Martin Hammerchmidt

APPRENDRE LE C++

"Venez apprendre le C++ dans ce livre à la portée de tous ! Jamais apprendre le C++ n'aura été aussi simple !"

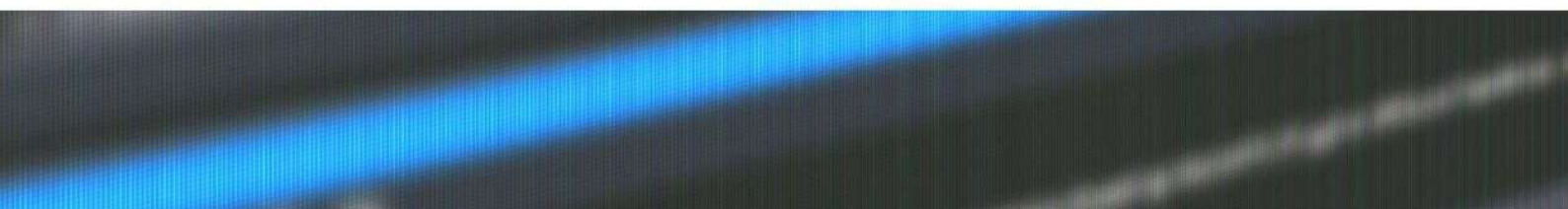


Table des matières

Introduction

Découvrir le C++

Venez découvrir le C++ dans ce livre learn-by-doing à la portée de tout le monde ! Jamais apprendre le C++ n'aura été aussi simple !

Préface

Bienvenue dans ce livre !

Si vous êtes ici, c'est probablement pour découvrir ou re-découvrir le C++. Et c'est, croyez moi, une très bonne idée.

En effet, dans ce livre nous allons partir des bases du C++ pour progresser lentement mais sûrement. Une fois ce livre acquis vous aurez amplement les connaissances nécessaires pour mener des projets de petites ou moyennes envergures. Et n'oubliez surtout pas que c'est en forgeant qu'on devient forgeron !

Ce livre est en cours d'écriture

Première chose très importante, ce livre est en cours d'écriture. J'ai commencé ce livre le 19 Juillet 2015 et je le met à jours aussi régulièrement que possible.

Pour qui est destiné ce livre ?

Ce livre est écrit dans un français très simple et est à la portée de tous. Le seul pré-requis pour ce livre est de savoir utiliser un ordinateur de manière aisée.

> Si un problème indépendant au livre intervient lors de votre apprentissage, n'hésitez surtout pas à *googliser* votre problème, peut être que vous trouverez la solution.

Que vous n'ayez absolument aucune connaissance en programmation ou que vous connaissez déjà le C++, ce livre saura vous satisfaire dans le sens où il vous servira de cours complet ou d'aide mémoire.

Comment ce compose le livre ?

Nous allons commencer par les notions les plus simples et les plus basiques. Petit à petit le niveau montera et vous découvrirez la véritable puissance du C++.

Le nombre de notions que vous allez acquérir est vraiment énorme. Mais n'abandonnez pas trop vite la lecture de ce livre ! Si vous trouvez que ce que je vous montre est trop compliqué, ne vous inquiétez pas. Relisez le chapitre une fois ou passez à la suite et vous le comprendrez naturellement par la suite.

Ce livre est très concis

J'ai volontairement écrit ce livre de manière très concise afin de ne pas vous ennuyer. Seul "bémol" à ce choix : vous devez donner plus du votre encore. Vous devrez vraiment faire parti du livre et pas seulement être un spectateur. Les exemples que je donne ne sont pas là que pour accompagner de manière secondaire les explications, les exemples font partis intégrante des explications et sont même **plus importants** que les explications.

Ce livre est *example driven*

Vous verrez que je donne **beaucoup** d'exemples et que parfois mes explications sont très concises. J'insiste bien, vous devez **lire**, **comprendre** et **apprendre** plus par les exemples que par mes explications. Mes explications ne sont ici que pour vous aider à comprendre les exemples !

Le glossaire !

Pensez au glossaire ! La plupart des termes techniques sont indexés dans le glossaire. Celui-ci vous servira d'aide mémoire afin de ne pas oublier les divers termes essentiels que nous aborderons ici.

Un exemple de code

Si vous êtes sur téléphone ou sur tablette ou encore si vous avez un petit écran, l'affichage des lignes de code peut être problématique. Voici un exemple de code, les lignes de codes ne seront jamais plus longues que celles-ci :

```
std::string& longueLigne(std::string &ligne, int * ptr)
{
    std::cout << ligne << *ptr << std::endl << "Merci !" << std::endl
    return ligne;
}
```

(Notez que ce code est totalement inventé et n'a rien à voir avec le reste du livre)

Il faut savoir admettre

Dès les premières minutes de lecture de ce livre nous allons travailler sur du code C++. Sachez-le tout de suite, vous ne comprendrez pas tout tout de suite. Si je vous dit que nous verrons telle chose plus tard ou que telle chose sert à telle autre chose, admettez-le. Considérez-le comme acquis et n'y faites plus attention : nous reviendrons dessus plus tard.

Une remarque ? Une question ? Un problème ?

Si jamais j'ai fait une erreur, ou si vous avez une question je vous propose de me contacter sur Twitter @_MartinH_.

Les bases du C++

Les bases du C++

Qu'est-ce que le C++ ?

Qu'est-ce que le C++ ?

Qu'est-ce que l'on peut faire avec le C++ ?

Eh bien... Beaucoup de choses ! Théoriquement, avec le C++ on peut créer un OS, on peut faire des jeux vidéos, des applications desktop et mobiles. Si vous le souhaitez vous pouvez même faire des sites web ou une IA qui vous sert le café !

> Saviez-vous que la plupart des gros jeux vidéos AAA sont codés en C++ ?

Donc oui le C++ est un langage puissant qui offre pleins de possibilités à condition de le maîtriser.

D'ailleurs nous allons très vite créer des applications plus ou moins complexes avec des fenêtres, des boutons ou tout ce que vous voulez !

L'histoire du C++ !

Un jour, un homme au doux nom de *Bjarne Stroustrup* a trouvé qu'il manquait des choses au C (un autre langage de programmation). Il a donc dans un premier temps créé une modification du C qu'il a nommé *C with classes* car oui, l'une des grosses nouveautés du C++ par rapport au C ce sont les classes que nous verrons plus tard. Il a ensuite nommé le C with classes avec un nom plus court : **C++** . Ce nom représente bien le C++, c'est une évolution du C d'où l'appellation C plus plus. Parfois on le nomme aussi cpp. Et dans la foulé, en 1985 il montra au monde le C++ en "publiant" le langage ainsi qu'un premier livre expliquant comment l'utiliser.

> De nombreuses personnes pensent que le C et le C++ sont des langages similaires. C'est absolument et totalement faux, le C et le C++ sont radicalements différents, surtout plus récemment avec les nouvelles normes du C++.

La définition du C++ ?

Le C++ a le droit à une définition très simple :

> C++ est un langage de programmation compilé, permettant la programmation sous de multiples paradigmes comme la programmation procédurale, la programmation orientée objet et la programmation générique.

- *Wikipédia*

Etudions cette définition segment par segment.

> C++ est un langage de programmation compilé

Le C++ est un langage de programmation dans lequel nous allons *parler*. Parler comme en français ou en anglais mais dans un langage que l'ordinateur est capable de comprendre : en C++ tout simplement. Le C++ est un langage compilé, c'est à dire que nous allons écrire le C++ dans notre alphabet puis avec un *compilateur*, ce code sera traduit en .exe sous Windows par exemple.

Certains langages de programmations sont dits *interprétés*, c'est à dire qu'ils sont compilés lors de l'exécution et pas avant l'exécution, comme avec le C++.

> permettant la programmation sous de multiples paradigmes

Un paradigme de programmation est en quelque sorte la manière dont nous allons écrire le code. Il existe plusieurs paradigmes de programmations qui offrent divers méthodes de pensées pour créer notre application. Il faut savoir qu'un paradigme de programmation impose littéralement une méthode de pensée afin de vous aider lors de la conception. Nous verrons ces divers paradigmes plus tard, vous comprendrez mieux.

> comme la programmation procédurale, la programmation orientée objet et la programmation générique.

La programmation procédurale est la programmation que l'on fait de base en C ou en C++, le premier type de programmation que l'on apprend. La programmation orientée objet quant à elle permet d'abstraire votre code par sections indépendantes dans des boites. Par exemple, avec la programmation procédurale, *grosso-modo*, tout le code est contenu dans la même boite. Avec la programmation orientée objet, nous prendrons soin de placer les différents bouts de code dans des boites indépendants que nous lierons entre elles.

La programmation générique quant à elle, assez complexe au début, permet de travailler sur des données sans faire de distinction du type de la donnée. Si vous ne connaissez pas le C++, vous ne comprenez probablement pas ce que ça veut dire mais nous le verrons à la fin de ce livre.

Et donc ?

Cette définition ne veut ainsi pas forcément dire grand chose pour vous. C'est pour cette raison que je vous annonce que nous allons pouvoir commencer notre apprentissage du C++ avec la programmation procédural dès maintenant !

Il ne nous reste plus qu'une chose à faire, installer un compilateur car, oui, le C++ est un langage compilé.

Installons un IDE

Installons un IDE

Un IDE ?

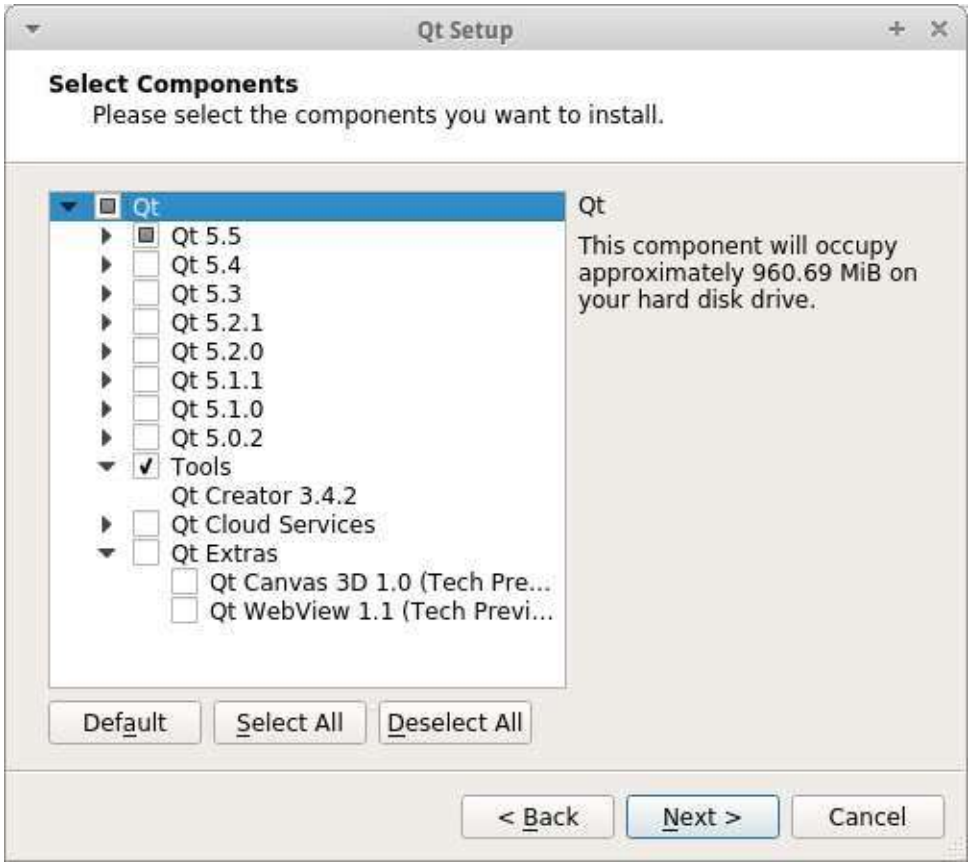
Ce terme signifie *Integrated Development Environment*. C'est un peu comme un éditeur de texte avancé qui vous aidera lorsque vous programmerez. Par exemple il complétera tout seul vos instructions ou corrigera vos erreurs.

Pour l'installer je vais directement vous proposer d'installer Qt, un outil magique que nous utiliserons par la suite. Qt vous installera en même temps le compilateur si vous êtes sur Windows et vous installera aussi Qt Creator : un IDE.

Comment installer Qt ?

Rien de plus simple. Non, vraiment !

Allez sur cette page et cliquez sur le gros bouton vert. Un téléchargement se lancera, une fois qu'il sera fini lancez le fichier obtenu puis laissez vous guider. Les paramètres par défaut devraient être largement suffisants. Assurez-vous tout de même que la case Tools est cochée ainsi que la dernière version de Qt.



Ensuite l'installation commence, prenez un petit café ou un jus multi-vitaminé le temps que celle-ci se termine !

Sur linux

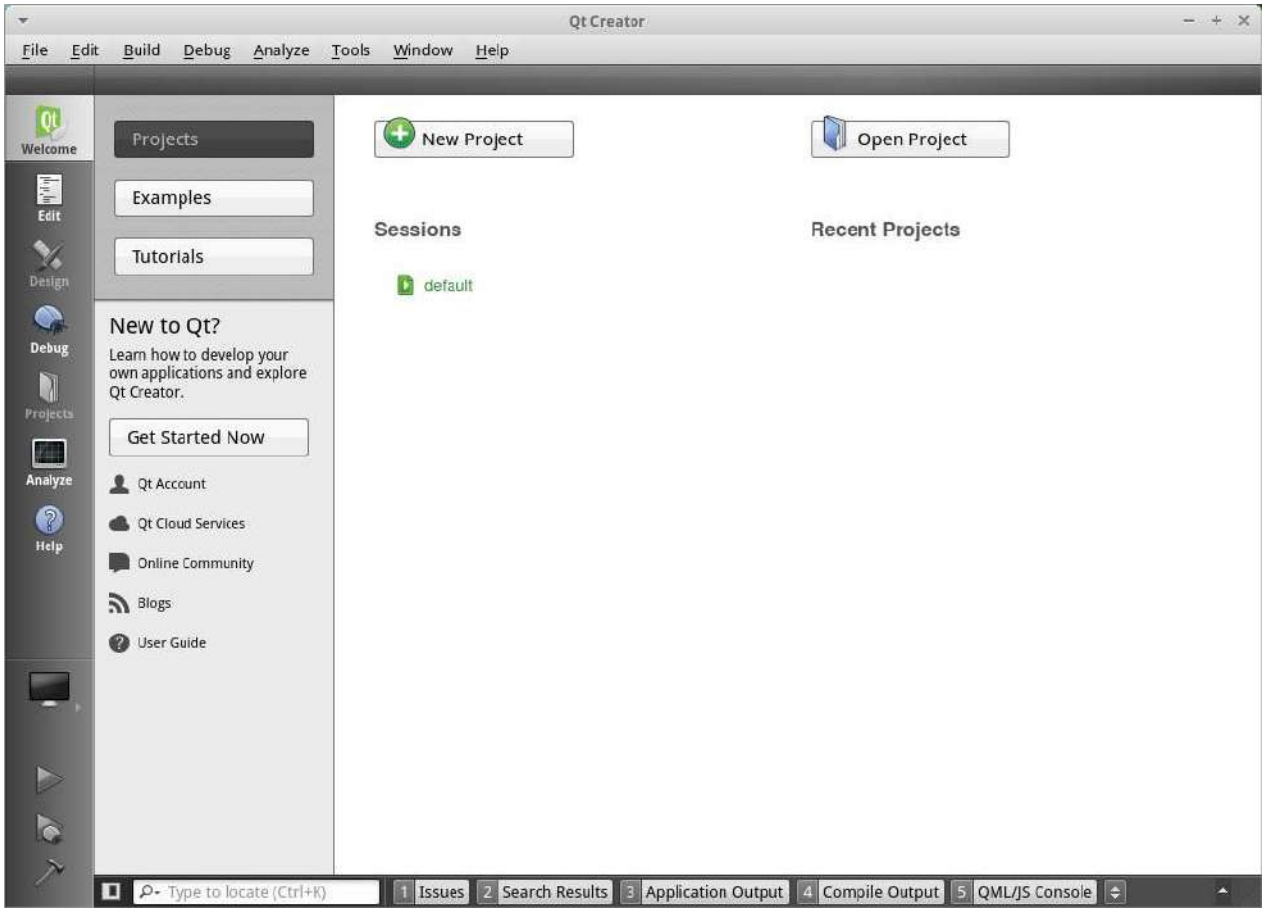
Notez que sur linux vous devez vous assuré d'avoir *build-essential* d'installé. Ainsi sur les distributions utilisant Apt, voici la commande à utiliser :

```
apt install build-essential
```

Adaptez la en fonction de votre distribution.

Lançons Qt Creator !

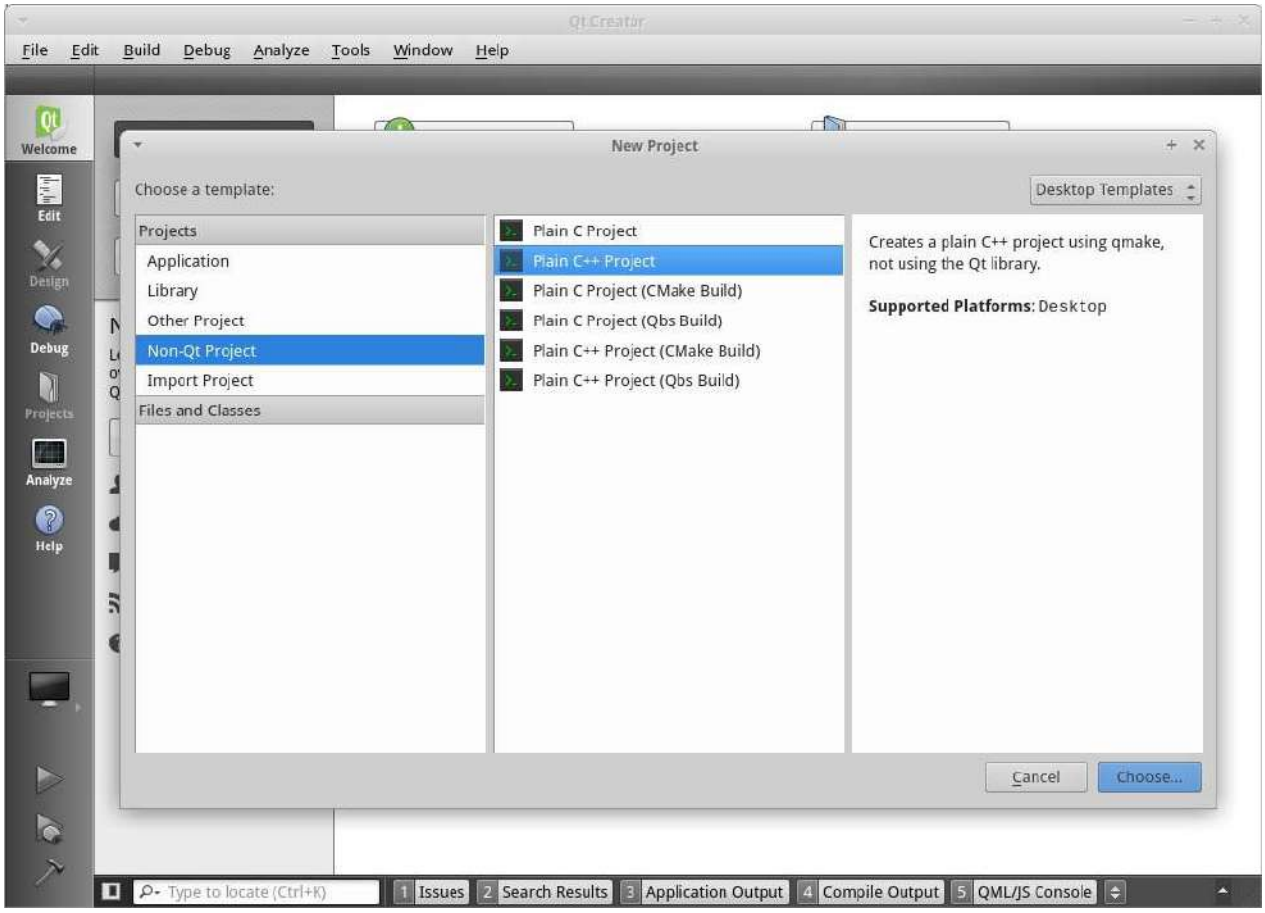
Si vous êtes arrivé jusque ici, bravo ! Maintenant je vous propose de lancer Qt Creator. Découvrons ce qu'il a dans le ventre et créons notre premier projet C++.



Enfin le moment tant attendu ! Ladies and Gentlemans, je vous présente Qt Creator ! Et croyez moi, vous allez l'aimer. Ou tout du moins il vous facilitera tellement la vie que vous ne pourrez plus vous en passer !

