

# Débuter en C++ moderne - Tome 1

## Cours de C++ moderne

### Création d'applications en C++ moderne par la pratique

## Avant propos

- Introduction
- Comment suivre ce cours ?
- Comment réaliser les exercices de ce cours ?
- Demander de l'aide et aider les autres
- Que faire en cas de problème lors de la compilation des codes d'exemples ?

## Premiers pas

### Langage

- Programme C++ minimal
- Le programme "hello world"
- Les littérales chaînes de caractères
- Notes sur la qualité logicielle

### Compléments

- Un langage vivant
- Explorer la documentation

## Bases du calcul numérique

### Langage

- Les nombres entiers
- Logique binaire et calcul booléen
- Les nombres réels
- Initialisation, concaténation et conversion des chaînes

### Compléments

- L'algèbre de Boole
- Les nombres complexes
- Les nombres à virgule fixe
- Les nombres rationnels

## Conserver les valeurs en mémoire

### Langage

- Utiliser la mémoire avec les variables
- Constantes et énumérations
- L'inférence de type
- Manipuler les types
- Obtenir des informations sur les types
- Créer des nouveaux types

## **Compléments**

- [Les nombres pseudo-aléatoires](#)

## **Collections et algorithmes**

Les chapitres suivants sont encore en cours de rédaction, voire à l'état d'ébauche. N'hésitez pas à faire des remarques ou poser des questions sur le forum de [Zeste de Savoir](#) ou de [OpenClassroom](#).

## **Langage**

- [Les collections de données](#)
- [Introduction aux algorithmes standards](#)
- [Les catégories d'algorithmes standards](#)
- [Les itérateurs](#)
- [Les foncteurs](#)
- [D'autres collections](#)

## **Compléments**

- [Les tableaux de bits](#)
- [Valarray](#)

## **Pratiquer**

- [Gérer des intervalles de valeurs](#)
- [Zip et unzip](#)
- [Jouons avec les chiffres](#)

# **Chaînes avancées et expressions régulières**

## **Compléments**

- [Les chaînes de caractères internationales](#)
- [Remplir une chaîne avec des caractères](#)
- [Les expressions régulières 1](#)
- [Les expressions régulières 2](#)
- [Les expressions régulières 3](#)
- [Regex - suite](#)

## **Pratiquer**

- [Analyse lexicale et syntaxique](#)
- [Évaluation d'expressions](#)
- [Créer un modèle de document](#)

## **Entrées et sorties**

## **Langage**

- [La ligne de commande](#)
- [Entrée console](#)
- [Les fichiers](#)

## **Compléments**

- [XML](#)

- Lire et écrire des feuilles de calculs
- Lire et enregistrer des images

## Créer ses algorithmes

### Langage

- Créer des fonctions
- Les conditions
- Les boucles
- Fonctions génériques
- Les références
- Gérer les erreurs dans les fonctions

### Compléments

- Les paires, tuples et structures
- Mesurer les performances
- Séparer définition et implémentation
- Les classes et fonctions variadiques
- La Pile d'appel des fonctions
- Variables globales et statiques, mémoire static, string table

### Pratiquer

- Sampler
- Jouer au Poker
- Algorithmes d'extraction de sous-chaînes
- Résoudre des intégragrammes
- Jouer au Scrabble

- Analyseur simple d'expressions régulières
- imagerie : génération d'arbres avec L-System
- implémenter des générateurs aléatoires cryptographiques

## Concevoir une bibliothèque

### Langage

- documentation, commentaire, codes d'exemple
- tests unitaires, testabilité, qualité logicielle
- Concevoir une bibliothèque, interface publique, réutilisabilité.  
“Easy to use correctly, hard to use incorrectly” - Scott Meyers.  
Design interface : s'adapter aux conventions qui existent. Etre consistant
- modules, conception en couches, utilisation de n-1 et n+1
- namespace, dépendances
- signature de fonctions, name lookup
- pré-condition et post-condition
- physical design
- variables globales, étude de std::cout

<https://www.famkruithof.net/uuid/uuidgen>

## Les outils de développement

- Mettre en place une chaîne de compilation
- Les environnement de développement
- Les compilateurs
- Les débogueurs
- La validation statique du code
- La validation du style du code

- Créer une documentation
- Les gestionnaires de versions
- Les tests
- Déployer une application
- Le suivi des problèmes

## **Les classes à sémantique de valeur**

### **Langage**

- Créer des classes
- Créer et copier des objets
- Surcharger les opérateurs
- Conversion de types  
<http://cpptruths.blogspot.de/2015/11/covariance-and-contravariance-in-c.html>
- \* allocation et libération de ressources, raii

### **Compléments**

- Swap
- Les principes SOLID
- Les design patterns
- Les objets-fonctions
- chaining methods

### **Pratiquer**

- Classe à sémantique de collection
- Les ranges et vues sur les collections

- [Itérateur personnalisé](#)
- [Les design patterns Wrapper et Facade](#)
- traits : adapter les type\_traits pour fonctionner avec vos classes

## Les classes à sémantique d'entité

### Langage

- [La sémantique d'entité](#)
- [L'héritage](#)
- [Polymorphisme et substitution](#)
- [Fonctions virtuelles](#)
- [invariant de classe](#)

### Compléments

- [La Pile et le Tas](#), création et destruction
- théorie des graphes, arbres
- héritage multiple
- points de variation : comment faire varier le comportement d'un programme ? (directive de compilation, template police/traits, héritage, DP stratégie) Quand utiliser quelle technique ? Impact sur la qualité du code (maintenabilité, évolutivité, etc)
- durée de vie et propriétaire des objets

### Pratiquer

- jeu d'échec
- évaluation de script (if, for, etc)

- article citation format (type medline)
- <http://progdupeu.pl/forums/sujet/205/banque-dexercices>
- factory. Créer une fonction factory avec switch, avec create, avec allocator
- système de gestion d'événements

## Bibliothèques externes

- Utiliser une bibliothèque
  - internationalisation : ICU
  - interface utilisateur : Qt, SFML
  - réseau : boost.asio, POCO, QtNetwork
  - XML : QDom
  - base de données : QSql
  - web : wt
  - script : boost.python, QtScript
- 

## Le C++03

- Le C++03
- Les chaînes de caractères style C
- tableaux style C
- pointeurs style C, surcharge opérateurs d'accès → \* &, etc + surcharge opérateur new et delete, pointeur de fonction

## Pratiquer

- créer un système d'allocation mémoire (tas). Allocator, utilisation

- des espaces libres, défragmentation
- buffer avec vector

## Documents de travail

- Liste des projets d'application
  - <http://gameprogrammingpatterns.com/>
  - <http://git-scm.com/book/fr/v1>
  - GPU Gems 1, 2 et 3, gratuits, complet et en ligne
  - <http://www.antigrain.com/>
  - <https://www.cgal.org/>
  - <http://www.gnu.org/software/gsl/>
  - <https://gmlib.org/>
  - <http://openclassrooms.com/forum/sujet/a-la-recherche-de-petits-projets#message-85317109>
  - <http://deptinfo.unice.fr/~julia/AL/>
- outils d'analyse statique : clang-analyzer, <http://oclint.org/>
- <http://www.red3d.com/cwr/boids/>
- <http://www.patorjk.com/software/taag/#p=display&f=Big%20Money-sw&t=abc>

Pixelator

<https://travis-ci.org/>

[https://en.wikipedia.org/wiki/Dynamic\\_recompilation](https://en.wikipedia.org/wiki/Dynamic_recompilation)

<https://github.com/facebook/folly/tree/master/folly>

<http://www.akkadia.org/drepper/cpumemory.pdf> et similaires

# À suivre...

À suivre...

Cours, C++

# Introduction

## Remerciements

Puisqu'il faut bien commencer quelque part, je vais commencer par la partie que les lecteurs survolent généralement. Bien que ce livre est le fruit du travail d'un seul auteur, beaucoup de personnes ont contribué, directement ou indirectement, à ce cours.

En premier lieu, il faut remercier les personnes qui ont pris le temps de relire, de corriger et de proposer des remarques pour améliorer ce cours.

Je dois remercier aussi tous ceux qui partagent leurs connaissances sur les forums C++, ceux avec qui j'ai pu avoir de nombreuses discussions sur les problématiques spécifiques du C++, sur la conception et les cycles de vie des logiciels en général, sur la pédagogie pour enseigner la programmation.

Il me faut remercier aussi tous les débutants motivés, qui améliorent les choses par leurs questions. Le fait d'en voir certains qui arrivent débutants sur les forums C++ et qui, quelques mois plus tard, montrent qu'ils ont assimilé les notions que l'on a essayé de transmettre et qui aident à leur tour les autres débutants est une source de motivation importante pour continuer à prendre du temps pour rédiger des cours, articles ou réponses détaillées aux questions des débutants.

Et bien sur, je ne peux m'empêcher de remercier tous les enseignants qui n'ont pas su suivre l'évolution du C++, de sa philosophie ou du génie logiciel en général...

## De quoi va parler ce cours ?

Donner un titre à un livre est assez complexe. Il faut qu'il soit

suffisamment informatif, mais pas trop long pour être percutant. J'avais, dans un premier temps, choisi comme titre “Cours pratique de création d'application en C++ moderne”, ce qui me semblait être le plus descriptif possible. Malheureusement, ce titre est très long et je suis passé à “Cours de C++ moderne”.

Pourquoi j'avais choisi ce premier titre ? Voyons les termes utilisés, un par un :

## Cours

À la différence d'un article ou d'un tutoriel, un cours est conçu selon une approche et un objectif pédagogique. Le but n'est pas d'être un “dictionnaire” exhaustif du langage C++, comme le livre de Bjarne Stroustrup (le créateur du langage C++) : [The C++ Programming Language](#). Le but n'est pas non plus d'être une source de référence, lorsque vous avez un doute sur un syntaxe particulière, comme le site [cppreference.com](#).

D'ailleurs, plus généralement, ce cours n'est pas destiné à être votre seule source d'apprentissage du C++. Il existe beaucoup d'ouvrages et de sites sur le C++, destinés aux débutants comme aux développeurs confirmés, allant des problématiques spécifiques du C++ jusqu'aux différents domaines d'application (calcul scientifique, jeux vidéos, interfaces graphiques, etc).

Je ne vais pas citer tous les ouvrages intéressants à lire, il y en a trop. Vous pouvez avoir un aperçu sur [la page dédiée aux livres](#) sur mon site. Un point important quand même : n'hésitez surtout pas à poser des questions sur les forums AVANT d'acheter un livre. Même si les “bons” livres sont nombreux, les “mauvais” sont encore plus nombreux (en particulier en français). Ce cours ne doit pas faire exception, n'hésitez pas à poser des questions sur celui-ci avant de le suivre.

Le but de ce cours est de vous donner les éléments d'information de base pour démarrer votre apprentissage du C++. Cela veut dire par exemple que vous n'allez pas voir toutes les syntaxes possibles, mais vous saurez que vous n'avez pas vu toutes les syntaxes possibles. Vous saurez qu'il existe d'autres façon de faire les choses, chaque approche ayant ses

avantages et défauts.

## **Pratique**

Un langage de programmation ne s'apprend pas dans les livres ou dans un cours. La programmation s'apprend par la pratique. Vous trouverez de nombreux exercices dans ce cours, dans le but de vous permettre de pratiquer chaque notion que vous aurez apprise. Une notion qui n'est pas pratiquée est une notion qui n'est pas assimilée, lire ce cours sans pratiquer les exercices ne vous permettra pas d'apprendre correctement.

Ce cours contient deux types de mise en application : les exercices et les projets.

Les exercices sont des utilisations directes du chapitre que vous venez d'étudier. Ils consistent en des codes à corriger ou modifier pour obtenir le résultat attendu. Il ne faut généralement que modifier ou ajouter quelques lignes de code et ils peuvent être réalisés en quelques minutes chacun.

Les projets nécessitent plus de travail. Ce sont des codes que vous allez créer sur le long terme, que vous allez conserver et modifier plusieurs fois au cours de votre progression dans ce cours. Vous pouvez même travailler à plusieurs sur un projet. Le but est de vous apprendre des notions importantes, mais qui ne sont pas accessibles lorsque vous travaillez sur de petits projets "jetables" : tout ce qui fait qu'un programme est de qualité, d'un point de vue professionnel.

## **Création d'application**

L'objectif de ce cours n'est pas de vous apprendre le C++. Plus précisément, l'objectif n'est pas de vous apprendre simplement la syntaxe d'un langage de programmation.

Le but de ce cours n'est pas non plus de vous apprendre toutes les syntaxes possibles, toutes les problématiques que l'on peut rencontrer dans le C++. D'abord, parce que je ne les connais pas toutes :). Et si quelqu'un se lançait dans un tel ouvrage, cela s'appellerait "l'encyclopédie du C++", il contiendrait probablement plus de 10.000

pages et serait obsolète au bout de six mois.

La programmation d'une application nécessite d'autres connaissances que la syntaxe, comme le cycle de vie des applications, la conception, l'algorithmie, savoir tester son code. Et plus généralement, tout l'écosystème du C++ : les outils, les bibliothèques, les méthodes de développement logiciel.

## Moderne

Le C++ est un langage en constante évolution. La norme actuelle date de 2014 et la prochaine évolution du C++ est déjà planifiée en 2017. De nombreux cours sur internet ou dans les formations se basent sur l'ancienne norme du C++, le C++03 qui date de 2003. De plus, ces cours font généralement l'impasse sur ce qui fait la qualité d'un code, en particulier la gestion des erreurs.

Lorsqu'un débutant écrit un code d'apprentissage de quelques dizaines de lignes de code, ne pas respecter les bonnes pratiques ne sera pas critique. Mais dans un projet réel, dans le monde professionnel, cela sera catastrophique. C'est l'une des raisons qui fait que le C++ a parfois la réputation d'être un langage complexe. Et de mauvaises habitudes acquises lors de l'apprentissage seront difficiles à oublier par la suite.

[Sommaire principal](#) [Chapitre suivant](#)

# Comment suivre ce cours ?

## Les chapitres

Ce cours est divisé en modules, eux-mêmes divisés en chapitre. Les chapitres sont regroupés en trois catégories : “Langage”, “Complément” et “Pratique”.

Les chapitres “Langage” constituent la trame principale de ce cours. Ils se suivent et devraient être lu de façon linéaire. L'étude d'un chapitre suppose que les chapitres “Langage” précédents ont été assimilés.

Les chapitres “Complément” présentent des notions importantes, mais qui peuvent être étudiées sans suivre la trame principale du cours. Vous pouvez par exemple les passer dans une première lecture et y revenir dessus par la suite.

Les chapitre “Pratiques” sont des compléments abordés sous forme de travaux dirigés, qui nécessitent que vous réalisiez vous même les codes d'exemple.

## Les codes d'exemple

Chaque chapitre se termine par une liste d'exercices et, optionnellement, des projets.

## Les encadrés

Plusieurs types, info, warning, erreur, Old C++

## **Les captures d'écran**

### **You verrez cela plus tard**

A plusieurs reprises, en particulier dans les premiers chapitres, vous verrez régulièrement des notions “qui seront vues dans un prochain chapitre”. Cela vous semblera peut être frustrant au début de ne pas comprendre certaines notions en détails, mais il est difficile de faire autrement.

Lorsque l'on présente une notion, il y a souvent d'autres notions liées, mais qui nécessitent trop d'explications pour être abordées tout de suite. Ces notions avancées ne sont pas nécessaire pour comprendre la suite du cours, elles servent avant tout à avoir une vision globale des problématiques.

Face à ces notions avancées, on peut gérer le problème de différentes façon :

- soit ne pas du tout citer ces notions avancées ;
- soit citer ces notions avancées, mais sans entrer dans les détails ;
- soit entrer dans les détails de ces notions avancées.

Dans le premier cas, cela veut dire que le lecteur ne sait pas qu'il apprend qu'une partie d'une notion plus globale, qu'il ne peut pas avoir une vision d'ensemble. La conséquence est que l'on voit certain développeur rester dans leurs idées préconçus, sans savoir qu'il existe d'autres façon de faire les choses.

Dans le troisième cas, le lecteur peut se perdre dans des détails annexes au sujet abordé dans un chapitre et donc ne pas en retenir les notions importantes qu'il doit acquérir à la fin de la lecture de ce chapitre. Ce qui au final peut nuire à l'apprentissage.

Dans ce cours, j'ai généralement choisi le second cas. Cela aide à avoir une vision globale, sans se perdre dans les détails. Même si le lecteur ne

comprend pas une notion, il est important qu'il sache qu'elle existe. Il est toujours possible de se renseigner sur des notions que l'on estime ne pas avoir assez vu en détail (et comme c'est un cours de base de C++, il n'est pas possible de voir toutes les notions en profondeur). Par contre, il est impossible d'approfondir par soi-même une notion si on ne sait pas qu'elle existe.

[\*\*Chapitre précédent\*\*](#) [\*\*Sommaire principal\*\*](#) [\*\*Chapitre suivant\*\*](#)

Cours, C++

# Comment réaliser les exercices de ce cours ?

Pour apprendre la programmation, vous devez pratiquer !

Cependant, il ne faut pas pratiquer n'importe quoi, n'importe comment. On voit beaucoup de débutant qui souhaitent apprendre le C++ pour réaliser un projet. L'intention est bonne. Peut-être est-ce votre cas ?

Le problème dans ce cas est que l'on a souvent tendance à se focaliser sur le projet et pas sur les notions apprises. Si une notion vue dans le cours ne trouve pas d'application pratique dans le projet, elle ne sera pas correctement assimilée, voir complètement oubliée. Même les notions importantes.

Ce cours part d'une idée simple : **Si on ne pratique pas une notion, on ne la connaît pas.**

C'est pour cela que ce cours vous propose de nombreux exercices, travaux pratiques et projets. Si vous avez une idée de projet, je vous conseille d'accepter l'idée qu'il faut le mettre temporairement de côté, le temps d'avoir appris les bases du C++ et éviter de vous perdre dans votre apprentissage.

Les solutions aux exercices ne sont volontairement pas données. La raison est qu'il n'existe pas toujours "la bonne réponse", mais plusieurs approches peuvent être correctes pour une même problématique. En proposant vos solutions sur les forums, vous pourrez obtenir des conseils détaillés sur les points à améliorer dans votre code.

Quelle est la différence entre exercices, travaux pratiques et projets dans ce cours ?

## Les exercices

Note importante pour compiler les codes d'exemple du cours

Le cours est maintenant basé sur la norme C++14. La ligne de commande indiqué dans les captures d'écran ont été réalisées pour le C++11. Il est donc nécessaire de compiler certains codes en changeant la ligne de commande. Je conseille également d'ajouter `-Weverything`, pour activer plus de messages d'avertissement.

```
clang++ -std=c++14 -Wall -Wextra -pedantic -Weverything -O2  
main.cpp && ./a.out
```

Chaque section d'un chapitre s'accompagne d'une série d'exercices, abordant la notion qui vient d'être expliquée. Je vous conseille de réaliser un maximum d'exercices. Les exercices sont rapides à réaliser (normalement quelques minutes) et ne nécessitent que quelques lignes de code.

Pour chaque exercice, un code est fourni. Vous devez modifier le code en suivant les instructions, le compiler et l'exécuter pour vérifier que le programme réalise la tâche demandée correctement. Si vous faites une erreur, le compilateur vous la signalera par un message d'erreur et il ne sera pas possible de lancer l'application. En cas de non respect des bonnes pratiques de codage, le compilateur signalera le problème avec un message d'avertissement. Le programme pourra quand même se lancer, mais sera susceptible d'avoir un comportement incorrect.

Dans ce cours, un exercice ne devra être considéré comme réalisé que lorsque vous n'aurez plus aucun message signalant un problème. (Ne vous inquiétez pas, vous apprendrez aussi à comprendre ces messages d'erreur).

Dans les chapitres suivants du cours, vous apprendrez à installer un environnement de développement complet pour compiler vos programme. Vous pourrez utiliser ces outils pour réaliser les exercices. Cependant, pour vous simplifier la tâche, vous pouvez également réaliser

une majorité des exercices donnés sur un éditeur en ligne. Pour cela, un lien est donné pour chaque exercice, par exemple : [Réaliser l'exercice](#). Cliquez dessus avec le bouton droit de la souris et choisissez “Ouvrir le lien dans un nouvel onglet” pour ouvrir l'éditeur en ligne Coliru :

The screenshot shows the Coliru Viewer application window. At the top, there's a toolbar with icons for file operations like Open, Save, and Print. Below the toolbar is a browser-style address bar with the URL "coliru.stacked-crooked.com/a/661ae729a3e83fd4". The main area is divided into two sections: a code editor on the left containing the following C++ code:

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     cout << "Bienvenue dans ce cours de C++ !" << endl;
6     return 0;
7 }
```

Below the code editor is a terminal window showing the command and its output:

```
clang++ -std=c++11 -Wall -Wextra -pedantic -O2 main.cpp && ./a.out
Bienvenue dans ce cours de C++ !
```

At the bottom of the window, there's a message: "This file can be also found using the Coliru command line: cat /Archive2/66/iae729a3e83fd4/main.cpp" followed by an "Edit" button.

Cliquez ensuite sur le bouton “Edit” en bas à droite pour modifier le code :

The screenshot shows the Coliru application window. The interface is similar to the viewer, with a toolbar at the top, a browser-style address bar, and a main area divided into code editor and terminal sections. The code editor contains the same C++ code as the previous screenshot. The terminal window shows the command and its output. In the bottom right corner of the main window, there is a button labeled "Edit". This button is highlighted with a red rectangle, indicating it is the target for the next click.

La fenêtre est divisée en trois parties :

- la partie du haut contient le code de l'exercice ;

- la partie du milieu (en gris) affiche les messages d'erreur du compilateur et les messages de l'application ;
- la partie du bas contient les commandes permettant de lancer la compilation, puis le programme.

Suivez les instructions données et modifiez le code dans la première partie pour réaliser l'exercice (dans ce premier exemple, vous n'avez rien à modifier). Cliquez ensuite sur le bouton “Compile, link and run...” pour lancer la compilation et l'exécution :

The screenshot shows the Coliru IDE interface. The top bar includes a logo, a title bar with 'Coliru', a URL bar with 'coliru.stacked-crooked.com', and a menu bar with 'Restore', 'defaults', 'Help', 'Feedback', 'Editor', 'Command', 'Q&A', and 'Read Write'. The main area has tabs for 'Donate' and 'Coliru'. The code editor contains the following C++ code:

```

1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     cout << "Bienvenue dans ce cours de C++ !" << endl;
6     return 0;
7 }

```

The output window below the editor shows the program's output: "Bienvenue dans ce cours de C++ !". At the bottom, the terminal window displays the command: "clang++ -std=c++11 -Wall -Wextra -pedantic -O2 main.cpp && ./a.out". To the right of the terminal are two buttons: "Compile, link and run..." and "Share!".

Dans cet exemple, le code permet d'afficher le message suivant “Bienvenue dans ce cours de C++ !”

Vous pouvez lancer la compilation autant de fois que vous le souhaitez. Si vous avez un doute sur la syntaxe à utiliser, le compilateur pourra vous indiquer le type d'erreur et vous aider à trouver la solution. Une erreur pouvant générer plusieurs messages, vous devez corriger les erreurs dans l'ordre donné par le compilateur.

Dans de nombreux exercices, vous ne devrez modifier que les parties du code indiquées par @@. Par exemple, un exercice pourra vous demander de remplacer par l'une des propositions données entre les @@ :

*// Choisissez la syntaxe correcte parmi les syntaxes proposées*

```
int main() {  
    @@int, double, string ou bool@@ i { 123 };  
}
```

Il faut dans ce cas par exemple écrire :

```
// Choisissez la syntaxe correcte parmi les syntaxes  
proposées
```

```
int main() {  
    int i { 123 };  
}
```

Dans d'autres cas, il faudra remplacer les @@@@ par la syntaxe correcte :

```
// Choisissez la syntaxe correcte parmi les syntaxes  
proposées
```

```
int main() {  
    @@@@ i { 123 };  
}
```

Dans d'autres cas, il faudra remplacer @@@@@@ par une ou plusieurs lignes, sans modifier les autres lignes :

