pour tout le reste n'oubliez pas : il y a la doc ! Et les forums du Site du Zéro aussi, oui oui, c'est vrai.

Partie 3 : Annexes

Besoin d'aller encore plus loin ?

Lisez donc les annexes !

Ce que vous pouvez encore apprendre

Qu'on se le dise : bien que le tutoriel C++ s'arrête là, vous ne savez pas tout sur tout. D'ailleurs, personne ne peut vraiment prétendre tout savoir sur le C++ et toutes ses bibliothèques.

L'objectif n'est pas de tout savoir, mais d'être capable d'apprendre ce dont vous avez besoin lorsque c'est nécessaire.

Si je devais moi-même vous apprendre tout sur le C++, j'y passerais toute une vie (et encore, ça serait toujours incomplet). J'ai autre chose à faire, et j'en serais de toute façon incapable.

Du coup, plutôt que de tout vous apprendre, j'ai choisi de vous enseigner de bonnes bases tout au long du cours. Cette annexe a pour but, maintenant que le cours est fini, de vous donner un certain nombre de pistes pour continuer votre apprentissage. 😊

J'ai découpé ce chapitre en 3 parties :

- Ce que vous pouvez encore apprendre sur le langage C++ lui-même.
- Ce que vous pouvez encore apprendre sur la bibliothèque standard du C++, que nous avons seulement effleuré ici.
- Ce que vous pouvez encore apprendre sur la bibliothèque Qt.

Cette annexe est seulement là pour vous présenter de nouvelles notions, pas pour vous les expliquer. Ne soyez donc pas surpris si je suis beaucoup plus succinct que d'habitude. Imaginez cette annexe comme un sommaire de ce qu'il vous reste à apprendre. 😊

... sur le langage C++

Le langage C++ est suffisamment riche pour qu'il vous reste encore de nombreuses notions à découvrir. Certaines d'entre elles sont particulièrement complexes, je ne vous le cache pas, et vous n'en aurez pas besoin tout le temps.

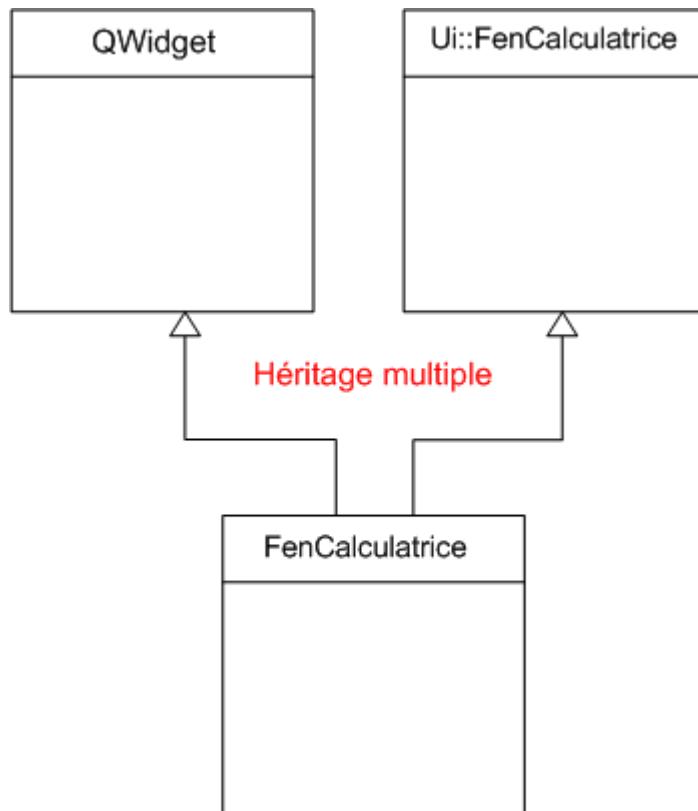
Toutefois, au cas où vous en ayez besoin un jour, je vais vous présenter rapidement ces notions. A vous ensuite d'approfondir vos connaissances, par exemple en lisant des [tutoriels écrits par d'autres Zéros sur le C++](#), en lisant des livres dédiés au C++, ou tout simplement en faisant une recherche Google. 😊

Voici les notions que je vais vous présenter ici :

- L'héritage multiple
- Le masquage de méthodes
- Les exceptions
- Les templates
- Les namespaces
- Les classes et fonctions amies
- Les méthodes virtuelles pures et les classes abstraites

L'héritage multiple

L'héritage multiple consiste à hériter de plusieurs classes à la fois. Nous avons déjà fait cela dans la partie sur Qt, pour pouvoir utiliser une interface dessinée dans Qt Designer :



Pour hériter de plusieurs classes, il suffit de mettre une virgule entre les noms de classe, comme on l'avait fait :

Code : C++ - [Sélectionner](#)

```
class FenCalculatrice : public QWidget, private Ui::FenCalculatrice
{
};
```

C'est une notion qui paraît simple mais qui, en réalité, est très complexe.