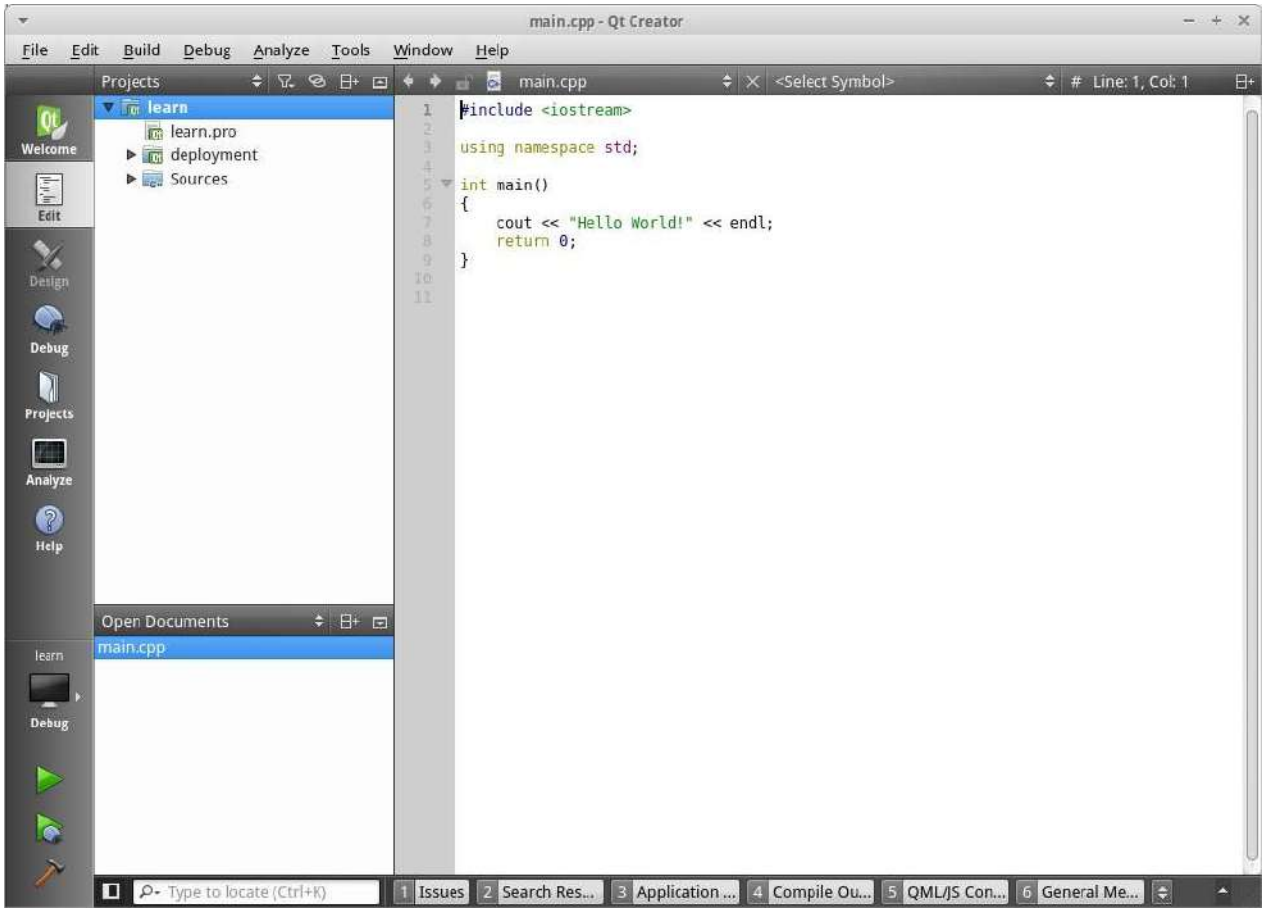
Peut être que votre version de Qt Creator sera différente de la mienne ou dans une autre langue, cela importe peu les manipulations seront probablement les mêmes.

Cliquez sur *New Project* ou sur *Nouveau Projet* en français.

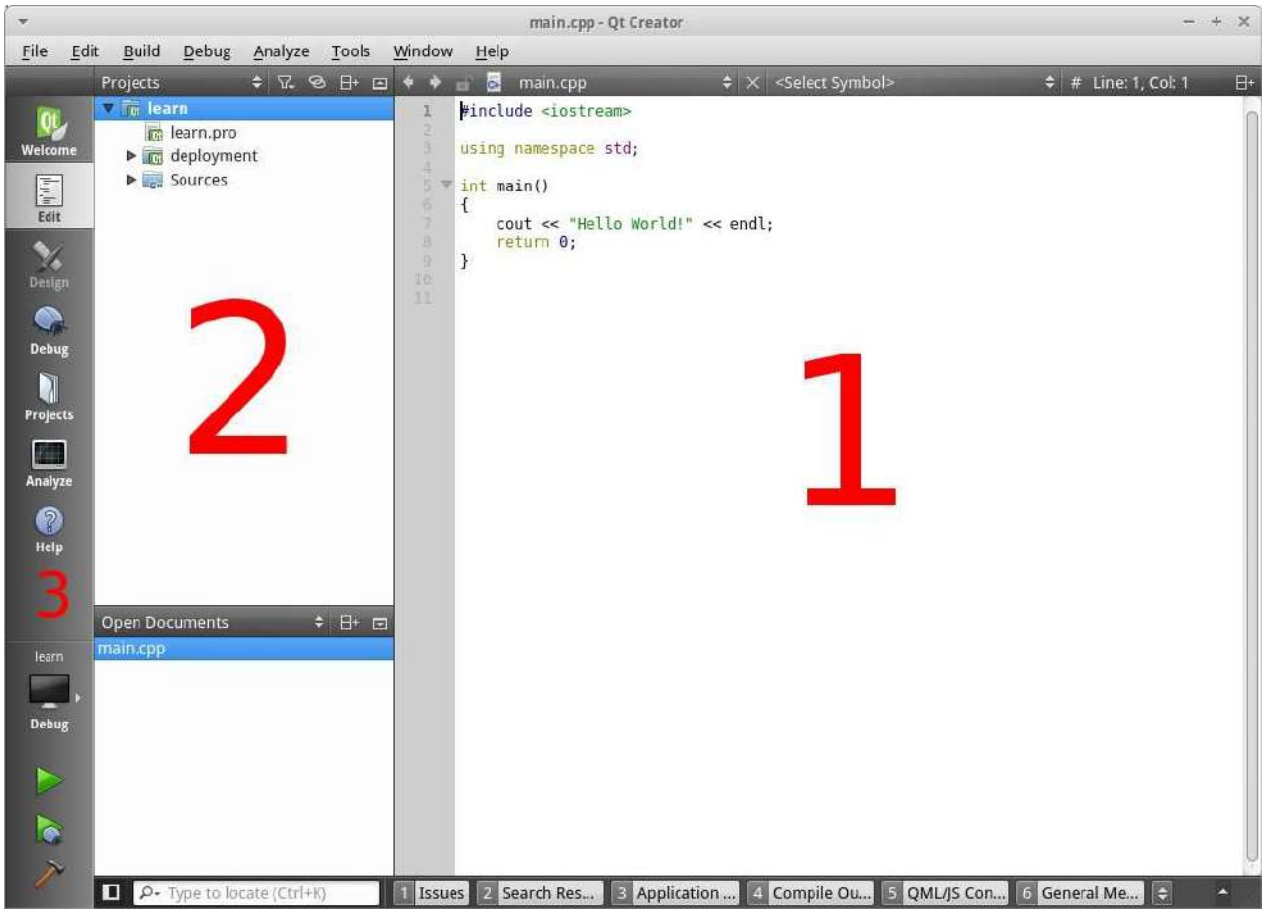


Une nouvelle fenêtre devrait apparaître. Allez dans les projets non Qt à gauche puis sélectionnez un projet C++ comme sur l'image. Enfin, validez. Nous n'utilisons pas Qt tout de suite, nous le découvrirons plus tard dans le livre.

Ensuite Qt Creator vous demandera un nom de projet, mettez par exemple *learn* pour cet apprentissage. Cliquez sur suivant à plusieurs reprises, puis le tant attendu éditeur apparaît enfin !



Nous pouvons découper Qt Creator en 3 parties :



La première partie contient le code. N'y faites pas encore attention, nous travaillerons dessus dans un prochain chapitre.

La seconde partie contient les fichiers ouverts et les fichiers de votre projet. Pratique pour naviguer entre les différents fichiers ! Qt Creator découpe vos applications en projets. Chaque fichier *.pro* représente un projet pour Qt Creator. Il contient la liste des fichiers, la configuration, etc. Nous le découvrirons plus tard !

Enfin, la troisième partie contient différentes options pour configurer son projet ou avoir de l'aide par exemple. Le plus important ici c'est les 4 boutons en bas : le marteau sert à compiler votre projet, la flèche verte sert à compiler votre projet *et* lancer votre application, tandis que la flèche verte avec une coccinelle permet de lancer le projet en mode debuggage. Chose que nous découvrirons plus tard aussi.

Vos premières lignes de C++

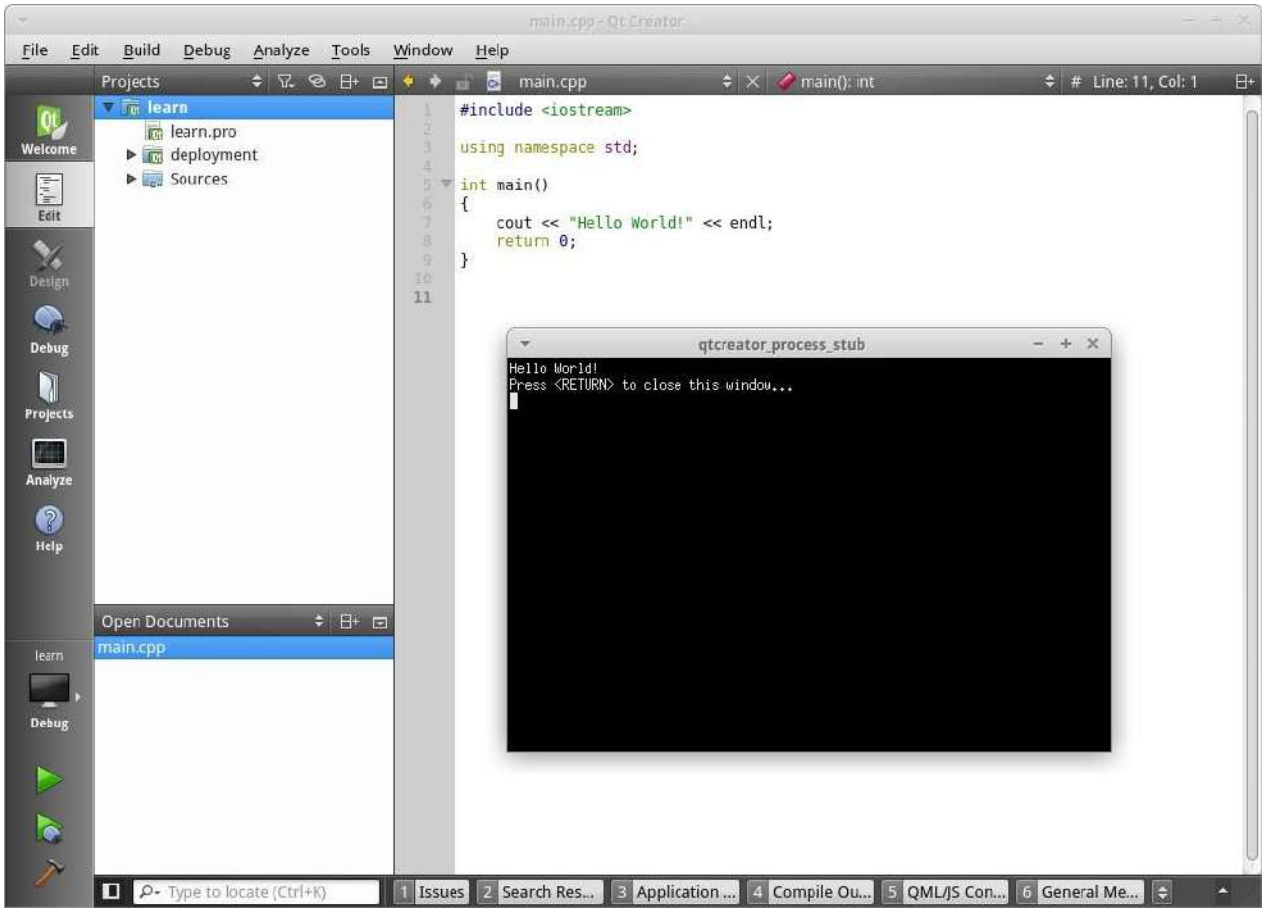
Vos premières lignes de C++

Enfin nous entrons dans le cœur du sujet !

Normalement vous avez ce code-ci à l'écran (après avoir créé un nouveau projet) :

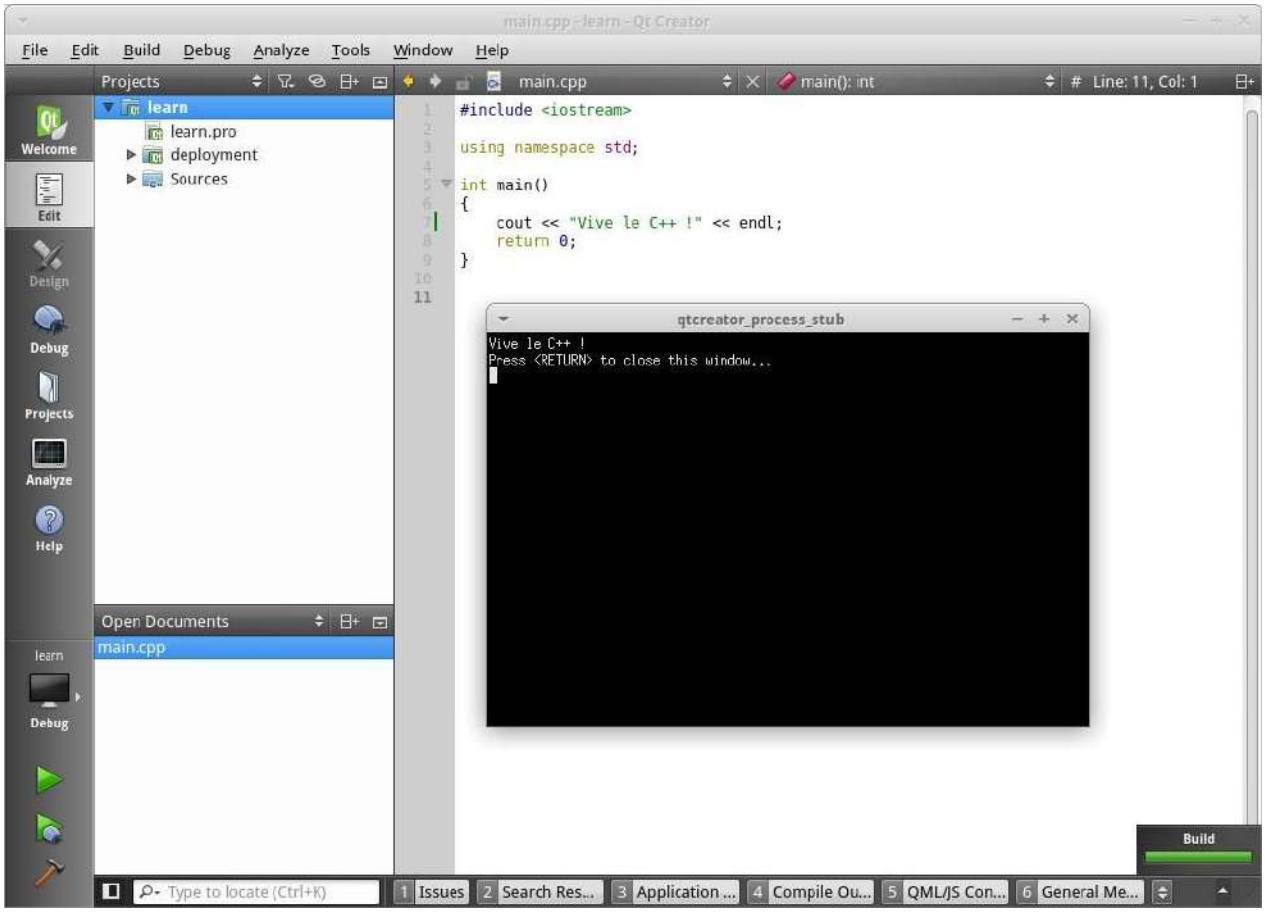
```
#include <iostream>

using namespace std;

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

Ce code vous parait peut être compliqué mais c'est en réalité le programme le plus simple que l'on peut écrire en C++. Je vous laisse cliquer sur la flèche verte, voici ce qui devrait apparaitre :



Et vous venez juste de lancer votre premier programme C++, **bravo !**

Maintenant je vous propose de remplacer le texte *Hello World!* par *Vive le C++ !* et de relancer l'application avec la flèche verte. Voici ce qui devrait apparaître cette fois ci :



Nous en concluons donc que la syntaxe `cout << "[UN TEXTE]" << endl;` permet d'afficher du texte à l'écran.

cout est un terme qui signifie *Console OUT*. En d'autres mots, ce terme permet de sortir des informations depuis l'ordinateur vers l'écran, d'où le sens des flèches. En effet les flèches ramènent le texte que l'on veut afficher vers le cout et donc vers l'écran, en quelque sorte. On appelle ceci un flux.

> Notez que `endl` permet de passer à la ligne suivante (*end line*).

Première règle : à la fin de chaque instruction, vous devez mettre un point virgule, qui marque la fin de cette même instruction.

Seconde règle : Ne JAMAIS oublier la première règle.

> Oublier un `;` est très dangereux. Déjà il empêchera votre logiciel de fonctionner, puis vous chercherez bêtement pendant 3 heures où est le problème jusqu'à ce que vous vous rendez compte que le problème est simplement un `;` qui manque. Croyez-moi, ceci est arrivé à tout le monde et vous arrivera ! Les effets secondaires de cet oubli sont très dangereux comme *la dépression, l'abandon de la lecture de ce livre* et bien d'autres... Soyez avertis ! Considérez cela comme la sélection naturelle, celui qui oublie le `;` n'arrivera pas à coder en C++ et sera naturellement éliminé. Alors prenez votre courage à deux main et n'oubliez jamais le `;` !

Troisième règle : le point de départ de toutes applications en C++, sauf exceptions, se trouve dans la fonction main.