```
// Choisissez la syntaxe correcte parmi les syntaxes  
proposées
```

```
int main() {  
    @@@@  
}
```

## Les travaux pratiques

A la fin de chaque chapitre, une série de travaux pratiques vous est proposée. Ces travaux pratiques consistent généralement à écrire un programme simple de quelques dizaines à quelques centaines de lignes. Les instructions sont moins détaillées que pour les exercices et

nécessiteront de votre part un travail de conception de votre programme.

Vous pouvez écrire vos programmes pour les travaux pratiques en utilisant [Coliru](#) et les partager en utilisant le bouton “Share!” en bas à droite pour obtenir une adresse URL que vous pouvez partager sur les forums.

Vous pouvez également écrire vos programmes en utilisant les outils de développement que vous souhaitez. Dans ce cas, pour partager vos codes sources et obtenir les avis des autres, vous pouvez utiliser un site de gestion de version du code tel que [GitHub](#).

Pour cela, vous devez créer un compte sur [GitHub](#) et créer un nouveau dépôt (*repository*) que vous pouvez par exemple nommer “cours-cpp”. Créez ensuite un répertoire pour chaque travail pratique et chaque projet.

#### [utilisation détaillée de github et git windows et linux](#)

Vous apprendrez dans la suite du cours à concevoir correctement vos programmes, mais voici déjà un conseil : n'hésitez pas à vous inspirer des codes C++ existants. Vous pouvez en particulier regarder comment sont conçues les bibliothèques comme la [bibliothèque standard](#) ou [Boost](#).

## **Les projets**

Le but des projets est de travailler sur un programme de taille relativement importante. Vous pourrez ajouter des fonctionnalités, corriger votre code et le faire évoluer pendant votre apprentissage de ce cours.

Plusieurs projets vous sont proposés. Choisissez en un ou deux (il est préférable de faire un projet correctement à fond, que de tenter de faire cinq projets et les faire à moitié) selon les thèmes que vous préférez.

Les projets nécessiteront de créer plusieurs fichiers, ce qui ne permet pas d'utiliser les éditeurs en ligne. Il faudra donc avoir installé les outils de développement nécessaires pour créer vos projets. Utilisez ensuite un outil de partage de code comme [GitHub](#) pour présenter vos projets sur

les forums et avoir les avis des autres développeurs.

Les différents travaux pratiques et projets sont issues des thématiques suivantes. Je vous conseille d'essayer de vous focaliser sur une ou deux thématiques pour commencer (en particulier sur la thématique "Généraliste"). Dès que vous aurez fini la lecture du cours et réalisé la majorité des exercices d'une thématique donnée, vous pourrez vous lancer dans les exercices d'une autre thématique.

Les thématiques sont les suivantes :

- Généraliste : cette thématique regroupe des algorithmes génériques, qui sont utilisés dans de nombreux cas d'applications.
- analyse de texte : dictionnaire, parser
- web : génération de pages html, xml, json, web application
- infographie : analyse d'image, rendering, raytracing, géométrie
- modélisation : créer un modèle, simulation
- cryptologie : mot de passe, décodage/encodage
- analyse de données : statistique, data mining, machine learning
- traitement du signal
- Vision par ordinateur (*computer vision*)

Voir l'annexe "Liste des exercices par thématiques" pour plus de détails.

[Chapitre précédent](#) [Sommaire principal](#) [Chapitre suivant](#)

Cours, C++

# Demander de l'aide et aider les autres

Comme indiqué dans le chapitre précédent, les solutions aux exercices ne sont pas données dans ce cours. Le but est de vous permettre d'apprendre à trouver des informations sur internet, savoir demander de l'aide sur les forums et savoir aider les autres.

Un site de référence pour trouver de l'aide sur internet est le site [cppreference](#). Vous trouverez de nombreuses références dans ce cours et apprendrez à utiliser ce site.

## Comment utiliser les forums d'entraide ?

Les forums internet permettent de partager vos codes avec d'autres développeurs et obtenir de l'aide et des conseils. Je suis en particulier présent sur les forums [Zeste de savoir](#), [OpenClassRooms](#) et [QtFr](#).

Cependant, pour demander de l'aide sur les forums, il faut respecter quelques règles de bonnes conduites :

- **Commencez par essayer de résoudre votre problème tout seul.**  
Vous devez apprendre par vous-même à trouver les informations nécessaires pour résoudre vos problématiques. Les forums sont une aide précieuse, surtout quand on débute, mais ils ne permettent pas de résoudre tous les problèmes.
- **Respectez les règles du forum.** En particulier, la politesse (bonjour, merci) et ne pas écrire en style SMS.
- **Choisissez un titre explicite.** Ne donnez jamais comme titre à vos messages "J'ai un problème", "Erreur C++" ou "J'ai besoin d'aide". Les personnes savent que vous avez un problème et que vous avez besoin d'aide, sinon vous ne poseriez pas votre

question sur un forum. Et comme vous posez votre question sur un forum C++, on se doute que c'est une erreur C++.

- **Faites un minimum d'effort dans la rédaction de vos messages : orthographe, grammaire et présentation du code.** Le minimum est d'utiliser les correcteurs d'orthographe inclus dans votre navigateur internet. Un message mal rédigé peut vraiment gêner la lecture, même si ce n'est pas le cas pour vous, par respect pour ceux qui vous liront, vous devez faire un effort. Pensez à utiliser un outil de mise en forme du code (par exemple [clang-format](#)).
- **Expliquez correctement vos problématiques.** Un adage dit qu'un problème correctement expliqué est à moitié résolu. On voit souvent des personnes trouver la solution à leur problème simplement en rédigeant le message sur le forum. Il faut en particulier donner le contexte, expliquer les erreurs obtenues et donner le code.
- **Ne donnez pas tout votre code.** Si votre programme fait plusieurs dizaines de lignes de code, voire plus, il sera long et fastidieux pour ceux qui vous liront de trouver les lignes de code correspondant à votre problème. Cela fait parti également de votre travail de réussir à situer correctement l'origine de votre problème. Vous ne devez donner dans l'idéal que le code minimal exécutable. Si vous devez donner la totalité de votre code, passez par un gestionnaire en ligne tel que [GitHub](#).
- **Donnez vos messages d'erreur.** Le pire que vous puissiez faire est de dire simplement "j'ai une erreur", cela ne sert strictement à rien. Ne donnez pas non plus un résumé du message d'erreur. Copiez collez directement les messages d'erreur tels quels. S'il y a beaucoup d'erreurs, ne copiez que les cinq premières.

## Aider les autres pour apprendre

<a href="#">Chapitre précédent</a>	<a href="#">Sommaire principal</a>	<a href="#">Chapitre suivant</a>
------------------------------------	------------------------------------	----------------------------------

## Cours, C++

# Que faire en cas de problème lors de la compilation des codes d'exemples ?

De nombreux codes sont donnés dans ce cours. Il peut arriver que vous ayez des problèmes de compilation. La raison est que le C++ n'est pas unique, il existe en fait plusieurs versions du C++ (normes) que l'on nomme C++03, C++11, C++14 et C++1z (il existe aussi un C++98, mais il est tellement ancien que tous les compilateurs actuels prennent en charge au moins le C++03 - sauf si vous travaillez sur un environnement particulier).

Selon le compilateur que vous utilisez, toutes ces normes ne sont pas forcément prises en charge et pas forcément en totalité.

Les codes sont écrits en C++14 et ont été testés sur coliru.com, donc avec le compilateur Clang version 3.5. Si cette version de Clang ne prend pas en charge une fonctionnalité, cela est indiqué dans le texte. Si vous utilisez un autre compilateur, il faudra vérifier que ce n'est pas un problème de prise en charge du C++14 sur votre compilateur.

Vous verrez par la suite comment installer une chaîne de compilation complète, quelques remarques ici sur les compilateurs.

## GCC et MingW

Compilateur en général fournit avec les IDE sous windows, par exemple Code::Block et Qt Creator. MingW est un portage de GCC sous windows, mais a une version de retard (GCC est en version 4.9, MingW en version 4.8.2). MingW ne supporte donc pas les dernières fonctionnalités du C++11 et encore moins le C++14. Utiliser une version de GCC pour windows ([lien](#))

De plus, par défaut, nouvelles normes non activées. Il faut le faire manuellement :

- ligne de commande = -std=c++1z, -std=c++14, -std=c++1y, -std=c++11, -std=c++03 ;
- dans Qt Creator : CONFIG += C++11 dans le .pro
- dans Code::Block : cocher l'option “activer le C++11” ou ajouter une directive de compilation (C++14)

## **Microsoft Visual**

Prendre le dernier MSVC 2013 CRTP2. Pas d'option à activer.

## **Clang**

Compilateur par défaut sur Mac (XCode). Installable dans les paquets sous linux. Problème sous windows (portage en cours)

<b><a href="#">Chapitre précédent</a></b>	<b><a href="#">Sommaire principal</a></b>	<b><a href="#">Chapitre suivant</a></b>
---	---	---

[Cours, C++](#)

Dans ce chapitre, vous allez entrer enfin dans le vif du sujet et écrire vos premiers programmes en C++. Le but sera d'avoir un aperçu du processus de compilation, d'afficher des messages et de faire quelques calculs simples.

## Programme C++ minimal

Créer une application en C++ n'est pas très compliqué. Commençons par voir un code C++ minimal, qui permet de créer un programme qui ne fait rien. Même s'il ne fait rien, nous allons pouvoir aborder quelques notions importantes.

### Tester l'environnement de compilation en ligne

Pour commencer, revoyons les étapes pour lancer un programme sur l'éditeur en ligne que vous allez utiliser dans ce cours (vous avez déjà réalisé cette procédure dans le chapitre [Comment réaliser les exercices de ce cours ?](#) pour tester les codes d'exemple). Cliquez sur le lien suivant, avec le bouton droit de la souris et choisissez "Ouvrir le lien dans un nouvel onglet" : [Tester le code C++ minimal](#). Ce lien permet d'ouvrir un environnement de développement en ligne, qui contient un éditeur pour écrire le code C++ et un compilateur pour transformer votre code C++ en programme exécutable.

Vous pouvez également copier le code suivant dans un éditeur en ligne ([Coliru](#), [IdeOne](#) ou autre), dans n'importe quel éditeur installé sur votre ordinateur (Qt Creator, Visual C++, Code::Block ou autre) ou dans un simple éditeur de texte.

main.cpp

```
int main() {  
}
```

Dans le début de ce cours, vous pouvez utiliser un éditeur en ligne, ce qui évite l'installation, dans un premier temps, d'un environnement de développement. Mais, bien sûr, la création d'un programme C++ professionnel nécessitera d'installer de nombreux outils pour faciliter le développement et apporter des garanties à la qualité du logiciel. Cela sera détaillé dans la suite de ce cours.

La nouvelle fenêtre ressemble à l'image suivante :

The screenshot shows a window titled "Coliru Viewer". At the top, there's a toolbar with icons for back, forward, refresh, and file operations. Below the toolbar, the address bar displays the URL "coliru.stacked-crooked.com/a/5c5f3c8f219a5f2f". The main area is divided into two sections: a code editor on the left containing the following C++ code:

```
1 int main() {  
2 }
```

and a terminal window on the right showing the compilation command:

```
clang++ -std=c++11 -Wall -Wextra -pedantic -O2 main.cpp && ./a.out
```

At the bottom of the terminal window, there's a note: "This file can be also found using the Coliru command line: cat /Archive2/5c/5f3c8f219a5f2f/main.cpp". To the right of this note is a small "Edit" button.

Mettre à jour les images pour utiliser le C++14 :

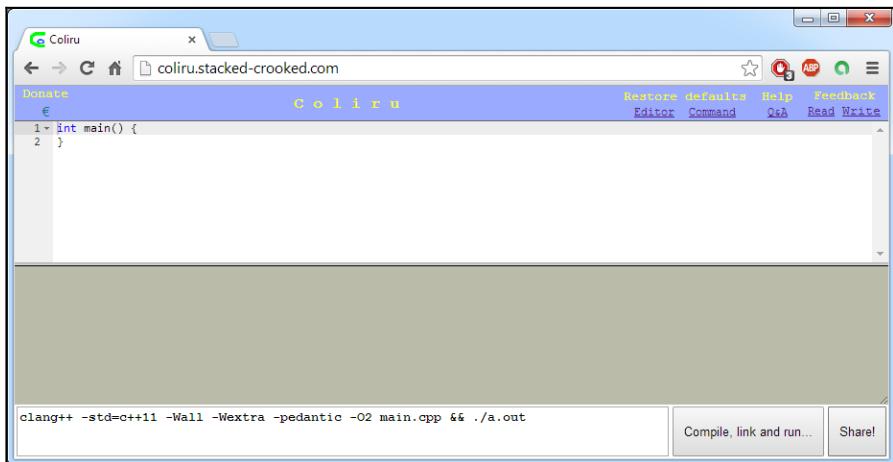
```
clang++ main.cpp -std=c++14 -Wall -Wextra -pedantic -O2 &&  
./a.out
```

Cette fenêtre se décompose en deux parties. De haut en bas :

- le code du programme ;
- le résultat de la compilation et l'exécution.

Pour le moment, vous ne pouvez pas modifier ce code, cette fenêtre est en fait une simple archive du code qui a été créé pour ce cours. Pour pouvoir modifier et tester ce code, vous devez cliquer sur le bouton "Edit" en bas à droite de la fenêtre.

L'aspect de la fenêtre change pour ressembler à l'image suivante :



Cette nouvelle fenêtre contient maintenant trois parties. De haut en bas :

- Dans la partie du haut, l'éditeur de code, qui contient le code C++ de votre programme. Contrairement à l'étape précédente, vous pouvez maintenant modifier ce code en cliquant dessus et en tapant le code sur le clavier.
- La partie centrale grise, le résultat de la compilation et de l'exécution du programme. Pour le moment, comme vous n'avez pas lancé le programme, cette partie est vide.
- Dans la dernière partie, les instructions pour la compilation et l'exécution du programme.

Pour lancer la compilation et l'exécution, il vous suffit de cliquer sur le bouton "Compile, link and run..." en bas à droite. Si vous le faites, vous verrez que l'éditeur change de couleur quelques instants, mais rien ne change dans la partie centrale grise.

La raison est simple : le code minimal donné ne fait rien et donc rien ne s'affiche, ce qui donne l'impression que rien ne se passe. Mais faites une petite modification du code (n'importe quoi, c'est pour tester) et relancez la compilation, vous verrez des messages s'afficher dans la partie grise :

The screenshot shows a web-based C++ compiler environment. At the top, there's a header with the Coliru logo, a 'Donate' button, and navigation links like 'Restore defaults', 'Help', 'Feedback', 'Editor', 'Command', 'Q&A', and 'Read Write'. Below the header is a code editor window containing the following C++ code:

```
1 un test
2
3 int main() {
4 }
```

Below the code editor is a terminal window showing the compilation errors:

```
main.cpp:1:1: error: unknown type name 'un'
un test
^
main.cpp:1:8: error: expected ';' after top level declarator
un test
      ^
;
2 errors generated.
```

At the bottom of the terminal window, there's a command line input field with the command `clang++ -std=c++11 -Wall -Wextra -pedantic -O2 main.cpp && ./a.out`, and two buttons: 'Compile, link and run...' and 'Share!'. The entire interface is set against a light blue background.

Même sans comprendre ce qui se passe, vous pouvez voir un mot dans la partie grise qui n'est pas compliqué à interpréter : "error". Le compilateur a analysé le code C++ fourni et a déterminé qu'il n'est pas valide. Il indique donc ce qu'il n'a pas compris (vous verrez par la suite, le compilateur est un ami qui va beaucoup vous aider en indiquant les erreurs que vous faites. Il faudra juste apprendre à le comprendre).

Ce résultat est normal, puisque l'on a écrit n'importe quoi dans le code ! Vous allez apprendre à écrire du code C++ correct dans la suite de ce cours, mais avant cela, vous devez comprendre les bases de la compilation d'un code C++ et de l'exécution d'un programme.

## La compilation d'un code C++

Fondamentalement, une application est une suite d'instructions donnée au processeur de votre ordinateur, qui lui indique les tâches qu'il doit accomplir. Un processeur ne connaît qu'un seul langage, appelé langage machine, spécifique à chaque type de processeur. Un langage machine est difficilement compréhensible pour les humains et il est donc très peu utilisé. C'est pour cela que les programmes sont écrits dans des langages plus compréhensibles pour les humains, appelés *langages de programmation* (le C++ est un exemple d'un tel langage).

Le problème est que le processeur est incapable de comprendre autre chose que le langage machine. Il faut donc passer par une étape qui convertit un code écrit dans un langage de programmation en programme en langage machine.

## **La compilation est le processus qui transforme le code C++ en programme exécutable par un ordinateur.**

Cette étape est donc indispensable pour tester vos programmes écrits en C++. La compilation est en fait constituée de plusieurs étapes (qui seront détaillées dans un prochain chapitre), chaque étape nécessitant l'utilisation d'un programme dédié. Vous verrez également dans un prochain chapitre différents outils de compilation, comment les installer et les utiliser. C'est pour éviter les problèmes liés à l'installation et l'utilisation de ces outils que vous utilisez dans un premier temps dans ce cours un environnement de compilation en ligne, qui possède déjà tous ces outils.

## **Classifications historiques des langages**

Historiquement, les langages étaient classifiés selon plusieurs critères. Par exemple, on distinguait les langages compilés et les langages interprétés. Ou les langages de haut niveau et bas niveau.

Avec un **langage compilé**, les étapes de compilation et d'exécution sont séparées dans le temps (comme vous venez de le voir pour le C++) alors que pour un **langage interprété**, la conversion en langage machine a lieu en même temps que l'exécution.

Un langage est dit de **bas-niveau** lorsqu'il est proche du langage machine, un langage est **haut-niveau** lorsqu'il est proche du langage humain. (les définitions de "haut" et "bas-niveau" ne sont pas formalisées et dépendent des auteurs. Peu importe ici, lire la suite).

De nos jours, cette distinction n'est plus forcément pertinente, beaucoup de langages peuvent être utilisés via compilation et interprétation, voire même un mélange des deux.

Revenons sur la compilation de notre code C++ minimal. Dans l'éditeur

utilisé, la compilation est réalisée en suivant les instructions données dans la partie du bas de la fenêtre. Cette partie (que vous pouvez également modifier) contient les instructions suivantes :

```
clang++ -std=c++11 -Wall -Wextra -pedantic -O2 main.cpp &&
./a.out
```

Ces instructions sont en fait des commandes Linux (le serveur utilisé par l'éditeur tourne sous Linux - mais les commandes sur Windows ou Mac OS X sont très proches). Si vous ne savez pas utiliser Linux en ligne de commande, ce n'est pas très grave (pour le moment... un développeur doit quand même avoir quelques bases sur l'utilisation des lignes de commandes).

Vous pourrez utiliser par la suite des outils qui se chargeront d'appeler ces instructions pour vous, de façon transparente. Mais gardez quand même en mémoire que quel que soit l'outil que vous utiliserez, celui-ci ne fera rien d'autre que d'appeler ces instructions, comme vous pourriez le faire vous-même (certains préfèrent d'ailleurs compiler manuellement leurs programmes en utilisant directement des lignes de commande).

Les instructions suivantes réalisent en fait deux tâches : lancer la compilation puis lancer l'exécution. Chaque étape est séparée par l'opérateur `&&` ou par un retour à la ligne. Ainsi, le code précédent peut également s'écrire :

```
clang++ main.cpp -std=c++14 -Wall -Wextra -pedantic -O2
./a.out
```

Remarque : dans l'éditeur, pour ajouter un retour à la ligne dans la partie du bas, il faut appuyer sur les touches `Shift` + `Entrée` du clavier. Appuyer simplement sur la touche `Entrée` lance la compilation et l'exécution.

La première ligne, qui nous intéresse pour le moment, permet de lancer la compilation. Cette ligne se décompose de la façon suivante :

```
<compilateur>      = clang++
<fichier à compiler> = main.cpp
```

```
<options de compilation> = -std=c++14 -Wall -Wextra  
-pedantic -O2
```

Voyons ces éléments en détail. La commande `clang++` permet de lancer un programme de compilation (le compilateur - qui convertit le code C++ en langage machine) appelé Clang. Ce compilateur est gratuit et l'un des plus à jour pour le support du C++. L'éditeur Coliru permet d'utiliser un autre compilateur appelé GCC, que vous pouvez utiliser en remplaçant `clang++` par `g++`.

La valeur `main.cpp` est le nom du fichier à compiler. Par défaut, le code dans l'éditeur Coliru est enregistré dans ce fichier, il faut donc indiquer à Clang qu'il doit compiler ce fichier.

Pour terminer, les options de compilation permettent de spécifier à Clang comment il doit compiler le code. Il existe plusieurs centaines d'options de compilation, il ne sera pas possible de toutes les détailler dans ce cours. Pour résumer les options utilisées ici, sachez que l'option `-std=c++14` permet d'activer le support de la norme la plus récente du C++, les options `-Wall -Wextra -pedantic` permettent de vérifier de nombreuses erreurs de programmation dans vos codes et l'option `-O2` permet d'optimiser le programme généré.

Il est important de connaître le processus et les options de compilation, cela sera détaillé dans la suite du cours. Cependant, pour les codes d'exemple et les exercices donnés dans ce cours, les instructions de compilation seront données, vous n'aurez pas besoin de les modifier.

Pour en savoir plus sur ces compilateurs, vous pouvez consulter les pages correspondantes de Wikipédia : [Clang](#) et [GCC](#). Vous pouvez également consulter les sites officiels : [Clang](#) et [GCC](#), en particulier les documentations pour connaître les options de compilation utilisables.

Notez aussi qu'il existe un dernier compilateur très utilisé, celui fourni par Microsoft : [Visual Studio](#). Ce n'est pas le compilateur qui fournit le meilleur support du C++14 - bien qu'ils aient fait de grands progrès sur ce point - mais il reste incontournable sur Windows.

## L'exécution d'un programme C++

Par défaut, le programme généré par la compilation s'appelle `a.out`. L'instruction `./a.out` permet donc de lancer son exécution. (Pour rappel, le code actuel ne fait rien, l'exécution ne produit aucun message dans la fenêtre de sortie de Coliru pour le moment).

Le schéma suivant résume l'ensemble des étapes de compilation et d'exécution :



Il est possible de changer le nom du programme généré en utilisation l'option `-o` (pour *output*, qui signifie *sortie*), par exemple :

```
clang++ main.cpp -o mon_programme -std=c++14 -Wall -Wextra  
-pedantic -O2  
./mon_programme
```

## La fonction main

Revenons maintenant sur le code C++ de notre programme. Ce code minimal définit une fonction appelée `main`, qui ne fait rien :

```
int main() {  
}
```

Pour le moment, vous ne savez pas créer une fonction en C++ et ce code peut vous paraître obscur. Savoir créer des fonctions est un point important de votre apprentissage du C++, mais il n'est pas possible de voir cela en détail dans un seul chapitre. Cela sera détaillé par la suite.

Mais pas de panique. Maîtriser les fonctions n'est pas nécessaire pour créer une fonction `main`. En effet, la façon d'écrire une fonction `main` (on

parle de sa *signature*) est définie par la norme C++. Donc, en pratique, vous devrez simplement copier à l'identique cette fonction pour créer un programme C++ de base.

Fondamentalement, un programme est simplement une suite d'instructions que l'on donne à l'ordinateur, pour réaliser une tâche. Tout comme il est souvent possible de décomposer une tâche en plusieurs sous-tâches plus simples, un programme sera décomposé en fonctions, qui sont des suites d'instructions destinées à réaliser une sous-tâche particulière.

Une fonction pourra appeler d'autres fonctions, qui pourront à leur tour appeler d'autres fonctions, jusqu'à ce que l'ensemble des fonctions appelées réalisent le travail demandé.

La fonction `main` est un peu particulière puisqu'elle est appelée par le système d'exploitation. Elle est obligatoire dans un programme. Si vous ne créez pas de fonction `main` ou si vous en créez plusieurs, le système d'exploitation ne saura pas comment lancer votre application et produira une erreur. Par contre, vous ne pouvez pas vous-même appeler cette fonction, seul le système peut l'appeler.

Vous apprendrez par la suite à créer des fonctions et leurs syntaxes en détail, mais pour comprendre la fonction `main`, voici quelques explications. Comme indiqué ci-dessus, une fonction est une suite d'instructions. Elle peut être appelée par une autre fonction. L'appel d'une fonction se déroule en trois étapes :

- le code appelant appelle la fonction à partir de son nom, en lui transmettant des informations si besoin ;
- la suite d'instructions correspondant à la fonction appelée est exécutée ;
- la fonction se termine en transmettant éventuellement une réponse à la fonction appelante.

Ces trois étapes apparaissent dans la déclaration d'une fonction, avec la syntaxe suivante :

```
| InformationsRetournées NomDeLaFonction(InformationsEnvoyées) |
```

```
{  
    Suite d'instructions  
}
```

La déclaration d'une fonction commence par définir les informations retournées par la fonction (encore appelé *paramètre de retour* de la fonction) lorsqu'elle se termine. Il ne peut y avoir qu'un seul paramètre de retour de fonction (mais ce point n'est pas limitant, puisqu'un paramètre peut contenir plusieurs informations). Lorsqu'une fonction ne retourne aucune information, le paramètre de retour est `void`. Dans le cas de la fonction `main`, celle-ci retourne toujours une information de type `int`, qui signifie un nombre entier (*integer* en anglais).

Vient ensuite le nom de la fonction. Le nommage des fonctions (et plus généralement de tous les éléments que vous allez pouvoir définir dans un programme C++) suit des règles spécifiques, qui seront détaillées plus tard. Pour faire simple, un nom est une suite de caractères alphanumériques (n'importe quelle lettre de l'alphabet sans accentuation et n'importe quel chiffre, sauf en première position), avec des majuscules et/ou des minuscules. Certains "mots" sont réservés par la norme C++, il vous est interdit de les utiliser. Il est également habituel d'écrire ses programmes en anglais, je vous conseille de faire de même.

On voit ici que l'écriture d'un programme C++ ne suit pas simplement des règles imposées par le langage (la norme), mais également les habitudes et bonnes pratiques mises en place avec le temps. Suivre ces règles permet :

- de fournir un cadre commun à tous les développeurs C++, pour faciliter la communication ;
- de simplifier la création du code, en proposant une approche "classique" de faire les choses ;
- d'apporter des garanties sur la qualité du code, pour qu'il fasse exactement ce que l'on attend de lui.

Il est classique de voir des débutants ne pas suivre ces bonnes pratiques durant leur phase d'apprentissage et justifier cela en disant que lorsqu'ils

travailleront sur de vrais projets, ils suivront ces règles. C'est une mauvaise idée de faire comme cela : une pratique (bonne ou mauvaise) acquise durant sa phase d'apprentissage sera très dure à faire évoluer. Il est important de pratiquer correctement dès le début (même si cela peut sembler faire perdre du temps au début, le temps d'acquérir des habitudes de programmation sera largement rattrapé).

Les informations d'entrée (ou *paramètres d'entrée*) sont définies à la suite du nom de la fonction, entre parenthèses. Lorsqu'il n'y a pas de paramètres d'entrée, on met simplement les parenthèses sans rien dedans (les parenthèses doivent toujours être présentes). Remarquez qu'il ne peut y avoir qu'un seul paramètre de retour de fonction, mais que l'on peut avoir plusieurs paramètres d'entrée.

Pour terminer, la partie la plus importante : la suite d'instructions, dans un bloc de code défini par des accolades `{` et `}`. Chaque instruction se termine par un point-virgule. Dans le code d'exemple de ce chapitre, le bloc d'instructions est vide, il n'y a que les accolades, le programme ne fait rien (mais vous verrez dès le prochain chapitre les bases pour écrire des instructions).

En fait, il existe plusieurs *signatures* pour la fonction `main` (i.e. plusieurs façons différentes d'écrire cette fonction). Cela permet en particulier au système d'envoyer des informations lors du lancement du programme, que vous pourrez utiliser dans votre code. Pour le moment, vous ne savez pas encore comment traiter ces informations, donc il n'est pas nécessaire de détailler ce point.

## Mise en forme du code

Vous avez maintenant les informations de base pour comprendre ce premier programme C++ (qui ne fait rien). Pour terminer ce chapitre, deux points importants, sur la présentation du code.

Dans un code C++, les espaces et les retours à la ligne ne sont pas pris en compte dans la compilation (sauf bien sûr si vous accollez deux termes

ensembles, si vous écrivez `intmain` sans espace, votre compilateur vous insultera copieusement). Vous pouvez donc ajouter des espaces et des retours à la ligne de façon à rendre votre code le plus lisible possible, sans que cela ne change votre programme.

Un point important à ne pas oublier : un code sera plus souvent lu qu'il n'est écrit ou modifié. Il faut donc privilégier la qualité de lecture d'un code, plutôt que d'essayer de gagner du temps à l'écriture (même pour un simple code de test ou d'apprentissage). Présenter correctement un code permet de gagner du temps sur le long terme.

Ainsi, le programme d'exemple peut s'écrire selon les façons suivantes :

```
int main(){}
```

```
int  
main()  
{  
}
```

```
int      main      (      )      {      }
```

Il faut trouver un compromis entre la concision (écrire un code qui sera le plus compact possible) et avoir un code aéré. Il est habituel de définir des règles d'écriture du code, pour faciliter la lecture et permettre à plusieurs personnes de comprendre le code des autres développeurs. Il existe plusieurs conventions pour ces règles, à vous de choisir celles qui vous conviennent.

**Peu importe les règles de codage que vous choisissez, le plus important est surtout d'avoir des règles et de les respecter.**

En particulier, un point important est le respect de l'indentation. L'indentation correspond aux espaces placés en début d'une ligne. Le début des lignes doit être aligné selon son niveau hiérarchique. Si on écrit la hiérarchie suivante :

```
niveau 1  
niveau 2  
niveau 3  
niveau 3  
niveau 2  
niveau 3  
niveau 1
```

Il est assez difficile de visualiser facilement à quel niveau hiérarchique correspond chaque ligne. Si on utilise une indentation pour distinguer chaque niveau, le code est beaucoup plus lisible :

```
niveau 1  
    niveau 2  
        niveau 3  
        niveau 3  
    niveau 2  
        niveau 3  
niveau 1
```

Vous trouverez des exemples de styles d'indentation du code dans [la page de Wikipédia](#) correspondante. Dans ce cours, j'utiliserais le style K&R, avec une indentation de quatre espaces.

Il existe des outils permettant de mettre en forme le code et vérifier que la présentation du code respecte les règles que vous avez fixé. Ces outils seront présentés dans la suite de ce cours.

## Commentaires du code

Pour comprendre un code, les premières informations que le lecteur verra sont les noms que l'on donne. Appeler par exemple ses fonctions `f1`, `f2` et `f3` n'aide pas du tout à comprendre à quoi servent ces fonctions. Par contre, appeler ces fonctions `add` (addition), `save` (enregistrer) ou `reset` (remettre à zéro) permet aux lecteurs d'avoir une idée de leur rôle.

**Il est important de prendre le temps de nommer correctement**

## **les choses, ce n'est pas une perte de temps.**

Cependant, il n'est pas toujours possible de trouver des noms significatifs (tout au moins, sans faire des noms de 100 caractères). Dans ce cas, il est possible d'ajouter des commentaires dans le code, qui seront ignorés par le compilateur (et donc ne changeront pas le comportement du programme généré) et seront destinés uniquement aux développeurs.

Il existe deux formes de commentaire. Les commentaires sur une ligne et les commentaires sur plusieurs lignes. Les commentaires peuvent être placés n'importe où dans votre code. Pour écrire un commentaire sur une ligne, vous devez utiliser deux barres obliques `/`/`/` suivies du commentaire. Pour un commentaire sur plusieurs lignes, il faut commencer le commentaire par une barre oblique puis un astérisque `/*` et terminer le commentaire par un astérisque puis une barre oblique `*/`.