En fait, la plupart des langages de programmation plus récents, comme Java et Ruby, ont carrément décidé de ne pas gérer l'héritage multiple. Pourquoi ? Parce que ça peut être utile dans certaines conditions assez rares, mais si on l'utilise mal (quand on débute) ça peut devenir un cauchemar à gérer.

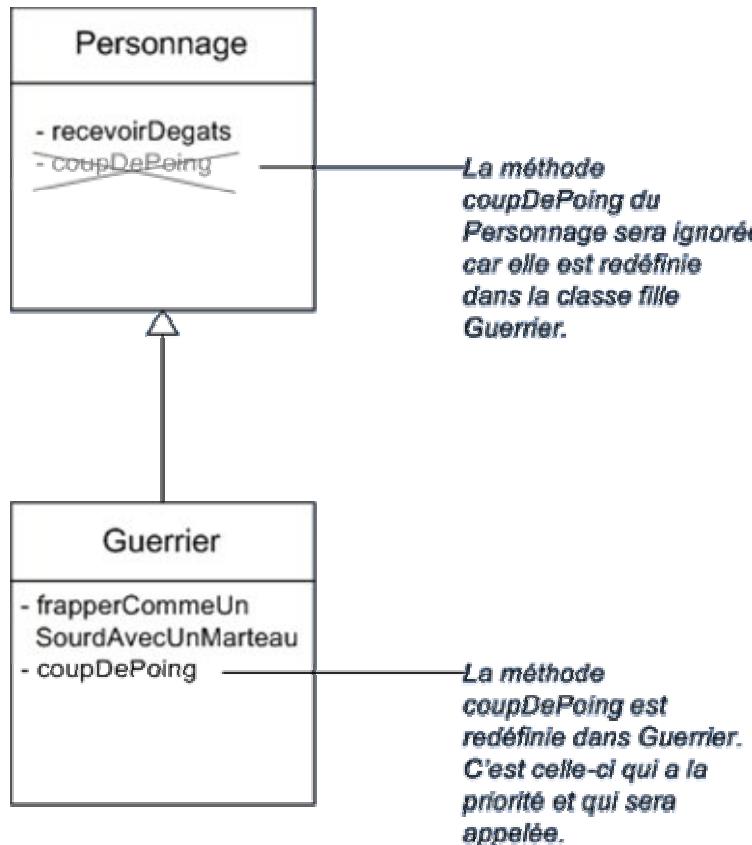
Bref, jetez un coup d'oeil à cette notion, mais juste un coup d'oeil de préférence, car vous ne devriez pas y avoir recours souvent.



Le masquage de méthodes

Lorsqu'une classe hérite d'une autre classe, il est possible d'avoir 2 méthodes du même nom (et du même prototype).

Prenons un exemple :



Ici, la classe Guerrier qui hérite de Personnage possède une méthode qui s'appelle elle aussi coupDePoing. Si notre guerrier fait un coupDePoing, c'est la méthode définie dans Guerrier qui sera utilisée, celle de Personnage sera ignorée (masquée).

Nanoc en parle plus en détail dans son [tutoriel sur l'héritage](#). N'hésitez pas à aller y jeter un oeil. 😊

Les exceptions

Les exceptions sont un mécanisme du C++ qui permet de gérer les erreurs.

En temps normal, pour vérifier qu'une fonction n'a pas eu d'erreur, celle-ci renvoie un booléen. Il nous suffit alors de tester la valeur de retour comme ceci :

Code : C++ - Sélectionner

```
if (maFonction())
{
    cout << "Ca a marché !";
}
else
{
    cout << "Erreur lors de l'appel de maFonction()";
}
```

Ca marche, mais ce n'est pas très pratique dans un gros programme. En effet, parfois on a besoin de faire "remonter" l'erreur à la fonction appelante, parce qu'on ne peut pas traiter l'erreur de suite. Ici, on mélange le code "utile" du programme avec la gestion des erreurs.

Un des gros intérêt des exceptions, c'est qu'elles vous permettent de regrouper vos erreurs. Cela fonctionne en 2 temps :

- On essaie (try) d'exécuter certaines fonctions.
- Si une de ces fonctions a généré une erreur, on la récupère (catch) et on gère l'erreur dans un endroit bien défini du code.