```
int main()
{
    ..
```

```
}
```

Nous verrons plus tard ce que c'est, contentez vous pour le moment d'écrire votre code entre les accolades qui suivent `int main()`.

Nous verrons le reste de ce code plus tard, désormais je vous propose d'entamer un nouveau concept, les variables, qui vont rapidement nous devenir indispensables.

Les variables

Les variables

Soyons clairs, ce chapitre est le **premier** vrai chapitre de programmation. On ne va plus observer et modifier, je vais vous apprendre syntaxes, des manipulations, des règles. Alors comprenez bien ce chapitre dans son intégralité, car partout où vous irez dans la programmation informatique, vous retrouverez des variables.

Une variable, c'est quoi ?

C'est une chose que vous allez aimer. Un concept que vous allez adorer. Une notion que vous [allez] utiliser tous les jours. Oui.

> Une variable permet de stocker une information. Une variable possède une durée de vie et est le plus généralement stockée dans la mémoire vive.

Tout simplement !

Par exemple imaginons que vous créez un jeu révolutionnaire et que votre personnage principal a un nombre de points de vie qui peut *varier*. Nous allons alors créer une variable pour sauvegarder ce nombre de points de vie pour ensuite l'afficher à l'écran.

> Techniquement parlant, les variables ne sont pas forcément stockées dans la mémoire vive, étant donné que les processeurs modernes possèdent des registres relativement grands qui agissent comme de la mémoire vive, mais encore plus rapide.

Et en C++, ça donne quoi ?

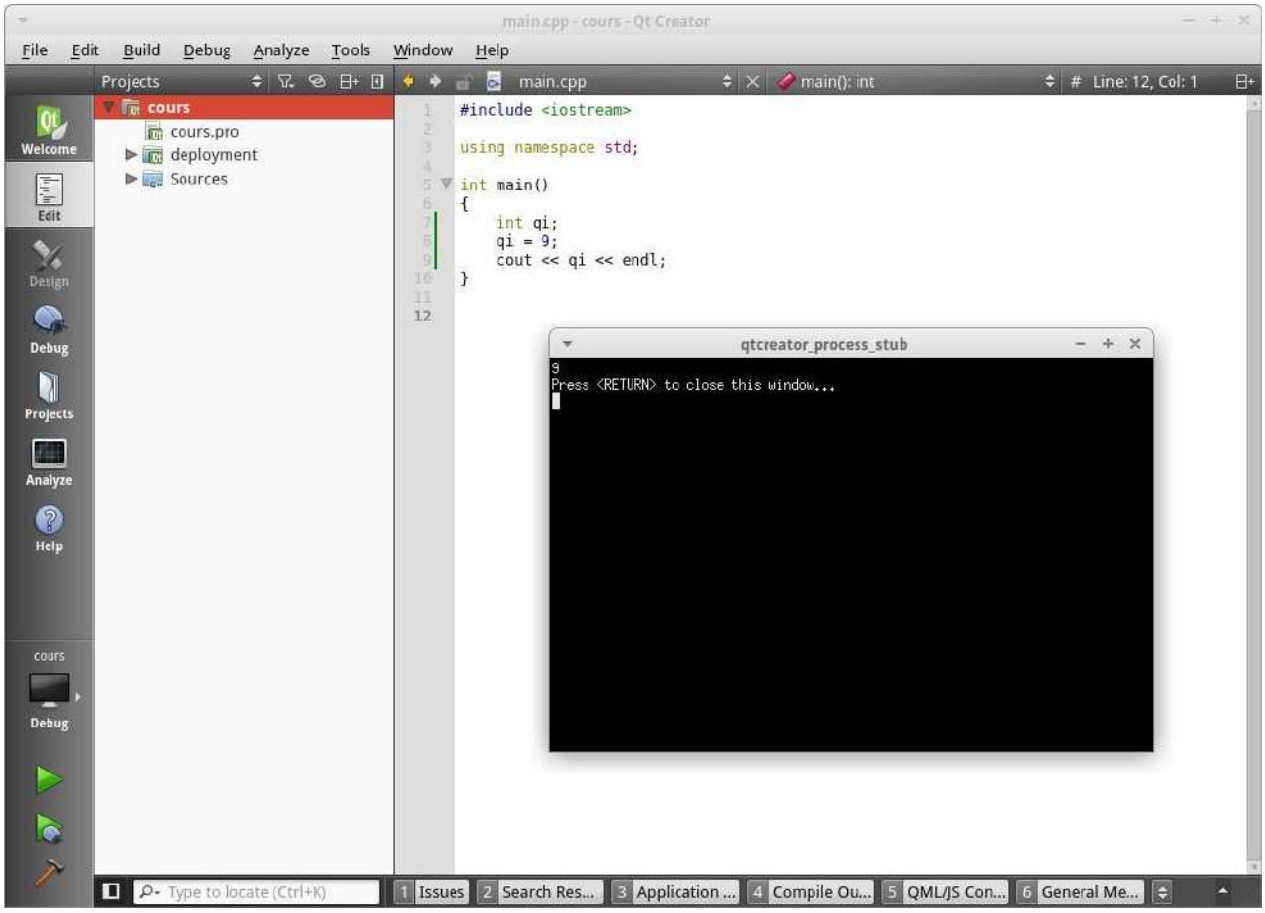
Tous les langages de programmations possèdent des variables. Mais chacun a ses spécificités.

Le C++ est un langage fortement typé, c'est à dire que vous **devez** préciser le type de la variable que vous déclarez (enfin presque, nous verrons dans un futur chapitre des subtilités). Pas d'inquiètude, c'est très simple. Les différents types sont par exemple les nombres, ou des caractères.

Prenons un exemple avec `int` qui représente un nombre (*integer* qui signifie entier en anglais).

```
int qi;
qi = 9;
cout << qi << endl;
```

Placez tout ceci dans le `main()` comme je l'ai précisé dans le chapitre précédent et voyez le résultat !



Pour tous les types que nous allons voir ici, vous pouvez les passer à `cout` via l'opérateur de flux `<<` afin d'afficher la variable dans la console.

Comme vous le voyez, créer une variable est assez simple, voici la syntaxe :

```
TYPE NOM;
```

Voyez plutôt ce code :

```
int x; // 1)
x = 0; // 2)

cout << x << endl;

x = x + 5; // 3)

cout << x << endl;
```

Première nouveauté ici, vous pouvez **commenter** votre code avec `//`. Placez un `//` n'importe où dans votre code et vous pourrez insérer sur tout le reste de la ligne du texte ou tout ce que vous voulez, ce ne sera pas pris en compte dans le logiciel.

Les commentaires utilisant `//` ne peuvent tenir que sur une ligne. Comment faire pour les commentaires multilignes ? C'est simple ! Utilisez `/*  */` ! La règle est simple : placez votre commentaire entre le `/*` et le `*/`. Le commentaire peut faire plusieurs lignes et être aussi grand que vous le voulez !

```
int i = 5;
```

```
/* Coucou
Je suis un commentaire
lala
*/

i = 8;

/* Rebonjour
 * Je suis un commentaire
 * avec plus de classe !
 * :-D !
 */
```

J'ai ici classé le code en 3 étapes.

- 1) On **déclare** la variable. On dit que elle doit être réservée dans la mémoire, désormais elle nous appartient.
- 2) On **initialise** la variable. En effet la variable était créée mais était inutilisable car elle n'avait aucune valeur. Ici on lui donne une valeur, c'est à dire zéro.
- 3) On **assigne** une nouvelle valeur à la variable. Cette valeur vaut `x + 5` c'est à dire `0 + 5` donc désormais `x` vaut 5.

Normalement si tout se passe bien votre console affichera 0 puis 5 sur des lignes différentes.

Attention, vous devez **toujours** initialiser une variable avant de l'utiliser. Sinon votre programme pourra réagir étrangement ou avoir des comportements indéfinis. Si vous le souhaitez vous pouvez lui donner une valeur dès sa déclaration, ce qui emboîte les deux premières étapes :

```
int x = 0;
```

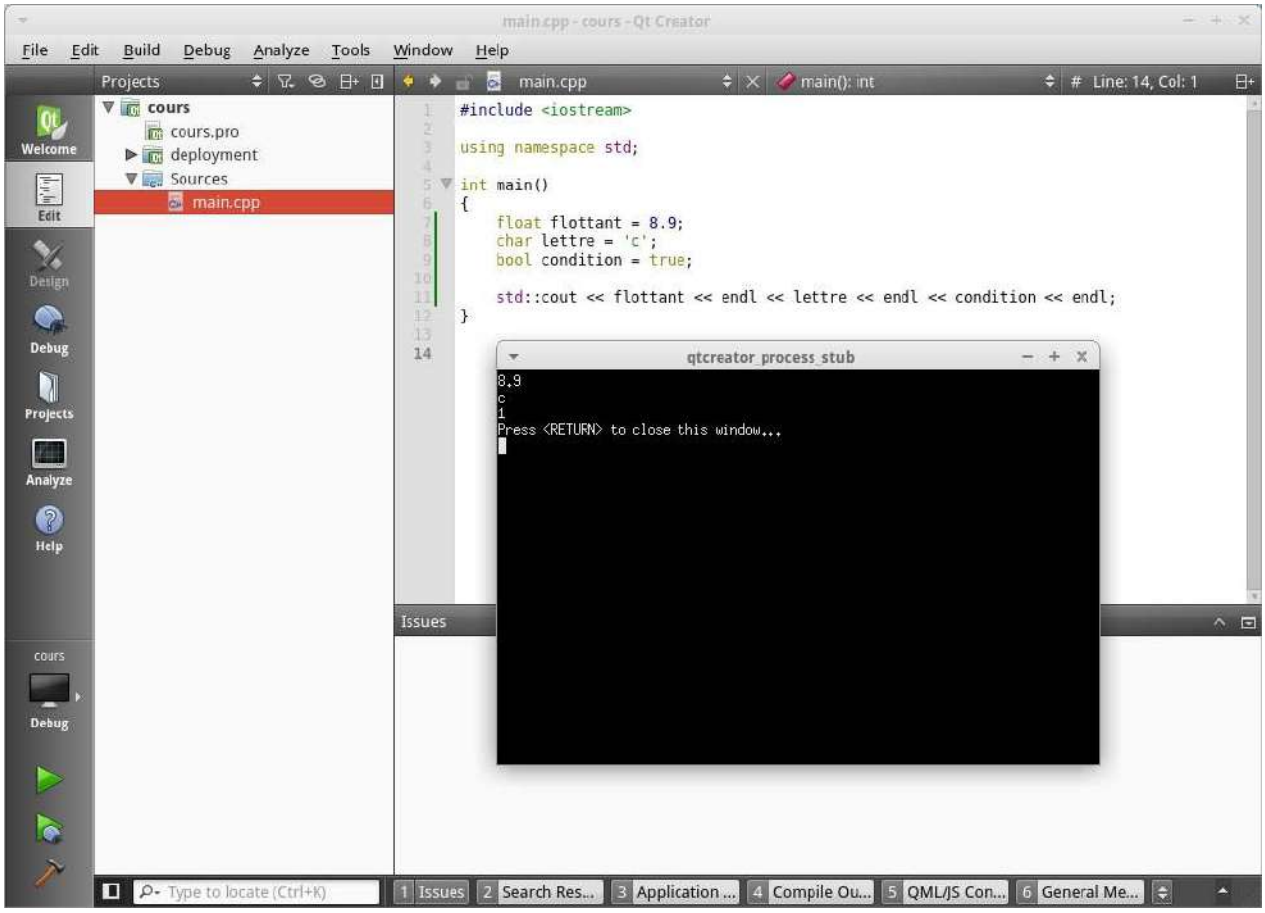
Voici les différents types de variables que vous pouvez utiliser (liste non exhaustive) :

Mot clé	Description	Taille en octet
long	Un très grand nombre entier	8
int	Un nombre entier	4
short	Un petit nombre entier	2
char	Représente un caractère ASCII ou un tout petit nombre	1
float	Représente à nombre à virgule (un nombre flottant)	4
double	Représente un nombre à virgule avec une *très* grande précision	8
bool	Représente vrai (true) ou faux (false)	1

Ce ne sont que des types primitifs c'est à dire qu'ils sont inclus de base dans le C++ sans aucune manipulation supplémentaire. Notez que chacun de ces types supportent les nombres négatifs. Voici des exemples d'utilisation de ces types :

```
float flottant = 8.9;
char lettre = 'c';
bool condition = true;

cout << flottant << endl << lettre << endl << condition << endl;
```

Ce qui affichera :



Notez que le type `bool` affiche 1 s'il vaut `true`, sinon il affiche 0. Nous verrons par la suite comment utiliser un boolean pour établir des conditions.

Bien sur, nous pouvons utiliser nos variables avec des opérateurs. Ces opérateurs nous permettent de calculer un peu tout et n'importe quoi, les voilà :

Nom	Symbole	Exemple
Addition	+	`int x = 5 + 8; // 13`
Soustraction	-	`int x = 5 - 8; // -3`
Multiplication	*	`int x = 5 * 8; // 40`
Division	/	`int x = 10 / 2; // 5`
Reste de la division (modulo)	%	`int x = 10 % 3; // 1`

Leur utilisation est très simple comme vous le voyez. Dans les exemples fournis, on calcule l'opération avec des valeurs fixes mais vous pouvez tout aussi bien mettre des variables du même type à la place des nombres.

Chacun de ces opérateurs a une variante. Au lieu d'écrire 30 lignes pour l'expliquer, je vais vous le démontrer avec un exemple :

```
int x = 5;
x += 8; // Revient à écrire x = x + 8;
x -= 8; // Revient à écrire x = x - 8;
x *= 8; // Revient à écrire x = x * 8;
//etc...
```

Se conclut ainsi le chapitre sur les variables. Gardez bien en tête que ce n'était qu'une très brève introduction et nous verrons les variables plus en détail plus tard. Pour l'instant je vous propose d'étudier un nouveau type, les `string` !

Le petit bonus

Et oui, même en programmation on a des surprises ! Après la syntaxe écourtée, je vous propose une syntaxe encore plus courte. Quand vous voulez incrémenter un nombre de seulement 1, vous devez écrire `x = x + 1;` ou encore mieux `x += 1;` mais il existe mieux ! Oui, vous pouvez écrire `++x;` qui aura exactement le même effet ! Ainsi voyez l'exemple suivant :

```
int x = 9;
++x;

cout << x << endl; // 10
```

Vous pourrez observer que `x` vaut bien 10.

> `x++;` fonctionne aussi mais a un comportement légèrement différent. Nous étudierons cette différence de comportement dans un futur chapitre.

Les chaînes de caractères string

Les strings

On vient juste de terminer le *premier* chapitre sur les variables et on enchaine sur... d'autres variables ! Et pour être plus précis sur les variables de type `string`.

Les `string` vous permettront de stocker des chaines de caractères c'est à dire des mots, des phrases ou même des romans entier. Leur utilisation est tout aussi simple que les autres type que nous avons vu.

> Par contre, attention ! Ce n'est *pas* un type primitif. Pour rappel un type primitif est un type qui existe de base en C++ sans la moindre action supplémentaire. Les `string` existent bien de base en C++ mais ont été codés en C++ avec les types primitif que nous avons vu précédemment.

Les inclusions

Parfois en C++, nous devrons inclure une nouvelle fonctionnalité ou un autre fichier que nous avons créé. Rien de plus simple, il suffit qu'en haut de votre fichier texte vous insérez la syntaxe suivante :

```
#include <UN FICHIER STANDARD>
#include "UN FICHIER DE VOTRE PROJET"
```

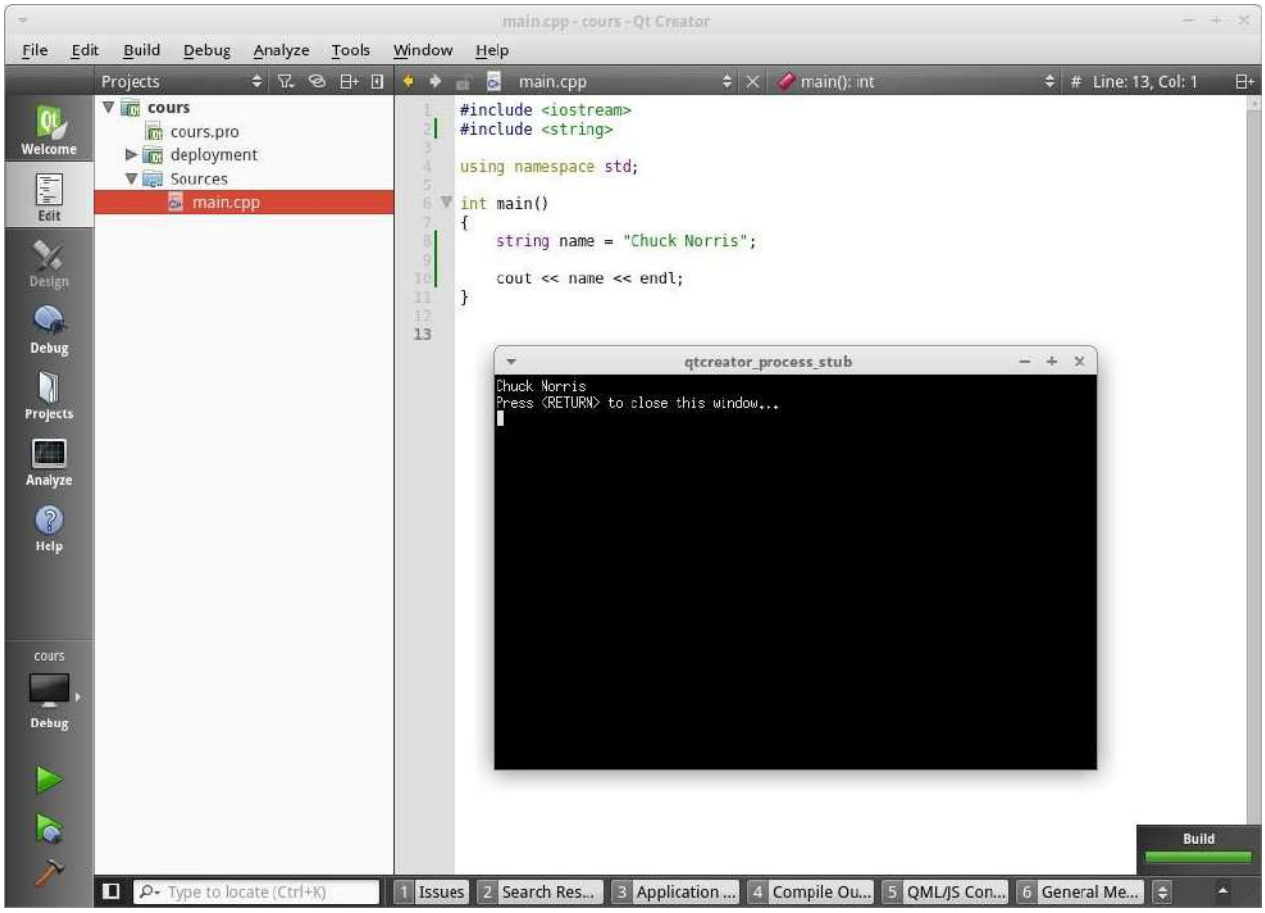
Par exemple, pour inclure les string dans notre fichier C++, nous écrirons : `#include <string>` mais pour inclure un fichier que nous créerons nous même, nous écrirons plutôt `#include "monfichier.cpp"`. Notez la différence ! Mais nous verrons comment fonctionne l'utilisation de plusieurs fichiers plus tard.

Comment utiliser les `string` ?

Rien de plus simple, prenez ce code et mettez le dans votre IDE.

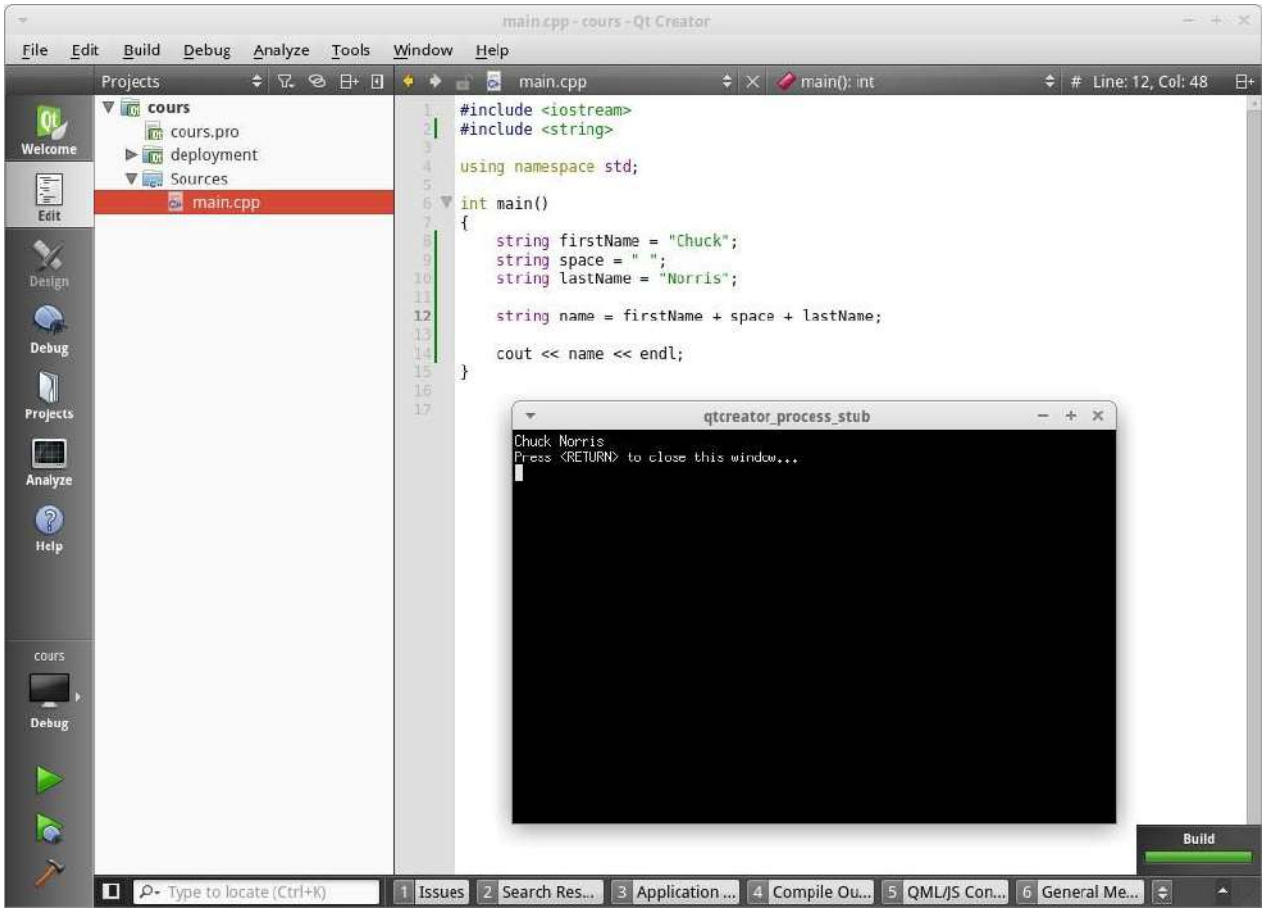
```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string name = "Chuck Norris";

    cout << name << endl;
}
```

Puis lancez l'application et que voyons nous ?



Et oui ! Chuck Norris s'affiche à l'écran, bienvenue dans la magie et la simplicité du C++ :-D !

Mais, fort heureusement, la puissance des `string` ne s'arrête pas là. Nous pouvons par exemple les concaténer, simplement en utilisant l'opérateur `+` entre deux `string`. Regardez plutôt :



Plutôt simple et efficace.

Je vais maintenant vous présenter une nouvelle notion. Vous devez considérer que les `string` sont un peu comme des grosses variables qui offrent pleins de possibilités. Par exemple, les `string` possèdent comme des informations, des propriétés qui peuvent être accédées via un `.` et avec des `()` à la fin !

Nous verrons le pourquoi du comment, contentez vous d'accepter que c'est une *méthode d'accès* à des informations ou à des actions. Par exemple pour connaitre la longueur de votre chaine de caractère, vous devez simplement écrire :

```
string name = "Martin";
cout << name.length() << endl;
```

Ici la notation est nouvelle. `length` veux dire en anglais *longueur*. On accède donc à la longueur de la chaine concernée via le `.` et on termine la propriété voulue par des `()` (vous comprendrez pleinement cette notation bientôt, ne vous inquiètez pas). Ainsi, écrire `name.length()` vous retourne la longueur de la chaine c'est à dire 6 dans notre cas.

Il existe d'autre méthodes comme `erase()` qui permettent de vider complètement la chaine et plein d'autres que nous verrons plus tard.

Attention : petite subtilité très importante ! Un `char` (un caractère) s'écrit avec des apostrophes alors qu'un `string` (une chaine) s'écrit avec des guillemets !