```
// un commentaire sur une ligne

int main() { // un commentaire en fin d'une ligne
    /* un
       commentaire
       sur
       plusieurs
       lignes */
}
```

## **Exercices**

### **Compilation et exécution**

Dans les commandes de compilation et d'exécution, il est possible d'utiliser la commande `echo` suivie d'un texte entre guillemets pour afficher un message.

1. Modifier les instructions de compilation pour afficher un message avant la compilation et un autre après l'exécution :

```
**** Compilation du programme C++ ****
...
```

```
**** Exécution du programme C++ ****
```

```
...
```

2. Modifier le code de compilation pour compiler deux fois le programme, avec les options d'avertissement et sans les options d'avertissement :

```
**** Compilation du programme C++ avec les avertissements
```

```
****
```

```
...
```

```
**** Compilation du programme C++ sans les avertissements
```

```
****
```

```
...
```

```
**** Exécution du programme C++ ****
```

```
...
```

3. Modifier le code de compilation et d'exécution pour nommer le programme “mon\_programme” au lieu de “a.out”.

## La fonction main

En fait, en plus des deux syntaxes possibles pour la fonction `main`, il existe une troisième syntaxe.

1. Trouvez dans la [documentation du langage C++](#) la page correspondant à la fonction `main`.
2. Trouvez ensuite la troisième syntaxe possible pour la fonction `main`.
3. Il est courant d'écrire au début de chaque code la licence d'utilisation du code, les coordonnées de l'auteur, la date de modification et d'autres informations utiles. Modifier le code de la fonction `main` pour ajouter ces informations sous forme de commentaires.

[Chapitre précédent](#) [Sommaire principal](#) [Chapitre suivant](#)

[Cours, C++](#)

# Le programme "hello world"

Dans le chapitre précédent, vous avez vu la structure de base d'un programme C++ et un aperçu du processus de compilation. Cependant, le code présenté ne faisait rien, ce n'était pas très intéressant. Dans ce chapitre, vous allez voir comment afficher un message.

Pour illustrer cette fonctionnalité en C++, nous allons prendre le programme "hello world" comme exemple. Ce programme permet simplement d'afficher le message "hello, world!". Il est traditionnellement utilisé pour montrer la syntaxe de base d'un langage informatique, ce qui explique qu'il possède son propre nom. Vous pouvez voir sur Wikipédia ce programme dans différents langages : [Wikipédia](#).

Le programme *hello world* en C++ est assez proche du programme minimal présenté dans le chapitre précédent. Vous pouvez ouvrir ce code dans [Coliru](#) et copier-coller le code dans l'éditeur de votre choix.

```
main.cpp
```

```
#include <iostream>

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

Par rapport au code minimal, vous pouvez voir que l'on a ajouté deux lignes. La première ligne, qui contient la directive `#include`, permet de spécifier des fonctionnalités utilisées par le programme. "iostream" fournit les fonctionnalités de base pour afficher un message et récupérer les textes saisies par l'utilisateur.

La quatrième ligne, commençant par `std::cout`, permet l'affichage proprement dit. On voit sans problème le message à afficher "hello, world!" en clair dans le code. Vous pouvez vous amuser à changer le texte et voir ce que cela donne. L'instruction `std::endl` permet d'indiquer la fin d'une ligne et donc de passer à la ligne suivante (`endl`)

pour “end line”, “fin de ligne”)

Ces deux lignes de code permettent d'utiliser le flux de sortie `std::cout` de la bibliothèque standard.

## La bibliothèque standard

Lorsque l'on parle du C++, il faut en fait distinguer deux choses : le langage C++ et la bibliothèque standard. Le langage C++ proprement-dit propose uniquement les fonctionnalités fondamentales indispensables pour écrire un programme. La conséquence est que de nombreuses fonctionnalités ne sont pas intégrées directement dans le langage (même l'affichage d'un message ne fait pas partie du langage).

Heureusement, lorsqu'une fonctionnalité n'existe pas dans le langage, cela ne veut pas dire que l'on ne peut pas utiliser cette fonctionnalité. Il est possible d'écrire des bibliothèques, qui fournissent de nouvelles fonctionnalités utilisables en C++ (ou dans d'autre langages, mais cela sort du cadre de ce cours).

Cela veut dire aussi que si vous créez un programme qui propose des fonctionnalités intéressantes, vous pouvez également créer une bibliothèque pour que d'autres développeurs utilisent vos fonctionnalités (ou que vous puissiez vous même utiliser ces fonctionnalités dans plusieurs de vos programmes). La création d'une bibliothèque sera vue par la suite.

**La bibliothèque standard fait partie intégrante du C++, il est indispensable d'apprendre à l'utiliser en même temps que le langage, l'un ne va pas sans l'autre.**

Pour utiliser une fonctionnalité de la bibliothèque standard, il faut dans un premier temps le spécifier au compilateur, en utilisant la directive de pré-processeur `#include`. L'une des syntaxes de cette directive est la suivante (il en existe d'autres, mais qui ne seront pas utilisée avec la bibliothèque standard) :

```
#include <nom_fichier>
```

## Les directives de pré-processeur

Une directive de pré-processeur permet de paramétriser le comportement du pré-processeur lors de la compilation. Il existe différentes directives, vous en verrez plusieurs dans ce cours. Une directive s'écrit toujours avec un dièse suivi de la directive et d'éventuels paramètres optionnels.

Il n'est pas possible d'expliquer le fonctionnement de la directive `#include` sans expliquer avant le fonctionnement en détail du pré-processeur. Cela sera vu dans un chapitre dédié à la compilation en profondeur.

Pour afficher un message, on utilise un objet particulier de la bibliothèque standard, nommé `std::cout`. Si vous regardez dans la [documentation](#), vous voyez au début de la page qu'il est écrit : "Defined in header <iostream>".

The screenshot shows a web browser displaying the [cppreference.com](https://en.cppreference.com/w/cpp/io/basic_ostream) website. The URL in the address bar is `https://en.cppreference.com/w/cpp/io/basic_ostream`. The page title is "std::cout, std::wcout". The content area includes the following text:

Defined in header `<iostream>`

```
extern std::ostream cout;      (1)
extern std::wostream wcout;    (2)
```

The global objects `std::cout` and `std::wcout` control output to a stream buffer of implementation-defined type (derived from `std::streambuf`), associated with the standard C output stream `stdout`.

These objects are guaranteed to be initialized during or before the first time an object of type `std::ios_base::Init` is constructed and are available for use in the constructors and destructors of static objects (as long as `<iostream>` is included).

Unless `sync_with_stdio(false)` has been issued, it is safe to concurrently access these objects from multiple threads for both formatted and unformatted output.

Once initialized, `std::cout` is tie()'d to `std::cin` and `std::wcout` is tie()'d to `std::wcin`, meaning that any input operation on `std::cin` executes `std::cout.flush()` (via `std::basic_istream::sentry`'s constructor).

Once initialized, `std::cout` is also tie()'d to `std::cerr` and `std::wcout` is tie()'d to `std::wcerr`, meaning that any output operation on `std::cerr` executes `std::cout.flush()` (via `std::basic_ostream::sentry`'s constructor) (since C++11).

Cela vous indique quel fichier il faut inclure pour utiliser `std::cout` : le fichier `iostream` :

```
#include <iostream>
```

"iostream" correspond à **Input/Output stream**, ce qui signifie "flux d'entrée et sortie". "Entrée" et "Sortie" doivent être compris du point de vue du programme : "entrée" d'information depuis l'extérieur vers

l'intérieur du programme (par exemple saisie d'un texte par l'utilisateur ou la lecture d'un fichier) et "sortie" d'information depuis le programme vers l'extérieur (par exemple afficher un message à l'écran ou enregistrer dans un fichier). Vous verrez juste en dessous pourquoi on parle de "flux".

Lorsque vous utiliserez une fonctionnalité de la bibliothèque standard que vous ne connaissez pas, vous pourrez de la même manière aller rechercher dans la documentation quel fichier inclure.

## L'espace de nom std

En C++, chaque chose doit avoir un nom unique, pour permettre au compilateur de les identifier correctement. Donner un nom n'est pas très compliqué, vous verrez par la suite les quelques règles à respecter. Lorsque l'on a un petit programme de quelques centaines ou milliers de ligne, cela pose pas trop de problème pour trouver des noms uniques. Mais dans le cas d'un programme de plus grande taille ou utilisant différentes bibliothèques, cela peut devenir très compliqué.

Pour éviter cette contrainte, le C++ permet de regrouper les noms dans un espace dédié : les espaces de noms (*namespace*). En créant un espace de noms, vous évitez les conflits entre les noms, ce qui peut simplifier vos codes. La bibliothèque standard utilise un espace de noms appelé `std`. Vous apprendrez par la suite à créer des espaces de noms, mais pour l'instant, voyons comment utiliser l'espace de noms de la bibliothèque standard.

Pour utiliser l'objet `cout` de la bibliothèque standard, il faut donc préciser que celui-ci provient de l'espace de noms `std`. Plusieurs solutions sont possibles, selon le contexte. Premièrement, vous pouvez déclarer l'espace de noms `std` à chaque utilisation d'une fonctionnalité de la bibliothèque standard, en utilisant l'opérateur `::` (comme vous l'avez vu dans les codes précédents) :

main.cpp

```
#include <iostream>
```

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

Dans ce cas, il faut faire précéder chaque utilisation de `cout` et de `endl` avec l'espace de noms.

La deuxième solution est de déclarer que vous allez utiliser une fonctionnalité d'un espace de noms en utilisant le mot-clé `using`. Lorsque le compilateur rencontre ensuite `cout`, il saura qu'il faut utiliser l'objet `std::cout` :

```
main.cpp
#include <iostream>
using std::cout;

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

Vous pouvez remarquer ici que seul l'objet `cout` est déclaré en utilisant `using`. `endl` n'étant pas déclaré de cette manière, il faut l'écrire en utilisant la première syntaxe.

Pour terminer, il est possible d'activer un espace de noms globalement, en utilisant `using namespace`.

```
main.cpp
#include <iostream>
using namespace std;

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

Vous remarquez ici qu'il n'est plus nécessaire d'écrire `std::` devant `cout` et `endl` (ou n'importe quelle autre fonctionnalités de la bibliothèque standard).

Cette syntaxe semble intéressante, puisque cela fait gagner du temps.

Cependant, cela signifie que l'on perd l'intérêt des espaces de noms : si deux espaces de noms proposent le même identifiant, il y aura un conflit. Le message d'erreur généré par le compilateur ne sera pas forcément explicite, puisqu'il ne détectera pas le conflit d'espace de noms. Les erreurs générées seront du type "déclaration ambiguë" ou "déclaration multiple".

Il est possible de limiter la portée de `using` en le déclarant à l'intérieur de la fonction :

main.cpp

```
#include <iostream>

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

et :

main.cpp

```
#include <iostream>

int main() {
    using namespace std;
    cout << "Hello, world!" << endl;
}
```

Il est préférable de limiter l'utilisation de la syntaxe avec `using namespace`. Dans ce cours, nous utiliserons systématiquement la première syntaxe, pour des raisons de compréhension. Dans vos codes, utilisez de préférence l'une des deux premières syntaxes.

## Les flux standards

Si vous avez regardé un peu la documentation de `std::cout`, vous avez peut-être remarqué qu'il existe d'autres flux de sortie :

- `cout` et `wcout` pour les messages standard ;

- `cerr` et `wcerr` pour les messages d'erreur ;
- `clog` et `wclog` pour les messages de log.

Par défaut, ces flux s'affichent tous dans le terminal, vous pouvez utiliser n'importe lequel. Le programme suivant :

### main.cpp

```
#include <iostream>

int main() {
    std::cout << "le flux cout" << std::endl;
    std::wcout << "le flux wcout" << std::endl;
    std::cerr << "le flux cerr" << std::endl;
    std::wcerr << "le flux wcerr" << std::endl;
    std::clog << "le flux clog" << std::endl;
    std::wclog << "le flux wclog" << std::endl;
}
```

affiche :

```
le flux cout
le flux wcout
le flux cerr
le flux wcerr
le flux clog
le flux wclog
```

Il est possible de faire en sorte de récupérer spécifiquement un flux, par exemple pour l'enregistrer dans un fichier. Certains éditeur de code récupèrent par exemple les messages affichés à l'aide de `std::cerr` pour afficher les messages d'erreur.

Même si utiliser n'importe quel flux ne change rien à votre programme (si les flux ne sont pas spécifiquement récupérés), il est préférable de respecter le rôle de chaque flux et d'utiliser `std::cerr` pour les messages d'erreur, `std::clog` pour les messages d'information et `std::cout` pour les messages standards.

## Internationalisation

Le `w` signifie que le flux prend en charge les caractères étendus (“w” pour *wide*), c'est-à-dire les caractères avec accent ou provenant d'un alphabet différent de l'anglais. La gestion de l'internationalisation est un peu complexe en C++, en particulier à cause de la multiplicité des normes de codage et la prise en charge très variable selon le système d'exploitation. Cela fera l'objet d'un chapitre dédié.

Si vous faites le test avec Clang dans Coliru, cela affichera les accents correctement, mais ça ne sera pas toujours le cas (en particulier sous Windows).

main.cpp

```
#include <iostream>

int main() {
    std::cout << "àâäéèëïïôöùû" << std::endl;
}
```

Comment fonctionne un flux ? Imaginer un employé de bureau qui reçoit des dossiers. Il a une grande pile de dossiers, il prend le plus ancien, le traite, puis passe au suivant. Peu importe si les dossiers arrivent un par un ou en paquet, il les prend toujours un par un.

Les flux standards fonctionnent sur le même principe : ils reçoivent des données (les caractères à afficher), ils prennent le premier arrivé, l'affiche puis passent au suivant. L'opérateur permettant d'envoyer des données à un flux est l'opérateur `<<` :

main.cpp

```
#include <iostream>

int main() {
    std::cout << "hello";
}
```

Ce code ne doit pas être compris comme signifiant “afficher ‘hello’”, mais comme “envoyer les caractères ‘h’, ‘e’, ‘l’, ‘l’ et ‘o’ dans le flux standard ‘std::cout’”.

Il est possible d'envoyer plusieurs valeurs en série de données dans un flux, en les séparant par plusieurs opérateurs `<<` :

main.cpp

```
#include <iostream>

int main() {
    std::cout << "hello" << "world";
}
```

Ce code signifie “envoyer 'hello' et 'world' dans 'std::cout'”, ce qui peut être développé en “envoyer les caractères 'h', 'e', 'l', 'l', 'o', 'w', 'o', 'r', 'l', 'd' dans 'std::cout'”. (Remarquez l'absence d'espace entre “hello” et “world”).

Il revient au même d'envoyer les données sur plusieurs lignes :

main.cpp

```
#include <iostream>

int main() {
    std::cout << "Hello";
    std::cout << "world";
}
```

Voir même d'envoyer les caractères un par un :

main.cpp

```
#include <iostream>

int main() {
    std::cout << 'h';
    std::cout << 'e';
    std::cout << 'l';
    std::cout << 'l';
    std::cout << 'o';
    std::cout << 'w';
    std::cout << 'o';
    std::cout << 'r';
    std::cout << 'l';
    std::cout << 'd';
}
```

}

Tous les codes précédent affichent :

helloworld

## Programmation impérative

Il existe plusieurs façons de concevoir un programme informatique, appelés [paradigme de programmation](#). Le C++ est un langage qui autorise l'utilisation de plusieurs paradigme, il est multi-paradigme. Il supporte en particulier la programmation impérative, que va être détaillé dans la suite, la [programmation générique](#) et la [programmation objet](#), qui seront vues plus tard dans ce cours, la [programmation fonctionnelle](#), qui sera simplement citée, etc.

La [programmation impérative](#) décris un programme comme une suite d'instructions, qui modifie l'état du programme. Cette approche permet de suivre un programme pas-à-pas, depuis le début du programme (correspondant à la fonction `main` en C++) jusqu'à sa fin (lorsque la fonction `main` se termine en C++).

Les instructions sont exécutées une par une, dans l'ordre où elle apparaissent dans le code. Une instruction commence dès que la précédente se termine.

Vous pourrez également entendre parler de [programmation procédurale](#) et de [programmation structurée](#), toutes deux appartenant au paradigme impératif. Dans la programmation procédurale, le programme est décomposé en "procédures" (en C++, on parle de "fonctions") qui sont des suites d'instructions. Dans la programmation structurée, le programme n'est pas linéaire, mais ajoute les concepts de boucles (une suite d'instruction qui est répétée) et de conditions (une suite d'instruction qui peut être exécutée ou non). Cela sera détaillé dans la suite de ce cours.

Les ordinateurs modernes sont généralement capables d'exécuter plusieurs instructions en même temps, voire plusieurs programme en même temps. Cela complique forcement la compréhension du déroulement d'un programme. Pour simplifier les explications, ce cours se base sur une situation parfaite et abstraite, dans laquelle les instructions sont exécutées une par une.

Voyons sur un code simple comment suivre le déroulement d'un programme C++. Lorsque vous exécutez le programme suivant, que se passe-t-il ?

main.cpp

```
#include <iostream>

int main() {
    std::cout << "ligne 1" << std::endl;
    std::cout << "ligne 2" << std::endl;
    std::cout << "ligne 3" << std::endl;
}
```

Pour commencer, le système lance le programme et appelle la fonction `main`. Dans cette étape, il se passe beaucoup de choses, mais qui concernent le système (et qui sont relativement complexes). Cela sort du cadre de ce cours. Mais vous pouvez considérer que le programme commence au niveau de la ligne contenant le `main`.

A cette étape, le programme n'a encore rien écrit dans la console (le système peut avoir écrit des choses, mais cela ne concerne pas le programme).

Une fois que le programme est lancé, la première instruction est exécutée. Cette première instruction est la première ligne contenant le `std::cout`, ce qui produit l'affichage de texte dans la console. La console affiche donc à la fin de cette étape :

ligne 1

L'étape suivante est la seconde ligne contenant `std::cout` et la console affiche donc :

```
ligne 1  
ligne 2
```

La troisième étape est la ligne contenant le dernier `std::cout` :

```
ligne 1  
ligne 2  
ligne 3
```

Pour terminer, le programme arrive à la fin de la fonction `main`, correspondant à l'accolade fermante `}`. Le programme se termine et le système reprend la main.

Pour simplifier la lecture du code, chaque instruction est écrite sur une ligne, se terminant par un point-virgule `;`. Mais si une ligne contient plusieurs instructions, séparées par des points-virgules, cela ne change pas le déroulement du programme : chaque instruction est exécutée une par une. La présentation du code n'influence pas le déroulement du programme.

Pour suivre le déroulement d'un programme C++, vous avez deux garanties :

- chaque instruction est exécutée ;
- les instructions sont exécutées dans l'ordre.

Il n'est donc pas possible d'obtenir les résultats suivants dans la console :

```
ligne 1  
ligne 3
```

ou

```
ligne 1  
ligne 3  
ligne 2
```

Le comportement d'un programme sera donc prédictible (sauf erreur de programmation) et constant.

## Exercices

- Testez le code en “oubliant” de mettre la directive de compilation `#include`. Quels messages d'erreur sont produit ? Trouvez-vous que les messages sont explicites et permettent de trouver facilement le problème ?
- Testez le code en “oubliant” de mettre l'espace de noms `std` devant `cout` et `endl`. Même questions que précédemment.
- Testez d'autres erreurs dans le code et regardez les messages d'erreur produits.

<a href="#">Chapitre précédent</a>	<a href="#">Sommaire principal</a>	<a href="#">Chapitre suivant</a>
------------------------------------	------------------------------------	----------------------------------

Todo : tabulation, retour à la ligne, quoted. Caractère spéciaux sous forme hexa.

# Les littérales chaînes de caractères

## Chaînes de caractères et caractères

Comme vous pouvez remarquer, on distingue en C++ les caractères uniques et les chaînes de caractères (un ensemble de caractères). Un caractère unique sera écrit avec un guillemet simple droit, une chaîne de caractères sera écrite avec des guillemets doubles droits :

```
main.cpp
#include <iostream>

int main() {
    std::cout << "hello"; // une chaîne
    std::cout << 'h'; // un caractère
}
```

Si vous essayez de mettre plusieurs caractères entre guillemets droits, vous obtiendrez généralement un message d'avertissement. Par exemple, le message est avec Clang dans Coliru :

```
main.cpp
#include <iostream>

int main() {
    std::cout << 'hello, world!' << std::endl;
}
```

affichera :

```
main.cpp:4:18: warning: character constant too long for its
type
```

```
    std::cout << 'hello, world!' << std::endl;
        ^
```

1919706145

Remarquez bien que cela provoque un avertissement (*warning*) et non une erreur : le programme s'exécute quand même. La chaîne est interprétée comme un nombre, qui est affiché en dessous de l'avertissement (1919706145 dans l'exemple précédent). Les avertissements signifient que le compilateur ne sait pas si le code contient une erreur ou si c'est intentionnel de la part du développeur.

**You must never ignore warnings. In this course, a program that displays warnings will not be considered correct.**

Les chaînes de caractères ne sont qu'une suite de caractères, sans aucun sens particulier pour le C++. Cela implique que ce que vous mettez dedans ne sera pas évalué. Si vous écrivez par exemple :

```
main.cpp
#include <iostream>

int main() {
    std::cout << "std::endl" << std::endl; // code dans une
chaîne
    std::cout << std::endl << std::endl; // code hors
d'une chaîne

    std::cout << "1+2" << std::endl; // expression
dans une chaîne
    std::cout << 1+2 << std::endl; // expression
hors d'une chaîne
}
```

affiche :

```
std::endl
```

```
1+2
```

On voit dans cet exemple simple que les chaînes “`std::endl`” et “`1+2`” s'affichent exactement tel quel, alors que le même code en dehors de la chaîne est évalué.

## Les caractères spéciaux

Si vous vous êtes amusé à tester différents messages à afficher, vous avez peut-être essayé d'afficher une barre oblique inversée `\` (backslash) ou des guillemets `"`. Si ce n'est pas le cas, essayez maintenant :

main.cpp

```
#include <iostream>

int main() {
    std::cout << "hello " world" << std::endl;
}
```

Le compilateur ne va pas du tout aimer ce code et produira des erreurs et avertissements :

```
main.cpp:4:28: error: expected ';' after expression
    std::cout << "hello " world" << std::endl;
                                         ^
;

main.cpp:4:34: warning: missing terminating '"' character
[-Winvalid-pp-token]
    std::cout << "hello " world" << std::endl;
                                         ^

main.cpp:4:29: error: use of undeclared identifier 'world'
```

```
std::cout << "hello " world" << std::endl;  
^
```

```
1 warning and 2 errors generated.
```

L'erreur est simple à comprendre avec un peu de logique. En C++, une chaîne de caractères est délimitée par des guillemets. Le compilateur va rencontrer le premier guillemet et va l'interpréter comme le début d'une chaîne (et donc tout ce qui suit sera considéré comme des caractères faisant partie de la chaîne).

```
"hello " world"  
^ ^ ^  
1 2 3
```

Le compilateur continue de parcourir le code et va arriver au second guillemet. Le problème est qu'il ne sait pas qu'il y a encore un guillemet ensuite, il va considérer que ce guillemet est la fin de la chaîne. Tout ce qui suit sera donc considéré comme du code C++ et non comme des caractères à afficher. Or, comme la suite n'est pas du code C++ valide, le compilateur produit une erreur.

Pour résoudre ce problème, il faut pouvoir dire au compilateur que l'on souhaite utiliser le caractère guillemet et non indiquer la fin de la chaîne. Pour cela, il faut utiliser un caractère spécial, le caractère d'échappement `\`. Ce caractère sera interprété de la façon suivante par le compilateur lorsqu'il est rencontré dans une chaîne : “attention, le caractère suivant est un caractère spécial”.

En particulier, le caractère `\"` est interprété comme étant le caractère guillemet et non la fin d'une chaîne. Avec cette correction, le code devient :

```
main.cpp  
#include <iostream>  
  
int main() {  
    std::cout << "hello \" world" << std::endl;
```

```
}
```

affiche :

```
hello " world
```

Remarquez que l'ensemble des deux caractères `\` et `"` est interprété comme étant un seul caractère.

Un autre problème se présente. Puisque le caractère `\` est considéré comme étant le caractère d'échappement, comment afficher le caractère `\` dans une chaîne ? La solution est simple, il suffit d'utiliser le caractère '`\`'.

Encore une fois, il s'agit bien d'une séquence d'échappement, qui correspond à un seul caractère dans la chaîne et qui est constitué du caractère d'échappement `\` puis de l'identifiant `\`.