Exemple :

Code : C++ - Sélectionner

```
try // On essaie d'exécuter les lignes de codes qui suivent
{
    maFonction();
    autreFonction();
    encoreUneAutreFonction();
}
catch (const string& erreur) // Si une erreur est survenue dans les lignes précédentes, c
{
    cout << "Erreur ! Raison : " << erreur;
}
```

Il faut que les fonctions maFonction(), autreFonction() et encoreUneAutreFonction() envoient un message d'erreur à l'aide du mot-clé throw si jamais elles rencontrent une erreur. Dans ce cas, si une erreur est détectée, les instructions situées dans le bloc catch seront exécutées.

Pour plus d'informations sur les exceptions, je vous recommande la lecture du [tutoriel de Nanoc](#).

Pour information, les exceptions ne sont pas bien utilisables avec la bibliothèque Qt, qui ne les prend pas correctement en charge. Vous ne pouvez donc pas vraiment utiliser d'exceptions dans un programme Qt.

Les templates

Les templates sont une notion pratique et puissante du C++ que nous avons déjà rencontrée, sans vraiment expliquer le fonctionnement derrière.

Imaginez que vous ayez besoin d'écrire une fonction (ou une classe) qui doit pouvoir accepter n'importe quel type de paramètre, et qui doit pouvoir retourner n'importe quel type. En fait, vous avez besoin d'écrire une fonction évolutive qui s'adapte à tous les types de données.

Voici par exemple une fonction d'addition qui utilise le principe des templates :

Code : C++ - Sélectionner

```
template<class T> T addition(T element1, T element2)
{
    T somme = element1 + element2;
    return somme;
}
```

La fonction annonce dès le début qu'elle utilise le principe des templates. Le <class T> indique que le symbole T représente n'importe quel type de variable.

Dans la fonction ensuite, on voit que celle-ci retourne un élément de type T et qu'elle en reçoit 2 de type T eux aussi. En fait, à la compilation, le "T" sera remplacé par le type nécessaire. Par exemple, T sera remplacé par int si vous appelez la fonction comme ceci :

Code : C++ - Sélectionner

```
int a = 10, b = 15, c = 0;
c = addition(a, b); // T sera remplacé par int car a, b et c sont des int
```

Les templates peuvent aussi être utilisés par des classes. Cela permet à la bibliothèque standard du C++ (et à Qt aussi d'ailleurs) de créer des classes qui gèrent des tableaux d'objets à taille variable.

Dans le chapitre sur le réseau, nous avions par exemple utilisé un tableau QList qui utilisait le principe des templates :

Code : C++ - Sélectionner

```
QList<QTcpSocket *> clients;
```

Les QList sont des tableaux qui acceptent n'importe quel type de données et dont la taille peut varier. Entre les chevrons < et >, on indique à la classe ce qu'on va stocker à l'intérieur (ici des pointeurs sur QTcpSocket). Grâce aux templates, les QList peuvent donc stocker n'importe quel type de données !

Vous pouvez retrouver une [explication plus détaillée des templates par foester](#) dans son tutoriel. 😊

Les namespaces

Souvenez-vous. Dès le début du tutoriel C++, je vous ai fait utiliser les objets cout et cin qui permettent d'afficher un message dans la console et de récupérer le texte saisi au clavier.

Voici le tout premier code source C++ que vous aviez découvert :

Code : C++ - Sélectionner

```
#include <iostream>

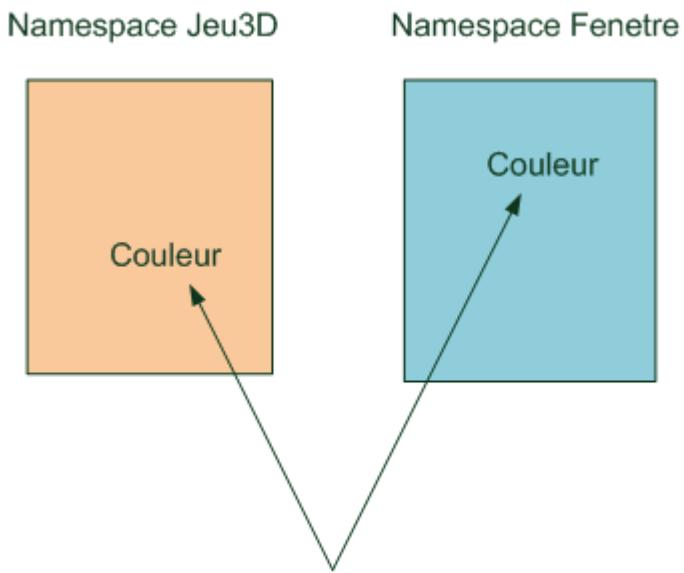
int main()
{
    std::cout << "Hello world!" << std::endl;
    return 0;
}
```

Le préfixe "std::" correspond à ce qu'on appelle un namespace, c'est-à-dire espace de nom en français. Les namespaces sont utiles dans de très gros programmes où il y a beaucoup de noms de classes et de variables différents.