```
char letter = 'a';
string word = "Hi!";
```

La subtilité est très importante, sinon votre programme risque d'avoir des comportements indéfinis.

Dialoguons avec l'utilisateur !

Dialoguons avec l'utilisateur !

C'est bien beau de pouvoir afficher toute sorte de choses dans la console, mais maintenant nous aimerions dialoguer avec l'utilisateur.

Par exemple, demandons son nom, demandons ce qu'il fait aujourd'hui, on peut demander absolument tout ce que l'on souhaite !

> Ce chapitre reste assez basique et ne traitera pas de l'intégralité de ce sujet.

Il existe deux méthodes pour réceptionner des informations depuis l'utilisateur dans la console.

La première méthode

Nous avons utilisé jusque ici le fameux `cout` (qui pour rappel se prononce *c-out* : Console OUT). Automatiquement par analogie nous allons découvrir le `cin` (*c-in*, Console IN). Voici un exemple d'utilisation tout bête :

```
#include <iostream>

using namespace std;

int main()
{
    int revenu;

    cout << "Quel est votre revenu mensuel ?" << endl;
    cout << "Mon revenu mensuel est ";
    cin >> revenu;

    cout << endl << "Eh beh, " << revenu
         << " euros, c'est un grand revenu !" << endl;
}
```

Ici, deux découvertes. Alors oui, on peux décomposer un `cout` sur plusieurs lignes comme je viens de le faire.

> N'oubliez pas, c'est le `;` qui délimite la fin d'une instruction (dans le cas présent c'est l'instruction `cout`). Sauter une ligne ne délimitera jamais une instruction !

Ensuite, on peut voir le fameux `cin`. Rien de plus simple vous le voyez, il suffit d'écrire `cin` avec les chevrons dans le sens opposé de `cout` (car le flux va dans l'autre sens). Essayez ce programme chez vous, rentrez un nombre lorsqu'il vous sera demandé et voyez la magie opérer ! Profitons en pour demander l'âge de l'utilisateur. Regardez plutôt :

```
int revenu;
int age;

cout << "Bonjour ! Quel est votre age ?" << endl;
```

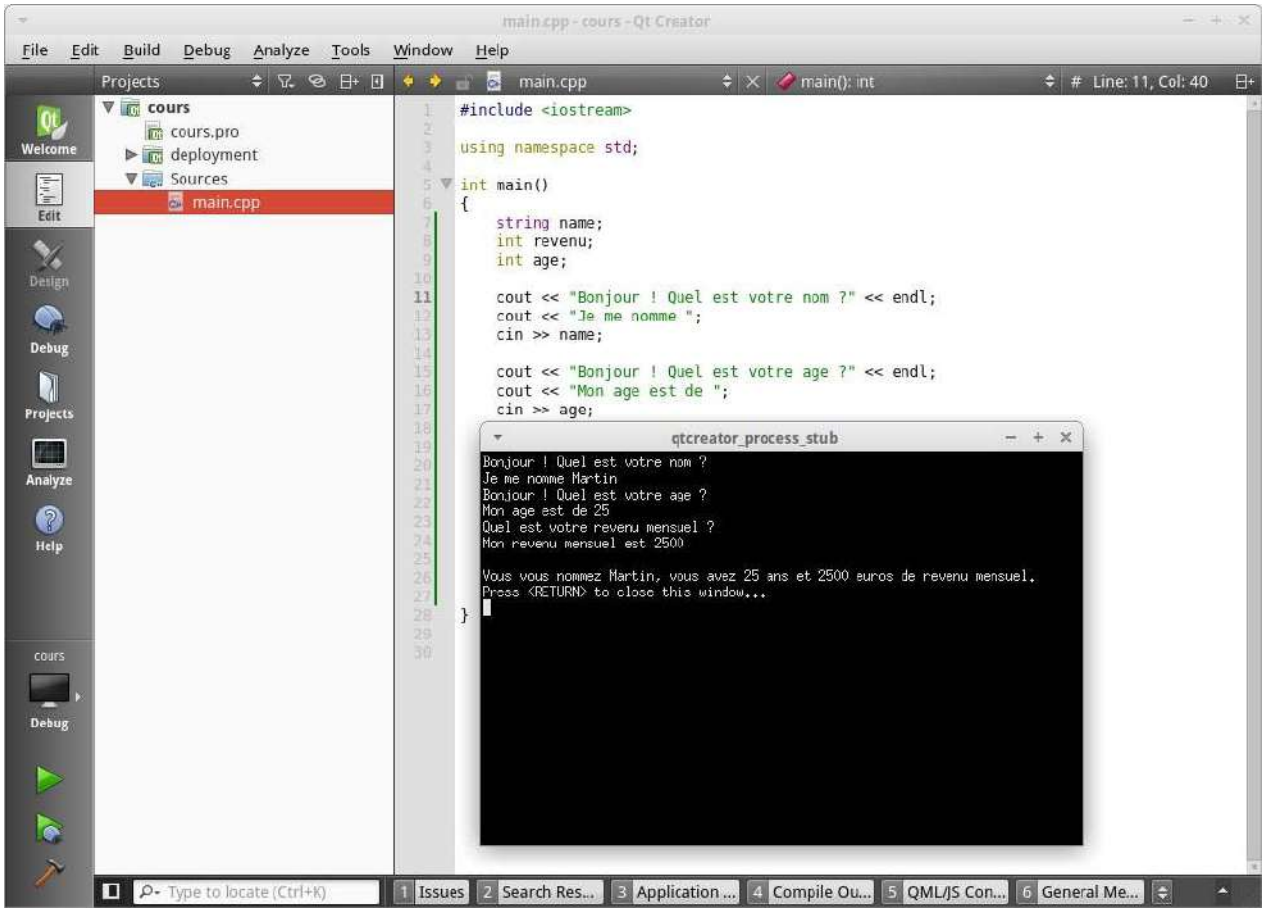
```
cout << "Mon age est de ";
cin >> age;


cout << "Quel est votre revenu mensuel ?" << endl;
cout << "Mon revenu mensuel est ";
cin >> revenu;

cout << endl << "Vous avez " << age << " ans et " << revenu << " eur
```

Nous ne verrons pas les cas où l'utilisateur n'entre pas les valeurs voulues.

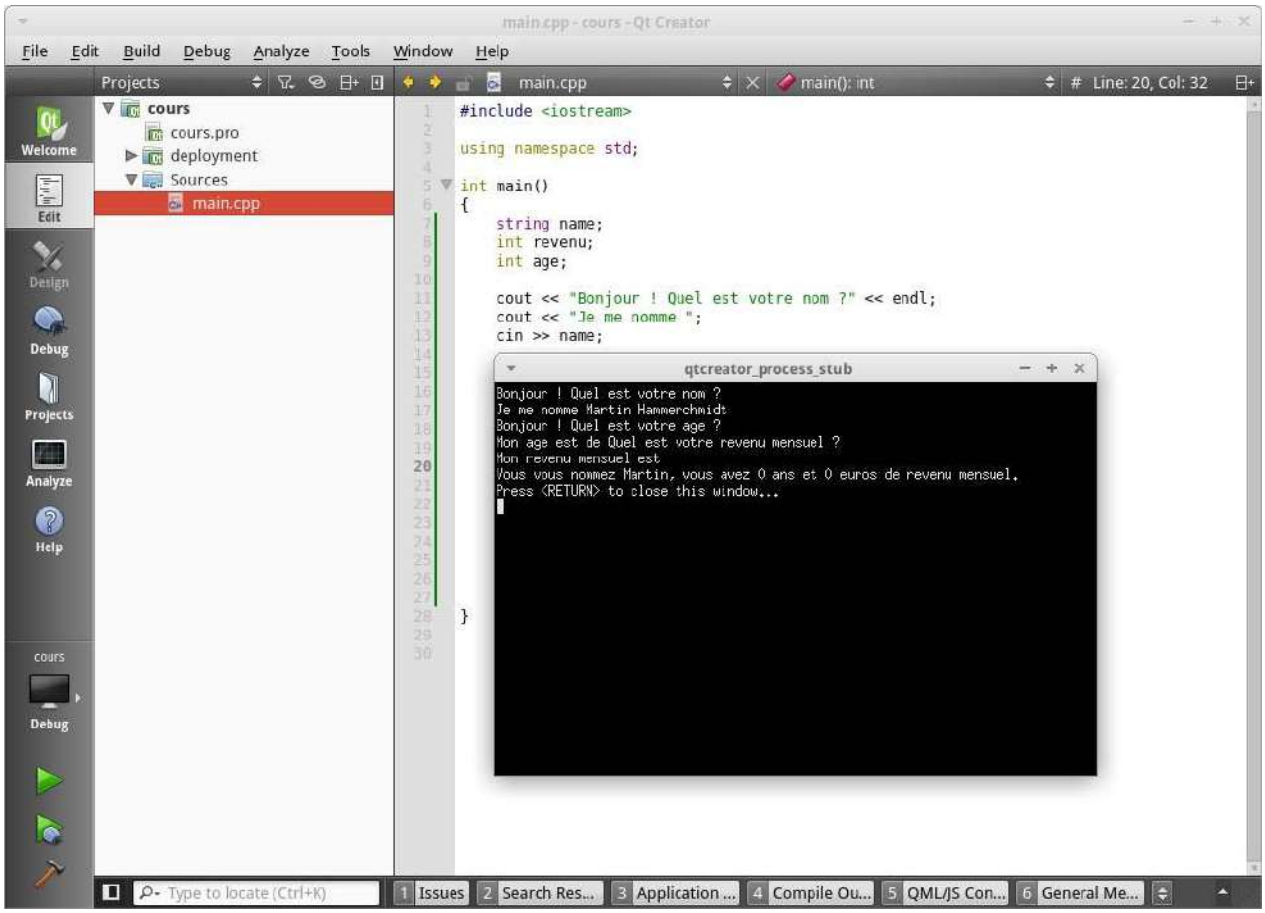
Maintenant, demandons lui son nom et son prénom. Rien de plus simple à première vue, rajoutons des éléments à notre code :

```
string name;
int revenu;
int age;

cout << "Bonjour ! Quel est votre nom ?" << endl;
cout << "Je me nomme ";
cin >> name;

cout << "Bonjour ! Quel est votre age ?" << endl;
cout << "Mon age est de ";
cin >> age;


cout << "Quel est votre revenu mensuel ?" << endl;
cout << "Mon revenu mensuel est ";
cin >> revenu;

cout << endl << "Vous vous nommez " << name
     << ", vous avez " << age
     << " ans et " << revenu << " euros de revenu mensuel."
     << endl;
```

Lancez le programme et entrez par exemple juste votre prénom. Tout fonctionne parfaitement !



Mais que se passe-t-il si vous insérez votre prénom *et* votre nom ?



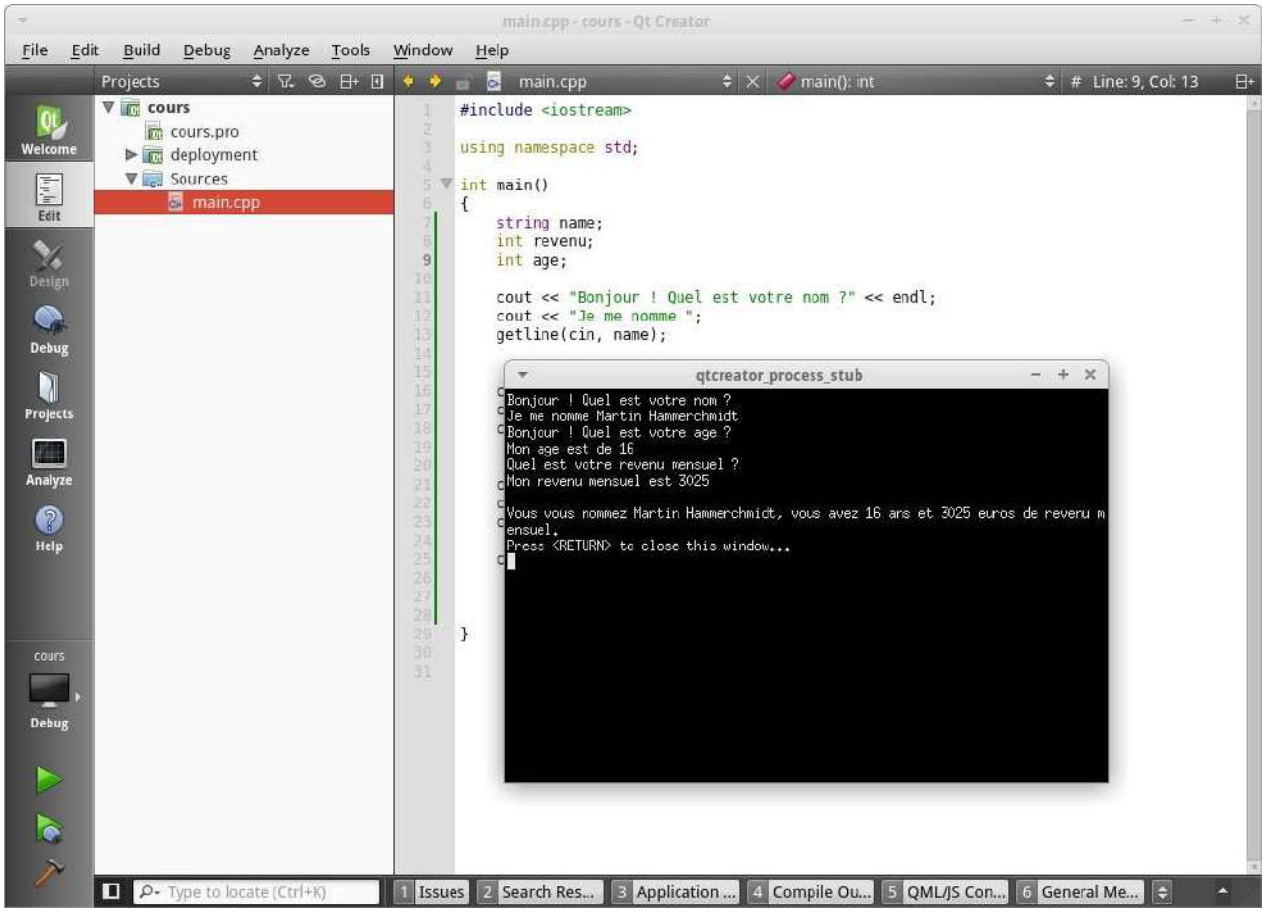
Le logiciel part en fanfare ! En effet il semble ne pas prendre en compte le second mot que vous avez entrer et ignore complètement les autres `cin`. La raison à cela est simple : `cin` se limite à un mot par un mot. Et du coup, comme il garde un mot un réserve (votre nom que vous avez mis après votre prénom) il attend que vous lui demandiez de remplir un `string` pour vider son *buffer* hors ce n'est pas ce que nous voulons. Nous voulons récupérer plusieurs mots et donc une ligne en quelque sorte.

La seconde méthode

Fort heureusement, le C++ nous propose une fonction juste pour ça ! Je vous présente `getline`. Comme toujours, l'utilisation de cet outil reste assez simple. En effet la syntaxe est `getline(cin, VARIABLE);`. Ainsi, voici notre exemple précédent corrigé :

```
string name;
int revenu;
int age;

cout << "Bonjour ! Quel est votre nom ?" << endl;
cout << "Je me nomme ";
getline(cin, name);


cout << "Bonjour ! Quel est votre age ?" << endl;
cout << "Mon age est de ";
cin >> age;


cout << "Quel est votre revenu mensuel ?" << endl;
cout << "Mon revenu mensuel est ";
cin >> revenu;

cout << endl << "Vous vous nommez " << name
     << ", vous avez " << age
     << " ans et " << revenu << " euros de revenu mensuel."
     << endl;
```

Et hop ! Tout fonctionne correctement. Comme vous le voyez, `getline` met automatiquement ce que l'utilisateur a entré dans la variable spécifié comme `cin` .



Voilà. Ce chapitre reste vraiment basique car d'autres notions sont nécessaires pour aller plus loin. Mais ne vous inquiétez pas, nous verrons tout au fur et à mesure. Entamons maintenant les **conditions** !

Les conditions

Les conditions

Allez on garde du courage, on ne désespère pas ! Ce chapitre et le chapitre suivant seront très simples !

Jusqu'ici, notre programme était cantonné à un déroulement fixe, et prévisible. Si seulement nous pourrions rajouter comme des modules qui s'exécutent seulement et seulement si une condition x est vérifiée...

La structure générale

Mais rassurez vous, c'est possible en C++ ! Et c'est même enfantin. Les conditions (ou *structures de contrôle* techniquement parlant) sont un composant que l'on retrouve dans absolument tous les langages de programmation, à ma connaissance.

Voici leur syntaxe :

```
if(condition)
{
    // ...
}
else if(condition2) // facultatif
{
    // ...
}
else // facultatif
{
    // ...
}
```

Si vous êtes un tantinet anglophone, vous comprendrez aisément le principe. Les conditions en C++ se forment avec trois différents blocs. Le premier bloc est obligatoire et détermine le début d'une structure de contrôle. C'est ce `if` qui signifie *si* en anglais.

Son utilisation reste simple, vous écrivez `if` puis entre parenthèses vous mettez votre condition (par exemple "si l'utilisateur à 3 ans") et enfin entre les accolades `{` et `}` vous écrivez le code à exécuter en cas de succès.

Concrètement, la condition est une *expression* qui peut être plein de choses (nous allons voir juste après). Si elle est évaluée comme étant **true**, le code entre accolades sera exécuté, sinon l'expression sera évalué comme étant **false** et le code entre accolades ne sera pas exécuté.

Premièrement, cette expression peut être une comparaison entre deux valeurs, selon le tableau suivant :

Signe	Exemple	Correspondance
`==`	`if(3 == 4)`	Egal à
`!=`	`if(3 != 4)`	Différent de

`<`	`if(3 < 4)`	Inférieur à
`<=`	`if(3 <= 4)`	Inférieur ou égal à
`>`	`if(3 > 4)`	Supérieur à
`>=`	`if(3 >= 4)`	Supérieur ou égal à

Ainsi en pratique voici plusieurs exemples :

```
if(3 == 3)
{
    cout << "3 est bien egal a 3" << endl;
}

// On peut aussi utiliser des variables

int a = 3;
int b = 4;

if(a == b) // Ne s'exécutera pas
{
    cout << "a et b sont identiques" << endl;
}
else if(a != b) // S'exécutera
{
    cout << "a est different de b" << endl;
}

// Le else if permet de créer un nouveau bloc si le bloc précédent
// est invalide

if(a < b) // S'exécutera
    cout << "a est plus petit que b" << endl;
else if(a > b) // Ne s'exécutera pas
    cout << "b est plus petit que a" << endl;

// Une forme raccourcie est possible, sans les accolades mais uniquer
// pour insérer une seule instruction.