```
main.cpp
```

```
#include <iostream>

int main() {
    std::cout << "hello \\ world" << std::endl;
}
```

affiche :

```
hello \ world
```

La liste des séquences d'échappement est donnée dans la documentation : [Escape sequences](#).

## Les chaînes de caractères brutes

Dans certains cas, une chaîne va contenir de nombreux caractères spéciaux (par exemple, le chemin d'un fichier sous Windows, qui peut contenir plusieurs `\` ou des expressions régulières). Dans ce cas, il devient fastidieux (c'est pas faux !) de devoir ajouter systématiquement le caractère d'échappement `\` devant chaque caractère spécial.

Pour éviter cela, il est possible d'utiliser une chaîne brute, dans laquelle les caractères spéciaux sont ignorés. Une chaîne brute doit commencer (par défaut) par `R"(` et se terminer par `)"`, tout ce qui se trouve entre les deux sera interprété comme des caractères à afficher.

main.cpp

```
#include <iostream>

int main() {
    std::cout << R"(Je fais des " tests pour \ apprendre le C++ !)" << std::endl;
}
```

affiche :

```
Je fais des " tests pour \ apprendre le C++ !
```

Remarquez bien que les parenthèses en début et fin de chaînes (celles correspondantes à `R"(` et `)"`) ne font pas partie de la chaîne et ne sont pas affichées.

Si vous êtes attentif (et que vous cherchez un peu les problèmes...), vous avez peut être testé un code similaire au suivant :

main.cpp

```
#include <iostream>

int main() {
    std::cout << R"(Un autre )" test pour apprendre le C++ ! )"
    << std::endl;
}
```

On retrouve le problème précédent, à savoir que l'on a une séquence de caractères dans la chaîne qui est similaire à la séquence de caractères utilisée pour indiquer la fin de la chaîne. Ce code retourne, comme on peut s'y attendre, des erreurs :

```
main.cpp:4:30: error: expected ';' after expression
    std::cout << R"(Un autre )" test pour apprendre le C++ !
) " << std::endl;
```

```
^  
;  
main.cpp:4:31: error: unknown type name 'test'  
    std::cout << R"(Un autre )" test pour apprendre le C++ !  
)" << std::endl;  
^  
main.cpp:4:40: error: expected ';' at end of declaration  
    std::cout << R"(Un autre )" test pour apprendre le C++ !  
)" << std::endl;  
^  
main.cpp:4:61: warning: missing terminating '"' character  
[-Winvalid-pp-token]  
    std::cout << R"(Un autre )" test pour apprendre le C++ !  
)" << std::endl;  
^  
1 warning and 3 errors generated.
```

Heureusement, il existe une solution : il est possible d'ajouter une séquence de caractères quelconque dans les guillemets et les parenthèses délimitant la chaîne. Cette séquence de caractères permet de spécifier des délimiteurs différents de n'importe quelle séquence contenu dans la chaîne et donc d'éviter les ambiguïtés. Par exemple :

```
main.cpp  
#include <iostream>  
  
int main() {  
    std::cout << R"*(Un autre )" test pour apprendre le C++ ! )  
*" << std::endl;  
}
```

affiche le résultat attendu :

```
Un autre )" test pour apprendre le C++ !
```

Ici, la chaîne brute est délimitée par les séquences `R"*(` et `)*"`.

## Exercices

Modifier le code suivant pour afficher les messages demandés. [Faire l'exercice](#)

## main.cpp

```
#include <iostream>

int main() {
    // Modifier la ligne suivante pour afficher "Bienvenue
tout le monde !"
    std::cout << "Hello, world!" << std::endl;

    // Ajouter UNE ligne pour afficher "Bienvenue !" et
    "Tout le monde !" sur DEUX lignes
    std::cout << @@@@@

    // Ajouter DEUX lignes pour afficher "Bienvenue !" et
    "Tout le monde !" sur UNE ligne
    std::cout << @@@@@
    std::cout << @@@@@

    return 0;
}
```

**Exos** : utiliser tabulation pour afficher un tableau

**Exos** : donner des chaînes à afficher et écrire le code correspondant

## Exos : Afficher un tableau avec bordures

## main.cpp

```
#include <iostream>

int main() {
    std::cout << " "
    std::cout << " "
    std::cout << " "
    std::cout << " "
    std::cout << " "
}
```

}

affiche :

12	34	56
ab	cd	ef

Faire pareil, avec double bordure.

### Exos : ASCII art

[Chapitre précédent](#) [Sommaire principal](#) [Chapitre suivant](#)

Cours, C++

# Notes sur la qualité logicielle

Vous avez appris à réaliser des calculs et à afficher le résultat avec `cout`. C'est un bon début, puisque par définition, c'est le rôle d'un ordinateur de réaliser automatiquement des calculs et opérations logiques. Cependant, on voit vite une limite. Imaginez que vous devez réutiliser le résultat d'un premier calcul dans un second calcul, comment faire ?

Actuellement, la seule solution que vous connaissez est de réécrire le premier calcul dans le second calcul. Par exemple, pour additionner 123 et 456, puis multipliez le résultat par 789, vous devez écrire :

```
cout << 123 + 456 << endl;
cout << (123 + 456) * 789 << endl;
```

Sur un calcul aussi simple, cela ne semble pas poser de problème particulier. Le problème pourrait se présenter pour des calculs plus complexes. Mais en fait, même sur des calculs simples, cette approche est limitante. Imaginons par exemple que vous souhaitez utiliser le résultat du premier calcul dans une série d'autres calculs. Par exemple :

```
cout << 123 + 456 << endl;
cout << (123 + 456) * 789 << endl;
cout << (123 + 456) * 428 << endl;
cout << (123 + 456) * 384 << endl;
cout << (123 + 456) * 126 << endl;
cout << (123 + 456) * 842 << endl;
cout << (123 + 456) * 962 << endl;
cout << (123 + 456) * 107 << endl;
cout << (123 + 456) * 640 << endl;
cout << (123 + 456) * 862 << endl;
cout << (123 + 456) * 364 << endl;
```

Cela commence à être pénible de devoir réécrire le premier calcul à chaque ligne. Imaginez maintenant que vous ne souhaitez plus calculer la somme de 123 et 456, mais la somme de 321 et 654. Que se passe-t-il ?

Il faut modifier chaque ligne, cela prend du temps et vous risquez de faire une erreur en recopiant le premier calcul.

On comprend vite que cette méthode, bien que correcte au niveau syntaxe, pose des problèmes. L'écriture d'un code C++ qui compile n'est pas suffisant pour créer un programme correct. Il faut également écrire un code qui respecte un certain nombre de règles de qualité pour les logiciels (*Software quality*).

Il existe plusieurs définitions des règles de qualité logicielle, nous allons voir rapidement celle de la norme [ISO/IEC 9126](#). Un logiciel de qualité doit être :

- **fonctionnel**, c'est-à-dire qu'il doit répondre aux besoins des utilisateurs ;
- **fiable**, c'est-à-dire qu'il doit donner les résultats attendus ;
- **convivial**, c'est-à-dire qu'il doit être facile d'utilisation ;
- **efficace**, c'est-à-dire qu'il doit être le plus performant possible en utilisant le minimum de ressources possible ;
- **maintenable**, c'est-à-dire qu'on doit pouvoir facilement le corriger ou lui ajouter des fonctionnalités ;
- **portable**, c'est-à-dire qu'on doit pouvoir l'utiliser facilement dans différents environnements.

Ces critères sont importants parce qu'ils permettent de distinguer un "bon" code d'un "mauvais" code. Le critère "ça compile" n'est pas suffisant.

Dans un vrai programme, il ne sera bien sûr pas toujours possible de respecter tous ces critères, il faudra donc faire des choix sur les critères que vous allez respecter et ceux qui seront moins importants dans un contexte de travail donné. Mais le point principal est qu'il faudra toujours que la décision de ne pas respecter des critères de qualité soit une décision consciente et réfléchie, c'est-à-dire que vous aurez parfaitement identifié en quoi votre code ne respecte pas ces critères et quel impact cela aura sur la qualité de votre logiciel.

Si on revient au code d'exemple précédent, on comprend alors que ce

code ne respecte pas l'un des critères de qualité logiciel : la maintenabilité. En effet, si on souhaite modifier le premier calcul, il est nécessaire de modifier plusieurs lignes de code (et dans un programme complexe, cela peut se traduire par la modification de plusieurs milliers de ligne de code). Dans un code correctement maintenable, il ne faudrait changer idéalement qu'une seule ligne, celle contenant le premier calcul.

### **DRY Don't Repeat Yourself**

On utilise souvent l'acronyme DRY (ne pas se répéter) pour résumer cette problématique. Même s'il est assez simple de copier-coller des lignes de code plusieurs fois, ce qui peut devenir compliqué - et source potentielle d'erreurs - lorsque vous aurez besoin de modifier les lignes de code que vous avez copié. Il sera donc préférable de factoriser votre code au maximum.

<a href="#">Chapitre précédent</a>	<a href="#">Sommaire principal</a>	<a href="#">Chapitre suivant</a>
------------------------------------	------------------------------------	----------------------------------

[Cours, C++](#)

Allez lire Wikipédia...

# Un langage vivant

## Histoire du C++

- années 80, Bjarne Stroustrup, amélioration du C (C with class)
- 1998 première normalisation
- 2003 : TR, mise à jour technique
- 2011 : C++11
- futur : 2014 et 2017

## Le comité de normalisation

- Study group, working group
- TS
- suivre les évolutions, isocpp.org

# Explorer la documentation

Le langage C++ et la bibliothèque standard sont extrêmement riches, il est difficile (et inutile) d'apprendre toutes les subtilités et détails dès le début. Cela viendra avec l'expérience. Par contre, il est important pour votre apprentissage d'acquérir une certaine autonomie, en particulier pour trouver les informations dont vous aurez besoin. Ce cours vous donnera les éléments de compréhension indispensables, mais c'est dans votre intérêt de ne pas hésiter à faire vos propres recherches et lire des sources complémentaires.

L'une des sources les plus importantes est bien sûr la "documentation".

## Le site [cppreference.com](#)

La référence officielle du C++ est le document publié par l'organisme international de normalisation (ISO). Ce document est assez austère à lire (plus de 1300 pages), il est avant tout destiné à ceux qui implémentent les compilateurs. Il est payant, mais vous pouvez consulter le document de travail du comité C++, qui contient également les propositions pour le futur standard ([Working Draft, Standard for Programming Language C++](#)).

En pratique, vous n'allez donc pas utiliser ce document. Il existe différents sites internet qui présentent les spécifications du C++, de façon plus abordable. Dans ce cours, nous allons utiliser le site [[cppreference.com](#)]. Ce n'est pas la documentation officielle à proprement parler, mais c'est l'une des plus à jour et complète.

La page principale est divisée en deux parties. En haut, la documentation du C++, et celle du C en dessous. Nous allons bien sûr utiliser celle concernant le C++.

\* screenshot page principale

La documentation du C++ se décompose en plusieurs parties. En pratique, on peut distinguer trois parties : le langage proprement dit (<http://en.cppreference.com/w/cpp/language>), les fichiers d'en-tête (<http://en.cppreference.com/w/cpp/header>) et la bibliothèque standard (toutes les autres parties en fait).

## Faire une recherche dans la documentation

Vous allez utiliser la documentation dans deux situations. Soit vous recherchez les détails techniques d'une fonctionnalités du C++ à partir d'un mot-clé, soit vous recherchez quels outils du C++ permettent de réaliser une idée que vous avez en tête.

Partir d'un mot-clé est beaucoup plus simple. Il suffit de rechercher dans un moteur de recherche ce mot-clé (par exemple en tapant "C++11 mot-clé") pour trouver des références sur le sujet (évitez de rechercher dans un premier temps "C++ mot-clé", il existe beaucoup de références obsolètes sur internet. Ne faites cela que dans un second temps, lorsque la recherche sur le C++11 a échouée).

Lorsque vous lirez ce cours ou lorsque vous poserez des questions sur les forums internet, on vous donnera probablement des mots-clés, comme par exemple `string` pour les chaînes de caractères ou `vector` pour les tableaux.

Sur [cppreference.com](http://en.cppreference.com), vous avez un champ de recherche en haut à droite. Il n'est pas nécessaire de préciser "C++" ici, puisque la recherche se focalisera sur le site. Vous pouvez donc taper directement votre mot-clé. Si vous tapez `vector` par exemple, vous allez obtenir la page principale de `std::vector` : <http://en.cppreference.com/w/cpp/container/vector>. Si vous essayez avec `string`, plusieurs pages peuvent correspondre, vous obtiendrez donc une page listant les différents résultats possibles <http://en.cppreference.com/mwiki/index.php?title=Special%3ASearch&search=string> (pour les chaînes de caractères, c'est la première qui nous intéresse : `std::string` [http://en.cppreference.com/w/cpp/string/basic\\_string](http://en.cppreference.com/w/cpp/string/basic_string)).

## **Trouver la fonctionnalité du C++ correspond à une idée**

La documentation du C++ (et la bibliothèque standard) est organisée par thématiques. Il suffit généralement d'avoir une idée du module qui sera utilisé pour retrouver facilement le mot-clé correspondant. Mais pour cela, il faut être un peu familier avec l'organisation de la documentation, pour savoir où chercher. Il n'y a pas 36 méthodes pour cela : il faut utiliser la documentation le plus souvent possible, même pour rechercher une fonctionnalité que vous connaissez déjà.

Les titres de modules sont suffisamment explicites, nous n'allons pas les détailler maintenant. Ils seront vus tout au long de ce cours, en fonction des besoins. Pour résumer les modules :

- “Utilities library” : les utilitaires pour gérer les types, la mémoire, les erreurs, les dates et temps, etc. ;
- “Strings library” : les chaînes de caractères ;
- “Containers library” : les collections de données (tableau, liste, etc.) ;
- “Algorithms library” : les algorithmes génériques ;
- “Iterators library” : les itérateurs pour manipuler les collections et algorithmes de façon générique ;
- “Numerics library” : la manipulation des nombres (complexes, aléatoires, etc.) ;
- “Input/output library” : les entrées et sorties (console et fichier) ;
- “Localizations library” : l'internationalisation ;
- “Regular expressions library” : les expressions régulières ;
- “Atomic operations library” : les opérations atomiques ;
- “Thread support library” : le multithreading.

Bien sûr, de nombreux concepts cités ici vous sont inconnus. Pas de panique, tout sera détaillé dans la suite de ce cours.

## **Le langage C++**

La première partie intéressante est la documentation concernant le langage proprement dit. C'est probablement la partie la plus importante de votre apprentissage (sans de bonnes bases sur le langage, vous ne saurez pas utiliser correctement la bibliothèque standard), mais c'est paradoxalement la plus petite partie de la documentation (comparé à la bibliothèque standard). Et probablement celle que vous utiliserez le moins, tout au moins dans un premier temps.

La raison est simple : comme le langage est indispensable à connaître, tout ce dont vous aurez besoin de connaître sera expliqué dans le cours. Cela ne veut pas dire que vous maîtriserez la totalité du langage dans ce premier cours (il faudra un peu plus de temps), mais que si une fonctionnalité n'est pas abordée dans ce cours, elle ne vous sera probablement pas nécessaire (sauf si vous travaillez sur un code qui ne provient pas de ce cours ou si vous êtes curieux — mais dans ce cas, ne vous perdez pas trop dans vos explorations, essayez de suivre globalement le parcours proposé dans ce cours).

Au contraire, la bibliothèque standard contient de nombreuses fonctionnalités simples à comprendre (quand on connaît les bases) et qui nécessitent surtout d'apprendre par la pratique. Tous les détails de la bibliothèque standard ne seront donc pas détaillés, il vous sera proposé à la place des travaux dirigés pour explorer et utiliser ces fonctionnalités. D'où l'importance de la documentation dans ce cas.

Pour autant, si vous avez la curiosité d'explorer cette page de documentation, ne vous privez pas. Et n'hésitez pas à poser des questions sur les forums. Et pour vous aider à faire vos propres recherches, vous trouverez dans ce cours les mots-clés en anglais correspondant aux notions qui sont présentées.

## **La liste des fichiers d'en-tête**

La page des fichiers d'en-tête <http://en.cppreference.com/w/cpp/header> contient tous les fichiers de la bibliothèque standard que vous pouvez

utiliser avec `#include`. Lorsque vous rencontrez une fonctionnalité, vous pouvez regarder dans le fichier d'en-tête correspondant, pour savoir les fonctionnalités qui sont liées.

Les fichiers d'en-tête commençant par "c" sont en fait des importations de la bibliothèque standard du C, disponibles pour des raisons de compatibilité.

Un fichier d'en-tête de la bibliothèque standard du C sera appelé en C en utilisant une syntaxe différente du C++. Le nom ne contient pas le préfixe "c" et contient l'extension ".h" :

```
#include <math.h>
```

Vous trouverez peut-être ce type d'inclusion dans de vieux codes C++ sur internet. Ne reprenez pas directement le code à l'identique, mais remplacez-le par le fichier d'en-tête C++ correspondant (par exemple `<cmath>` dans cet exemple). Votre code pourra fonctionner si vous oubliez de le faire, mais cela posera des problèmes de conflits avec le code C++ sur le long terme.

## Savoir lire une page de documentation

Voyons plus en détail le contenu d'une page de la documentation. Pour cela, regardons la page de `std::cout` : <http://en.cppreference.com/w/cpp/io/cout>.

- screenshot

Le premier point important est que les pages sont organisées sous forme hiérarchique (un peu comme les chapitres et sous-chapitres d'un livre). Vous pouvez voir en haut de la page la liste suivante :

- C++
- Input/output library
- std::basic\_ostream

Cela signifie que `std::cout` est défini dans `std::basic_ostream`, qui fait partie du module “entrée et sortie” de la partie “C++”. Si on souhaite trouver les fonctionnalités similaires, vous pouvez regarder dans `std::basic_ostream` et/ou dans “Input/output library”.

En dessous, vous avez une autre ligne importante, dont on a déjà parlé :

Defined in header `<iostream>`

Cela signifie que cette fonctionnalité est définie dans le fichier d'en-tête `<iostream>`. Il suffit donc d'écrire la ligne suivante au début d'un code pour utiliser cette fonctionnalité :

`#include <iostream>`

Pour terminer, vous aurez systématiquement une description de la fonctionnalité (en anglais) et des codes d'exemple. Certaines fonctionnalités sont très complexes et très riches, n'hésitez pas à étudier le code fourni si vous avez des problèmes.

## Lire l'anglais

Il est possible que vous ne soyez pas habitué à lire l'anglais. Pas de panique, il est possible de s'en sortir autrement... mais ce n'est pas une bonne idée. L'anglais est omniprésent en informatique et plus encore en programmation. Il est classique que la documentation et les articles de référence soient en anglais et il est préférable d'écrire ses codes en anglais aussi.

Si vous lisez ce cours, c'est que normalement vous souhaitez apprendre correctement le C++, ce qui va prendre des années. Et que vous apprenez principalement en autodidacte. Donc vous avez le temps et de bonnes raisons de faire les choses correctement. Même si vous prenez une journée pour lire une simple page, ce n'est pas grave. Le principal est de s'y mettre progressivement. Et tout de suite. Cela viendra avec la pratique.

[Chapitre précédent](#) [Sommaire principal](#) [Chapitre suivant](#)

## Cours, C++

# Les nombres entiers

Dans les chapitres précédents, vous avez appris à utiliser `std::cout` pour afficher des messages. Les messages sont des chaînes de caractères encadrées par des guillemets droits ". Une valeur écrite directement dans le code C++ est appelée une **littérale**. Il existe d'autres types de littérales, par exemple :

- les nombres entiers : `0, 1, 2, -1, -2` ;
- les nombres réels : `1.0, 2.1, -5.12, 1.457` ;
- les booléens, qui représente une valeur à deux états : `true` (vrai) et `false` (faux) ;
- les caractères : `'a', 'b', 'c'` ;
- les chaînes de caractères (que vous avez déjà vu) : "hello, world", "salut tout le monde".

Remarquez bien la différence entre les caractères, qui s'écrivent avec des guillemets droits simples, et les chaînes, qui s'écrivent avec des guillemets droits doubles. Notez aussi que les nombres réels s'écrivent avec un point (notation anglaise) et non avec une virgule (notation française).

Vous pouvez afficher ces littérales directement en utilisant `std::cout`, comme vous l'avez vu pour les chaînes de caractères.

main.cpp

```
#include <iostream>

int main() {
    std::cout << 1 << std::endl; // nombre
    entier
    std::cout << 3.1415 << std::endl; // nombre
    réel
    std::cout << true << std::endl; // booléens
    std::cout << 'a' << std::endl; // caractère
```

```
    std::cout << "hello, world!" << std::endl; // chaîne de
caractères
}
```

Nous reviendrons par la suite sur les spécificités de chaque type de littérales, voyons pour le moment les nombres entiers plus en détail.

## Écrire des nombres entiers

Les nombres entiers ne devraient pas vous poser de problème, ce sont les nombres que vous utilisez pour compter depuis l'école maternelle. La forme la plus simple pour écrire un nombre entier est d'écrire une série continue de chiffres entre 0 et 9.

```
main.cpp
#include <iostream>

int main() {
    std::cout << 1 << std::endl; // affiche le nombre 1
    std::cout << 2 << std::endl; // affiche le nombre 2
    std::cout << 3 << std::endl; // affiche le nombre 3
    std::cout << 4 << std::endl; // affiche le nombre 4
}
```

Pour écrire un nombre entier négatif, vous devez ajouter le signe moins devant le nombre.

```
main.cpp
#include <iostream>

int main() {
    std::cout << -1 << std::endl; // affiche le nombre -1
    std::cout << -2 << std::endl; // affiche le nombre -2
    std::cout << -3 << std::endl; // affiche le nombre -3
    std::cout << -4 << std::endl; // affiche le nombre -4
}
```

Lorsque vous écrivez de très grands nombres, il peut être difficile de les lire. Par exemple :

### main.cpp

```
#include <iostream>

int main() {
    std::cout << 123456789123456789 << std::endl;
}
```

La raison est que le cerveau humain est capable de reconnaître des groupes de quelques caractères, mais pas un seul bloc de 18 caractères. Il n'arrive donc pas, en un coup d'œil, à identifier si ce nombre est de l'ordre du million, du milliard ou autre.

Pour faciliter la lecture, il est possible d'ajouter un caractère guillemet droit simple ' pour séparer un grand nombre en plusieurs groupes. Par habitude, on sépare en groupes de trois chiffres :

### main.cpp

```
#include <iostream>

int main() {
    std::cout << 123'456'789'123'456'789 << std::endl;
}
```

Notez bien la différence entre le guillemet droit simple utilisé pour écrire une littérale de type caractère et le séparateur numérique. Si la littérale commence par un un guillemet, elle sera considérée comme étant un caractère.

### main.cpp

```
#include <iostream>

int main() {
    std::cout << 4'56 << std::endl; // ok, séparateur
    numérique
    std::cout << '456 << std::endl; // erreur, guillemet
    manquant (1)
    std::cout << '456' << std::endl; // erreur, plusieurs
    caractères (2)
}
```

La première erreur se produit du fait que le compilateur pense que le guillemet correspond au début d'une littérale caractère, mais ne trouve pas la fin.

```
main.cpp:5:18: warning: missing terminating ' character  
[-Winvalid-pp-token]  
    std::cout << '456 << std::endl;  
          ^
```

La seconde erreur correspond à une littérale caractère qui contient plusieurs caractères.

```
main.cpp:6:18: warning: multi-character character constant  
[-Wmultichar]  
    std::cout << '456' << std::endl;  
          ^
```

Vous pouvez écrire des nombres très grand de cette manière, mais il existe une limite. Si vous écrivez un nombre trop grand, vous aurez un message d'erreur signalant que ce nombre est trop grand.