Quand vous avez beaucoup de noms différents dans un programme, il y a un risque que 2 classes aient le même nom. Par exemple, vous pourriez utiliser 2 classes Couleur dans votre programme : une dans votre bibliothèque "Jeu3D" et une autre dans votre bibliothèque "Fenetre".

Normalement, avoir 2 classes du même nom est interdit... sauf si ces classes sont chacune dans un namespace différent ! Imaginez

que les namespaces sont comme des "boîtes" qui évitent de mélanger les noms de classes et de variables.



Ces 2 classes portent le même nom mais ça ne pose pas de problème car elles sont dans des namespaces différents.

Si la classe est dans un namespace, on doit placer le nom du namespace en préfixe devant :

Code : C++ - [Sélectionner](#)

```
Jeu3D::Couleur rouge; // Utilisation de la classe Couleur située dans le namespace Jeu3D
```

3

5
6

4

Le namespace "std" est utilisé par toute la bibliothèque standard du C++. Il faut donc mettre ce préfixe devant chaque nom issu de la bibliothèque standard (cout, cin, endl, string...).

Il est aussi possible, comme je vous l'avais dit, d'utiliser la directive "using namespace" au début du fichier :

Code : C++ - [Sélectionner](#)

```
using namespace std;
```

Grâce à ça, dans tout le fichier le compilateur saura que vous faites références à des noms définis dans le namespace std. Cela vous évite d'avoir à répéter "std::" partout.

Certains programmeurs préfèrent éviter d'utiliser "using namespace" car, en lisant le code ensuite, on ne sait plus vraiment à quel namespace le nom se rapporte.

Pour plus d'informations, lisez le [tutoriel sur les namespaces de Vanger](#).

Les classes et fonctions amies

Vous vous souvenez de la notion de portée ? Je l'espère en tout cas !
Non ? Si je vous dis "public", "private", "protected", ça vous revient ?

Ah, j'aime mieux ça. 😊

Ces mots-clés servent à définir la portée d'un attribut ou d'une méthode dans une classe :

- Une méthode "public" pourra être appelée depuis n'importe quelle fonction de votre code.

- Une méthode "private" ne pourra être appelée que par une autre méthode de votre classe..
- Une méthode "protected" est identique à private, à la différence près que les classes filles y auront quand même accès elles aussi.

A cette liste, il faudrait rajouter la portée "friend". Je n'en ai pas parlé dans le cours car c'est une notion à double tranchant : elle peut être très pratique dans certains cas, mais si vous l'utilisez mal vous risquez de ne plus respecter le principe d'encapsulation (et donc coder comme des malpropres ).

Pour ceux qui veulent en savoir plus et qui me promettent de ne pas utiliser abusivement friend, je les autorise à lire le [tutoriel sur l'amitié de Nanoc](#) (friend = ami au cas où vous n'auriez pas percuté ).

Les méthodes virtuelles (pures) et les classes abstraites

Il y aurait beaucoup à dire sur les notions de méthodes virtuelles pures, méthodes virtuelles, classes virtuelles et classes abstraites. La pire erreur serait de mettre tout dans le même paquet. Ces notions sont chacune très différentes les unes des autres et ne signifient pas du tout la même chose à chaque fois (quel intérêt d'avoir mis des noms similaires ? Bah juste pour vous embrouiller ).

Ce que vous devez retenir c'est que ce sont des notions avancées de C++ dont on n'a besoin que dans des cas très précis, donc rares.

Je vous conseille de vous attarder en premier sur la notion de méthodes virtuelles pures : ce sont des méthodes que l'on définit sans implémenter. Dans ce cas, la classe est dite "abstraite" et on ne peut pas l'instancier (on ne peut pas créer d'objets avec). L'implémentation de la méthode virtuelle pure doit alors être faite dans une classe fille.

Cette notion est expliquée par cysboy dans son [tutoriel Java](#). Oui c'est pour du Java je sais, mais le principe est le même et son tutoriel a l'avantage de montrer le principe à l'aide de nombreux schémas.

Pour des explications spécifiques au C++, jetez un oeil au [cours de Christian Casteyde](#) ou à la [FAQ C++ de Developpez.com](#).

Je vous laisse faire des recherches supplémentaires à ce sujet si le coeur vous en dit, mais ne commencez pas par apprendre ça car ce sont des notions tout de même assez délicates.

... sur la bibliothèque standard

Comme le C, le C++ propose une bibliothèque standard. L'avantage par rapport à une bibliothèque externe comme Qt, c'est justement que cette bibliothèque est installée par défaut sur tous les ordinateurs. Pas besoin de livrer des DLL supplémentaires.