if(a == b) // Ne s'exécutera pas
    cout << "a et b sont identiques" << endl;
else // S'exécutera
{
    cout << "a et b ne sont pas identiques" << endl;
    cout << "Ceci est un else" << endl;
}

// Le else se lancera que si les précédentes conditions n'ont pas été
// validées.
```

Je n'utilise pas d'accents car la console Windows a, de base, un peu de mal à les afficher.

Les `bool`

On en avait parlé il y a quelque chapitres, revenons dessus. C'est un type assez particulier dans le sens

où il ne peut contenir que deux valeurs : `true` et `false`. Si vous mettez un `bool` qui vaut `true` dans une condition, celle-ci sera exécuté, sinon non. Un `bool` peut être utilisé de multiples manières, voyez ci-contre :

```
bool a = true;
bool b = false;
bool c = 0; // False
bool d = 1; // True
bool e = 80; // True
bool f = -80; // True
bool g = a != b; // True
bool h = a; // True
bool i = !a; // False
```

Et oui, regardez bien le dernier exemple ! En réalité, les opérateurs de comparaisons que nous avons eu plus haut retourne un booléen et vous pouvez donc stocker le résultat dans une variable si l'envie vous prend. Ainsi vous pouvez directement passer une variable `bool` à une condition si cela vous arrange. Et devinez quoi ! Automatiquement, puisque un nombre non nul se "transforme" en `true` et un nombre nul en `false`, vous pouvez directement passer un `int` ou un `long` ou ce genre de variables dans une condition !

Pratique non :-D ?

Les petits bonus

Car j'aime vous offrir des cadeaux et que je pense à vous, je vais vous offrir une astuce. En effet, vous pouvez mettre plusieurs expressions dans une condition avec les opérateurs logiques qui concatènent (un drôle de mot, oui oui) deux `bool` en un seul ! Voici un tableau les présentant :

Mot clé	Explication	Exemple
&&	Opérateur logique ET	`true && true // True` et `true && false // False`
||	Opérateur logique OU	false || true // True et false || false // False
!	Opérateur logique NON	`!true // False` et `!false // True`

Ainsi, par exemple, on peut les utiliser de la manière suivante :

```
bool a = true;
bool b = false;
bool c = true;

if(a && (b || c)) // Valide
    cout << "Condition 1 valide" << endl;

if(!a) // Invalide
    cout << "Condition 2 valide" << endl;

if(!b) // Valide
    cout << "Condition 3 valide" << endl;
```

Voilà vous savez à peu près tout sur les conditions, ne vous inquiétez pas s'il vous reste des zones d'ombre, nous les éclairerons dans les prochains chapitres au fur et à mesure de la pratique.

Les boucles

Les boucles

Parfois, dans la logique de votre application, vous aurez besoin de répéter une action x fois. Par exemple vous devez dire bonjour 30 fois. Vous n'allez pas écrire `cout << "Bonjour !" << endl;` 30 fois ! Ce serait du suicide pur et simple.

Non, les boucles permettent de faire cela automatiquement et super simplement. Il existe *grosso-modo* 3 types de boucles que nous allons voir ici.

La boucle `while`

La boucle `while` est clairement la plus simple. Elle fonctionne comme une condition (si si vous savez, le `if()`). Tant que l'expression sera valide (`true`), le contenu du `while` s'exécutera (le code entre le `{` et le `}`). Voici un exemple assez simple :

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string message;

    while(message != "je valide")
    {
        cout << "Vous devez ecrire 'je valide'." << endl;
        getline(cin, message)
    }
}
```

Ce programme va se contenter de demander à l'utilisateur de rentrer le texte "je valide". S'il ne le rentre pas, il va lui demander encore et encore.

> `while` en anglais se traduit ici par *Tant que*. On peut donc lire cette boucle par *Tant que* `message` *n'est pas égal à "je valide", affiche un texte et je remplis la variable* `message` *par le texte que l'utilisateur entrera.*

> *Mais attention !* Faites bien attention à l'expression que vous entrerez dans la boucle car si elle reste valide à l'infini (donc tout le temps `true`), vous serez dans une boucle infinie qui ne cessera jamais ! Elle tournera sans jamais s'interrompre. Votre expression, au bout d'un moment, devra évaluer à `false` au bout d'un moment afin de sortir de la boucle.

La boucle `do...while`

La boucle `do...while` est une variante de la boucle `while`. Observez ce code :

```
string message;

do
{
    cout << "Vous devez ecrire 'je valide'." << endl;
    getline(cin, message);
} while(message != "je valide");
```

La boucle `do...while` fonctionne exactement comme la boucle `while` à une différence près. En effet si l'expression de votre boucle évalue à `false` dès le début, la boucle ne sera jamais exécuté, pas même une seule fois. L'intérêt de la boucle `do...while` est de permettre à la boucle de s'exécuter au moins une fois quoi qu'il arrive. Par exemple :

```
while(false)
    cout << "Bonjour!" << endl;
```

Ici, *Bonjour!* ne s'affichera jamais à l'écran.

```
do
    cout << "Bonjour!" << endl;
while(false);
```

Alors que ici, *Bonjour!* s'affichera une seule fois.

Vous le voyez, j'ai aussi enlevé les accolades ici, si le code tient sur une seule ligne, vous pouvez les enlever. Mais vous n'êtes pas obligé, tout est question de goût.

La boucle `for`

Voyons directement un exemple :

```
for(int i = 0; i < 30; ++i)
{
    cout << "Bonjour!" << endl;
}
```

Comme vous le voyez, ce code dit *Bonjour!* un total de 30 fois. La boucle `for` est particulièrement adaptée pour faire un compteur, par exemple.

Cette boucle fonctionne en trois étapes :

- L'initialisation `int i = 0;` : on déclare le compteur, cette étape est effectuée une seule fois.
- La condition `i < 30;` : si c'est condition est fausse, la boucle s'arrête.
- L'action `++i` : on incrémente notre compteur. Mais on n'est pas limité à une simple incrémentation on peut faire `i += 2` par exemple. En fait, vous pouvez mettre n'importe quel instruction ici, mais juste une seule. La plupart du temps on va incrémenter un compteur. **Surtout ne mettez jamais de `;` pour cette troisième étape!**

Ainsi cette boucle est particulièrement adaptée pour effectuer une action x fois ou encore pour garder en mémoire le nombre d'itérations de la boucle nécessaires avant que celle-ci ne se termine. Voici un exemple :

```
#include <iostream>
#include <string>
```

```
using namespace std;

int main()
{
    int i; // Initialisation dans la boucle
    int init = 50;
    int limit = 675;

    for(i = init; i < limit; ++i);

    cout << "La difference entre " << limit << " et "
         << init << " est de " << i - init << "." << endl;
}
```

Cet exemple est assez compliqué, je vous propose de le mettre dans votre IDE et de l'analyser. C'est un algorithme assez simple, complètement inutile mais il présente bien le principe. Il calcule la différence entre deux nombres (`init` et `limit`).

Vous pouvez remarqué que je n'ai pas mis de bloc de code avec la boucle `for`, j'ai directement mis un point virgule. C'est aussi possible avec la boucle `while` : ça permet de simplement "attendre" que la boucle soit terminée sans rien faire (ou dans le cas de la boucle `for`, en faisant les calculs du compteur).

Le petit bonus

Comment faire si, lorsque nous sommes dans une boucle, nous souhaitons en sortir, **tout de suite** ? Eh bien je vous répondrai : **break;** ! En effet, si l'instruction `break;` est exécutée dans une boucle, le programme sortira aussitôt de la boucle pour continuer la suite du code. Essayez !

Les fonctions

Les fonctions

Les fonctions sont un autre élément majeur du C++. Jamais vous ne pourrez les éviter alors il vaut mieux les maîtriser sur le bout des doigts ! Mais, comme toujours, ne vous inquiétez pas. Les fonctions sont l'un des éléments du C++ les plus simples à comprendre et à utiliser !

A quoi sert une fonction ?

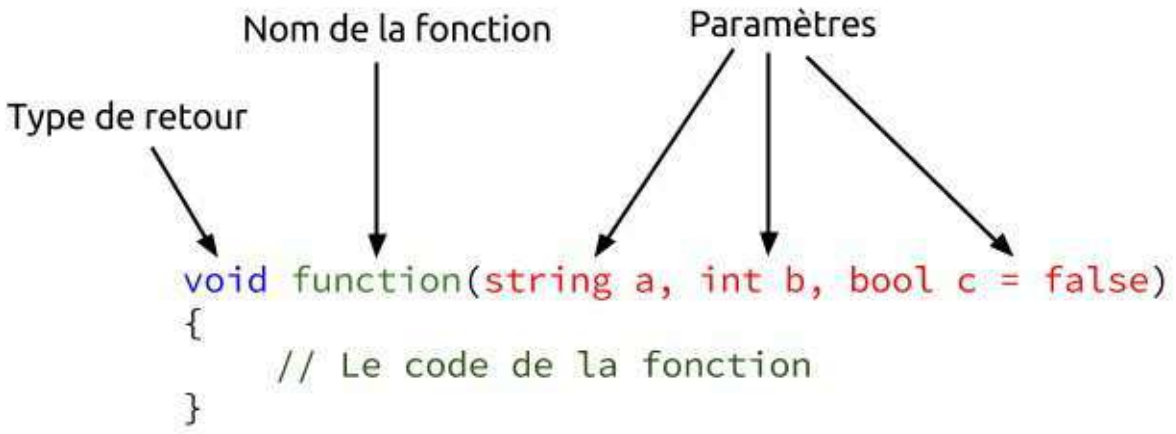
Il est impossible d'écrire tout son code dans la fonction `main()` car parfois vous devrez effectuer la même action 100 fois, 200 fois voir beaucoup plus ! De plus, la fonction pourrait faire plusieurs milliers de lignes de code, ce qui serait tout simplement illisible.

Une fonction va vous permettre de "stocker" du code pour pouvoir le relancer autant de fois que vous le souhaitez. Un peu comme une mallette que vous ouvrez dès que vous le souhaitez !

Comment fonctionne une fonction ?

C'est ici que les choses se compliquent. Pour créer une fonction, vous devez appliquer sa syntaxe hors de la fonction `main()`, avant la fonction `main()` pour être plus précis.

Voici comment se structure sa syntaxe :



Le nom de la fonction

Le nom de la fonction est très important car c'est avec ce nom que nous allons l'appeler. Par exemple, si vous nommez votre fonction `openWindow`, dans notre code nous pourrons l'appeler en écrivant l'[instruction](#) `openWindow();`. Choisissez le judicieusement, il doit représenter ce que fait la fonction !

Les paramètres

Parfois, dans la fonction, vous aurez besoin d'informations. Par exemple vous créez une fonction `sayHello` qui dit bonjour à la personne désigné. Pour désigner cette personne, la fonction doit connaitre son nom, et c'est ici que les paramètres interviennent. Les paramètres sont des variables qui sont passées dans la fonction et que la fonction peut utiliser, par exemple le nom d'une personne.

Pour désigner les paramètres, vous devez écrire le type du paramètre ainsi que son nom, ce qui aura pour effet de créer la variable correspondante automatiquement. Vous pouvez demander autant de paramètres que vous voulez, séparez-les simplement par des virgules. Par défaut, les paramètres sont obligatoires, mais si vous voulez les rendre optionnels, vous devez simplement les *initialiser* comme une variable en plaçant un `=` puis la valeur par défaut.

Le type de retour

Une fois l'exécution de la fonction terminée, vous pouvez retourner à l'appelant une valeur. Pour faire cela, vous devez spécifier le type de retour (`int`, `double`, `string`, ...). Ensuite, pour renvoyer la valeur acquise, vous devez clôturer votre fonction avec l'instruction `return [valeur à retourner];`. Par exemple `return 0;`. Attention, cette instruction terminera la fonction en cours, même s'il reste du code à exécuter par la suite.

> Si vous ne voulez rien retourner à l'appelant, spécifiez le type `void` qui signifie *vide* ou *rien* en anglais.

Des exemples !

Vous n'avez probablement pas pu assimiler tout ce que je viens de présenter, c'est normal. Voici des exemples que je vous propose d'analyser et de lancer chez vous.

```
#include <iostream>
#include <string>

using namespace std;

void sayHello(string name)
{
    cout << "Hello " << name << "!" << endl;
}

int main()
{
    sayHello("Martin");
    sayHello("Benjamin");
    sayHello("Chuck Norris");
}
```

Ce programme dira bonjour trois fois. Voici une variante avec les paramètres par défaut :

```
#include <iostream>
#include <string>

using namespace std;

void sayHello(string name = "outsider")
{
    cout << "Hello " << name << "!" << endl;
}

int main()
{
```

```
    sayHello("Martin");
    sayHello("Benjamin");
    sayHello();
    sayHello("Chuck Norris");
}
```

Et enfin, voici un exemple avec les types de retour :

```
#include <iostream>

using namespace std;

int addition(int x, int y)
{
    return x + y;
}

int subtraction(int x, int y)
{
    int result = x - y;
    return result;
}

int main()
{
    cout << "5 - 4 + 6 = ";

    int result = addition(subtraction(5, 4), 6);
    cout << result << endl;
}
```

Bon, on est d'accord, ces programmes sont purement théoriques et n'ont absolument aucune utilité. Je vous laisse comprendre par l'exemple.

La fonction spéciale `main()`

Cette fonction retourne un entier et ne prend pas de paramètres (enfin nous verrons ce cas particulier plus tard). Cette fonction est spéciale car votre OS l'appelle automatiquement quand vous voulez lancer votre programme C++.

> La fonction retourne un entier, mais par convention, on retourne 0 la plupart du temps, ou rien du tout ce qui n'est pas très correct en vérité. L'entier retourné permet à l'OS de savoir si l'application c'est bien déroulé ou non.

Nous étudierons cette fonction spéciale en long et en large plus tard.

Les espaces de noms

Les espaces de noms

Vous utilisez les espaces de noms depuis le début de ce livre sans même vous en rendre compte.

Que sont les espaces de noms ?

Imaginez que votre application est contenue dans des boîtes. Chaque boîte est un espace de nom et contient des fonction, des variables, enfin tout ce que vous voulez !

Les espaces de noms permettent de *classer*, de *ranger* le code dans des zones indépendantes les unes des autres. Une fois que vous avez rangé vos fonctions dans un espace de nom, pour pouvoir utiliser ces fonctions, vous allez devoir dire "Utilise cet espace de nom" (ou utiliser l'opérateur de portée). Et c'est la fameuse instruction `using namespace std;` !

L'espace de nom `std`

Avec le C++ est fournie une série d'outils plus ou moins utiles qui permettent de faire absolument plein de choses. Tous ces outils, pour ne pas "concurrencer" vos propres outils ou d'autres outils venant d'un tiers sont rangés dans l'espace de nom `std`.

> `std` est le diminutif de `standard`. Ce sont les outils *standard* du C++, ou de son vrai nom la *librairie standard*.

Les deux manières d'accéder à un espace de nom

Ainsi pour utiliser un espace de nom vous devez écrire `using namespace` suivi de son nom puis d'un `;` (car c'est une instruction !). A partir de ce point, tout le reste du fichier actuel aura un accès intégral aux fonctions de l'espace de nom que vous avez inclus. Par exemple `using namespace std;` ou `using namespace sf;` pour la SFML.

Mais, il existe une autre manière d'accéder à un composant d'un espace de nom, via *l'opérateur de portée*. Nous ne l'avons pas encore vu jusqu'ici. Voici un code qui devrait vous rappeler des souvenirs :

```
#include <iostream>

using namespace std;

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

Le fameux `Hello world` ! Nous allons donc supprimer l'instruction `using namespace std;` afin d'utiliser à la place l'opérateur de portée.

```
#include <iostream>
```

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

Comme vous pouvez le deviner, `cout` et `endl` sont deux composants de l'espace de nom `std` et sont inclus depuis le fichier `iostream`.

L'utilisation de l'opérateur de portée est simple. Voici la syntaxe : `ESPACE_DE_NOM::COMPOSANT`. Vous écrivez le nom de l'espace de nom suivi de deux doubles points (le fameux opérateur de portée). Une fois que l'opérateur de portée est inséré, vous vous retrouvez **dans** cet espace de nom le temps **d'une** instruction. Ce qui vous donne le loisir d'utiliser un composant de cet espace de nom, par exemple `cout` !

Ainsi l'utilisation de `cout` avec l'opérateur de portée se fait de cette manière : `std::cout`.

Nous étudierons la création d'espaces de noms dans un futur chapitre.

Les tableaux

Les tableaux

Imaginons que nous devons stocker une liste de nombres, comment faire ? Nous pourrions déclarer manuellement tous les entiers, mais comment faire si nous devons stocker une liste de 100 entiers ? On ne va pas écrire 100 fois `int nombre1 = 0; int nombre2 = 0;`, ce serait bien trop long.

Les tableaux sont là pour nous sauver ! Ce sont comme des variables spéciales qui peuvent contenir plusieurs variables, comme par exemple une liste de nombres, une liste de noms, ou une liste de ce que vous voulez !

L'étude des tableaux se fait en deux étapes : les tableaux statiques et les tableaux dynamiques.

Les tableaux statiques

Les tableaux statiques type C

Les tableaux statiques type C

En C++, il existe de nombreuses méthodes pour créer des tableaux, chaque méthode ayant ses particularités. Nous allons étudier 3 méthodes ici et dans les prochains chapitres, et nous découvrirons les autres prochainement.

Comme vous le savez, le C++ hérite du C. Tout ce qui existe en C existe aussi en C++ ! Cela doit vous faire une belle jambe vu que vous ne connaissez pas le C. Mais certaines notions du C sont tellement ancrés en C++ malgré tout qu'elles sont encore utilisés. Ici nous allons étudier les tableaux statiques, héritage du C.

> Nombreuses sont les personnes qui utilisent encore ces tableaux, et il n'est pas interdit de continuer à les utiliser. Nous verrons dans le prochain chapitre l'équivalent en C++ des tableaux statiques.

Comme je vous l'ai dit, les tableaux servent à stocker plusieurs valeurs dans une seule variable. Vous l'avez peut être remarqué, les variables que nous avons étudié jusqu'ici ne contiennent qu'une valeur : un nombre ou une chaine de caractère.

Ici nous allons découvrir comment stocker plusieurs nombres dans une seule variable.

La particularité des tableaux statiques (contrairement aux tableaux dynamique) est que leur taille est fixe. Si vous décidez d'y placer 42 éléments, le tableau aura **toujours** 42 éléments.

L'opérateur `[]`

Cet opérateur est l'élément clé qui va nous permettre d'utiliser et de créer des tableaux. Pour rappel, la syntaxe de définition d'une variable classique est :

```
TYPE NAME;
```

Désormais, pour créer un tableaux, la syntaxe sera :

```
TYPE NAME[SIZE];
```

Où `TYPE` est le type des éléments contenu dans le tableau, `NAME` le nom du tableau et `SIZE` la taille du tableau (le nombre d'éléments).

Notez les crochets qui entourent `SIZE` juste après le nom de la variable. C'est le fameux opérateur `[]`.

`SIZE` est une variable et doit être une variable *constante*. C'est une notion que nous n'avons pas encore vu, contentez vous de savoir qu'une variable constante ne peut être modifié et se crée de cette manière :

```
int const size = 8;
int x[size];
```

`SIZE` doit être constante car la taille du tableau est constante et que cette variable nous sera utile encore plus tard.

> Pour déclarer une variable constante, vous devez utiliser la syntaxe suivante : `TYPE const NAME;`.

L'initialisation !

Oui, comme une variable classique il ne faut pas oublier d'initialiser chaque éléments de notre tableau, sinon votre application subira des comportements indéfinis. Il existe deux méthodes pour initialiser notre tableau.

Avec la liste d'initialisation

C'est la méthode la plus simple mais la moins flexible, on peut utiliser la liste d'initialisation. Voici un exemple :