```
main.cpp  
#include <iostream>  
  
int main() {  
    std::cout << 123456789123456789123456789123456789 << std  
    ::endl;  
}
```

produira l'erreur :

```
main.cpp:4:18: error: integer literal is too large to be  
represented in any integer type  
    std::cout << 123456789123456789123456789123456789 <<  
    std::endl;  
          ^
```

L'existence d'une valeur maximale limite est liée à la représentation des nombres dans la mémoire des ordinateurs et à la notion de type de données. Vous verrez cela dans les prochains chapitres.

Il existe des outils qui permettent d'utiliser des nombres avec autant de chiffre que vous souhaitez, mais il ne sera pas possible d'utiliser dans ce cas les nombres entiers tel que définit dans ce chapitre. Vous verrez par la suite quelques bibliothèques qui permettent de faire cela et pourrez implémenter ce type de fonctionnalités.

## Les nombres décimaux, hexadécimaux, octaux et binaires

Les nombres entiers que vous utilisez habituellement s'écrivent à partir de dix chiffres (0 à 9). C'est pour cette raison que l'on parle de système décimal (du latin *decimus*, qui signifie "dixième") et de base 10. Mais ce n'est pas la seule façon d'écrire les nombres et il existe d'autres systèmes numériques.

Imaginons par exemple que vous souhaitez écrire des nombres en utilisant que huit chiffres (0 à 7 - base 8). Dans ce cas, nous pouvons compter de la façon suivante : 0, 1, 2, 3, 4, 5, 6, 7. Arrivé au huitième chiffre, nous ne pouvons pas écrire "8", puisque ce chiffre n'est pas autorisé dans ce système. Donc, il faut passer à un nombre à deux chiffres : 0, 1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 15, 16, 17, 20, 21, etc.

En C++, il est possible d'écrire et afficher des nombres écrits selon des bases différentes de 10. Pour des raisons historiques et matériel, les ordinateurs savent manipuler les nombres en base 2 (binaire), 8 (octal), 10 (décimal) et 16 (hexadécimal). Pour écrire un nombre dans une base différente de 10, il faut commencer le nombre par le chiffre 0 puis un caractère optionnel pour spécifier la base : rien pour octal, `x` ou `X` pour l'hexadécimal et `b` pour le binaire. Les chiffres autorisés pour écrire un nombre dépendent de la base utilisée : 0 à 7 pour l'octal, 0 à 9 et a à f (ou A à F) pour l'hexadécimal et 0 et 1 pour le binaire.

Le code suivant permet d'afficher la valeur de 10 selon la base :

```
main.cpp
```

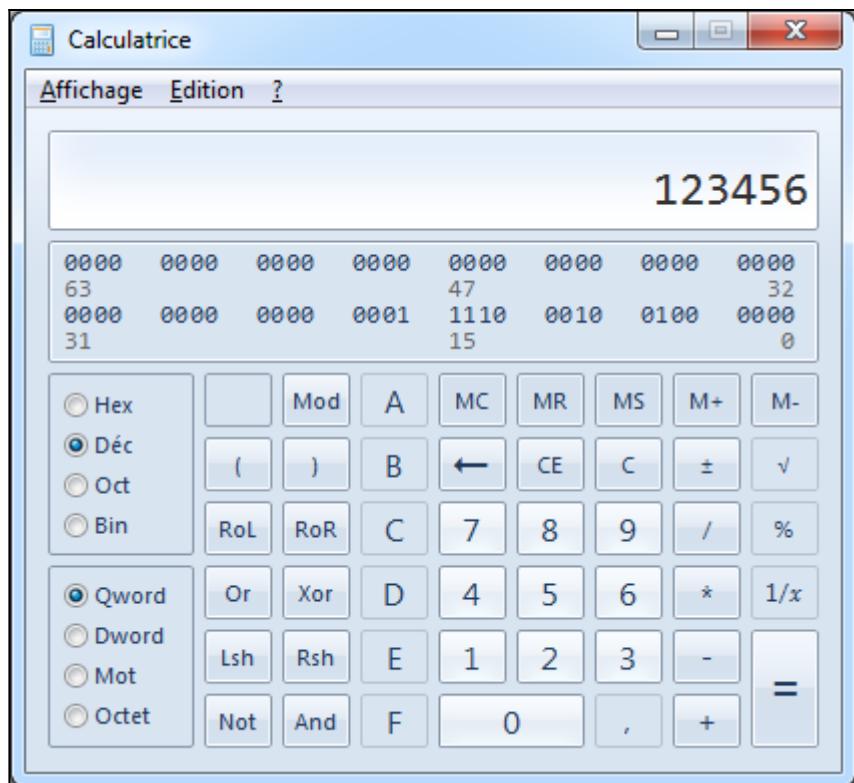
```
#include <iostream>
```

```
int main() {
    std::cout << 10 << std::endl;
    std::cout << 010 << std::endl;
    std::cout << 0x10 << std::endl;
    std::cout << 0b10 << std::endl;
}
```

affiche :

```
10
8
16
2
```

Il est possible de réaliser ces conversions à la main, mais c'est plus simple d'utiliser un logiciel de conversion. Il existe des convertisseurs en ligne, vous pouvez également utiliser la calculatrice de Windows en sélectionnant le mode "Programmeur" dans le menu "Affichage".



Il faut donc bien faire attention à la base utilisée lorsque l'on écrit un nombre.

Base	Système	Préfixe	Chiffres	Exemple
2	binaire	0b	0 et 1	0b01
8	octal	0	0 à 7	001234567
10	décimal		0 à 9	1234567890
16	hexadécimal	0x	0 à 9 et a à f	0x0123456789abcdef
		0X	0 à 9 et A à F	0X0123456789ABCDEF

Comme vous l'avez vu dans les codes précédents, les nombres sont affichés après conversion en base 10. C'est le mode de fonctionnement par défaut de `std::cout`, mais il est possible de modifier ce comportement. Les directives (*I/O manipulator*) suivantes permettent de modifier la base utilisée par `std::cout` pour afficher les nombres :

- `std::oct` pour afficher en utilisant la base 8 ;
- `std::dec` pour afficher en utilisant la base 10 ;
- `std::hex` pour afficher en utilisant la base 16.

Il n'existe pas de directive permettant d'afficher les nombres en binaire.  
Vous pourrez réaliser cela dans un exercice, dans la suite du cours.

main.cpp

```
#include <iostream>

int main() {
    std::cout << std::hex;
    std::cout << 10 << std::endl;
    std::cout << 010 << std::endl;
    std::cout << 0x10 << std::endl;
    std::cout << 0b10 << std::endl;
}
```

affiche :

```
a
8
10
2
```

Comme vous le voyez, une directive continue de s'appliquer sur toutes les lignes suivantes, tant que vous ne donnez pas une directive modifiant une nouvelle fois l'affichage. Vous pouvez écrire les directives sur une ligne différente (comme dans le code précédent), sur la première ligne ou à chaque ligne, cela ne change rien au résultat.

Faites bien la distinction entre des directives comme `std::endl`, qui ont un effet immédiat, et les directives comme `std::hex`, qui ont un effet permanent, jusqu'à ce qu'une autre directive modifie le comportement.

Si vous affichez des nombres en utilisant plusieurs bases différentes, il peut être difficile de savoir à quelle base correspond chaque nombre affiché. Il est possible d'afficher la base utilisée pour l'affichage avec la

directive `std::showbase`. Pour ne plus afficher la base, vous pouvez utiliser la directive `std::noshowbase`.

main.cpp

```
#include <iostream>

int main() {
    std::cout << 10 << std::endl;
    std::cout << 010 << std::endl;
    std::cout << 0x10 << std::endl;
    std::cout << 0b10 << std::endl;
    std::cout << std::endl;

    std::cout << std::hex << std::showbase;
    std::cout << 10 << std::endl;
    std::cout << 010 << std::endl;
    std::cout << 0x10 << std::endl;
    std::cout << 0b10 << std::endl;
}
```

affiche :

```
10
8
16
2

0xa
0x8
0x10
0x2
```

## Les opérations arithmétiques sur les entiers

Pouvoir écrire et afficher des nombres entiers est un début, mais il est possible d'aller plus loin et de réaliser des calculs dessus. Pour cela, le C++ propose plusieurs opérateurs naturels de l'arithmétique, comme l'addition `+`, la soustraction `-` ou la multiplication `*`.

### main.cpp

```
#include <iostream>

int main() {
    std::cout << "123 + 456 = " << 123 + 456 << std::endl;
    std::cout << "123 - 456 = " << 123 - 456 << std::endl;
    std::cout << "123 * 456 = " << 123 * 456 << std::endl;
}
```

affiche :

```
123 + 456 = 579
123 - 456 = -333
123 * 456 = 56088
```

Faites bien la distinction entre l'opération écrite entre guillemets “123 + 456”, qui est donc interprétée comme une chaîne de caractères et non évaluée, et la même chose en dehors des guillemets, qui sera interprétée comme une expression mathématique et évaluée.

Il est également possible de calculer des divisions entières avec l'opérateur division / et de calculer le reste d'une division entière avec l'opérateur modulo %. Pour rappel, la division entière permet de calculer le résultat d'une division en utilisant uniquement des nombres entiers (c'est le premier type de division que vous avez appris à l'école). Ainsi :

- avec une division entière : 11 divisé par 4 donne 2 et reste 3 ;
- avec une division réelle : 11 divisé par 4 donne 2,75.

### main.cpp

```
#include <iostream>

int main() {
    std::cout << "10 / 2 = " << 10 / 2 << std::endl;
    std::cout << "10 / 3 = " << 10 / 3 << std::endl;
    std::cout << "10 / 4 = " << 10 / 4 << std::endl;

    std::cout << "10 % 2 = " << 10 % 2 << std::endl;
```

```
    std::cout << "10 % 3 = " << 10 % 3 << std::endl;
    std::cout << "10 % 4 = " << 10 % 4 << std::endl;
}
```

affiche :

```
10 / 2 = 5
10 / 3 = 3
10 / 4 = 2
10 % 2 = 0
10 % 3 = 1
10 % 4 = 2
```

En C++, l'opérateur `/` est utilisé pour faire les divisions entières et réelles. Il faut donc particulièrement faire attention de ne pas se tromper lorsque l'on écrit une expression, sous peine d'avoir des résultats étranges.

`main.cpp`

```
#include <iostream>

int main() {
    std::cout << "11 / 4 = " << 11 / 4 << std::endl;
// division entière
    std::cout << "11.0 / 4.0 = " << 11.0 / 4.0 << std::endl;
// division réelle
}
```

affiche :

```
11 / 4 = 2
11.0 / 4.0 = 2.75
```

Les opérations mathématiques suivent la sémantique habituelle, en particulier pour l'ordre d'évaluation et l'utilisation des parenthèses. Pour rappel, pour évaluer une expression mathématique (une série de plusieurs opérations), l'ordre d'évaluation est le suivant :

- en premier, évaluer les expressions entre parenthèse ;
- ensuite, évaluer les multiplications et les divisions ;
- pour terminer, évaluer les additions et les soustractions.

Ainsi, l'expression ci-dessous s'évalue de la façon suivante :

$$\begin{aligned}
 & 5 - (2 + 3) / 4 \\
 = & 5 - 5 / 4 \\
 = & 5 - 1 \\
 = & 4
 \end{aligned}$$

Pour exercice, écrivez le code C++ correspondant pour évaluer cette expression et vérifier que le résultat est correct.

De la même façon, les opérateurs du C++ suivent les règles arithmétiques habituelles :

- la commutativité (sauf pour la division) :  $a + b = b + a$  ;
- l'associativité :  $a + (b + c) = (a + b) + c$  ;
- la distributivité :  $a * (b + c) = a * b + a * c$ .

On peut vérifier facilement ces règles avec un programme C++ :

`main.cpp`

```
#include <iostream>

int main() {
    std::cout << "Commutativité :" << std::endl;
    std::cout << "2 + 3 = " << 2 + 3 << std::endl;
    std::cout << "3 + 2 = " << 3 + 2 << std::endl;

    std::cout << "Associativité :" << std::endl;
    std::cout << "2 + (3 + 4) = " << 2 + (3 + 4) << std::endl;
    std::cout << "(2 + 3) + 4 = " << (2 + 3) + 4 << std::endl;

    std::cout << "Distributivité :" << std::endl;
    std::cout << "2 * (4 + 3) = " << 2 * (4 + 3) << std::endl;
}
```

```
    std::cout << "2 * 4 + 2 * 3 = " << 2 * 4 + 2 * 3 << std  
    ::endl;  
}
```

affiche :

```
Commutativité :  
2 + 3 = 5  
3 + 2 = 5  
Associativité :  
2 + (3 + 4) = 9  
(2 + 3) + 4 = 9  
Distributivité :  
2 * (4 + 3) = 14  
2 * 4 + 2 * 3 = 14
```

## Les comportements indéfinis

Un dernier point pour terminer, essayez de diviser un nombre entier par zéro :

```
main.cpp  
#include <iostream>  
  
int main() {  
    std::cout << "1 / 0 = " << (1 / 0) << std::endl;  
}
```

Ce code affichera un avertissement (*warning*) à la compilation, vous informant que la division par zéro n'est pas définie :

```
main.cpp:4:20: warning: division by zero is undefined  
[-Wdivision-by-zero]  
    std::cout << (1/0) << std::endl;  
           ^~
```

On parle en C++ de “comportement indéfini”, *undefined behavior* en anglais, que l'on abrège parfois avec les initiales “UB”. Cela signifie qu'un programme utilisant un tel code aura un comportement aléatoire et

pourra faire n'importe quoi : provoquer une erreur, donner un résultat quelconque ou encore sembler fonctionner correctement. Les comportements indéfinis sont difficiles à corriger, puisque cela ne produit pas forcément un message d'erreur clair.

Ainsi, l'exécution du programme précédent se déroule sans erreur à l'exécution. Par contre, la valeur affichée par `std::cout` est aléatoire (dans Coliru.com) et n'a pas de sens en termes mathématiques :

```
1 / 0 = 4196864
```

Le C++ étant un langage permissif, le compilateur ne va pas nécessairement vérifier que vous écrivez du code incorrect. C'est de votre responsabilité de vérifier que le code ne provoque pas de comportement indéfini. Pour cela, voici quelques aides :

**Respectez les bonnes pratiques de codage.** Comme le C++ est permissif, le développeur a le droit d'écrire du code correct ou du code incorrect. Cela peut paraître évident, mais il vaut mieux le dire : il est préférable d'écrire du code correct que du code incorrect ! Il y a souvent plusieurs façons en C++ de résoudre un même problème, certaines approches posent davantage de problèmes que d'autres.

Ce cours de C++ n'a pas pour vocation à vous apprendre tout le C++. Il se focalise, volontairement, sur les syntaxes les plus récentes et les plus sûres. Et même en utilisant ces approches modernes, il y aura des précautions à prendre pour éviter les problèmes. Vous apprendrez à écrire du code le plus sécurisé possible.

**Faites vous aider par le compilateur.** Celui-ci peut faire certaines vérifications, comme dans le code précédent, et vous avertir lorsque vous écrivez un code qui pose potentiellement un problème. Tous les compilateurs ne font pas les mêmes vérifications et les contrôles effectués dépendent des options utilisées. Les options `-Wall -Wextra -pedantic` qui vous ont été présentées dans coliru.com activent les principales vérifications, mais il en existe d'autres. Il peut être intéressant de compiler un code avec plusieurs compilateurs pour avoir un maximum de messages d'avertissement.

**Utilisez les outils de vérifications statique et dynamique.** Ce sont des outils qui font plus de vérifications que le compilateur. Les premiers vérifient le code directement, alors que les seconds travaillent sur le programme après compilation (donc durant l'exécution).

Non seulement ils permettent de vérifier le respect des règles du langage C++, mais également que certaines pratiques de codage sont respectées. Dans l'objectif d'écrire des programmes C++ modernes, il ne faut pas hésiter à utiliser ce type d'outils. Vous verrez dans la suite de ce cours comment installer et utiliser certains de ces outils.

## Exercices

c f

<http://openclassrooms.com/forum/sujet/exercices-dans-le-cours-de-gbdivers>

Remarque : le but de ces exercices est de faire des calculs arithmétiques de base. C'est assez simple, ce n'est pas vraiment de la programmation (c'est surtout pour jouer avec l'affichage et les opérations de base). Mais c'est quelque chose que l'on doit savoir faire (même à la main, avec crayon et papier)

## Conversion en nombre de n'importe quelle base

[http://fr.wikipedia.org/wiki/Syst%C3%A8me\\_d%C3%A9cimal](http://fr.wikipedia.org/wiki/Syst%C3%A8me_d%C3%A9cimal)

opérations à faire pour les calculs ?

Le but de cet exercice est de convertir un nombre d'une base donnée en une autre base. (comme c'est le début du cours, il ne faudra utiliser que les opérations mathématiques et l'affichage, donc pas de boucles et fonctions). Par exemple convertir 1234 en hexadécimal ou 0xabc en décimal. (le but est de pratiquer les conversions de bases, pas d'écrire un code de conversion réutilisable). Tu peux prendre n'importe quel nombre dans n'importe quelle base (en utilisant les préfixes), l'afficher dans une autre base (en utilisant std::hex, std::dec, etc), puis écrire un code qui affiche la conversion, mais faite manuellement.

## Calculer l'indice de colonne et de ligne d'une table

Cet exercice consiste à calculer la ligne et la colonne correspondant à un index donné (c'est très utile pour les tableaux à plusieurs dimensions). Pour cet exo, imaginez un tableau suivant :

0	1	2	3
4	5	6	7
8	9	10	11

(on appelle ce tableau "row major" puisque les données sont en ligne en premier, cf [http://en.wikipedia.org/wiki/Row-major\\_order](http://en.wikipedia.org/wiki/Row-major_order)). Si on donne un élément du tableau (par exemple 6), il faut trouver le numéro de ligne et le numéro de colonne correspondant, en utilisant les opérateurs / et %. Faire de même avec d'autres tableaux (dont des tableaux en column major). Remarque : en C++, on compte à partir de 0. Donc les colonnes et lignes s'écrivent 0, 1, 2, ... (et pas 1, 2, 3, ...)

Préliminaire : Je veux l'indice du tableau 1D qui correspond à la case (x,y) de mon tableau 2D, sachant que j'ai rangé mes lignes l'une derrière l'autre, d'abord la première ligne, ensuite la seconde ...

Pour la découvrir empiriquement, quelle est l'indice de la première colonne de la première ligne? Quel est l'indice de la dernière colonne de la première ligne? Quel est l'indice de la première colonne de la seconde ligne? quelle est l'indice de la dernière colonne de la seconde ligne?, quel est l'indice de la dernière colonne de la dernière ligne?, Quel est l'indice de la première colonne de la dernière ligne? et pour finir quel est l'indice d'une case (x,y) quelconque du tableau 2D?

Sachant qu'a chaque indice du tableau 1D il y a une seule position (x,y) qui correspond, et que chaque position (x,y) correspond un seul indice du tableau 1D, si tu connais la formule dans un sens, découvrir la formule dans l'autre sens ne sera pas très difficile ;).

A partir d'un élément du tableau (par exemple  $i = 6$ ,  $i$  pour index), tu as besoin de connaître la largeur ( $w = 4$ ,  $w$  pour width) et la hauteur ( $h = 3$ ,  $h$  pour height) du tableau et d'utiliser les opérateurs division entière / et modulo %. Tu dois obtenir pour  $i = 6$ ,  $x = 2$  (numéro de colonne) et  $y = 1$

(numéro de ligne).

Si ta formule est bonne, cela veut dire qu'en appliquant la même formule sur un autre indice du tableau ou un autre tableau, le numéro de ligne et de colonne devraient être bons. Est-ce que c'est le cas ? Essaie par exemple, avec le même tableau, avec l'indice 7 par exemple (cela doit donner ligne=1 et colonne=3)

Remarque : Une division entière (comme on l'apprend en primaire : image) peut s'écrire :

```
dividende = diviseur * quotient + reste
```

La division entière et le modulo sont les opérations qui permettent d'avoir le quotient et le reste :

```
division(dividande, diviseur) = quotient  
modulo(dividande, diviseur) = reste
```

En C++, on va écrire :

```
std::cout << "Quotient = " << (dividende / diviseur) <<  
std::endl;  
std::cout << "Reste = " << (dividende % diviseur) <<  
std::endl;
```

## Des compteurs finis

Un autre exercice avec modulo : imaginons que l'on a un compteur qui compte jusqu'à 100. Comment transformer ce compteur en compteur qui compte de 0 à 10 en boucle ? (il doit retourner 0, 1, ..., 8, 9, 0, 1, ...)

Faire de même pour compter de 3 à 8 en boucle.

<a href="#">Chapitre précédent</a>	<a href="#">Sommaire principal</a>	<a href="#">Chapitre suivant</a>
------------------------------------	------------------------------------	----------------------------------

# Logique binaire et calcul booléen

La représentation en base 2, le binaire, possède une importance particulière en informatique. Vous avez vu dans les chapitres précédents que vous pouvez représenter les nombres entier sous forme binaire (une suite de 0 et de 1) en utilisant le préfixe `0b`. Vous pouvez également écrire des valeurs booléennes en utilisant les mots-clés `true` (vrai) et `false` (faux).

Cette forme de logique est tellement importante qu'un chapitre complet lui est consacré (et vous reviendrez plusieurs fois sur ces notions durant le cours).

En électronique, il est facile de représenter des valeurs binaires en utilisant des tensions différentes. Par exemple, on va définir que l'état "vrai" sera représenté par une tension de +5V et l'état "faux" par une tension de 0V. (En pratique, les valeurs prises seront très variables selon les composants de l'ordinateur, mais le principe reste le même.)

Ces valeurs binaires utilisées en interne sont appelées "bits" et correspondent au plus petit élément d'information que peut manipuler un ordinateur. Tous les autres type de données (aussi bien les nombres que les chaînes de caractères) sont définis à partir d'une représentation interne en bits. Même les nombres binaires et les booléens que vous utilisez en C++ sont représentés par des bits dans l'ordinateur.

Il est possible en C++ de manipuler directement les représentations internes des valeurs, mais cela est beaucoup plus complexe et moins sécurisé que de manipuler les types du C++. Vous n'aurez besoin de faire cela que dans des cas très spécifiques (programmation de micro-contrôleur, optimisation bas niveau), mais vous verrez comment faire cela.

La notation des valeurs binaires "vrai" et "faux" est purement arbitraire. Vous pouvez définir que les valeurs binaires sont "haut" et "bas", "droite"

et “gauche” ou n’importe quoi d’autre. Le plus important est que cela représente deux états différents. Dans du code C++, vous avez deux manières de représenter les valeurs binaires :

- avec 0 et 1, pour représenter un nombre entier ;
- avec `false` et `true`, pour représenter un booléen.

## Les booléens

Voyons dans un premier temps les valeurs binaires, encore appelées booléens (nom donné en l’honneur du mathématicien [George Boole](#), qui a créé cette branche des mathématiques).

## Afficher une valeur booléenne

Pour écrire une valeur booléenne, il faut utiliser les mots-clé `true` (vrai) et `false` (faux). Par défaut, `std::cout` affiche ces valeurs avec respectivement 1 et 0. La directive `std::boolalpha` permet d'afficher les booléens en clair et la directive `std::noboolalpha` permet d'arrêter de représenter les booléens.

```
main.cpp
#include <iostream>

int main() {
    std::cout << std::boolalpha;
    std::cout << "false = " << false << std::endl;
    std::cout << "true = " << true << std::endl;
    std::cout << std::noboolalpha;
    std::cout << "false = " << false << std::endl;
    std::cout << "true = " << true << std::endl;
}
```

affiche :

```
false = false
```

```
true = true  
false = 0  
true = 1
```

## Les opérateurs logiques

Les booléens ne se manipulent pas comme des nombres entiers. En effet, cela n'a pas de sens de faire des opérations arithmétiques dessus. Les booléens permettent un nombre limité d'opérations logiques, qui prennent un ou deux booléens et retourne un nouveau booléen.

Faire des opérations arithmétiques sur les booléens ne provoquera pas d'erreur de compilation, vous pouvez donc écrire par exemple `true + 2`. La raison est que les valeurs booléennes sont représentée en interne par des nombres (généralement 0 et 1) et que cela a un sens, **pour l'ordinateur**, de faire ce type d'opération. Mais cela n'a pas de sens en termes de logique (quel sens pourrait-on donner à l'expression `true + 2` ?).

En C++, on utilisera les mots-clés `true` et `false`. Le respect des types est l'une des forces du C++, encore faut-il les utiliser correctement. Le C++ est un langage permissif, il autorisera à écrire `true + 2`, mais cela brisera la sémantique des booléens.

La première opération booléenne est la négation “NON” `!`, qui transforme `true` en `false` et `false` en `true` :

```
main.cpp  
#include <iostream>  
  
int main() {  
    std::cout << std::boolalpha;  
    std::cout << "!false = " << !false << std::endl;  
    std::cout << "!true = " << !true << std::endl;  
}
```

affiche :

```
!false = true
!true = false
```

La seconde opération est la conjonction `&&`, qui prend deux booléens et retourne vrai uniquement si les deux booléens valent vrai. Cette opération est également appelée “AND” ou “ET”, puisque pour que le résultat soit vrai, il faut que le premier booléen soit vrai ET que le second booléen soit vrai.

main.cpp

```
#include <iostream>

int main() {
    std::cout << std::boolalpha;
    std::cout << "false AND false = " << (false && false) << std::endl;
    std::cout << "false AND true = " << (false && true) << std::endl;
    std::cout << "true AND false = " << (true && false) << std::endl;
    std::cout << "true AND true = " << (true && true) << std::endl;
}
```

affiche :

```
false AND false = false
false AND true = false
true AND false = false
true AND true = true
```

La dernière opérateur est la disjonction `||`, qui prend également deux booléens et retourne vrai si au moins un des deux booléens est vrai. Cette opération est également appelée “OR” ou “OU”, puisque pour que le résultat soit vrai, il faut que le premier booléen soit vrai OU que le second booléen soit vrai.

main.cpp

```
#include <iostream>

int main() {
    std::cout << std::boolalpha;
    std::cout << "false OR false = " << (false || false) <<
std::endl;
    std::cout << "false OR true = " << (false || true) <<
std::endl;
    std::cout << "true OR false = " << (true || false) <<
std::endl;
    std::cout << "true OR true = " << (true || true) << std
::endl;
}
```

affiche :

```
false OR false = false
false OR true = true
true OR false = true
true OR true = true
```

Il existe en théorie deux versions de la disjonction. L'opérateur `||` retourne vrai si les deux opérandes sont vraies, on dit que c'est un "OU inclusif". Il faut donc comprendre le "OU" de la façon suivante : "le premier booléen est vrai OU le second booléen est vrai OU les deux sont vrais".

La seconde version de la disjonction est le "OU exclusif" ou "XOR". Dans ce cas, il faut prendre le "OU" au sens strict : "le premier booléen est vrai OU le second booléen est vrai, mais pas les deux en même temps".

Il n'existe pas en C++ d'opérateur "Ou exclusif", mais il est possible de le simuler avec les autres opérateurs.

Ces opérateurs peuvent être résumé dans un tableau (appelé table de vérité) :

a	b	<code>!a</code>	<code>a &amp;&amp; b</code>	<code>a    b</code>
0	0	1	0	0

0	1	1	0	1
1	0	0	0	1
1	1	0	1	1

Pour terminer, les opérateurs logiques du C++ fonctionnent en utilisant l'évaluation paresseuse (*lazy evaluation*). Cela permet d'évaluer les opérandes uniquement si nécessaire. Imaginons les opérations suivantes, dans lesquelles “expression complexe” est un code quelconque qui prend du temps pour être évalué :

```
a && (expression complexe)
b || (expression complexe)
```

Avec le tableau précédent, on peut remarquer que si `a` est faux, alors le résultat de `a && (expression complexe)` sera toujours faux, quelque soit la valeur de `expression complexe`. Dans ce cas, il n'est pas nécessaire d'évaluer “expression complexe”, puisque sa valeur ne change pas le résultat.

De la même façon, si `b` est vrai dans la seconde expression, le résultat de `b || (expression complexe)` sera toujours vrai quelque soit la valeur de `expression complexe`, il n'est pas nécessaire d'évaluer `expression complexe`.

Cette technique permet de gagner en performances, en évitant de faire des calculs inutiles, mais cela implique une contrainte : il ne faut JAMAIS mettre dans une expression logique des calculs qui peuvent modifier le comportement du programme. Il sera plus sûr de séparer ces calculs et les opérations logiques dans le code.

## Les opérateurs de comparaison

Généralement, vous n'aurez pas à écrire `true` et `false` directement dans vos codes, vous manipulerez des booléens générés par des tests. Un test est simplement une expression (une suite d'instructions et de calculs) qui retourne un booléen. Les opérateurs logiques permettent de

combiner des tests simples pour former des tests plus complexes, voire très complexes.

Une méthode classique pour écrire une expression retournant un booléen est d'utiliser les opérateurs de comparaison. Comme leur nom l'indique, ces opérateurs permettent de comparer des valeurs et de retourner vrai ou faux, selon le résultat de cette comparaison. Ces opérateurs sont les suivants :

- l'opérateur “EST ÉGAL À” `==` permet de tester si deux valeurs sont égales ;
- l'opérateur “EST DIFFÉRENT DE” `!=` permet de tester si deux valeurs sont différentes ;
- l'opérateur “EST SUPÉRIEUR À” `>` permet de tester si la première valeur est supérieure à la seconde ;
- l'opérateur “EST SUPÉRIEUR OU ÉGAL À” `>=` permet de tester si la première valeur est supérieure ou est égale à la seconde ;
- l'opérateur “EST INFÉRIEUR À” `<` permet de tester si la première valeur est inférieure à la seconde ;
- l'opérateur “EST INFÉRIEUR OU ÉGAL À” `<=` permet de tester si la première valeur est inférieure ou est égale à la seconde.

Ces opérateurs peuvent s'appliquer sur des nombres entiers, des réels ou des caractères, par exemple :

main.cpp