La bibliothèque standard du C++ est très riche (et parfois complexe). Je vous ai présenté quelques notions de base à ce sujet dans mon cours :

- Les entrées / sorties avec cin et cout.
- Les chaînes de caractères à taille variable, avec la classe string.

Ca, c'est vraiment un tout petit bout de la bibliothèque standard C++ qui propose en fait de nombreux autres outils. Comme pour la classe string, ces outils sont plutôt faciles à utiliser et très pratiques, mais ils font appels à des mécanismes complexes en interne.

Les différentes parties de la bibliothèque standard du C++

La bibliothèque standard du C++ peut être découpée comme ceci :

- La bibliothèque de flux : c'est elle qui gère les flux d'entrée / sortie cin et cout, en utilisant en particulier les opérateurs << et >>.
- La gestion des chaînes de caractères : avec la classe string.
- La bibliothèque standard du C : elle est aussi utilisable en C++. En effet, certaines fonctions basiques (comme les fonctions

mathématiques) n'ont pas été réécrites en C++ car cela n'aurait eu aucun intérêt. Par conséquent, on utilise toujours les fonctions du C.

- La bibliothèque standard de templates, aussi appelée STL (Standard Template Library) : c'est une grosse partie de la bibliothèque standard du C++ qui utilise massivement le principe des templates que je vous ai présenté un peu plus haut.

La STL

C'est clairement le plus gros morceau de la bibliothèque standard du C++. Il s'agit d'un ensemble de classes et d'algorithmes qui utilisent les templates pour créer des conteneurs capables de stocker n'importe quel type de données.

Présentation de la STL

Quand on veut stocker plusieurs objets du même type en temps normal, on utilise généralement un tableau :

Code : C++ - [Sélectionner](#)

```
int tableauEntiers[10];
string tableauChaines[5];
```

Le problème, c'est que ce genre de tableau a une taille fixe. Le premier ne peut stocker que 10 entiers, et le second que 5 chaînes (string) maximum.

Parfois, on ne sait pas combien de données différentes notre tableau va contenir. Ca peut très bien être 3 comme 300... Imaginez par exemple un jeu en ligne : on ne sait pas combien de joueurs vont participer à l'avance quand on écrit le code. Rappelez-vous de notre programme de Chat : on ne sait pas combien de gens vont chatter.

Je sais ! On n'a qu'à créer un très grand tableau pour être sûr de pouvoir tout stocker !

`int tableauEntiers[999999];` et hop là ! Le problème est réglé ! 😊

Ca va pas la tête ? 😬

En faisant ça, vous consommez beaucoup trop de mémoire inutilement. Ce n'est pas du tout efficace.

En fait, le top serait d'avoir un tableau dont la taille change en fonction des besoins. Il fait 3 cases s'il y a 3 éléments, il s'agrandit automatiquement à 4 cases si on ajoute un élément, se réduit à 2 cases si on en enlève un.

Figurez-vous que c'est tout le principe de la STL. Elle propose des classes "conteneur" qui se comportent comme des tableaux dynamiques (= à taille variable). Le truc, c'est que la STL propose de très nombreuses classes conteneur. On aurait pu se dire qu'une aurait suffit, mais en fait non. 😊

Tout dépend de vos besoins. La [FAQ STL de Developpez.com](#) propose [un schéma](#) qui fait un peu peur au début mais qui résume en fait très bien la situation.

Tout dépend comme vous le voyez si vous avez besoin de créer un tableau ordonné ou si au contraire l'ordre n'a pas d'importance. Tout dépend aussi de la fréquence à laquelle vous allez insérer des éléments dans les conteneurs, de la façon dont vous voulez les lire, etc.

Dans tous les cas, les conteneurs sont capables de stocker n'importe quel type de données grâce aux templates.

Liste des classes de la STL

Voici une liste des différents conteneurs proposés par la STL :

- stack
- queue
- priority_queue
- list
- vector

- deque
- map
- set
- multi_map
- multi_set

Chacun d'entre eux a ses spécificités. Le [schéma](#) dont je vous ai parlé vous permet de faire votre choix parmi la jungle des conteneurs disponibles.

Un exemple simple : la classe vector

Commencez par exemple par la classe vector, c'est un bon début pour découvrir la STL.

On peut se servir de vector comme d'un tableau à taille dynamique. Voici un exemple d'utilisation, pour un vector qui va stocker des int :

Code : C++ - [Sélectionner](#)

```
vector<int> tableauEntiers; // Création d'un tableau d'entiers à taille variable

tableauEntiers.push_back(14); // Insère 14
tableauEntiers.push_back(27); // Insère 27
tableauEntiers.push_back(83); // Insère 83

cout << tableauEntiers[1]; // Affiche 27
```

Le vector ne nécessite donc pas que l'on définisse une taille lors de sa création. On ajoute des éléments à la fin du tableau avec la méthode push_back(). On peut ensuite accéder à n'importe quel élément du vector comme si c'était un tableau (merci la surcharge de l'opérateur [] !).