```
int const size = 3; // 3 éléments dans le tableau
int tableau[size] = { 0, 1, 2 };
```

Une liste d'initialisation pour des valeurs (car il existe d'autres type de liste d'initialisation) se construit de cette manière : on ouvre le `{`, on insère les valeurs séparés par une virgule puis on ferme le `}`.

> Si vous voulez initialiser tout votre tableau à la valeur par défaut (`0` pour un nombre), il suffit de ne mettre aucune valeur dans la liste d'initialisation (c'est à dire `int tableau[size] = {};`).

En itérant le tableau

Vous vous souvenez de la boucle `for` ? Oui, cette boucle basé sur un système de comptage ! Eh bien nous pouvons l'acceuillir à bras ouvert ici ! Elle est parfaitement adapté pour itérer à travers le tableau.

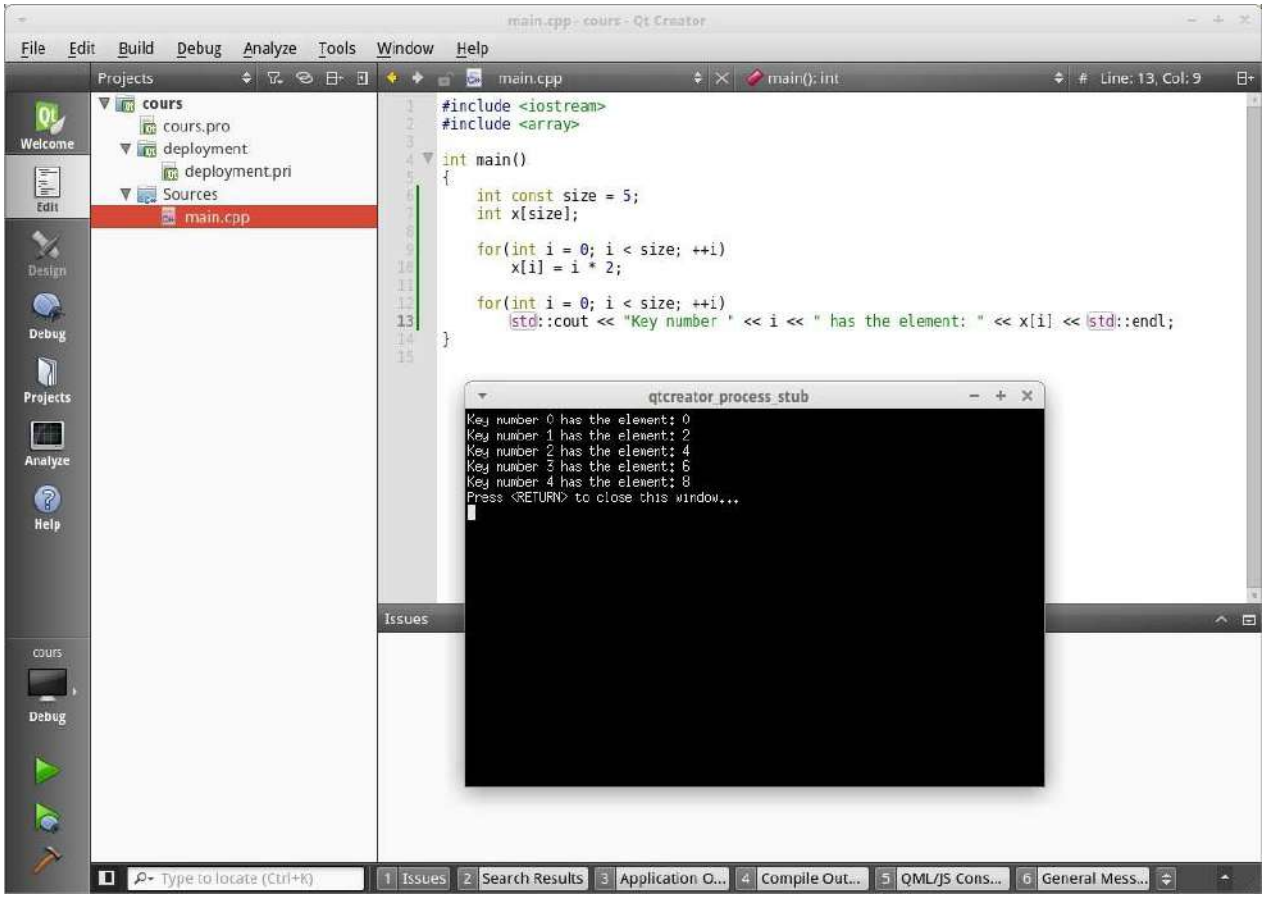
Itérer est un nouveau terme qui signifie **parcourir un tableau** dans le cas présent. Ici on va itérer un tableau d'entiers `x` qui contient `size` éléments afin de lui assigner une valeur. Voici la procédure avec la boucle `for` qui devrai vous être familière :

```
int const size = 5;
int x[size];

for(int i = 0; i < size; ++i)
    x[i] = i * 2;

// On effectue des actions, des calculs...

for(int i = 0; i < size; ++i)
    std::cout << "Key number " << i << " has the element: " << x[i] <
```

Le principe est simple ici. On crée un tableau `x` de taille 5. Ensuite on crée un compteur `i` qui est compris entre 0 et 5 exclus. Ce compteur `i` représente donc l'élément que nous sommes en train d'itérer. On accède à un élément avec l'opérateur `[]` encore une fois en indiquant entre les crochets l'index (le numéro) de l'élément que nous voulons.

Nous initialisons chacun des éléments du tableau à la valeur `i * 2` (0, 2, 4, 6, 8). Après on peut imaginer qu'on fait des calculs, des choses sur le tableau et enfin on affiche l'intégralité du tableau en

itérant dessus. Vous devriez obtenir ceci :



Et c'est ici que nous pouvons remarquer un détail **très important**. Le numérotage des éléments commence à **0** dans un tableau. Donc, si vous avez 10 éléments, ceux-ci seront numérotés de 0 à 9. L'élément numéro 10 n'existera pas.

Vous ne devez jamais utiliser un tableau avec un index supérieur à sa taille car votre application pourra (encore une fois) avoir des comportements indéfinis.

Accéder à un élément

Vous l'aurez compris, pour accéder à un élément dans le tableau il faut utiliser l'opérateur `[index]`. `index` peut être un nombre ou une variable représentant un entier. `index` ne peut bien sur pas dépasser la taille du tableau ou être inférieur à zéro.

Ainsi, voici plusieurs méthodes à suivre pour accéder à l'élément numéro 4 d'un tableau :

```
int const size = 10;
int x[size] = {};

cout << x[4] << endl;
cout << x[3 + 1] << endl;

int three = 3;
cout << x[three + 1] << endl;
```

Rien de bien compliqué, vous le voyez bien ! Maintenant, voyons ces même tableaux mais en version C++.

Les tableaux statiques C++11

Les tableau statiques en C++

C++11

Qu'est-ce que le C++11 ? Excellente question. Le C++ est un langage en constante évolution, et des *mises à jours* officielles sont régulièrement créés. La dernière mise à jour majeure est le C++11, sortie en 2011. Une autre mise à jour, mineure celle ci, a été établie en 2014, c'est le C++14. La prochaine grosse mise à jour qui apportera plein de nouveautés sortira en 2017 avec le C++17.

Le C++11 a apporté de très nombreuses choses, dont les `array`.

Notez que pour pouvoir profiter du C++11, il faut avoir des logiciels récents. En effet, seul les compilateur récents peuvent compiler un code C++11. Mais ne vous inquiétez pas, nous sommes aujourd'hui en 2016 et je doute que votre compilateur n'est pas compatible avec le C++11.

Les mêmes propriétés que les tableaux statiques du C

Les tableaux statiques du C++ proposent les même propriété que celle du C, avec des fonctionnalités en plus. Le nombre d'élément inclus au tableau est **fixe**, vous ne pourrez pas rajouter ou supprimer un élément. Le type d'élément que contient le tableau est **fixe** lui aussi, si vous y mettez des `int`, ce sera toujours des `int`.

> Les tableaux statiques du C++ s'utilisent de la même manière que ceux du C, mis à part les différences que nous allons étudier ici.

Comment les utiliser ?

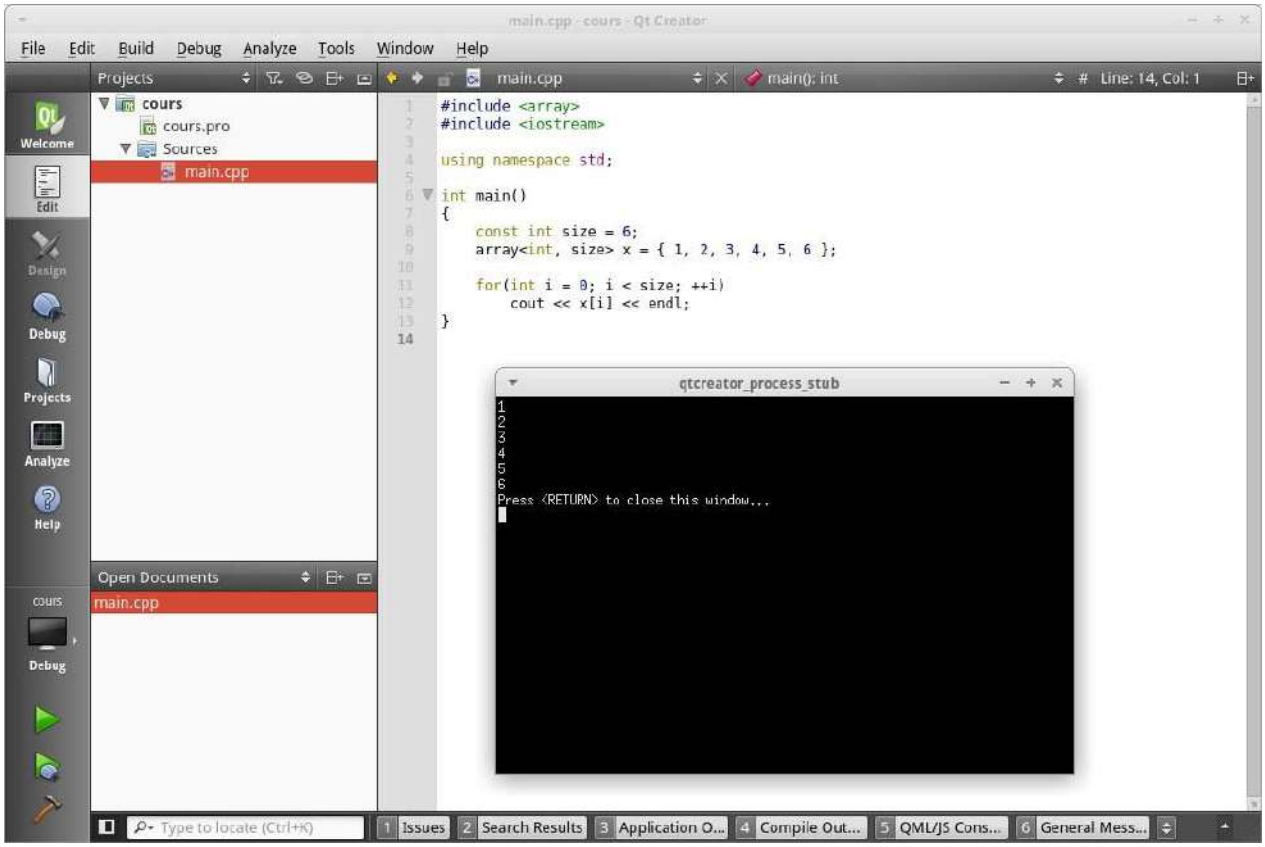
Tout d'abords, nous devons incluer un nouveau fichier, le fichier **array**.

```
#include <array>
```

Ensuite nous les utilisons comme suit :

```
#include <array>
#include <iostream>

int main()
{
    const int size = 6;
    array<int, size> x = { 1, 2, 3, 4, 5, 6 };

    for(int i = 0; i < size; ++i)
        cout << x[i] << endl;
}
```

Ce qui affichera, vous l'aurez deviné :



L'utilisation est la même que les tableaux en C. Par contre la création est différente, voici la syntaxe :

```
array<TYPE, SIZE> NAME;
```

On déclare tout d'abord une variable de type `array` donc insère le mot clé `array` au début et le nom de la variable à la fin, normal. Par contre, grosse subtilité ici, voici une nouvelle syntaxe que nous étudierons en détail plus tard mais que nous utiliserons souvent.

Juste après `array`, sans mettre d'espace vous devez ouvrir les chevrons et indiquer quel type le tableau contiendra puis quel sera sa taille. Le tout séparé par une virgule.

Ainsi cela nous donne par exemple :

```
array<short, 5> x;
```

Cela permet au tableau de savoir ses deux paramètres immuables qu'on ne peux pas changer une fois qu'il est créé : sa taille et son type.

Les bonus de ce tableau

Oui, ce tableau nous offre des bonus, en plus des tableaux statiques du C. Le membre `size()` vous permet d'avoir la taille du tableau.

```
array<int, 5> x = {};
cout << x.size() << endl;
```

Ceci affichera 5, par exemple. De la même manière, `front` et `back` vous permettent d'accéder respectivement au premier et au dernier élement.

```
array<int, 5> x = {};
```

```
int first = x.front();
int last = x.back();
```

Il existe d'autres choses comme ça, nous les verrons plus tard.

Mais ces tableaux ont un défaut, leur taille est fixe. Nous ne pouvons pas supprimer ou rajouter des éléments à la volé ce qui est un problème majeur tout de même. Donc après avoir étudié les tableaux statique, nous allons étudier les tableaux dynamiques.

Les tableaux dynamiques

Les tableaux dynamiques

La mise en bouche

Imaginez, vous êtes un développeur dans une jeune startup qui est en train de créer un jeu révolutionnaire. Dans ce jeu, vous contrôlez un personnage qui possède une seule et unique arme : un lanceur de boules.

Bien entendu, le joueur a la possibilité d'utiliser avec l'arme. Mais que ce passe t'il lorsque qu'il presse la détente ? Une boule est tirée. Bien entendu, chaque boule réagit à la physique et doit donc rester dans le monde virtuel.

Comment stocker les boules ? On ne peut pas utiliser un tableau statique car ceci ont une taille fixe hors notre joueur peut tirer autant de boules qu'il le souhaite. Il faut donc utiliser un tableau à taille variable. Ces tableaux sont des tableaux dynamiques.

Nous allons voir le tableau dynamique le plus simple, le `vector`. Il en existe d'autres, que nous verront par la suite.

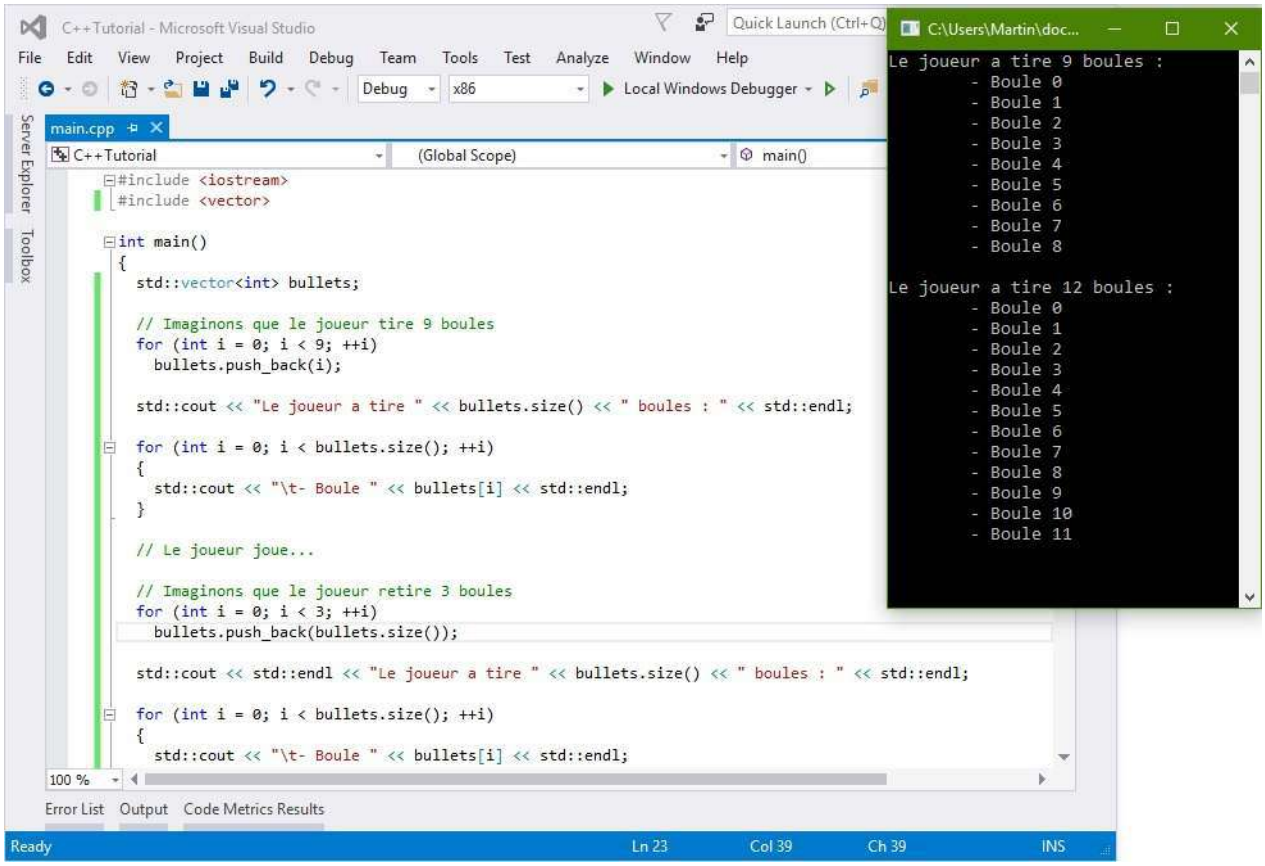
Le `vector` est très simple d'utilisation, voici un premier exemple déjà très complet :

```
#include <iostream>
#include <vector>

int main()
{
  std::vector<int> bullets; // La taille du tableau est 0

  // Imaginons que le joueur tire 9 boules
  for (int i = 0; i < 9; ++i)
    bullets.push_back(i);

  std::cout << "Le joueur a tire " << bullets.size() << " boules : "

  for (int i = 0; i < bullets.size(); ++i)
  {
    std::cout << "\t- Boule " << bullets[i] << std::endl;
  } // La taille du tableau est 9

  // Le joueur joue...

  // Imaginons que le joueur retire 3 boules
  for (int i = 0; i < 3; ++i)
    bullets.push_back(bullets.size());

  std::cout << std::endl << "Le joueur a tire " << bullets.size() <<

  for (int i = 0; i < bullets.size(); ++i)
  {
```

```
    std::cout << "\t- Boule " << bullets[i] << std::endl;
  } // La taille du tableau est 9 + 3 = 12
}
```

Nous n'avons bien sur pas un jeu vidéo ici, mais c'est tout comme. Imaginez donc que le joueur joue et que à deux reprise il tire des boules. La première fois, il tire 9 boules, et la seconde fois il tire 3 boules.

Voici le résultat dans la console :



Oui bon c'est vrai, j'ai changé d'OS et de logiciel. Mais chut :p

J'en convient, il y a, ici, beaucoup de nouvelles choses. Mais ne vous inquiétez pas, rien d'insurmontable.

Et dans les détails ?

Premièrement, comme toujours, on inclus la fonction que l'on souhaite :

```
#include <vector>
```

A partir de maintenant, je vais appeler le `vector` un vecteur.

La création et l'initialisation d'un vecteur est tout à fait similaire à un tableau statique en C++. Voici la syntaxe :

```
vector<TYPE> NAME;
```

Contrairement à ce que nous avons vu jusqu'ici, pas besoin de spécifier de taille ! Et oui, ce tableau a une taille variable et est donc vide à la base il n'attend qu'à être rempli.

Entre les <>, vous devez insérer le type que contiendra le vecteur, et, pour rappel, il ne pourra enregistrer que ce type précis de données.

Voyons maintenant ce qu'il se passe dans le code étape par étape. Premièrement, on simule le joueur qui tire 9 boules. Pour ce faire ont fait une boucle comme nous avons vu dans un précédent chapitre, qui ajoute 9 boules dans le tableau. Ici, les boules sont en fait des `int`, représentant le numéro de la boule.

Une fois fait, on fait le point dans la console et on affiche le nombre de boules puis on itère le tableau afin d'afficher le numéro de chacune des boules.

> Le `\t` dans la chaîne de texte que nous affichons représente simplement une tabulation. En effet, lorsque la console affiche ce caractère, selon les paramètres du système, elle effectue une tabulation plus ou moins grande. C'est pratique pour hiérarchiser les informations.

La fonction `size()` permet d'obtenir sa taille. Utilisez la comme suit :

```
int size = bullets.size();
```

Ainsi, `size` aura comme valeur le nombre d'éléments que contient le tableau.

Les fonctions push_back() et pop_back()

La fonction `push_back()` est le coeur même d'un vecteur. En effet, cette fonction permet d'ajouter un élément au tableau. Cet élément sera positionné à la dernière place, c'est à dire :

```
int a = 5;
int b = 10;

vector<int> v;

v.push_back(a);
// a est à la position v.size() - 1, soit à la position 0

v.push_back(b);
// b est à la position v.size() - 1, soit à la position 1
```

Au contraire, la fonction `pop_back()`, supprime le dernier élément.

La liste d'initialisation

Voici une autre syntaxe pour initialiser un vecteur :

```
vector<int> v = { 1, 2, 3, 4, 5 };
v.size(); // 5
```

Comme vous le voyez, on peut déclarer un vecteur puis l'initialiser avec des valeurs de manière assez élégante. La syntaxe est la suivante :