```
#include <iostream>

int main() {
    std::cout << std::boolalpha;
    std::cout << "12.34 == 23.45 ? " << (12.34 == 23.45) <<
std::endl;
    std::cout << "12.34 != 23.45 ? " << (12.34 != 23.45) <<
std::endl;
    std::cout << "12.34 > 23.45 ? " << (12.34 > 23.45) <<
std::endl;
    std::cout << "12.34 >= 23.45 ? " << (12.34 >= 23.45) <<
std::endl;
```

```
    std::cout << "12.34 < 23.45 ? " << (12.34 < 23.45) <<
std::endl;
    std::cout << "12.34 <= 23.45 ? " << (12.34 <= 23.45) <<
std::endl;
}
```

affiche :

```
12.34 == 23.45 ? false
12.34 != 23.45 ? true
12.34 > 23.45 ? false
12.34 >= 23.45 ? false
12.34 < 23.45 ? true
12.34 <= 23.45 ? true
```

Pour les nombres, ces opérateurs ne posent pas de difficultés particulières, leur fonctionnement correspond à ce que vous connaissez en mathématique.

Les comparaison sur les chaînes de caractères est possible, mais nécessite quelques précautions. Vous verrez cela en détail dans le chapitre sur les chaînes (lequel ?).

Comme le langage C++ est permissif, la comparaison de valeurs de type différent ne produira pas forcément une erreur de compilation. Par exemple comparer un entier et un caractère :

```
#include <iostream>

int main() {
    std::cout << std::boolalpha << ('a' < 42) << std::endl;
}
```

Par contre, cela n'a pas de sens en termes de sémantique (comme on dit, on ne compte pas ensemble des pommes et des poires), il ne faut donc pas écrire ce type de code. En revanche, vous pouvez combiner le résultat de plusieurs comparaisons sur des valeurs de type différent en utilisant les opérateurs logiques :

```
#include <iostream>

int main() {
    std::cout << std::boolalpha << (( 'a' < 'z') && (123 >=
456)) << std::endl;
}
```

## Exercices

Evaluer le résultat de ces expressions (à la main, pas avec du code) :

a	b	$\neg a \ \&\& b$	$\neg a \     b$	$\neg a \ \&\& \neg b$	$\neg a \     \neg b$
0	0	?	?	?	?
0	1	?	?	?	?
1	0	?	?	?	?
1	1	?	?	?	?

Quelle combinaison des opérateurs logiques de base permet d'obtenir un OU Exclusif ? Ecrivez le code C++ correspondant.

a	b	XOR
0	0	0
0	1	1
1	0	1
1	1	0

## Représentation binaire des entiers

Comme vous l'avez vu dans les chapitres précédents, la façon dont vous écrivez un nombre dans le code et la façon dont il est affiché dans la console sont indépendant. Par défaut, l'écriture d'un nombre et son affichage se font en utilisant la base 10 (décimal), mais vous pouvez changer la base lors de l'écriture en utilisant un préfixe (`0b`, `0` et `0x`) et lors de l'affichage en utilisant une directive (`std::oct`, `std::dec` et `std::hex`).

### main.cpp

```
#include <iostream>

int main() {
    std::cout << std::showbase << 0b101010 << std::endl;
    std::cout << std::hex << 0b101010 << std::endl;
}
```

affiche :

```
42
0x2a
```

Il n'existe pas de directive pour afficher les nombres directement en binaire, on utilise souvent à la place la représentation hexadécimale. La raison est qu'il est relativement facile de faire la conversion entre hexadécimale et le binaire. En effet, un chiffre hexadécimal correspond exactement à quatre chiffres binaires, il faut donc utiliser la conversion suivante :

hexadécimal	binaire	hexadécimal	binaire
0	0000	8	1000
1	0001	9	1001
2	0010	a	1010
3	0011	b	1011
4	0100	c	1100
5	0101	d	1101
6	0110	e	1110
7	0111	f	1111

Ainsi, pour convertir la valeur `0x42` en binaire, vous devez prendre le premier chiffre (`4`), le convertir en binaire (`0100`), puis faire la même chose avec le second chiffre (`2`, ce qui donne `0010`). La représentation binaire finale est donc `0b01000010`.

Il est quand même possible d'afficher la représentation binaire

d'un nombre, en utilisant la classe `std::bitset`. Cette classe sera étudiée en détail dans un chapitre Complément, mais pour le moment, vous pouvez utiliser la syntaxe :

main.cpp

```
#include <iostream>
#include <bitset>

int main() {
    std::cout << "0b" << std::bitset<8>(0b101010) << std::endl;
    std::cout << "0b" << std::bitset<8>(42) << std::endl;
}
```

affiche :

```
0b000101010
0b000101010
```

Le chiffre 8 correspond au nombre de bits à utiliser, pensez à l'adapter si vous utilisez des nombres entiers plus grands. Et n'oubliez pas la directive d'inclusion `bitset`.

Comme cela a été expliqué au début du chapitre, toutes les valeurs que manipule un ordinateur sont en fait codées interne en binaire. Cet encodage en binaire dans la mémoire de l'ordinateur est appelée *représentation binaire*. Vous n'aurez généralement pas besoin de manipuler directement les valeurs sous forme binaire, mais cette représentation est suffisamment importante pour que cela soit détaillé ici.

La conversion des nombres entiers décimaux positifs en binaire est relativement simple. Pour les autres types de données (valeurs entières négatives, nombres réels, chaînes de caractères, etc.), la conversion n'est pas aussi simple et naturelle. Il existe en fait différentes normes de conversion, qui expliquent comment convertir une valeur d'un type donné en sa représentation binaire. Et il existe souvent plusieurs normes pour un même type de données.

Un exemple classique de normalisation d'encodage concerne les caractères. Vous verrez qu'il existe des normes telles que "ASCII",

“Windows-1252” ou “UTF-8”. Au final, il existe des centaines de [formes différentes d'encodage](#).

Un point important à comprendre avec l'encodage : une même valeur pourra donner différentes représentations binaires, selon l'encodage utilisé. Et une même valeur binaire en mémoire pourra être représentée par différentes valeurs, de différents types, selon l'encodage utilisé. Ainsi, il est tout à fait possible de “convertir” un nombre réel en caractère en passant par une représentation binaire, mais cela n'aura généralement pas de sens.

Heureusement, le compilateur vérifie les types que l'on utilise lorsque l'on fait des conversions, pour que cela conserve un sens. Il faudra juste faire attention de ne pas empêcher le compilateur de faire son travail. Mais nous reviendrons là dessus plus tard.

Une séquence de 8 bits (ou de 2 chiffres hexadécimaux, c'est équivalent) est appelée un octet (1 o), 1024 o donnent 1 kibi-octet (1 Kio), 1024 Kio donnent 1 mébi-octet (1 Mio), 1024 Mio donnent 1 gibi-octet (1 Gio) et 1024 Gio donnent 1 tebi-octet (1 Tio).

Vous rencontrerez souvent une notation un peu différente, utilisant les préfixes du système métrique : kilo-, méga-, giga- et téra-. Généralement, ces suffixes seront équivalents, c'est-à-dire seront basé sur un rapport de 1 à 1024 entre deux unités de grandeur.

Cependant, dans le système métrique, le rapport devrait être de 1000 au lieu de 1024 et certains utilisent volontairement cette différence pour maintenir une ambiguïté chez le lecteur. Voir [Préfixe binaire](#) pour plus de détail.

## Les opérateurs arithmétiques

Maintenant que vous savez écrire et lire les nombres binaires, vous allez pouvoir les manipuler. Comme ce sont des nombres entiers, les opérateurs arithmétiques présentés dans le chapitre précédent peuvent être utilisés :

main.cpp

```
#include <iostream>

int main() {
    std::cout << 0b1010 + 0b1011 << std::endl; // addition
    std::cout << 0b0011 - 0b1101 << std::endl; // soustraction
    std::cout << 0b1000 * 0b1011 << std::endl; // multiplication
    std::cout << 0b1001 / 0b0010 << std::endl; // division entière
}
```

affiche :

```
21
-10
88
4
```

Notez bien que ces opérations donnent le même résultat que si vous aviez écrit les nombres en représentation binaire. Par exemple, pour la division :

main.cpp

```
#include <iostream>

int main() {
    std::cout << 0b1001 << std::endl;
    std::cout << 0b0010 << std::endl;
    std::cout << 9 / 2 << std::endl;
    std::cout << 0b1001 / 0b0010 << std::endl;
}
```

affiche :

```
9  
2  
4  
4
```

**Exos :** faire une addition et une multiplication binaire “à la main”.

main.cpp

```
#include <iostream>  
#include <bitset>  
  
int main() {  
    std::cout << " " << std::bitset<8>(0b0100100) << std::endl;  
    std::cout << "+ " << std::bitset<8>(0b0101001) << std::endl;  
    std::cout << " -----" << std::endl;  
    std::cout << " = " << std::bitset<8>(0b0100100 +  
0b0101001) << std::endl;  
}
```

affiche :

```
00100100  
+ 00101001  
-----  
= 01001101
```

## L'opérateur négation

En complément de ces opérateurs arithmétiques, il existe des opérateurs travaillant sur la représentation binaire, que l'on appelle opérateurs logiques bit à bit. Le premier opérateur est la négation `~`, qui permet d'inverser tous les bits d'un nombre (les 0 deviennent des 1 et les 1 deviennent des 0). Par exemple :

main.cpp

```
#include <iostream>
```

```
#include <bitset>

int main() {
    std::cout << " ~" << std::bitset<8>( 0b0100100 ) << std::endl;
    std::cout << "= " << std::bitset<8>( ~0b0100100 ) << std::endl;
    std::cout << std::endl;
    std::cout << " ~" << std::bitset<8>( 0b1001011 ) << std::endl;
    std::cout << "= " << std::bitset<8>( ~0b1001011 ) << std::endl;
}
```

affiche :

```
~00100100
=
11011011

~01001011
=
10110100
```

Si vous n'utilisez pas `std::bitset`, mais affichez en hexadécimal, le résultat est un peu différent :

```
main.cpp
#include <iostream>

int main() {
    std::cout << std::showbase << std::hex;
    std::cout << 0b1 << std::endl;
    std::cout << ~0b1 << std::endl;
    std::cout << std::endl;
    std::cout << 0b0110011 << std::endl;
    std::cout << ~0b0110011 << std::endl;
}
```

affiche :

```
0x1
0xffffffffe
```

```
0x33  
0xffffffffcc
```

Le résultat peut paraître surprenant. Si on réécrit ces deux nombres en binaire et qu'on les aligne avec les valeurs binaires d'origine, on obtient :

```
0b1 = 1  
0xfffffffffe = 1111 1111 1111 1111 1111 1111 1111 1110  
  
0b110011 = 11 0011  
0xffffffffcc = 1111 1111 1111 1111 1111 1111 1100 1100
```

ajouter un schéma, comme pour les opérateurs suivants

Si on se rappelle que les 0 devant un nombre peuvent être ignorés ( $1 = 01 = 001 = 0001$ , etc.), on comprend que l'opération est réalisée sur des nombres entiers de 32 bits (ou 4 octets), quelque soit le nombre de bits que l'on utilise pour écrire le nombre. Les 0 manquants devant le nombre sont ajoutés avant l'opération.

Cela signifie qu'en interne, ces nombres entiers sont représentés par défaut par 32 bits (4 octets), *quelque soit le nombre de bits utilisés pour les écrire*.

On pourrait penser que c'est du gâchis de mémoire d'utiliser 32 bits pour la représentation interne, alors que l'on écrit des nombres de 1 ou 6 bits. La raison est qu'un ordinateur est optimisé pour travailler avec des représentations de taille déterminée (généralement 32 ou 64 bits pour les ordinateurs de bureau). Le compilateur C++ adapte donc le nombre de bits en fonction de ce qui est le plus optimal pour l'ordinateur, mais il est possible de forcer l'utilisation de représentations de taille spécifique. Vous verrez cela dans un prochain chapitre.

## Le décalage de bits

Un autre type d'opérateur logique sont les opérations de décalage à droite `>>` et à gauche `<<`. Ces opération permettent de décaler les bits à droite ou à gauche d'un certain nombre de bits. Par exemple :

main.cpp

```
#include <iostream>

int main() {
    std::cout << std::hex << std::showbase;
    std::cout << (0b11011000 << 1) << std::endl; // décalage
de 1 bit à gauche
    std::cout << (0b11011000 << 2) << std::endl; // décalage
de 2 bit à gauche
    std::cout << std::endl;
    std::cout << (0b11011000 >> 1) << std::endl; // décalage
de 1 bit à droite
    std::cout << (0b11011000 >> 2) << std::endl; // décalage
de 2 bit à droite
}
```

Remarquez bien les parenthèses. Sans celle-ci, le compilateur ne pourra pas faire la différence entre les opérateur de décalage de bits `<<` et `>>` et les opérateurs de flux `<<` et `>>`, ce qui ne produira pas le comportement attendu.

Plus généralement, il faudra faire attention en C++ à la syntaxe, un même opérateur pouvant signifier des choses différentes selon le contexte.

affiche :

```
0x1b0
0x360

0x6c
0x36
```

Affichons les valeurs en binaire et alignons les pour mieux comprendre :

```
0b11011000 = 0000 0000 0000 0000 0000 0000 1101 1000
```

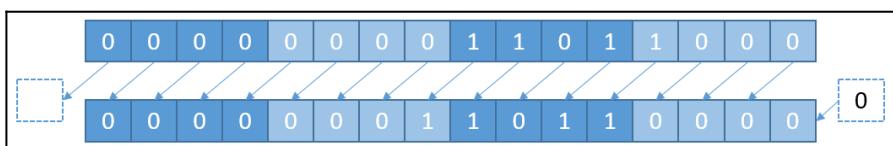
```
0x1b0 =      0000 0000 0000 0000 0000 0001 1011 0000 //  
décalage de 1 bit à gauche
```

```
0x360 =      0000 0000 0000 0000 0000 0011 0110 0000 //  
décalage de 2 bit à gauche
```

```
0x6c =      0000 0000 0000 0000 0000 0000 0110 1100 //  
décalage de 1 bit à droite
```

```
0x36 =      0000 0000 0000 0000 0000 0011 0110 //  
décalage de 2 bit à droite
```

Prenons par exemple le premier décalage (décalage de 1 bit à gauche). Avec un schéma, cela devrait être encore plus clair :



On voit :

- que la séquence de 0 et de 1 est identique ;
- qu'elle est décalée de 1 bit vers la gauche ;
- qu'un 0 est inséré à droite ;
- que le 0 à gauche est perdu.

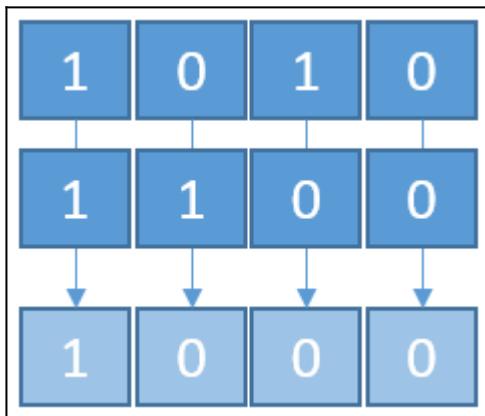
**Exos** : comparer les division et multiplication par 2, 4, etc avec les décalage

## Les opérateurs logiques bit à bit

Pour terminer, il existe les opérateurs logiques "AND" ("ET") `&`, "OR" ("OU") `|` et XOR ("OU Exclusif") `^` pour les nombres. Ils sont similaires aux opérateurs de même nom que vous avez vu précédemment pour les

booléens, sauf qu'ils s'appliquent sur chaque bit d'un nombre. Ainsi, le premier bit du résultat est calculé à partir du premier bit de chaque nombre, le deuxième bit du résultat à partir du deuxième bit de chaque nombre, et ainsi de suite. L'opérateur "OU exclusif" n'a pas d'équivalent pour les booléens, pour rappel il retourne vrai lorsque l'une des opérandes est vraie, mais pas les deux.

Par exemple, pour l'opérateur "AND", on aura le schéma suivant :



Le code suivant permet de vérifier les différents opérateurs logique :

main.cpp

```
#include <iostream>
#include <bitset>

int main() {
    std::cout << "  " << std::bitset<8>(0b1010) << std::endl;
    std::cout << "  " << std::bitset<8>(0b1100) << std::endl;
    std::cout << " -----" << std::endl;
    std::cout << "& " << std::bitset<8>(0b1010 & 0b1100) <<
std::endl; // AND
    std::cout << " | " << std::bitset<8>(0b1010 | 0b1100) <<
std::endl; // OR
    std::cout << "^ " << std::bitset<8>(0b1010 ^ 0b1100) <<
std::endl; // XOR
}
```

affiche :

```
00001010  
00001100  
-----  
& 00001000  
| 00001110  
^ 00000110
```

On retrouve les tables logiques données pour les booléens :

a	b	$\sim a$	$a \& b$	$a   b$	$a ^ b$
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

**Exos:** utilisation de mask avec opérateur logiques.

## Exercices

Dans un ordinateur, composé de transistors, ces derniers forment des portes logiques. Ces portes permettent de réaliser tous les calculs.

- pour 1 bit, réécrire l'addition avec les opérations logiques
- uniquement avec NAND
- pour 8 bits, idem

[Chapitre précédent](#) [Sommaire principal](#) [Chapitre suivant](#)

Cours, C++

A ajouter : setw, [syntaxe alternative : flags, setf, unsetf, precision, width]. Cf [std::ios\\_base](#)

## Les nombres réels

Vous avez vu dans les chapitres précédents comment écrire et manipuler les nombres entiers. Et vous avez vu que le C++ faisait la distinction entre les nombres entiers (nombres "sans virgules") et les nombres réels (nombre "avec virgule"). En particulier, lors d'une division, le résultat est totalement différent selon le type de nombre.

main.cpp

```
#include <iostream>

int main() {
    std::cout << "11 / 4 = " << 11 / 4 << std::endl;
// division entière
    std::cout << "11.0 / 4.0 = " << 11.0 / 4.0 << std::endl;
// division réelle
}
```

affiche :

```
11 / 4 = 2
11.0 / 4.0 = 2.75
```

Pour faire des calculs mathématiques, les nombres entiers ne seront généralement pas suffisants. Par exemple, si vous souhaitez calculer la circonférence d'un cercle à partir de son diamètre, il faudra multiplier celui-ci par le nombre Pi, qui vaut environ 3,14159265358979323846264338327950288... Ces types de nombres et de calculs ne peuvent pas être réalisés avec des nombres entiers, il faudra donc utiliser des nombres conçus spécialement dans ce but : les nombres à virgule flottante (*floating-point numbers*), que l'on appelle aussi nombres réels.

## Écrire des nombres réels

Les nombres à virgule flottante sont donc simplement des nombres qui s'écrivent avec des chiffres après la virgule, comme par exemple Pi.

Un rappel important : en français, les nombres réels sont écrits avec une virgule (*comma* en anglais). Le C++ est basé sur l'anglais, il utilise donc le point comme séparateur décimal. La virgule ayant un sens spécifique en C++, vous n'aurez peut-être pas de message si vous faites l'erreur. Faites donc attention sur ce point.

À partir de là, l'écriture de nombres réels est relativement simple, par exemple pour écrire quelques  [constantes de physique](#) :

main.cpp

```
#include <iostream>

int main() {
    std::cout << "Pi = " << 3.1415926535 << std::endl;
    std::cout << "Nombre d'or = " << 1.6180339887 << std::
endl;
    std::cout << "Vitesse de la lumière = " << 299792.458 << "
km/s" << std::endl;
}
```

affiche :

```
Pi = 3.14159
Nombre d'or = 1.61803
Vitesse de la lumière = 299792 km/s
```

Comme pour les nombres entiers, il est possible d'utiliser le guillemet droit simple ' pour faciliter la lecture des nombres contenant beaucoup de chiffres. Par convention, on écrit séparé généralement en blocs de 3 chiffres :

main.cpp

```
#include <iostream>
```

```
int main() {
    std::cout << "Pi = " << 3.141'592'653'5 << std::endl;
    std::cout << "Nombre d'or = " << 1.618'033'988'7 <<
std::endl;
    std::cout << "Vitesse de la lumière = " << 299'792.458
<< " km/s" << std::endl;
}
```

Lorsque la partie entière (la partie à gauche du point) ou la partie décimale (à droite du point) d'un nombre ne contient que des zéros, il est possible de ne pas les écrire. Par exemple, "0.123" est équivalent à ".123" et "123.0" est équivalent à "123.". Mais faites attention de ne pas oublier le point, sinon cela correspondra à un nombre entier :

main.cpp

```
#include <iostream>

int main() {
    std::cout << 0.123 << " = " << .123 << std::endl;
    std::cout << 123.0 << " = " << 123. << std::endl;
    std::cout << (123. / 5.) << " != " << (123 / 5) << std::
endl;
}
```

ce qui affiche :

```
0.123 = 0.123
123 = 123
24.6 != 24
```

Cependant, il faut se rappeler qu'un code sera plus souvent lu qu'écrit et il n'y a aucun intérêt à faire l'économie d'un caractère. Il est donc préférable d'écrire quand même les chiffres zéro pour la lisibilité. C'est ce que l'on fera systématiquement dans ce cours.

## Notation scientifique

Pour Pi ou la vitesse de la lumière  $c$ , cela ne pose pas de problème de les

écrire comme présenté au-dessus. Par contre, si l'on souhaite écrire par exemple la constante de perméabilité magnétique du vide  $\mu_0$ , il faudra écrire :

```
std::cout << "Perméabilité magnétique du vide = " <<  
    0.000'001'256'637'061'4 << " kg.m/A²/s²" << std::endl;
```

Ce qui commence à faire beaucoup de zéros. Si on veut écrire la constante de Planck, c'est encore plus compliqué : il faut écrire 34 zéros après le séparateur décimal. On ne va pas le faire, cela devient trop compliqué. Heureusement, le C++ prend en charge la notation scientifique des nombres réels.

La notation scientifique consiste à écrire un nombre sous la forme :

$\$ \$ \text{mantissee} \times 10^{\text{exposant}} \$ \$$

La mantisse (*mantissa* ou *significand* en anglais) est un nombre réel positif ou négatif et l'exposant (*exponent* en anglais) est un nombre entier positif ou négatif.

Pour obtenir la notation scientifique d'un nombre, il faut multiplier ou diviser ce nombre plusieurs fois par 10, jusqu'à obtenir un nombre supérieur ou à égal à 1 et strictement inférieur à 10. Le résultat est la mantisse, le nombre de fois que l'on a multiplié ou divisé par 10 est l'exposant.

Par exemple, si on prend le nombre 0.000123. On peut écrire :

```
0.000123 * 10 = 0.00123  
0.00123 * 10 = 0.0123  
0.0123 * 10 = 0.123  
0.123 * 10 = 1.23
```

Il faut donc multiplier 4 fois 0.000123 par 10 pour obtenir 1.23. Le nombre 0.000123 peut donc s'écrire :

$\$ \$ 1.23 \times 10^{-4} \$ \$$

Pour écrire un nombre en C++ en utilisant cette notation, il faut ajouter

l'exposant après le caractère “e” ou “E” dans l'écriture d'un nombre. Par exemple :

```
main.cpp
#include <iostream>

int main() {
    std::cout << "Perméabilité magnétique du vide = " <<
        1.256'637'061'4e-6 << " kg.m/A2/s2" << std::endl;
    std::cout << "Constante de Planck = " << 6.626'069
'57e-34 << " kg.m2/s" << std::endl;
}
```

affiche :

```
Perméabilité magnétique du vide = 1.25664e-06 kg.m/A2/s2
Constante de Planck = 6.62607e-34 kg.m2/s
```

Un dernier point sur la notation scientifique. Si vous écrivez :

```
main.cpp
#include <iostream>

int main() {
    std::cout << 123456789.0e10 << std::endl;
    std::cout << 0.0000000123456789e-10 << std::endl;
}
```

cela affichera :

```
1.23457e+18
1.23457e-18
```

On peut remarquer deux points sur le résultat affiché :

- la fin des chiffres (le 8 et le 9) ne sont pas affichés. On verra cette problématique dans la partie suivante, sur la mise en forme de l'affichage des nombres réels.
- les exposants utilisés ont été modifiés automatiquement lors de l'affichage.

C'est un point fondamental en informatique : la façon dont les données sont affichées ne correspond pas forcément à comment elles sont manipulées en interne (représentation binaire). C'est valable pour les nombres, mais également n'importe quel type de données.

## Mettre en forme la sortie

Il est possible de changer l'affichage des nombres entiers avec `std::cout` en utilisant des directives. Il existe également des directives pour modifier l'affichage des nombres réels.

La première chose que vous allez pouvoir modifier est l'utilisation de la notation scientifique ou non. La directive `std::scientific` force l'affichage en notation scientifique, `std::fixed` force l'affichage sans notation scientifique et `std::defaultfloat` affiche en utilisant la notation par défaut.

main.cpp

```
#include <iostream>

int main() {
    std::cout << "Pi" << std::endl;
    std::cout << " Notation scientifique : " << std::
scientific << 3.141'592'653'5 << std::endl;
    std::cout << " Notation fixe : " << std::fixed
<< 3.141'592'653'5 << std::endl;
    std::cout << " Notation par défaut : " << std::
defaultfloat << 3.141'592'653'5 << std::endl;
    std::cout << std::endl;

    std::cout << "Constante de Planck" << std::endl;
    std::cout << " Notation scientifique : " <<
std::scientific << 6.626'069'57e-34 << std::endl;
    std::cout << " Notation fixe : " << std::fixed
<< 6.626'069'57e-34 << std::endl;
    std::cout << " Notation par défaut : " <<
std::defaultfloat << 6.626'069'57e-34 << std::endl;
}
```

affiche :

```
Pi  
Notation scientifique : 3.141593e+00  
Notation fixe : 3.141593  
Notation par défaut : 3.14159
```

```
Constante de Planck  
Notation scientifique : 6.626070e-34  
Notation fixe : 0.000000  
Notation par défaut : 6.62607e-34
```

Notez bien que ces directives ne s'appliquent qu'aux nombres réels. Si vous écrivez un nombre entier, celui-ci ne sera pas affiché en notation scientifique :

main.cpp

```
#include <iostream>  
  
int main() {  
    std::cout << std::scientific << 3.0 << std::endl;  
    std::cout << std::scientific << 3 << std::endl;  
}
```

affiche :

```
3.000000e+00  
3
```

N'oubliez jamais ce point fondamental : le C++ est un langage de programmation à typage fort, les types ont une importance particulière.

Même si `3`, `3.0`, `"3"` et `'3'` représentent la même chose pour vous (le chiffre 3), ces valeurs ont des types différents (respectivement un nombre entier, un nombre réel, une chaîne de caractères et un caractère) et s'utilisent différemment en C++.

La directive `std::showpos` permet d'afficher le signe plus devant les

nombres positifs et la directive `std::noshowpos` permet de ne pas l'afficher.

La directive `std::showpoint` permet d'afficher le séparateur décimal et ajoute le nombre de zéros nécessaires après le séparateur décimal pour atteindre le nombre de chiffres fixés par la directive `std::setprecision` décrite plus bas. La directive `std::noshowpoint` permet de ne pas afficher le séparateur décimal pour les nombres dont la partie décimale est nulle.

### main.cpp