Pour connaître la taille actuelle du tableau, appelez la méthode size(). Dans notre exemple précédent, cette méthode aurait renvoyé 3.

Il y aurait beaucoup à dire encore sur les vector. Nanoc leur a justement dédié un [tutoriel](#) que je vous invite à lire. 😊

Les algorithmes

La STL propose de nombreuses fonctions capables d'effectuer des opérations sur ces conteneurs :

- Copier le tableau
- Trier les éléments dans le tableau
- Trouver l'élément le plus petit ou le plus grand
- Scinder ou fusionner un tableau
- Rechercher dans un tableau
- Supprimer les doublons

Bref, toutes les opérations de base sont déjà codées pour vous ! Il n'y a plus qu'à les utiliser.

Où trouver de la documentation ?

Comme je ne peux pas vous expliquer tout ça à moins d'y passer un temps fou, il faudra lire la documentation ou rechercher d'autres tutoriels.

Personnellement, bien qu'il n'y ait pas de "site officiel" comme pour Qt, j'ai tendance à utiliser ce site comme documentation de référence pour la bibliothèque standard du C++ :

www.cppreference.com

Vous y retrouverez en particulier la liste des classes de la STL et des algorithmes qui y sont proposés.

Bonne lecture 😊

... sur la bibliothèque Qt

Dans le cours, nous avons eu largement le temps d'étudier la bibliothèque Qt et de découvrir à quel point il était simple de créer des GUI (fenêtres).

Nous avons aussi découvert que cette bibliothèque était énorme, et qu'on devait plutôt parler de framework (ensemble de bibliothèques).

Je vous rappelle que Qt est constitué de plusieurs modules :

- GUI
- OpenGL
- Dessin
- Réseau
- SVG
- Scripts
- XML
- SQL
- Core

En ce qui nous concerne, nous avons eu l'occasion de bien faire le tour du module GUI (c'était le but !) et nous nous sommes initiés aussi un peu au réseau.

Malgré cela, nous n'avons pas tout vu sur le module GUI. D'autre part, nous avons seulement effleuré le module réseau, et nous n'avons pas du tout parlé des autres modules.

Je vais, dans cette annexe, vous présenter brièvement quelques-uns de ces modules. Je ne vais pas vous les expliquer (ce serait beaucoup trop long !), juste vous en parler pour vous donner quelques pistes.

Surtout, pensez à vous rendre sur [la doc](#) pour en savoir plus ! 😊

Module GUI : des petites fonctionnalités cachées

Il y a quelques widgets et fonctionnalités plus rares dont je n'ai pas eu l'occasion de parler. Je vais vous en présenter quelques-uns rapidement ici. Ils ne sont pas toujours utiles mais ça peut être bien de savoir qu'ils existent.

Cette liste des autres fonctionnalités à découvrir n'est pas complète, loin de là. Je ne connais pas tout. Je vous donne juste une idée des "petites choses" que vous pouvez découvrir si vous passez un peu de temps dans la doc. 😊

[QCalendarWidget](#) : un calendrier tout prêt

Le widget [QCalendarWidget](#) permet d'afficher un calendrier :



Si vous devez réaliser un agenda ou si l'utilisateur doit sélectionner une date, nul doute que ce widget vous fera gagner un temps fou !

QSplashScreen : pour faire patienter au démarrage

Parfois, certains programmes sont un peu longs à charger. Pour faire patienter l'utilisateur, on affiche un "splash screen", c'est-à-dire une petite image au centre de l'écran. C'est ce que fait Code::Blocks au démarrage par exemple.

Qt permet de créer un "splash screen" avec la classe [QSplashScreen](#). On l'utilise en général dans le main, juste avant d'ouvrir la fenêtre principale :

Code : C++ - Sélectionner

```
#include <QApplication>
#include <QTranslator>
#include <QLocale>
#include <QLibraryInfo>
#include <QSplashScreen>
#include <QPixmap>
#include "FenPrincipale.h"

int main(int argc, char* argv[])
{
    QApplication app(argc, argv);

    QSplashScreen splash(QPixmap("znavigo.png"), Qt::WindowStaysOnTopHint);
    splash.show();

    // Traduction des chaînes prédéfinies par Qt dans notre langue
    QString locale = QLocale::system().name();
    QTranslator translator;
    translator.load(QString("qt_") + locale, QLibraryInfo::location(QLibraryInfo::TranslationsPath));
    app.installTranslator(&translator);

    // Ouverture de la fenêtre principale du navigateur
    FenPrincipale principale;
    principale.show();

    return app.exec();
}
```

Résultat :

Fichier Navigation ?



http://www.siteduzero.com/

Le Site du Zéro, site communautaire de t...