```
vector<TYPE> NAME = { VALUES };
```

Vous savez déjà remplir `TYPE` et `NAME`. L'ajout cette fois ci est `VALUES`, où vous pouvez insérer toutes les valeurs que vous voulez (au moins une), séparé par une `,`.

Vous pouvez aussi écrire `vector<int> v = {};`. C'est tout à fait légal (le compilateur l'accepte) mais est inutile car c'est la même chose que `vector<int> v;`.

Les manipulations d'un vecteur

Un vecteur peut être manipulé de très nombreuses manières. Nous allons voir celles que nous avons pas déjà vu.

La fonction `at()`

Il existe deux moyens d'accéder à un élément avec un vecteur. Vous pouvez soit utiliser l'opérateur `[]`, de cette manière :

```
vector<int> v = { 1, 2, 3, 4, 5 };
cout << v[3] << endl;
```

Ce qui affichera `4`. Mais que se passe t'il si le nombre que vous entrez entre les `[]` est trop grand ? Soit votre application plante, soit elle va avoir des comportements indéterminés.

Pour pallier ce problème, nous avons la fonction `at()` qui s'utilise comme suit :

```
vector<int> v = { 1, 2, 3, 4, 5 };
cout << v.at(3) << endl;
```

Cela affichera aussi `4`. Vous devez mettre en paramètre (c'est à dire entre les parenthèses) le numéro de l'élément que vous souhaitez accéder. Si vous mettez un nombre trop grand ici, l'application ne plantera pas, une exception sera émise. Nous n'avons pas encore vu la gestion des exceptions donc contentez vous de retenir que les `[]` sont très dangereux si vous mettez un nombre trop grand alors que la fonction `at()` est sécurisée.

Les fonctions `front()` et `back()`

Ces deux fonctions sont très simples. La fonction `front()` permet d'accéder au premier élément, alors que la fonction `back()` permet d'accéder au dernier élément.

Les fonctions `empty()` et `clear()`

Au lieu d'utiliser une condition avec la fonction `size()` pour savoir si le vecteur est vide ou non, vous pouvez utiliser `empty()`. Ces deux codes produiront la même chose :

```
vector<int> v;
cout << v.size() == 0 << endl; // Affichera 1

cout << v.empty() << endl; // Affichera 1
```

Un peu par analogie, la fonction `clear()` va se contenter de vider tout le tableau. Faites très attention à son usage, son action est irréversible !

Beaucoup d'autres choses sont possibles avec les vecteur. Mais nous les verront dans un prochain chapitre ! Maintenant, il est temps pour voir de découvrir la vérité sur les variables ! A l'abordage moussaillons !

Encore plus sur les variables

Encore plus sur les variables

Les variables 2

Les variables, la suite

Les variables sont le pilier central du C++. Alors, vous vous en doutez nous n'aurions pas pu rester avec un seul chapitre sur les variables ! Non, je vais vous torturer avec 4 chapitres d'affilé sur les variables !

Courage moussaillon, je crois en vous !

Les allocations mémoires

Nous allons maintenant entrer dans un chapitre d'avantage technique. Nous n'écrirons que très peu de code ici !

Comme vous le savez, une variable est une information qui a une taille (en octet) et qui est stocké dans l'un des supports suivant :

- La mémoire vive (aussi nommé la RAM) (prioritaire)
- Les registres CPU (aussi nommé le cache CPU, L1, L2 et L3)
- Le SWAP (uniquement et seulement si la quantité de RAM n'est pas suffisante)

Le SWAP

Les utilisateurs de Linux connaissent d'avantage la notion de SWAP que les utilisateurs Windows. Mais, tous les systèmes modernes (Linux, Mac OS et Windows) connaissent et gèrent le SWAP à merveille.

Comment faire si votre ordinateur n'a plus de mémoire vive disponible ? Nous ne pouvons pas empêcher les nouvelles allocations mémoire, cela planterai le système. Nous ne pouvons pas libérer de mémoire, cela planterai aussi le système.

Non, la seule solution est de trouver de l'espace supplémentaire. Et cet espace est le SWAP, réservé dans le disque dur de votre ordinateur. Si votre OS ressent le besoin de l'utiliser, il l'utilisera comme de la mémoire vive. Dans votre programme, cela ne fait aucun changement ! En pratique, cela réduira juste le temps de réaction de l'ordinateur (car l'accès au disque dur est plus long que l'accès à la RAM) et cela pourra aussi saturer le taux de lecture/écriture du disque dur.

Le registre CPU

C'est ici un domaine que je ne maîtrise que *très peu* voir même pas du tout. Mais sachez toujours qu'en interne votre CPU crée pleins de mini-variables intermédiaires et que ces variables sont stockés dans le registre CPU. Ces données ne restent pas plus de quelques millisecondes, le temps que le CPU l'utilise et les détruise.

Aussi, si dans votre code vous créez une variable que vous utilisez et supprimez juste après, alors, *peut-être* votre CPU l'enregistrera dans son registre. Tout dépend de votre CPU, de votre compilateur et de l'environnement d'exécution.

La durée de vie des variables

Voici un sujet épineux. Ce sujet deviendra tabous lorsque nous commencerons les *pointeurs* !

Mais pour le moment, parlons du *scope* ou de la portée en français. Un scope se représente par une zone virtuelle dans votre code, généralement entre deux accolades ou imbriqués dans une instruction unique suite à une structure. Je m'explique :

```
int main()
{
  // Du code...
  // Encore du code...
}
```

Vous avez ici un superbe exemple d'un scope. Tout ce qui se trouve dans la fonction `main()` est exécuté dans le scope de `main()`.

```
int main()
{
    int x = 8;

    if(x < 10)
    {
      int y = 0;
      x = 5;
      // Du code...
    }

    // Du code...

    x = 9;
}
```

Ici un autre exemple. La variable `x` est créé dans `main()` qui est son scope. Par contre, `y` est aussi créé dans `main()`, mais son scope est la condition `if(x < 10)`. En effet, le scope d'une variable est la zone virtuelle la plus proche.

Savez vous ce qu'il se passe pour `y` quand le programme sors de la condition ? `y` est détruit. Il n'existe plus dans la mémoire et n'est plus accessible ailleurs.

Par contre, `x` est toujours utilisable après la condition, et est même utilisable dans la condition car le `if(x < 10)` est sous-jacent a scope de `main()`.

C'est ce que nous appelons la portée des variables. **Dès que le programme sors de la portée d'une variable, alors cette variable est détruite.**

Globalement, les différentes portées qui existent dans le C++, sont les fonctions, les conditions, et les boucles.

Les limites des variables et leur signatures

Voici pour rappel le tableau que je vous ai présenté dans le premier chapitre sur les variables :

Mot clé	Description	Taille en octet
long	Un très grand nombre entier	8
int	Un nombre entier	4
short	Un petit nombre entier	2

char	Représente un caractère ASCII ou un tout petit nombre	1
float	Représente à nombre à virgule (un nombre flottant)	4
double	Représente un nombre à virgule avec une *très* grande précision	8
bool	Représente vrai (true) ou faux (false)	1

Si vous êtes un peu curieux, vous vous êtes peut être posé la question jusque combien peut grandir un `long`, un `int`, un `short` ou les autres types. Avant de vous donner les limites pratique de ces différents nombre, je dois vous parler de la signature des variables.

Les signatures

Toutes les variables ne peuvent pas être signés, non. Seul les nombres peuvent être signés.

Un nombre **signé** est un nombre qui peut être négatif. Un nombre **non signé** peut uniquement être positif. Ainsi, pour un nombre non signé, il peut atteindre `2^n - 1` où `n` est le nombre de bits.

Prenons l'exemple du nombre entier `int`. Celui-ci a `4` octets, ce qui fait `4 * 8 = 32` bits (un octet = 8 bits). Ainsi, un `int` non signé peut aller jusque `2^32 - 1 = 4'294'967'295` ! Cela fait beaucoup tout de même !

> Nous soustrayons 1 dans le calcul car nous avons `2^n` combinaisons différentes. Le 0 étant inclus dans cette liste de combinaisons, nous devons alors le prendre en compte. Ce qui fait que la limites d'un nombre non signé est `2^n - 1` où `n` est le nombre de bits.

Les limites pour un nombre signé (et pouvant donc être négatif) sont :

- `-((2^16 / 2) - 1)`
- `(2^16 / 2) - 1`

Rien de compliqué, il suffit de couper la poire en deux, et d'en mettre une moitié avant 0 et l'autre après 0. Ainsi, voici un nouveau tableau listant les types numériques du C++ avec leurs limites :

Mot clé	Limites	Taille en octet
signed long	-9'223'372'036'854'775'807 jusque 9'223'372'036'854'775'807	8
unsigned long	0 jusque 18'446'744'073'709'551'615	8
signed int	-2'147'483'647 jusque 2'147'483'647	4
unsigned int	0 jusque 4'294'967'295	4
signed short	-32'767 jusque 32'767	2
unsigned short	0 jusque 65'535	2
signed char	-127 jusque 127	1
unsigned char	0 jusque 255	1

Comme vous le voyez nous atteignons de très gros nombre, au moins cela vous laisse la liberté de construire de grand nombre sans trop vous inquiéter des limites.

Quand vous créez une variable, si vous ne précisez pas sa signature, elle est signée de base.

Voici comment signer ou empêcher la signature d'une variable :

```
int a; // Nombre entier signé
signed int a; // Nombre entier signé
unsigned int a; // Nombre entier non signé
```

Si jamais une de vos variables dépasse l'une de ses limites, se produit alors ce que nous appelons un *overflow*. Cela ne va pas faire planter votre logiciel, votre nombre va simplement continuer d'évoluer en repartant du début ! Par exemple, si vous avez un `signed char` qui vaut `127` et que vous lui ajoutez `1`, alors il vaudra `-127`.

Les variables constantes

Vous êtes en train de créer un programme mathématique. Votre programme stocke la valeur de Pi. Vous le savez, Pi est une constante, nous ne devons donc pas la changer.

Comment faire pour préciser qu'une variable est constante en C++ ? C'est très simple, regardez :

```
const float Pi = 3.14;
```

Il suffit de préfixer le type par le terme `const`.

Cette fonctionnalité peut paraître inutile, mais que nenni ! En effet, le compilateur pourra effectuer certaines optimisations quand il sait qu'une variable ne peut pas être modifiée. De plus cela vous aide dans votre code, si vous faites une erreur et que vous essayez de modifier une constante, alors votre compilateur pour préviendra et vous pourrez régler l'erreur.

Ainsi, la règle est simple : dès que vous stocker une valeur qui ne change *jamais*, vous *devez* la déclarer comme constante.

Bon, c'est vrai. J'avoue. Il y avait beaucoup de technique dans ce chapitre ! J'espère que vous êtes toujours avec moi car nous allons maintenant parler d'adresse mémoires et de références ! Allons-y !

Les références

Les références

Les adresse mémoires

Avant de pouvoir utiliser les références, il faut comprendre un principe très important. Chaque donnée qui est enregistré en mémoire vive a une adresse.

Cette adresse est un nombre entier dont la taille varie selon votre OS, souvent représentée sous forme hexadécimale.

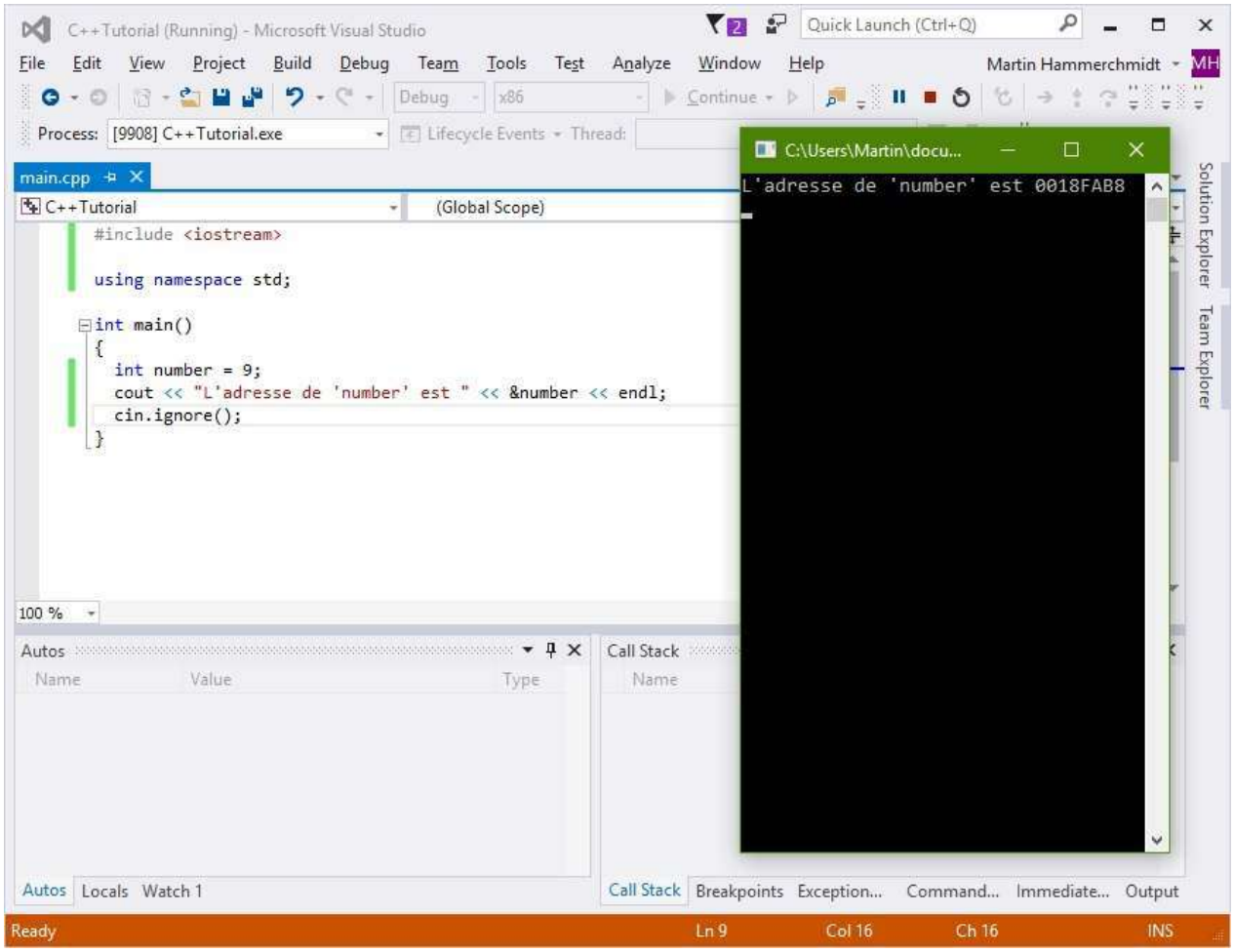
Ainsi, lorsque vous utilisez ou créez une variable, votre logiciel réserve un espace dans la mémoire et y enregistre les données. Ensuite, il peut accéder à cet espace à n'importe quel moment grâce à l'adresse.

Obtenir et utiliser l'adresse en C++

Vous vous en doutez, nous pouvons obtenir et manipuler cette adresse en C++. Voici comment obtenir l'adresse d'une variable :

```
int number = 9; // Nous créons la variable
cout << "L'adresse de 'number' est " << &number << endl;
```

Comme vous le voyez, pour obtenir l'adresse mémoire d'une variable il faut la préfixer d'une esperluette, une &. On appelle cela une référence.



D'autres manipulations avec les références :

```
int number = 9;
int &ref = number; // ref est une référence
cout << "Number vaut " << ref << endl;
```

En fait, il faut vous imaginer qu'une référence en C++ est un autre accès à une variable. En gros, il y a les variables maitres et les variables esclaves. Une variable normale (et non une référence) est maitre. Elle crée la donnée originale. Une référence est "esclave", elle représente juste un autre accès à la même donnée. C'est un alias en quelque sorte.

Vous vous rappelez quand je vous ai expliqué comment créer et utiliser des fonctions ? Nous allons étudier ça plus en détails.

Prenez cet exemple :

```
int add(int a, int b)
{
  return a + b;
}

int main()
{
  int a = 9;
  int b = 10;
  int c = add(a, b);
```

```
  // ...
}
```

Dans cet exemple, nous utilisons une fonction qui ajoute deux nombre et retourne le résultat (c'est inutile certes, mais c'est pour l'exemple). Que ce passe t'il en interne ?

- Le logiciel crée deux variables a et b et leur assignent une valeur.
- Le logiciel crée la variable c.
- Le logiciel copie les variables a et b dans deux nouvelles variables temporaire : a' et b'.
- Le logiciel additionne a' et b' et le stocke dans une nouvelle variable temporaire c'.
- Le logiciel supprime a' et b'.
- Le logiciel assigne à c la valeur de c'.
- Le logiciel supprime c'.

Comme vous le voyez, le défaut est que le logiciel crée plusieurs variables inutiles ici. En effet, il crée des variables temporaire pour a et b, qui sont inutiles.

L'explication à ce phénomène est simple : lorsque vous appelez une fonction, tout argument passé à cette fonction est copié, systématiquement. On ne peut pas faire autrement.

Alors, si nous passons de grosses variables comme cela, le logiciel risque d'avoir des problèmes de performances car à chaque fois il doit copier la variable originale.

Ainsi, la solution est de passer une référence. En effet, si vous passez une référence, l'adresse mémoire originale est passée, et au lieu de copier les données, la fonction utilisera la donnée originale directement.

```
int add(int &a, int &b)
{
  return a + b;
}
```

Et ici nous évitons alors la création de multiple variables inutiles. Quelle différence cela fait au niveau de la logique du code mis à par l'apparition d'une & dans la liste des paramètres de la fonction ?

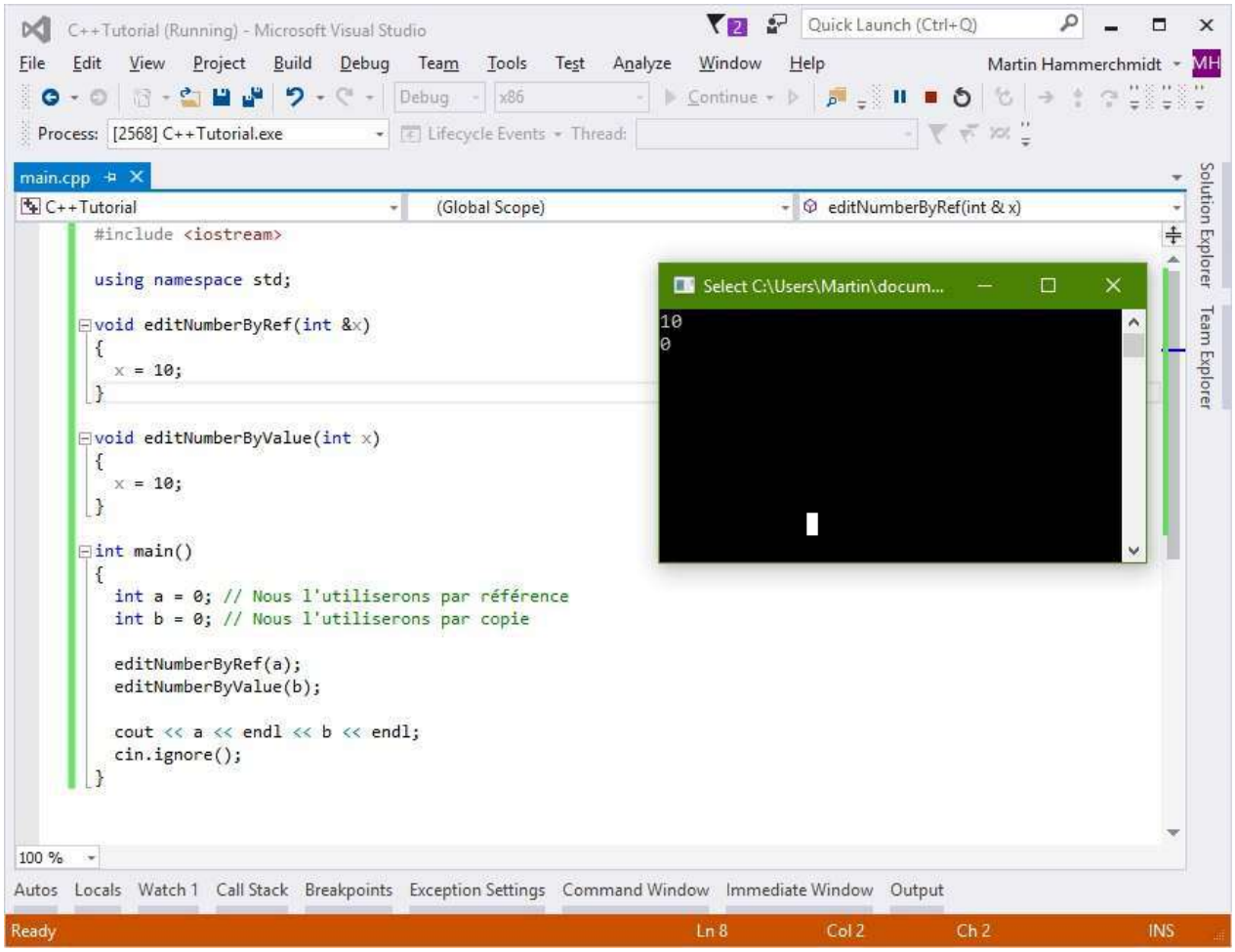
Si vous modifiez les paramètres passés par référence, alors ce sera aussi modifié pour l'environnement d'appel de la fonction puisque les deux environnements (la fonction et l'appel de fonction) utilisent la même variable.

Si vous modifiez les paramètres passés par valeur (eg. par copie directe), alors rien ne se passera pour l'environnement d'appel. Voyez cet exemple :