```
#include <iostream>

int main() {
    std::cout << std::showpos << 123.456 << std::endl;
    std::cout << std::showpos << 0.123 << std::endl;
    std::cout << std::showpos << 123.0 << std::endl;
    std::cout << std::noshowpos << 123.456 << std::endl;
    std::cout << std::endl;

    std::cout << std::showpoint << 123.0 << std::endl;
    std::cout << std::noshowpoint << 123.0 << std::endl;
}
```

affiche :

```
+123.456
+0.123
+123
123.456

123.000
123
```

Pour terminer, la directive `std::setprecision` permet de définir le nombre de chiffres significatifs à afficher (donc sans compter les zéros au début et à la fin des nombres réels). Cette directive prend un nombre entier en paramètre, correspondant aux nombres de chiffres à afficher.

Cette directive est disponible dans le fichier d'en-tête `iomanip`, il faut

donc l'inclure au début du programme avec la directive de préprocesseur `#include` :

main.cpp

```
#include <iostream>
#include <iomanip>

int main() {
    std::cout << "Pi (défaut) = " << 3.141'592'653'5 <<
std::endl;
    std::cout << "Pi (précision 0) = " <<
std::setprecision(0) << 3.141'592'653'5 << std::endl;
    std::cout << "Pi (précision 1) = " << std::setprecision(
1) << 3.141'592'653'5 << std::endl;
    std::cout << "Pi (précision 2) = " <<
std::setprecision(2) << 3.141'592'653'5 << std::endl;
    std::cout << "Pi (précision 5) = " << std::setprecision(
5) << 3.141'592'653'5 << std::endl;
    std::cout << "Pi (précision 10) = " <<
std::setprecision(10) << 3.141'592'653'5 << std::endl;
    std::cout << "Pi (précision 30) = " << std::setprecision(
30) << 3.141'592'653'5 << std::endl;
    std::cout << "Pi (précision 200) = " <<
std::setprecision(200) << 3.141'592'653'5 << std::endl;
    std::cout << "Pi (valeur invalide) = " << std::
setprecision(-1) << 3.141'592'653'5 << std::endl;
}
```

affiche :

```
Pi (défaut) = 3.14159
Pi (précision 0) = 3
Pi (précision 1) = 3
Pi (précision 2) = 3.1
Pi (précision 5) = 3.1416
Pi (précision 10) = 3.141592654
Pi (précision 30) = 3.14159265350000005412312020781
Pi (précision 200) =
3.1415926535000000541231202078051865100860595703125
Pi (valeur invalide) = 3.14159
```

Vous pouvez remarquer que si vous définissez une précision supérieure au nombre de chiffres que vous avez écrit dans votre littérale (par exemple 30 dans le code précédent), alors `std::cout` affichera des chiffres incorrects à la fin de votre nombre.

Si vous définissez une valeur très grande (200 dans le code précédent), alors le nombre de chiffres affichés sera limité à 50 maximum (le nombre maximal de chiffres peut varier selon le compilateur et le système).

Pour terminer, si vous définissez une valeur négative, alors `std::cout` affiche de nouveau les nombres en utilisant la précision par défaut (6 chiffres dans l'exemple précédent).

## Opérateurs arithmétiques

Comme pour les nombres entiers, il est possible de réaliser des calculs arithmétiques avec les nombres réels : addition `+`, soustraction `-`, multiplication `*` et division `/` (n'oubliez pas la différence entre division entière et réelle). L'opérateur modulo n'a pas de sens pour les nombres réels.

`main.cpp`

```
#include <iostream>

int main() {
    std::cout << "Addition : " << 12.34 + 56.78 << std::endl;
    std::cout << "Soustraction : " << 12.34 - 56.78 << std::endl;
    std::cout << "Multiplication: " << 12.34 * 56.78 << std::endl;
    std::cout << "Division : " << 12.34 / 56.78 << std::endl;
}
```

affiche :

```
Addition : 69.12
Soustraction : -44.44
Multiplication: 700.665
```

Division : 0.21733

De la même manière, vous pouvez utiliser les opérateurs de comparaison présentés dans le chapitre sur les entiers : est égal `==`, est différent `!=`, est supérieur `>`, est supérieur ou égal `>=`, est inférieur `<`, est inférieur ou égal `<=`.

main.cpp

```
#include <iostream>

int main() {
    std::cout << std::boolalpha;
    std::cout << "Est égale : " << (12.34 == 56.78) << std::endl;
    std::cout << "Est différent : " << (12.34 != 56.78) << std::endl;
    std::cout << "Est supérieur : " << (12.34 > 56.78) << std::endl;
    std::cout << "Est supérieur ou égal : " << (12.34 >= 56.78) << std::endl;
    std::cout << "Est inférieur : " << (12.34 < 56.78) << std::endl;
    std::cout << "Est inférieur ou égal : " << (12.34 <= 56.78) << std::endl;
}
```

affiche :

```
Est égale : false
Est différent : true
Est supérieur : false
Est supérieur ou égal : false
Est inférieur : true
Est inférieur ou égal : true
```

## Egalité de nombres réels et epsilon

Encore une fois, il faut insister sur un point très important : ce qui est affiché ne correspond pas forcément à ce qu'il y a en mémoire. Par

exemple :

main.cpp

```
#include <iostream>

int main() {
    std::cout << 3.0000000000001 << std::endl;
    std::cout << 3.0000000000000001 << std::endl;
}
```

affiche :

```
3
3
```

Si vous utilisez l'opérateur d'égalité sur ces nombres, vous aurez peut-être des surprises :

main.cpp

```
#include <iostream>

int main() {
    std::cout << 3 << " est égal à " << 3.0000000000001 << "
? "
    << std::boolalpha << (3 == 3.0000000000001) <<
std::endl;
    std::cout << 3 << " est égal à " << 3.0000000000000001
<< " ? "
    << std::boolalpha << (3 == 3.0000000000000001)
<< std::endl;
}
```

affiche :

```
3 est égal à 3 ? false
3 est égal à 3 ? true
```

Les nombres 3, 3.0000000000001 et 3.0000000000000001 sont affichés de la même façon (ce qui est normal, si vous vous souvenez du rôle de `std::setprecision`). Par contre, le résultat du test d'égalité change aussi, ce qui est plus surprenant.

Dans le premier cas, 3.0000000000001 est effectivement enregistré en mémoire comme différent de 3. Le test d'égalité échoue puisque l'ordinateur sait que ces nombres sont différents (même s'il affiche le même nombre). Dans le second cas, le nombre dépasse les capacités de l'ordinateur (plus précisément la façon dont les nombres sont enregistrés en mémoire). En effet, le nombre est arrondi en mémoire à 3 donc le test d'égalité n'échoue pas.

Il est facile de voir ici que les nombres entrés sont différents. Mais imaginez que vous réalisez des calculs scientifiques complexes. Cela pourrait générer des nombres qui semblent identiques (à l'affichage), mais qui sont en fait différents.

Pour éviter cela, on ne compare pas directement l'égalité de deux nombres réels en général. On va considérer que deux nombres sont égaux s'ils sont suffisamment proches, compte tenu d'une éventuelle erreur de précision. Dit autrement, il y a égalité si la valeur absolue de la différence entre les deux nombres est inférieure à une certaine erreur de précision maximale appelée *epsilon* (nous reviendrons sur la valeur absolue plus tard).

```
std::abs(nombre1 - nombre2) < epsilon
```

Epsilon sera choisie en fonction du type de calculs que vous réalisez, de la précision des nombres, de l'ordinateur, etc. Bref, elle va dépendre du contexte.

main.cpp

```
#include <iostream>
#include <cmath>

int main() {
    std::cout << 3 << " est égal à " << 3.0000000000001 << "
? "
        << std::boolalpha << (std::abs(3 -
3.000000000001) < 0.0001) << std::endl;
    std::cout << 3 << " est égal à " << 3.000000000000001
<< " ? "
        << std::boolalpha << (std::abs(3 -
```

```
3.0000000000000001) < 0.0001) << std::endl;  
}
```

affiche :

```
3 est égal à 3 ? true  
3 est égal à 3 ? true
```

## Infini et pas-un-nombre

Vous avez vu dans le chapitre sur les entiers que la division par zéro produisait un *undefined behavior* (comportement indéterminé en français). Essayons le même code en utilisant des nombres réels :

```
main.cpp  
-----  
#include <iostream>  
  
int main() {  
    std::cout << "1.0 / 0.0 = " << (1.0 / 0.0) << std::endl;  
}
```

affiche :

```
1.0 / 0.0 = inf
```

Contrairement à la division sur les nombres entiers, la division sur les nombres réels ne produit pas un comportement indéterminé. Le résultat affiché “inf” signifie *infinity* (infini en français), ce qui a un sens au niveau mathématique.

Si on modifie le code pour calculer la division de 0 par 0 :

```
main.cpp  
-----  
#include <iostream>  
  
int main() {  
    std::cout << "0.0 / 0.0 = " << (0.0 / 0.0) << std::endl;  
}
```

on obtient :

```
0.0 / 0.0 = nan
```

Le résultat affiché “nan” signifie *not-a-number* (pas-un-nombre en français). Ce résultat est obtenu lorsqu'un calcul n'a pas de sens mathématique. Mais du point de vue du C++, cela n'est pas une erreur (au sens d'erreur de calcul ou d'exécution).

Ces deux valeurs particulières “inf” et “nan” sont parfaitement définies en C++. Pour les écrire directement vous pouvez utiliser les macros `INFINITY` et `NAN` définies dans le fichier d'en-tête `cmath` :

`main.cpp`

```
#include <iostream>
#include <cmath>

int main() {
    std::cout << "Infini : " << INFINITY << std::endl;
    std::cout << "Pas-un-nombre : " << NAN << std::endl;
}
```

affiche :

```
Infini : inf
Pas-un-nombre : nan
```

Cependant, dans la majorité des cas, vous aurez plus souvent besoin de vérifier qu'un calcul ne produit pas une de ces valeurs. Pour cela, vous pouvez utiliser une des fonctions suivantes :

- `std::isinf()` pour tester si une valeur est infinie ;
- `std::isnan()` pour tester si une valeur est pas-un-nombre ;
- `std::isfinite()` pour tester si une valeur est finie, c'est-à-dire qu'elle n'est ni infinie, ni pas-un-nombre.

Ces fonctions sont également définies dans le fichier d'en-tête `cmath`, il faut donc l'inclure dans votre code.

`main.cpp`

```

#include <iostream>
#include <cmath>

int main() {
    std::cout << std::boolalpha;
    std::cout << "isinf(1.0 / 1.0) = " << std::isinf(1.0 /
1.0) << std::endl;
    std::cout << "isinf(0.0 / 1.0) = " << std::isinf(0.0 /
1.0) << std::endl;
    std::cout << "isinf(1.0 / 0.0) = " << std::isinf(1.0 /
0.0) << std::endl;
    std::cout << "isinf(0.0 / 0.0) = " << std::isinf(0.0 /
0.0) << std::endl;

    std::cout << std::endl;
    std::cout << "isnan(1.0 / 1.0) = " << std::isnan(1.0 /
1.0) << std::endl;
    std::cout << "isnan(0.0 / 1.0) = " << std::isnan(0.0 /
1.0) << std::endl;
    std::cout << "isnan(1.0 / 0.0) = " << std::isnan(1.0 /
0.0) << std::endl;
    std::cout << "isnan(0.0 / 0.0) = " << std::isnan(0.0 /
0.0) << std::endl;

    std::cout << std::endl;
    std::cout << "isfinite(1.0 / 1.0) = " << std::isfinite(
1.0 / 1.0) << std::endl;
    std::cout << "isfinite(0.0 / 1.0) = " << std::isfinite(
0.0 / 1.0) << std::endl;
    std::cout << "isfinite(1.0 / 0.0) = " << std::isfinite(
1.0 / 0.0) << std::endl;
    std::cout << "isfinite(0.0 / 0.0) = " << std::isfinite(
0.0 / 0.0) << std::endl;
}

```

affiche :

```

isinf(1.0 / 1.0) = false
isinf(0.0 / 1.0) = false
isinf(1.0 / 0.0) = true
isinf(0.0 / 0.0) = false

```

```
isnan(1.0 / 1.0) = false
isnan(0.0 / 1.0) = false
isnan(1.0 / 0.0) = false
isnan(0.0 / 0.0) = true

isfinite(1.0 / 1.0) = true
isfinite(0.0 / 1.0) = true
isfinite(1.0 / 0.0) = false
isfinite(0.0 / 0.0) = false
```

## Les fonctions mathématiques

Pour terminer avec l'utilisation de base des nombres réels, le C++ propose de nombreuses fonctions mathématiques usuelles, comme le calcul des puissances et des racines ou les fonctions trigonométriques. Nous n'allons pas toutes les voir dans ce chapitre, elles ne présentent pas de difficulté particulière d'utilisation. Ces fonctions sont définies dans le fichier d'en-tête `cmath`, vous trouverez la totalité des fonctions dans la documentation : [Common mathematical functions](#).

Voici quelques exemples d'utilisation :

main.cpp

```
#include <iostream>
#include <cmath>

int main() {
    std::cout << "Fonctions basiques" << std::endl;
    std::cout << "valeur absolue : " << fabs(-1.0) << std::endl;
    std::cout << "minimum : " << fmin(1.0, 2.0) << std::endl;
    std::cout << "maximum : " << fmax(1.0, 2.0) << std::endl;
    std::cout << std::endl;
    std::cout << "Fonctions exponentielles et logarithmiques"
<< std::endl;
    std::cout << "exponentielle naturelle : " << exp(1.0) << std::endl;
    std::cout << "logarithme naturel : " << log(1.0) << std::endl;
```

```

        std::cout << "logarithme décimal : " << log10(1.0) <<
std::endl;
        std::cout << std::endl;
        std::cout << "Fonctions puissances" << std::endl;
        std::cout << "puissance : " << pow(2.0, 3.0) << std::
endl;
        std::cout << "racine carrée : " << sqrt(2.0) << std::
endl;
        std::cout << "racine cubique : " << cbrt(2.0) << std::
endl;
        std::cout << std::endl;
        std::cout << "Fonctions trigonométriques" << std::endl;
        std::cout << "sinus : " << sin(1.0) << std::endl;
        std::cout << "cosinus : " << cos(1.0) << std::endl;
        std::cout << "tangente : " << tan(1.0) << std::endl;
        std::cout << "arc sinus : " << asin(1.0) << std::endl;
        std::cout << "arc cosinus : " << acos(1.0) << std::endl;
        std::cout << "arc tangente : " << atan(1.0) << std::endl;
        std::cout << std::endl;
        std::cout << "Fonctions hyperboliques" << std::endl;
        std::cout << "sinus hyperbolique : " << sinh(1.0) << std
::endl;
        std::cout << "cosinus hyperbolique : " << cosh(1.0) <<
std::endl;
        std::cout << "tangente hyperbolique : " << tanh(1.0) <<
std::endl;
        std::cout << "argument sinus hyperbolique : " << asinh(
1.0) << std::endl;
        std::cout << "argument cosinus hyperbolique : " << acosh(
1.0) << std::endl;
        std::cout << "argument tangente hyperbolique : " <<
atanh(1.0) << std::endl;
        std::cout << std::endl;
        std::cout << "Fonctions partie entière" << std::endl;
        std::cout << "partie entière par défaut : " << floor(1.3)
<< std::endl;
        std::cout << "partie entière par excès : " << ceil(1.3)
<< std::endl;
        std::cout << "troncature : " << trunc(1.3) << std::endl;
        std::cout << "arrondi : " << round(1.3) << std::endl;
}

```

affiche :

```
Fonctions basiques
valeur absolue : 1
minimum : 1
maximum : 2
```

```
Fonctions exponentielles et logarithmiques
exponentielle naturelle : 2.71828
logarithme naturel : 0
logarithme décimal : 0
```

```
Fonctions puissances
puissance : 8
racine carrée : 1.41421
racine cubique : 1.25992
```

```
Fonctions trigonométriques
sinus : 0.841471
cosinus : 0.540302
tangente : 1.55741
arc sinus : 1.5708
arc cosinus : 0
arc tangente : 0.785398
```

```
Fonctions hyperboliques
sinus hyperbolique : 1.1752
cosinus hyperbolique : 1.54308
tangente hyperbolique : 0.761594
argument sinus hyperbolique : 0.881374
argument cosinus hyperbolique : 0
argument tangente hyperbolique : inf
```

```
Fonctions partie entière
partie entière par défaut : 1
partie entière par excès : 2
troncature : 1
arrondi : 1
```

# Initialisation, concaténation et conversion des chaînes

Fondamentalement, l'informatique englobe tout ce qui concerne le traitement de l'information. Vous avez vu dans les chapitres précédents les bases du calcul numérique. Un autre type de données qui est souvent manipulé en informatique sont les chaînes de caractères. Les chapitres suivants détaillent leurs manipulations en C++.

## Les fonctions membres et non membres

parler plus tôt des fonctions. Et mieux expliquer, parce que là, c'est caca...

partie trop détaillée et pas à sa place

Les fonctions sont des traitements spécifiques que l'on peut appliquer sur les données. Vous avez déjà utilisé de telles fonctions dans les chapitres précédents, comme par exemple les fonctions mathématiques ou la fonction `size` pour connaître la taille d'une chaîne de caractères `string`.

Les fonctions peuvent s'écrire selon deux syntaxes différentes. Par exemple :

```
double const d { 123.456 };
std::cout << std::exp(d) << std::endl; // utilisation de la
fonction exp

std::string s{ "hello, world!" };
std::cout << s.size() << std::endl; // utilisation de la
fonction size
```

Le premier type de fonction, représenté par la fonction mathématique exponentielle `exp`, s'appelle une fonction non membre. Le service qu'elle rend (calculer l'exponentielle d'un nombre réel) s'applique à la valeur qui est passée en argument entre les parenthèses.

Le résultat du calcul est retourné directement par la fonction. Cela signifie que lors de l'évaluation de la fonction, la fonction est remplacée par le résultat du calcul. Ainsi, le code suivant :

```
double const result { std::exp(1.0) };
std::cout << std::exp(1.0) << std::endl;
```

est équivalent au code suivant, après évaluation de la fonction `exp` :

```
double const result { 2.71828182845905 };
std::cout << 2.71828182845905 << std::endl;
```

Chaque fonction possède une signature spécifique, qui définit son nom, le nombre d'arguments qu'elle prend, leur type respectif et le type de valeur qu'elle retourne. Une fonction peut également ne pas prendre d'argument ou ne pas retourner de valeur.

Le second type de fonction s'appelle une fonction membre et s'applique à un objet spécifique. Un exemple d'une telle fonction est la fonction `size` de la classe `string` :

```
std::string s{ "hello, world!" };
std::cout << s.size() << std::endl; // utilisation de la
fonction size
```

Ce type de fonction permet d'obtenir des informations sur un objet (c'est le cas de `size`) ou de modifier cet objet. Un exemple de fonction qui modifie un objet pourrait être la fonction `clear`, qui efface une chaîne de caractères.

```
std::string s{ "hello, world!" };
s.clear(); // on efface s
std::cout << s << std::endl;
```

Ce code n'affiche rien, puisque la variable `s` ne contient plus aucune

chaîne après l'appel de la fonction membre `clear`.

Une fonction membre peut également prendre aucun, un ou plusieurs arguments et retourner ou non une valeur.

Une fonction membre ne peut pas être appelée comme une fonction non membre. Si vous essayez d'appeler la fonction `size` seule ou en lui passant une chaîne en argument, cela produira une erreur :

```
size(s); // erreur  
s.size("test"); // erreur
```

Cependant, dans certains cas, il existe une fonction membre et une fonction non membre qui possède le même nom. C'est par exemple le cas de la fonction `begin`, que vous verrez plus tard. Vous pouvez donc écrire :

```
string s { "hello, world!" };  
s.begin();  
begin(s);
```

Les deux lignes avec `begin` sont équivalentes et font exactement la même chose. Par contre, n'oubliez pas qu'il s'agit bien de deux fonctions différentes (cela sera important lorsque vous écrivez vos propres fonctions si c'est pas important maintenant, faut il en parler maintenant ?).

Dans tous les cas, il ne faut pas hésiter à se référer à la documentation, pour connaître la signature de chaque fonction.

## Exercices

Lire la documentation de quelques fonctions, membre et non membre. Ecrire le code correct pour utiliser ces fonctions, même si on les connaît pas, en fonction de la documentation.

## Initialisation et assignation d'une chaîne

Les chaînes de caractères en C++ sont manipulées via la classe `std::string` de la bibliothèque standard. L'initialisation ou la modification d'une chaînes se fait en utilisant l'initialisation avec des crochets et l'opérateur d'affectation `=` :

main.cpp

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

int main() {
    std::string const s1 { "hello, world!" }; // initialisation avec une littérale
    std::string s2 {}; // initialisation par défaut
    s2 = "bonjour tout le monde !"; // affectation

    std::cout << "hello, world!" << std::endl; // afficher une littérale
    std::cout << s2 << std::endl; // afficher une variable
}
```

Un point important concernant les littérales chaînes de caractères : elles ne sont pas de type `string`, mais du type `const char *` hérité du C. L'utilisation de ce type n'est pas recommandé, sauf pour initialiser une chaîne de type `string`. La classe `string` apporte des garanties plus fortes que `const char *` (fuite mémoire) et offre beaucoup plus de fonctionnalités (que vous allez voir dans la suite de ce chapitre et dans les chapitres suivants).

### Les littérales string

Dans la prochaine norme du C++14, il sera possible de créer directement une littérale chaîne de caractères de type `string`, en ajoutant le suffixe "s" après la chaîne. Cela permettra d'éviter les conversions de `const char *` en `string` et permettra d'utiliser directement `auto` :

```
string s1 { "hello, world!"s };
auto s2 = "hello, world!"s;
```

## La concaténation

### Concaténation de chaînes de type string

En informatique, la concaténation est simplement l'opération qui consiste à créer une chaîne en associant plusieurs chaînes mises bout à bout. Par exemple, la chaîne “hello world” est la concaténation des chaînes “hello” et “ world”.

Dans de nombreux langages, l'opérateur `+` appliqué sur des chaînes permet de réaliser la concaténation. C'est également le cas pour les chaînes de type `string` en C++ :

```
main.cpp
#include <iostream>
#include <string>

int main()
{
    std::string const s1 { "hello" };
    std::string const s2 { " world" };
    std::string const s3 { s1 + s2 }; // concaténation de s1
et s2
    std::cout << s3 << std::endl;
}
```

affiche :

```
hello world
```

Il est possible de concaténer autant de chaînes que l'on souhaite de cette manière :

### main.cpp

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

int main()
{
    std::string const s1 { "Bonjour" };
    std::string const s2 { " tout" };
    std::string const s3 { " le monde" };
    std::string const s4 { s1 + s2 + s3 };
    std::cout << s4 << std::endl;
}
```

affiche :

```
bonjour tout le monde
```

La concaténation est utilisable bien sûr lors de l'initialisation (comme ci-dessus), mais également pour modifier une chaîne avec l'opérateur d'affectation `=` :

### main.cpp

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

int main()
{
    std::string const s1 { "hello" };
    std::string const s2 { " world" };
    std::string s3 {};
    s3 = s1 + s2;
    std::cout << s3 << std::endl;
}
```

Il est également possible de modifier une chaîne en ajoutant d'autres chaînes à la suite. La méthode évidente est d'utiliser les opérateurs de concaténation `+` et d'affectation `=` :

### main.cpp

```
#include <iostream>
```

```
#include <string>

int main()
{
    std::string s1 { "hello" };
    std::string const s2 { " world" };
    s1 = s1 + s2;
    std::cout << s1 << std::endl;
}
```

Cette opération, qui consiste à réaliser un calcul sur lui-même, est assez classique et le C++ fournit l'opérateur `+=` pour simplifier cette écriture.

```
s1 += s2; // est équivalent à s1 = s1 + s2;
```

Une autre approche est d'utiliser la fonction membre `append` (*ajouter*) pour la concaténation. La fonction `append` permet de réaliser plus de choses que l'opérateur `+`, vous verrez cela dans les exercices.

```
s1.append(s2);
```

## Concaténation d'une chaîne avec autre chose

Il est possible d'utiliser la concaténation avec d'autres types que les chaînes de type `string`, mais avec des restrictions. Il est possible d'ajouter un caractère à une chaîne par exemple :

```
main.cpp

#include <iostream>
#include <string>

int main()
{
    std::string const s1 { "hello" };
    char c { '!' };
    std::string const s2 { s1 + c };
    std::cout << s2 << std::endl;
}
```

affiche :

```
hello!
```

Il est également possible de concaténer une chaîne avec une littérale chaîne ou une littérale caractère :

main.cpp

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

int main()
{
    std::string s1 { "hello" };

    s1 += " world"; // concaténation d'une littérale chaîne
    s1 += '!';      // concaténation d'une littérale
                     // caractère

    std::cout << s1 << std::endl;
}
```

affiche :

```
hello world!
```

En revanche, si vous essayez de concaténer plusieurs littérales chaînes ou caractère, le compilateur va produire une erreur. Par exemple :

main.cpp

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

int main()
{
    std::string const s1 { "hello" + " world" }; // erreur
    std::cout << s1 << std::endl;
}
```

produit l'erreur :

```
main.cpp:6:36: error: invalid operands to binary expression
('const char *' and 'const char *')
    std::string const s1 { "hello" + " world" };
                           ~~~~~ ^ ~~~~~
1 error generated.
```

Si vous essayez avec une littérale caractère, le compilateur produit un avertissement, mais pas d'erreur. Pour lui, appliquer l'opérateur `+` sur une littérale chaîne et une littérale caractère a un sens. Mais ce n'est pas une concaténation.

```
std::string const s1 { "hello" + '!' };
```

affiche l'avertissement :

```
main.cpp:6:36: warning: adding 'char' to a string does not
append to the string [-Wstring-plus-int]
    std::string const s1 { "hello" + '!' };
                           ~~~~~^~~~~~
main.cpp:6:36: note: use array indexing to silence this
warning
    std::string const s1 { "hello" + '!' };
                           ^
                           &      [     ]
1 warning generated.
```

Pour comprendre ce problème, il faut se rappeler un point : les littérales chaînes de caractères ne sont pas de type `string`, mais de type `const char *`. Et ce type, hérité du C, ne propose tout simplement pas les mêmes fonctionnalités que le type `string` pour gérer les chaînes de caractères. Ce qui explique pourquoi le C++ a introduit le type `string` (en C, il existe des fonctions spécifiques pour réaliser la concaténation, ce qui est plus lourd à utiliser que l'opérateur `+`). Afin de résoudre ce problème, le C++14 introduit un “s” après la chaîne afin de clarifier ces cas ambigus.