Le Site du Zéro

Ici, on apprend tout à partir de zéro...

[Accueil](#) / [Tutoriels](#) / [Forums](#) / ...

[Tutoriels](#)

Vous débutez ? C'est ici qu'on commence !

Site Web

XHTML / CSS
PHP / MySQL

Programmation

Langage C
Langage C++
Langage Java
Nouveau !

Système alternatif

Linux

Prêt



zNavigo

- Qu'est-ce que c'est ? : Vous êtes sur le Site du Zéro ? Si vous vous sentez un peu perdu, n'hésitez pas à consulter le plan ou à lire le Manuel du Zéro.
- Qui appelle-t-on "les Zéros" ? : c'est vous, les visiteurs de ce site. Nous vous donnons ce surnom amical car tous nos cours partent de "Zéro" : aucune connaissance n'est requise pour pouvoir les lire. Vous pouvez voir la liste des cours officiels et la liste des cours rédigés par les Zéros.
- Ça coûte combien ? : tout cela est gratuit. En effet, nous voulons qu'un maximum d'entre vous puisse en profiter.

Évènements

« Juillet 2008 »

Dernières News

Le concours Sudoku avance à grands pas

Derniers tutoriels

Un wiki avec Yaws en 13 minutes 37

Sondage

Envisagez-vous de faire ou non une fois des études de...

Le splash screen peut être arrêté en cliquant dessus.

Après, libre à vous de l'arrêter automatiquement au bout d'un certain temps, il faut juste chercher dans la doc comment faire.

Afficher une icône dans le system tray

Pour certaines applications résidentes en mémoire, il peut être utile de placer une icône dans le system tray, oui là à côté de l'horloge vous savez. 😊

Qt permet justement de le faire avec QSystemTrayIcon :



Le mieux pour apprendre à s'en servir est de jeter un oeil à l'[exemple fourni dans la doc de Qt](#).

Module réseau : utilisez des classes de haut niveau

Dans notre découverte du réseau, nous avons utilisé des QTcpSocket et un QTcpServer. C'est une gestion assez bas niveau des paquets et il nous a fallu apprendre un peu comment le réseau fonctionnait.

On aurait pu parler des paquets UDP aussi, mais on les utilise vraiment dans des cas spécifiques.

En revanche, ce qu'on n'a pas vu, c'est qu'il y a des classes de plus haut niveau qui vous évitent d'avoir à manipuler les paquets TCP directement. Je pense en particulier à :

- QHttp : vous permet d'utiliser le protocole HTTP et donc de télécharger des pages web ou des fichiers via le web.
- QFtp : vous permet de télécharger et d'envoyer des fichiers par FTP. Vous pourriez créer votre propre client FTP comme Filezilla par exemple. 😊

Ces classes sont beaucoup plus faciles à utiliser que celles que nous avons vues, donc n'hésitez pas à y jeter un oeil. Elles sont brièvement introduites dans la [page d'accueil du module](#) réseau sur la doc de Qt.

Module SQL : accès aux bases de données

Si votre programme doit enregistrer de nombreuses données, il peut être utile de les stocker dans une base de données. C'est un système puissant pour enregistrer des informations, mais il faut connaître le langage SQL pour écrire et lire des informations dedans.

Qt propose tout ce qu'il faut pour se connecter à une base de données dans votre programme, mais il n'inclue pas la base de données... ce sera à vous de l'installer. En clair, si vous utilisez MySQL comme base de données, il faudra d'abord aller installer MySQL sur le [site officiel](#) avant de pouvoir établir une connexion avec dans votre programme.

MySQL est un système de gestion de base de données puissant mais évitez d'y avoir recours systématiquement dans vos programmes. Ce serait un peu utiliser un tank équipé de missiles nucléaires pour tuer une mouche.

Parfois, stocker les meilleurs scores dans un jeu pour être facilement fait dans des fichiers (avec QFile par exemple) sous forme de texte simple ou au format XML (je vais en parler un peu plus loin). Inutile de sortir l'artillerie lourde MySQL pour ça.

Si toutefois vous avez vraiment besoin d'une base de données mais que vous ne voulez pas utiliser MySQL qui est un peu gros, jetez un œil du côté de [SQLite](#) qui est tout léger (mais un peu moins complet).

Une fois que vous avez installé votre système de gestion de base de données sur votre ordinateur, vous pouvez découvrir comment y faire appel depuis Qt. Le mieux est de lire l'[introduction au module QSql](#) sur la doc. En tout cas c'est ce que je ferais à votre place.