```
void editNumberByRef(int &x)
{
  x = 10;
}

void editNumberByValue(int x)
{
  x = 10;
}

int main()
{
```

```
  int a = 0; // Nous l'utiliserons par référence
  int b = 0; // Nous l'utiliserons par copie

  editNumberByRef(a);
  editNumberByValue(b);

  cout << a << endl << b << endl;
  cin.ignore();
}
```

Le résultat est :



Donc prenez bien garde à l'utilisation que vous faites de vos variables. Si vous les passez par copie, surveillez bien que la variable n'est pas trop grosse (un `int` par exemple c'est bon, un petit `string` aussi, mais un gros `string`, préférez le passer par référence). Si vous passez une variable par référence, faites très attention quand vous la modifier.

Voilà, vous savez à peu près tout sur les références. Maintenant, entamons l'un des sujets les plus redoutés du C et du C++ ! Les pointeurs !

Les pointeurs

Les pointeurs

Autant en C que en C++, les pointeurs sont... mythiques.

Beaucoup prennent peur et s'enfuient lâchement à la vu des pointeurs. Mais, la vérité sur les pointeur c'est qu'il faut bien les comprendre. Une fois assimilés et compris, vous verrez, c'est ultra-simple.

Le principe

On pourrais confondre les pointeurs avec les références. En effet, ils permettent tous les deux d'accéder à une donnée qu'ils ne possèdent pas.

Mais, leur fonctionnement est complètement différent. Avec les variables normales et les références, nous ne contrôlons pas nous même la mémoire. En effet, vous créez une variable normale et votre logiciel alloue une zone de la mémoire automatiquement pour vous. Si votre variable est détruite, alors la zone de la mémoire allouée l'est aussi instantanément.

Avec les pointeurs vous contrôlez vous même la mémoire. Et toute la difficulté est là ! Vous devez allouer et libérer la mémoire vous même, manuellement. Si vous allouez de la mémoire mais que vous oubliez de la libérer, vous aurez alors une fuite de mémoire.

Les fuites de mémoires sont mortelles ! Imaginez un logiciel qui consomme 4 Gb de mémoire vive alors qu'en réalité il n'en utilise que 200 Mb. Pourquoi ? Parce que il a alloué de la mémoire mais a oublié de la libérer. Vous comprendrez mieux après.

La pratique

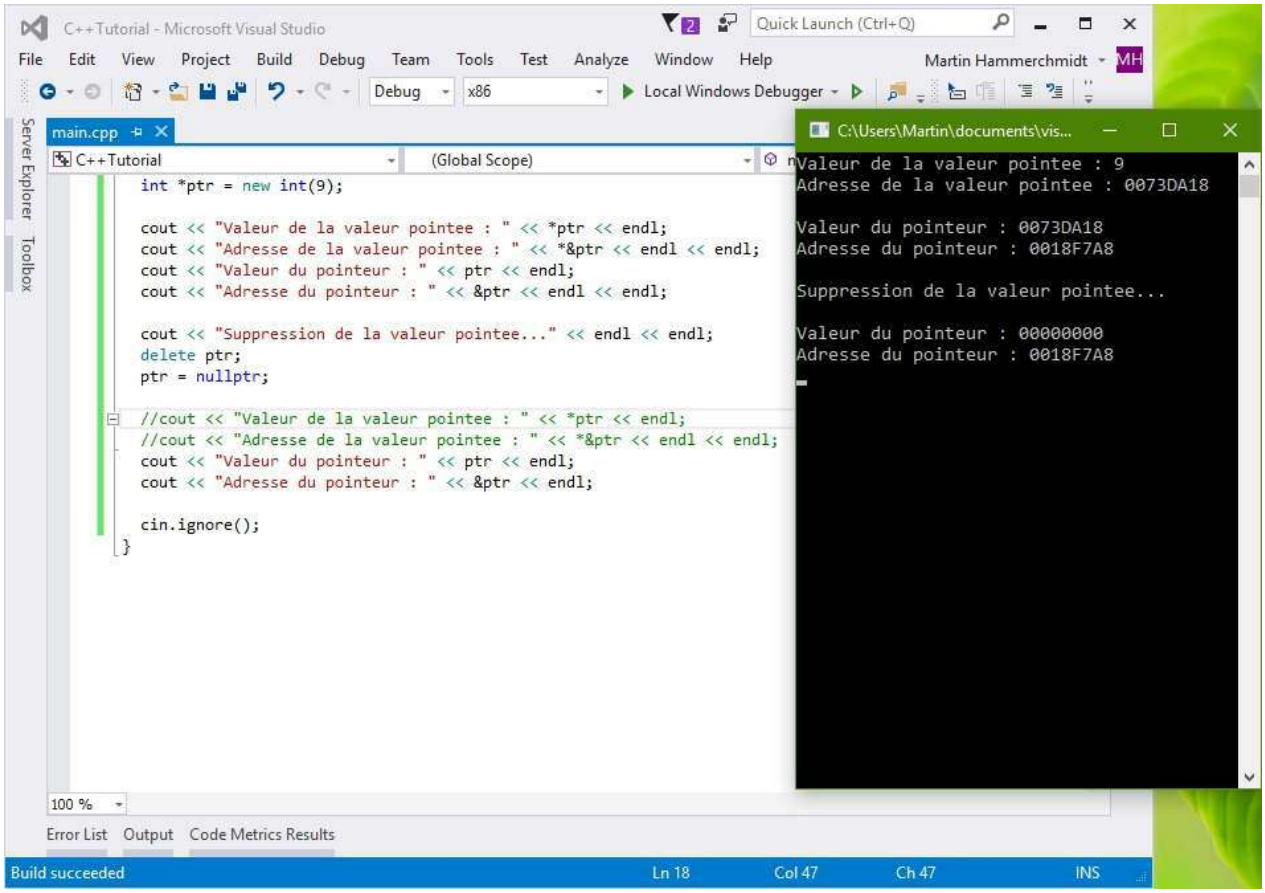
En pratique un pointeur est une variable à part entière. Plus en détail, c'est un nombre. Et ce nombre représente une adresse mémoire.

Je m'explique. Comme nous l'avons vu dans le chapitre précédent, toute variable dans la mémoire vive a une adresse qui lui est propre. Nous allons donc créer une variable dans la mémoire sans lui attribuer d'accès direct (donc pas de variable la représentant directement) : une sorte de variable fantôme. Bien sur cette variable aura une adresse mémoire, vous le savez. Nous allons donc récupérer cette adresse et la donner à un pointeur. Et le rôle d'un pointeur est de sauvegarder cette adresse. Un pointeur est donc une sorte de nombre. Et nous pourrons accéder à la variable fantôme *via* le pointeur.

Voici un code d'exemple ainsi que son résultat.

```
int *ptr = new int(9);

cout << "Valeur de la valeur pointee : " << *ptr << endl;
cout << "Adresse de la valeur pointee : " << *&ptr << endl << endl;
cout << "Valeur du pointeur : " << ptr << endl;
cout << "Adresse du pointeur : " << &ptr << endl << endl;

cout << "Suppression de la valeur pointee..." << endl << endl;
delete ptr;
ptr = nullptr;
```

```
//cout << "Valeur de la valeur pointee : " << *ptr << endl;
//cout << "Adresse de la valeur pointee : " << *&ptr << endl << endl
cout << "Valeur du pointeur : " << ptr << endl;
cout << "Adresse du pointeur : " << &ptr << endl;

cin.ignore();
```



Il est très important de comprendre à la perfection ce que vous voyez ici.

La création des pointeurs

Il est maintenant temps pour vous d'apprendre à utiliser les pointeurs. Voici comment créer un pointeur :

```
TYPE * NAME;
```

En C++, le symbole caractéristique du pointeur est une `*`. A chaque fois que vous verrez une étoile en C++, c'est qu'un pointeur n'est pas très loin. Comme vous le savez, le pointeur est un *nombre* qui pointe sur une donnée. Donc comme toujours en C++, nous devons donner un type à cette donnée.

Je vous ai dit que les pointeurs permettent de gérer manuellement la mémoire. En fait c'est *faux*. Ce ne sont pas les pointeurs qui permettent de gérer la mémoire mais les opérateurs `new` et `delete`. Comme leur nom l'indique, `new` permet de d'allouer de la mémoire alors que `delete` permet de la récupérer.

Voici comment allouer de la mémoire :

```
new TYPE;
```

Cela s'appelle **l'allocation dynamique**. L'opérateur `new` va donc allouer dans votre mémoire une zone pour le type `TYPE` que vous avez demandé. Et, cette instruction retourne l'adresse mémoire de l'allocation fraîchement effectuée. Or, comme vous le savez, un pointeur sauvegarde les adresses mémoires. Alors fusionnons les !

```
TYPE * NAME = new TYPE;
```

Cette instruction va simplement allouer une zone mémoire du type `TYPE`, et il assignera alors au pointeur `NAME` l'adresse de cette mémoire pour que vous puissiez l'utiliser. Si vous souhaitez, vous pouvez aussi assigner directement une valeur après l'initialisation :

```
TYPE * NAME = new TYPE(VALUE);
```

Ainsi, voici un exemple en pratique (cf. le code au dessus)

```
int *ptr = new int(9);
```

> Attention ! Si vous initialisez un pointeur sans l'assigner (donc sans lui donner de valeur), vous faites erreur. Vous devez **toujours** assigner une valeur à votre pointeur. En effet, le pointeur risque de pointer sur une variable aléatoire sur votre ordinateur, et vous pourriez alors créer des instabilités sur votre système. Ainsi, si vous créez un pointeur que vous voulez laisser vide, faites comme suit : `int *ptr = nullptr`. La constante `nullptr` provient du C++11 et est essentielle.

L'utilisation des pointeurs

Maintenant que nous avons notre pointeur, comment pouvons nous l'utiliser ? Rien de plus simple.

Pour accéder à la valeur que le pointeur pointe, utilisez l'opérateur `*` (oui encore).

Regardez plutôt :

```
int *ptr = new int(9);
int value = *ptr;
```

Alors, `value` vaudra `9`. Par contre, si vous enlevez l'étoile, vous accéderez à l'adresse mémoire de la valeur pointée. Par exemple :

```
int *ptr = new int(9);
cout << *ptr << endl; // Affichera 9
cout << ptr << endl; // Affichera l'adresse mémoire de la valeur poi
```

C'est tout ! Et oui c'est aussi simple !

> L'opérateur étoile que nous appliquons alors aux données permet de suivre les adresses mémoires. Ainsi, si vous appliquez l'opérateur étoile sur un nombre qui représente l'adresse d'une variable, alors vous accéderez à la variable. L'opérateur étoile permet ainsi d'effectuer des lectures de la mémoire arbitraire.

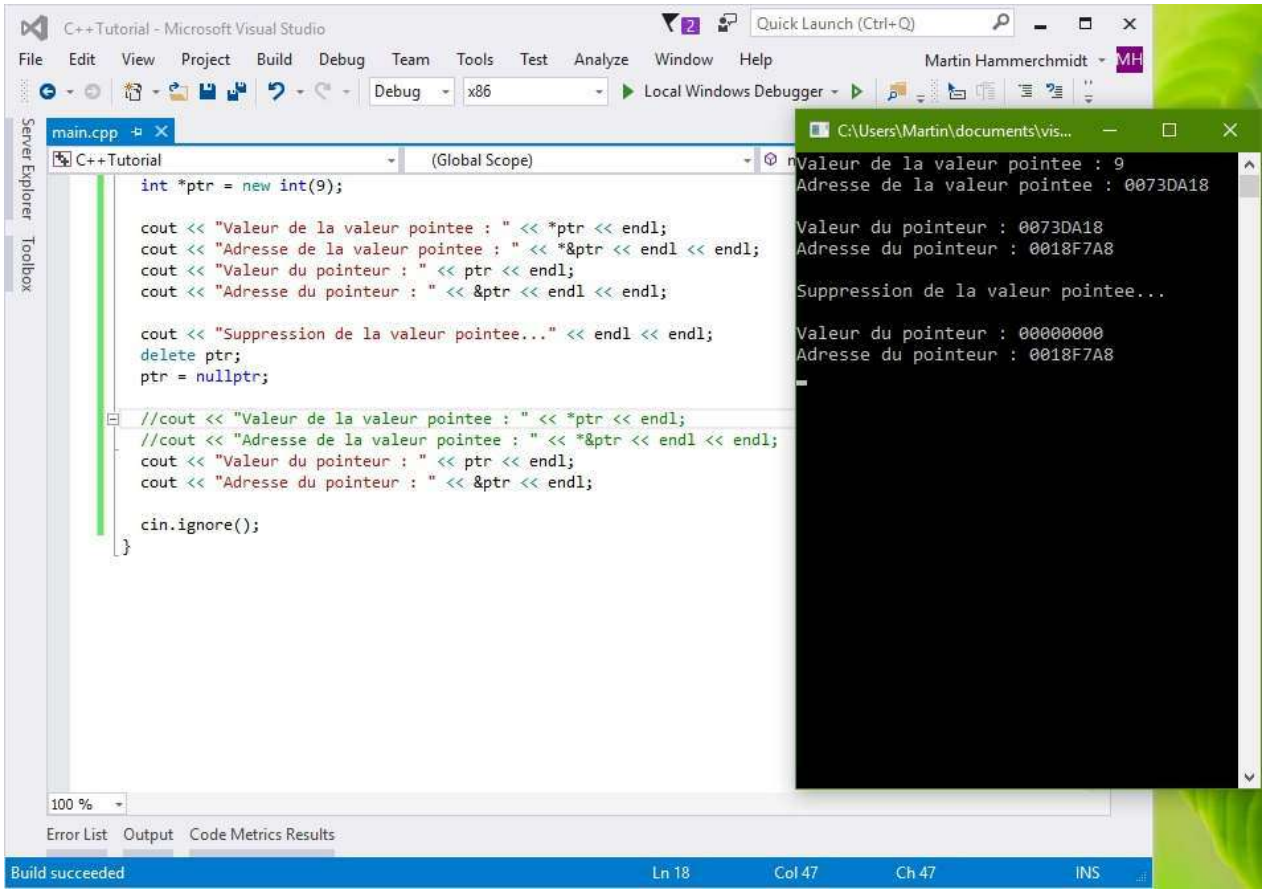
La suppression de la mémoire allouée

Si vous allouez de la mémoire dynamiquement avec l'opérateur `new`, celle-ci ne sera jamais libéré autrement que par vous même. Vous **devez** donc la libérer dès qu'elle n'est plus nécessaire. Pour ce faire, utiliser l'opérateur `delete`. Il suffit de donner à cet opérateur une adresse mémoire (ou un pointeur), et il va libérer la mémoire associé.

```
int *ptr = new int(9); // Allocation de la mémoire
delete ptr; // Libération de la mémoire
```

Analysons l'exemple

Je vous redonne l'exemple :



Voici l'analyse en détail instruction par instruction.

- Nous allouons un `int`, nous lui assignons la valeur `9` et nous créons et assignons un pointeur sur cette même mémoire.
- Nous affichons la valeur du nombre grâce à l'opérateur `*`.
- Nous affichons l'adresse mémoire du nombre en pointant dessus puis en demandant son adresse (voir chapitre précédent) grâce aux deux opérateurs `*&`.
- Nous affichons la valeur du pointeur en lui même (valeur identique à l'adresse mémoire du nombre comme vous pouvez le voir).
- Nous affichons l'adresse du pointeur en lui même.
- Nous supprimons le nombre et remettons le pointeur à zéro.
- Nous ne pouvons plus afficher la valeur pointée ni l'adresse de la valeur pointée car celle-ci n'existe plus. Si vous essayez d'exécuter ces deux lignes, alors votre application plantera.
- Nous affichons la valeur du pointeur en lui même (qui vaut donc `0`, c'est à dire `nullptr`).
- Et enfin nous affichons l'adresse du pointeur qui n'a pas changé.

Si vous avez bien suivi, vous devriez désormais être capable de comprendre et d'utiliser cet exemple. Voici maintenant des petites astuces d'utilisations.

Lier un pointeur à une variable classique

Si un jour vous aimeriez que votre pointeur pointe sur une variable normale, vous devrez utiliser l'opérateur &, qui pour rappel permet d'obtenir l'adresse mémoire d'une variable.

```
int x = 10;
int *ptr = &x;
cout << *ptr << endl; // Affichera 10;
```

> Notez que vous devez utiliser `delete` que si la mémoire a été créé par l'opérateur `new`. Ici, vous n'avez pas besoin de libérer la mémoire par vous même car le propriétaire de la mémoire est une variable classique : quand elle sera détruite, la mémoire le sera aussi.

Les pointeurs et les fonctions

Il peut être très utile d'utiliser les pointeurs avec des fonctions. En effet, en somme, un pointeur n'est qu'un nombre donc très rapide à copier. Beaucoup plus rapide que copier une grosse variable ! Donc n'hésitez pas à passer des pointeurs dans les fonctions.

```
int add(int * a, int * b)
{
  return *a + *b;
}

int main()
{
  int *a = new int(9);
  int *b = new int(12);

  int c = add(a, b);

  delete a;
  a = nullptr;
  delete b;
  b = nullptr;
}
```

Le pourquoi du comment !

Je vais vous fournir une petite dernière explication pour ceux qui n'auraient pas encore très bien compris.

Voici une variable classique :

```
int x = 0;
```

Ici, `x` représente la variable en elle même. `x` **est** la variable. Ainsi, si `x` est détruit, alors la mémoire est libéré.

Voici un pointeur :

```
int *ptr = new int(0);
```

Nous avons ici `ptr`. C'est un nombre qui représente l'adresse mémoire d'une autre variable (un nombre aussi ici). Si `ptr` est détruit, le nombre qui représente l'adresse mémoire de la variable pointée est aussi supprimé, mais pas la variable pointée en elle même.

```
delete ptr;
```

Cette instruction ne vas pas détruire `ptr`. Elle va détruire `*ptr`, c'est à dire la variable que pointe `ptr`. Ainsi, la variable pointée est détruite donc elle n'est plus accessible. N'oubliez pas de faire un `ptr = nullptr` dans ce cas, sinon `ptr` devient un pointeur invalide !

Vous allez peut être me détester, mais après cette grosse leçon je vais vous informer que... j'ai mieux que les pointeurs ! Les *pointeurs intelligents*.

Glossaire

Glossaire

buffer

Sorte de mémoire temporaire qui attend d'être lu. Une fois lu, l'information disparait. Par exemple les entrées utilisateurs.

- 2.6. Dialoguons avec l'utilisateur !

compilateur

Le compilateur transforme votre code source en un exécutable lançable par votre système d'exploitation.

- 2.1. Qu'est-ce que le C++ ?
- 2.2. Installons un IDE
- 2.11.2. Les tableaux dynamiques
- 2.11.1.2. Les tableaux statiques C++11
- 2.12.1. Les variables 2

instruction

Une instruction en C++ ordonne d'effectuer une action. Par exemple 'int x;' est une instruction qui déclare une variable. Une instruction est toujours terminé par un point virgule.

- 2.3. Vos premières lignes de C++
- 2.12.3. Les pointeurs
- 2.10. Les espaces de noms
- 2.6. Dialoguons avec l'utilisateur !
- 2.8. Les boucles
- 2.9. Les fonctions
- 2.12.1. Les variables 2

librairie

Une librairie est un "kit" qui contient des outils dans un but spécifique. Par exemple une librairie 2D pour afficher de la 2D, une librairie de cryptographie et bien plus encore.

- 2.10. Les espaces de noms

mémoire vive

Plus communément appelé RAM, cette mémoire est volatile et a une vitesse d'accès extrêmement rapide.

- 2.12.2. Les références

- 2.12.3. Les pointeurs
- 2.4. Les variables
- 2.12.1. Les variables 2