En quelques minutes de lecture de cette seule page, vous devriez déjà savoir vous connecter à la base de données et exécuter des requêtes SQL (mais attention, il faut connaître le langage SQL avant !).

Module XML : pour ceux qui doivent gérer des données au format XML

Le XML est un langage générique qui est à la base de nombreux autres langages, comme XHTML (qui permet de créer des pages web).

Le principe de XML peut être très vite compris si vous avez déjà fait du XHTML avant. En gros, c'est vous qui définissez vos propres balises :

Code : XML - [Sélectionner](#)

```
<bibliothèque>
    <livre>
        <auteur>J.R.R. Tolkien</auteur>
        <titre>Le seigneur des anneaux</titre>
    </livre>
    <livre>
        <auteur>R. Barjavel</auteur>
        <titre>La nuit des temps</titre>
    </livre>
</bibliothèque>
```

Les données sont placées entre des balises que vous définissez. L'avantage du XML est qu'il est facile à lire (enfin, tant que le fichier n'est pas trop gros ou trop complexe).

Vous pouvez vous servir de cette technique pour organiser vos données dans des fichiers sans avoir recours à une base de données. D'autre part, le XML est un format d'échange devenu courant de nos jours, et il est possible que quelqu'un vous "envoie" des données au format XML que vous devrez traiter dans votre programme.

Pour lire le contenu d'un document XML comme celui ci-dessus (et pour écrire du XML aussi), il y a le module `QtXml` qui permet de faire cela facilement. Il vous faudra acquérir avant un peu de théorie sur le fonctionnement de XML (DOM, SAX, XQuery, DTD, XML Schema...). Il vaut mieux être rodé sur la théorie de XML avant de s'y lancer sinon vous n'en profiterez pas. 😊

Je vous conseille de lire cette petite [introduction à XML](#) sur le Site du Zéro avant de faire des recherches plus approfondies. [Wikipedia](#) est une bonne source de départ aussi.

Une fois que vous connaissez un peu mieux le fonctionnement de XML, direction la [page d'accueil du module QtXml](#) pour découvrir les outils que Qt met à votre disposition pour lire et écrire du XML. Il y a de quoi faire, et encore une fois je vous le rappelle, mieux vaut être armé et connaître XML avant de se lancer là-dedans !

Module Core : toutes les fonctionnalités de base de Qt

Le module `QtCore` contient des classes de base de Qt qui n'ont pas de rapport avec les GUI et qui peuvent donc être utilisées dans un programme purement console.

Dans ce module, on trouve un certain nombre de classes que vous connaissez déjà :

- `QString` : gestion des chaînes de caractères.
- `QByteArray` : une suite d'octets (on s'en est servi dans le programme de Chat pour construire des paquets).
- `QFile` : accès aux fichiers.
- `QLocale` : permet d'accéder aux habitudes de représentation des nombres et chaînes dans différentes langues.
- `QList` : une liste capable de stocker un tableau à taille dynamique (cette classe est une version "Qt" de ce qui se fait dans la STL dont je vous ai parlé plus haut).
- `QUrl` : représente une URL.

Voilà quelques exemples de classes du module `QtCore` que vous avez déjà utilisées. Comme vous le voyez, ces classes font partie du "coeur" de Qt et pas du module GUI car elles peuvent être réutilisées dans tous les autres modules.

Jetez donc un œil à la [liste des classes du module QtCore](#). Il y a de quoi faire, et on retrouve notamment de nombreuses versions "Qt" de classes présentes dans la STL (il y a même un [QVector](#) !).

Bonne pêche !

J'espère que cette annexe aura rempli son rôle : vous aider à regarder dans de nouvelles directions. L'inconnu, ça fait un peu peur au début, mais on s'y fait très vite vous verrez. 😊

Comme vous avez pu le voir, tout ce que vous pouvez faire en C++ (et en programmation en général) est tellement riche qu'on n'aurait jamais assez d'une vie pour tout connaître. J'espère que vous me comprenez maintenant. 😊

Plutôt que de tout apprendre, essayez plutôt de découvrir une nouvelle notion à la fois. Si vous vous éparpillez trop, vous aurez du mal à bien assimiler ces connaissances.

Bon courage, et bonne continuation ! 😊

Le cours de C++ s'arrête là !

J'espère que vous aurez appris au moins autant de choses que vous ne l'espériez, et surtout que vous avez formé votre esprit à être capable de programmer en toutes circonstances par la suite.

N'hésitez pas à lire la dernière annexe "Ce que vous pouvez encore apprendre", qui vous donne de nombreuses ouvertures pour continuer votre apprentissage si vous le désirez. 😊