## Conversions entre chaînes et nombres

De la même façon, si vous avez essayé de concaténer une chaîne avec

un nombre, vous avez obtenu un avertissement lors de compilation. Par exemple :

```
main.cpp
#include <iostream>
#include <string>

int main()
{
    std::string s1 { "hello " };
    s1 += 1234.56;
    std::cout << s1 << std::endl;
}
```

afficher le message :

```
main.cpp:7:11: warning: implicit conversion from 'double' to
'char' changes value
from 1234.56 to 127 [-Wliteral-conversion]
    s1 += 1234.56;
    ~~~~~ ^~~~~~
hello □
```

Cette opération est acceptée par le compilateur, mais il convertit le nombre en un caractère de type `char` (pour rappel, le type `char` est un type de nombre entier interprété comme un caractère). Cette conversion implicite d'un nombre en `char` est autorisée, mais ne produit pas le résultat attendu.

Pour réaliser correctement la concaténation d'un nombre dans une chaîne, il faut dans un premier temps convertir le nombre en `string` en utilisant la fonction `std::to_string` :

```
s1 += std::to_string(1234.56);
```

Après conversion, la chaîne est correctement affichée :

```
hello 1234.560000
```

Il est également possible de réaliser l'opération inverse, c'est-à-dire

convertir une chaîne contenant un nombre en un nombre. Il y a cependant une différence importante : lorsque l'on utilise la fonction `to_string`, le compilateur sait, lors de la compilation, quel type de nombre il doit convertir en chaîne :

```
double const d { 123.456 };
std::to_string(d); // conversion d'un double en string

int const i { 123456 };
std::to_string(i); // conversion d'un int en string
```

Dans le cas d'une conversion d'une chaîne `string` en un nombre, le compilateur n'a aucun moyen de déduire le type du nombre qu'il faut convertir à la compilation (il faudrait qu'il parcourt le contenu de la chaîne, ce qui n'est possible que lors de l'exécution). Il faut donc indiquer explicitement le type du nombre que l'on souhaite obtenir à partir d'une chaîne `string`, ce qui explique pourquoi il existe plusieurs fonctions pour convertir une chaîne en un nombre, alors qu'il n'existe qu'une seule fonction pour l'opération inverse.

La signature des fonctions de conversion suit un même modèle, il est relativement simple de se souvenir de la fonction à utiliser en fonction du type que l'on souhaite obtenir. Le nom des fonctions s'écrit :

- “s” pour `string` ;
- “to”, que l'on peut traduire par “en”, “vers” ;
- un identifiant pour le type : “i” pour `int`, “f” pour `float`, etc.

Le tableau suivant résume les différentes fonctions :

Fonction	Type du nombre	Identifiant de type
<code>stoi</code>	<code>int</code>	<code>i</code>
<code>stol</code>	<code>long int</code>	<code>l</code>
<code>stoll</code>	<code>long long int</code>	<code>ll</code>
<code>stoul</code>	<code>unsigned long int</code>	<code>ul</code>
<code>stoull</code>	<code>unsigned long long int</code>	<code>ull</code>
<code>stof</code>	<code>float</code>	<code>f</code>
<code>stod</code>	<code>double</code>	<code>d</code>

stold	long double	ld
-------	-------------	----

### main.cpp

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

int main()
{
    std::string int_str { "123" };
    int const i { std::stoi(int_str) };
    std::cout << i << std::endl;

    std::string double_str { "123.456" };
    auto const d = std::stod(double_str);
    std::cout << d << std::endl;
}
```

affiche :

```
123
123.456
```

La chaîne à convertir peut contenir des espaces devant le nombre à convertir et n'importe quel caractère après le nombre sans que cela ne pose de problème de conversion. Par contre, si le nombre est précédé d'un caractère autre qu'un espace, cela produit une erreur d'exécution :

```
std::string int_str { " 123abc" }; // ok
std::string int_str { "a123" };    // erreur
```

La seconde ligne produit l'erreur suivante ("argument invalide") :

```
terminate called after throwing an instance of
'std::invalid_argument'
  what():  stoi
bash: line 8: 30337 Aborted                  (core dumped)
./a.out
```

## Exercices

- utiliser les autres formes de la fonction `append`

<a href="#">Chapitre précédent</a>	<a href="#">Sommaire principal</a>	<a href="#">Chapitre suivant</a>
------------------------------------	------------------------------------	----------------------------------

[Cours, C++](#)

# L'algèbre de Boole

Vous avez vu dans les chapitres précédents comment utiliser les booléens et les opérations de base : ET, OU, NON et OU-EXCLUSIF. En fait, l'utilisation des booléens est plus riche et complexe que cela. Ils sont à la base d'une branche des mathématiques, appelée [Algèbre de Boole](#), en l'honneur de son créateur, le mathématicien George Boole.

Avoir des bonnes bases avec cet algèbre est important pour tous les développeurs, et plus encore pour les développeurs C++, c'est pourquoi nous allons voir cela plus en détail.

## Quelques rappels

Pour rappel, un booléen est une variable logique, qui peut prendre deux valeurs. Peu importe comment sont nommées ces valeurs, vous rencontrez parfois : "vrai" et "faux", "oui" et "non", "haut" et "bas", "positif" et "négatif", 0 et 1, "Titi" et "Grosminet", etc.

Le C++ utilise les mots-clés `true` (vrai) et `false` (faux) pour représenter les booléens.

Certaines valeurs dans la liste ont un sens historique. En particulier, "vrai" et "faux" correspondent au fait que les booléens représentent le résultat d'une proposition logique. Par exemple "la terre est plus grosse que le soleil".

En C++, une proposition logique (que l'on appelle aussi expression logique) sera écrite en utilisant les opérateurs logiques (que vous avez déjà vu) :

Opérateur	Synonyme	Opérateur C++
ET	Conjonction	<code>&amp;&amp;</code>
OU	Disjonction	<code>  </code>

NON	Négation	!
-----	----------	---

Pour rappel, voici la table de vérité de ces opérateurs (elle est redonnée pour vous éviter de retourner dans le chapitre [Logique binaire et calcul booléen](#), mais il faudra la connaître par cœur par la suite. Mais pas d'inquiétude, vous la connaîtrez à force de l'utiliser) :

a	b	!a	a && b	a    b
0	0	1	0	0
0	1	1	0	1
1	0	0	0	1
1	1	0	1	1

Il est classique de définir d'autres opérateurs logiques par combinaison des opérateurs de base. Par exemple :

- NON-OU (*NOR*) correspond à `!(a || b)` ;
- NON-ET (*NAND*) correspond à `!(a && b)` ;
- OU-Exclusif (*XOR*) correspond à `(a && !b) || (!a && b)` ;
- NON-OU-Exclusif (*XNOR*) correspond à `!( (a && !b) || (!a && b) )`.

Ces opérateurs n'ont pas d'équivalent direct en C++, mais il est facile de les écrire en utilisant les opérateurs de base du C++.

## Propriétés des opérateurs logiques

L'algèbre de Boole définit un certain nombre de propriétés. Il est assez facile d'écrire un code C++ pour les vérifier. Un code d'exemple est donné pour la première propriété, vous devrez écrire les codes C++ correspondant aux autres propriétés comme exercice.

## Ordre d'évaluation

Une expression contenant plusieurs opérateurs (et donc plus de deux opérandes) est évaluée en évaluant chaque opérateur un par un. Dans le

cas des opérateurs logiques, les opérateurs sont évalués en suivant les règles suivantes :

- les opérateurs sont évalués dans l'ordre : NÉGATION, ET puis OU ;
- lorsque les opérateurs sont identiques, ils sont évalués de gauche à droite.

Par exemple :

```
a || b && c
```

pourra s'écrire de la façon suivante :

```
a || (b && c)
```

Un autre exemple :

```
a || b || c
```

pourra s'écrire de la façon suivante :

```
(a || b) || c
```

Ces règles s'appliquent aux opérateurs logiques. Vous avez déjà vu que les opérateurs arithmétiques ont aussi un ordre d'évaluation spécifique (multiplication et division avant addition et soustraction). Vous verrez par la suite les règles d'évaluation pour l'ensemble des opérateurs existant en C++.

Dans tous les cas, il est plus simple et moins ambiguë d'utiliser les parenthèses pour exprimer clairement une expression.

**Exercice** : évaluer l'ordre des opérations suivantes, en ajoutant les parenthèses pour isoler chaque opérateur, sans changer le résultat de l'expression.

```
a && b && c
```

```
a || b || c
```

```
a || b && c
```

```
a && b || c  
!  
!a && b  
a || !b  
  
a && b || c && d  
a || b && c || d
```

## Complémentarité

Dans certains cas, l'utilisation de l'opérateur négation peut se simplifier. Le cas le plus simple est la double-négation qui donne le même résultat :

```
!!a == a
```

Sous forme de table de vérité :

a	!a	!!a
0	1	0
1	0	1

Lorsque l'opérateur négation est utilisé avec d'autres opérateurs, on peut également simplifier dans certains cas :

```
a && !a == false  
a || !a == true
```

Sous forme de table de vérité :

a	!a	a && !a	a    !a
0	1	0	1
1	0	0	1

## Idempotence

La première notion est relativement simple : appliquer un opérateur en utilisant deux fois la même valeur est équivalent à utiliser directement la

variable. Si on regarde la table de vérité précédente, on peut remarquer que lorsque **a** et **b** sont égaux, alors le résultat des opérations logiques aura la même valeur :

a	b	a && b	a    b
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

De façon plus formelle, on peut écrire :

$\$ \$ \text{A} \sim \text{ET} \sim \text{A} \sim = \sim \text{A} \$ \$ \$ \text{A} \sim \text{OU} \sim \text{A} \sim = \sim \text{A} \$ \$$

## Eléments neutres

Un élément neutre est une valeur booléenne qui ne va pas changer le résultat d'une expression logique. Pour l'opérateur ET, on peut remarquer dans la table de vérité que si **a** est vrai, le résultat de l'expression aura la même valeur que **b** :

a	b	a && b
0	0	0
0	1	0
1	0	0
1	1	1

Pour l'opérateur OU, ça sera faux qui est l'élément neutre :

a	b	a    b
0	0	0
0	1	1
1	0	1
1	1	1

On peut écrire :

Ces relations permettent de simplifier une expression. Si vous démontrez qu'une opérande est toujours vraie ou toujours fausse, vous pouvez simplifier certaines expressions en supprimant l'opérateur et l'opérande.

## Absorption

L'absorption peut être considérée comme l'inverse de l'élément neutre : si  $a$  est faux, le résultat de l'opérateur ET sera toujours faux.

a	b	a	&&	b
<b>0</b>	<b>0</b>		<b>0</b>	
<b>0</b>	<b>1</b>		<b>0</b>	
1	0		0	
1	1			1

Pour l'opérateur OU, si  $a$  est vrai, le résultat sera toujours vrai :

a	b	a		b
0	0	0		
0	1		1	
<b>1</b>	<b>0</b>		<b>1</b>	
<b>1</b>	<b>1</b>		<b>1</b>	

## L'écriture formelle :

\$\$ \text{A} \sim \text{ET} \sim \text{Faux} \sim = \sim \text{Faux} \$\$  
\$\text{A} \sim \text{OU} \sim \text{Vrai} \sim = \sim \text{Vrai}\$

Cette propriété est particulièrement intéressante en C++, puisque cela permet de ne pas évaluer une expression, si possible. Comme expliqué dans le chapitre [Logique binaire et calcul booléen](#), le C++ utilise la *lazy evaluation* (évaluation paresseuse) lorsque c'est possible. Pour évaluer les expressions suivantes :

\$\$ \text{A} \sim \text{ET} \sim \text{Expression complexe} \$\$ \$\$ \text{B} \sim \text{OU} \sim \text{Expression complexe} \$\$

Si **A** est *faux*, le résultat sera toujours *faux*, quelle que soit la valeur de l'expression complexe. Le programme C++ va donc évaluer la valeur de **A** dans un premier temps. Si cette valeur est *faux*, le programme n'évaluera pas l'expression complexe et retournera directement *faux*. De même, si **B** est *vrai*, le programme n'évaluera pas l'expression complexe et retournera directement *vrai*.

## Commutativité

La commutativité signifie que l'on peut changer l'ordre des valeurs (opérandes) dans une expression, sans changer le résultat. Cette propriété est valide pour les opérateurs binaires (qui prennent deux opérandes, donc les opérateurs ET et OU), mais pas pour NON (opérateur unaire, qui ne prend qu'une opérande).

En utilisant une écriture plus formelle, on peut donc écrire :

\$\$ \text{A} \sim \text{ET} \sim \text{B} \iff \text{B} \sim \text{ET} \sim \text{A} \$\$  
\$\$ \text{A} \sim \text{OU} \sim \text{B} \iff \text{B} \sim \text{OU} \sim \text{A} \$\$

On peut vérifier cela en écrivant la table de vérité correspondant aux deux expressions :

a	b	$a \& b$	$b \& a$
0	0	0	0
0	1	0	0
1	0	0	0
1	1	1	1

Vérifions cela avec un code C++ :

```
main.cpp
```

```
#include <iostream>
```

```

int main() {
    std::cout << "(A ET B) est équivalent à (B ET A), pour A
faux et B faux ? " <<
        std::boolalpha << ((false && false) == (false &&
false)) << std::endl;
    //                                     A   ET   B           B   ET   A
}

```

affiche :

(A ET B) est équivalent à (B ET A), pour A faux et B faux ?  
true

**Exercice** : écrire les trois autres lignes, correspondant à A vrai et B faux, à A faux et B vrai et à A vrai et B vrai.

## Associativité

Comme vu précédemment, les opérateurs identiques sont évalués de gauche à droite. En fait, les opérateurs logiques sont associatifs et l'ordre d'évaluation ne change pas le résultat :

```

(a && b) && c == a && (b && c) == a && b && c
(a || b) || c == a || (b || c) == a || b || c

```

**Exercice** : compléter les tables de vérité suivantes :

a	b	c	a && b	(a && b) && c	b && c	a && (b && c)
0	0	0	...	...	...	...
a	b	c	a    b	(a    b)    c	b    c	a    (b    c)
0	0	0	...	...	...	...

## Distributivité

Pour terminer avec les propriétés de base des opérateurs logiques, la distributivité fait intervenir trois opérandes et deux opérateurs différents (contrairement à l'associativité).

$$(a \ \&\& b) \ ||\ c == (a \ ||\ c) \ \&\& (b \ ||\ c)$$

$$(a \ ||\ b) \ \&\& c == (a \ \&\& c) \ ||\ (b \ \&\& c)$$

Note : il est classique de noter l'opérateur ET comme une multiplication et l'opérateur OU comme une addition, du fait de leur similarité. Avec cette écriture, les opérateurs logiques deviennent :

$$\$ \$ ( \text{A} \sim \text{OU} \sim \text{B} ) \sim \text{ET} \sim \text{C} \iff ( \text{A} \sim + \sim \text{B} ) \sim \times \sim \text{C} \$ \$$$

Sous cette forme, on reconnaît plus facilement la propriété : “le produit d'une somme est égal à la somme des produits”. Celle-ci est connue sous le nom de “distributivité de la multiplication par rapport à l'addition” et permet d'écrire :

$$\$ \$ ( \text{A} \sim + \sim \text{B} ) \sim \times \sim \text{C} \iff \text{A} \sim \times \sim \text{C} \sim + \sim \text{B} \sim \times \sim \text{C} \$ \$$$

Si on revient aux opérateurs logiques, on obtient finalement :

$$\$ \$ \text{A} \sim \times \sim \text{C} \sim + \sim \text{B} \sim \times \sim \text{C} \iff ( \text{A} \sim \text{ET} \sim \text{C} ) \sim \text{OU} \sim ( \text{B} \sim \text{ET} \sim \text{C} ) \$ \$$$

Ce qui correspond bien à la propriété initiale. On parle donc de “distributivité de l'opérateur ET par rapport à l'opérateur OU”. Si vous vous souvenez de la propriété pour l'addition et la multiplication, vous pouvez retrouver facilement la propriété équivalente pour les opérateurs logiques.

Une remarque importante quand même : contrairement à l'addition et la multiplication, la propriété équivalente obtenue en inversant les opérateurs est vraie pour les opérateurs logiques : l'opérateur OU est distributif par rapport à l'opérateur ET. Ceci n'est pas vrai pour les opérateurs arithmétiques (l'addition n'est pas distributive par rapport à la

multiplication).

**Exercice** : compléter les tables de vérité :

a	b	c	a && b	(a && b)    c	a    c	b    c	(a    c) && (b    c)
0	0	0	...	...	...	...	...
a	b	c	a    b	(a    b) && c	a && c	b && c	(a && c)    (b && c)
0	0	0	...	...	...	...	...

## Lois de De Morgan

Les lois (ou théorèmes) de [De Morgan](#) permettent de remplacer un opérateur ET dans une expression logique par un opérateur OU et vice-versa.

$$!(a \&\& b) == !a \mid\mid !b$$

$$!(a \mid\mid b) == !a \&\& !b$$

## Exercices

- Écrire les opérateurs NON-ET, NON-OU, OU-Exclusif et NON-OU-Exclusif avec les opérateurs de base du C++.
- Écrire tous les opérateurs en utilisant uniquement l'opérateur NON-ET.
- Écrire un additionneur à un bit. Un additionneur va prendre deux valeurs booléennes et retournera la somme et la retenue de ces valeurs, suivant la table de vérité suivante :

a	b	Somme	Retenue
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

faire un schéma

- Écrire un additionneur quatre bits.
- Écrire un multiplicateur  $4 \times 4$  bits.
- Écrire un multiplexeur.

**[Chapitre précédent](#) [Sommaire principal](#) [Chapitre suivant](#)**

Cours, C++

# Les nombres complexes

L'une des utilisations majeures du C++ est le calcul numérique intensif. Pour cela, il est possible d'utiliser des fonctionnalités du langage (comme par exemple les nombres à virgule flottante que vous avez vus précédemment) ou des outils, plus adaptés, apportés par des bibliothèques spécialisées ou non. Vous découvrirez certains de ces instruments dans les projets d'exemple.

La bibliothèque standard fournit également quelques outils mathématiques. Vous allez voir dans ce chapitre un exemple permettant de manipuler des nombres complexes. Le but ici n'est pas de présenter mathématiquement les nombres complexes, mais de donner un aperçu de ce qu'une bibliothèque peut fournir comme outils.

## Rappels mathématiques

### explications bof bof

Pour commencer, un petit rappel sur les nombres complexes. Pour ceux intéressés par les détails, vous pouvez consulter la page de Wikipédia correspondante ([Nombre complexe](#)) ou consulter un cours de mathématiques.

Les équations du second degré peuvent s'écrire de la façon suivante :

$$\$\$ ax^2 + bx + c = 0 \$\$$$

Si vous vous souvenez de vos cours de lycée, pour résoudre cette équation, on calcule le discriminant (réalisant ou  $\$ \Delta \$$ ) donné par cette formule :

$$\$\$ \Delta = b^2 - 4ac \$\$$$

Si ce discriminant est positif, l'équation admet deux solutions réelles. S'il

est nul, elle admet une solution réelle double. Le dernier cas, qui nous intéresse plus particulièrement ici, est que si le discriminant est négatif, cette équation n'admet pas de solutions réelles.

Cependant, on peut définir les nombres complexes de la façon suivante :

$$z = x + iy$$

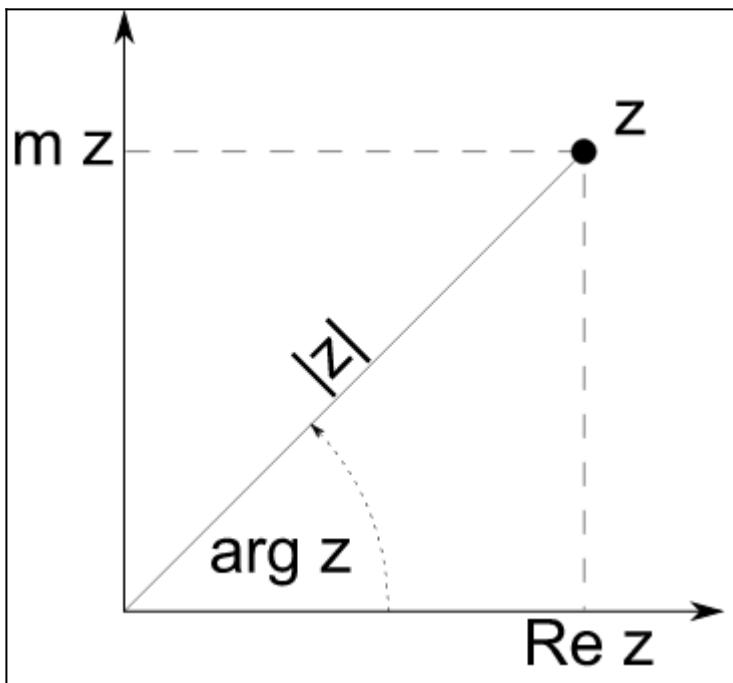
avec  $x$  et  $y$  réels et :

$$i^2 = -1$$

Dans ce cas, l'équation admet deux solutions complexes (conjuguées).

$x$  est la partie réelle d'un nombre complexe et  $y$  est la partie imaginaire.

Il est classique de représenter les nombres complexes sur un plan, de la façon suivante :



On peut également définir un nombre complexe par l'angle entre l'axe des abscisses et la droite passant par l'origine et le point, que l'on appelle "argument" d'un nombre complexe, et la distance entre l'origine et le point, que l'on appelle "module".

## Nombres complexes en C++

Les nombres complexes sont fournis par la classe `std::complex` de la bibliothèque standard. Comme toujours, la documentation de cette classe se trouve sur le site [cppreference](#). En consultant cette page, vous pouvez trouver au début le fichier d'en-tête à inclure : `<complex>`. Vous avez également des codes d'exemple à la fin.

## Écrire une littérale d'un nombre complexe

Pour rappel, une littérale est une valeur écrite directement dans un code. Pour écrire un nombre complexe en C++, une syntaxe a dû être définie pour cela. Une première approche peut être d'écrire directement un nombre complexe en suivant sa définition. Par exemple, pour le nombre :

`$$ z = 2 + 3 i $$`

On pourrait écrire :

```
2 + 3 * i;
```

Cette écriture est tout à fait valable. Cependant, sous cette forme, `i` correspond à l'écriture d'une variable (vous verrez cela par la suite), ce qui peut être limitant. Surtout que l'on a l'habitude en C++ d'utiliser la variable `i` comme indice (dans un tableau par exemple), d'où le risque de confusion. (Mais rien ne vous interdit par la suite, quand vous saurez créer une variable, de créer cette variable `i` avec le nombre complexe `z = 0 + 1 i`).

Pour éviter cette ambiguïté, une autre écriture a été choisie : un nombre imaginaire pur s'écrit sous forme d'une littérale réelle, suivie du suffixe `i`. Par exemple :

```
2.0 + 3.0i;
```

Dans ce code, la littérale `2.0` correspond à une littérale réelle et la littérale `3.0i` à un nombre imaginaire pur (en pratique, à une littérale réelle avec le suffixe `i`). Il est possible d'afficher directement un nombre complexe avec `std::cout` :

`main.cpp`

```
#include <iostream>
#include <complex>

int main() {
    using namespace std::literals;
    std::cout << "2+3i = " << (2.0 + 3.0i) << std::endl;
}
```

affiche :

```
2+3i = (2,3)
```

### **std::literals**

Il existe différents types de préfixes et suffixes permettant de modifier une littérale. Certains sont définis dans le langage et ne nécessitent donc pas d'espace de noms. D'autres sont définis dans la bibliothèque standard, dans une bibliothèque externe ou par les utilisateurs. Ils peuvent dans ce cas être dans un espace de noms.

C'est le cas ici du suffixe `i`, qui se trouve dans l'espace de noms `std::literals`. Il faut donc avertir le compilateur que vous souhaitez utiliser cet espace de noms pour qu'il puisse utiliser `i`. (Essayez de compiler sans le `using`.)

Généralement, vous avez trois syntaxes possibles pour utiliser une fonctionnalité définie dans un espace de noms. Pour rappel :

```
using namespace std;
using std::cout;
std::cout;
```

L'utilisation d'un préfixe ou d'un suffixe nécessite qu'il n'y ait rien entre la littérale et le modificateur. Donc, il n'est pas possible d'écrire par exemple :

```
3.0std::literals::i // erreur
```

C'est pour cette raison qu'il est plus pratique d'utiliser `using` pour travailler avec les modificateurs de littérales.

Vous voyez ici qu'un nombre complexe est affiché sous la forme `(partie réelle,partie imaginaire)`. On peut en particulier afficher `i` et vérifier que le carré de `i` vaut -1.

main.cpp

```
#include <iostream>
#include <complex>

int main() {
    using namespace std::literals;
    std::cout << "i = " << 1.0i << std::endl;
    std::cout << "i² = " << (1.0i * 1.0i) << std::endl;
}
```

affiche :

```
i = (0,1)
i² = (-1,0)
```

Le résultat affiché correspond bien aux valeurs attendues.

Notez bien que le nombre imaginaire `i` ne peut pas s'écrire directement `i` dans un code C++, puisque cela correspondrait à l'écriture d'une variable et non d'une littérale. Le `i` d'une littérale représentant un nombre imaginaire est un suffixe, il doit toujours suivre une littérale numérique.

## Comparer des nombres complexes

Il est possible de comparer l'égalité (ou l'inégalité) des nombres

complexes entre eux en utilisant les opérateurs `==` et `!=`, comme vous l'avez fait avec les nombres entiers et réels.

Par exemple :

main.cpp

```
#include <iostream>
#include <complex>

int main() {
    using namespace std::literals;
    std::cout << std::boolalpha << ((2.0 + 3.0i) == (2.0 +
3.0i)) << std::endl;
    std::cout << std::boolalpha << ((2.0 + 3.0i) != (3.0 +
2.0i)) << std::endl;
}
```

affiche :

```
true
true
```

Même si le calcul de  $i$  au carré donne 1, le résultat affiché correspond au nombre complexe  $(-1, 0)$ . Mathématiquement, cela est correct :

$\$ \$ -1 + 0 i = -1 \$ \$$

Il n'est possible de comparer les nombres complexes que par égalité ou inégalité. Les comparaisons d'ordre (plus petit, plus grand, etc.) n'ont pas de sens pour les complexes.

Cependant, n'oubliez pas que même si deux valeurs sont mathématiquement identiques, le C++ est basé sur un typage fort et différencie les valeurs en fonction de leur type. Ainsi, même si "-1" (nombre entier) est égal à "-1.0" (nombre à virgule flottante) et à "(-1,0)" (nombre complexe), ce sont des valeurs différentes en C++.

Vous pouvez tester ces égalités en utilisant l'opérateur d'égalité `==`. Commençons par la comparaison d'un nombre complexe et d'un nombre à virgule flottante :

### main.cpp

```
#include <iostream>
#include <complex>

int main() {
    using namespace std::literals;
    std::cout << std::boolalpha << ((1.0i * 1.0i) == -1.0)
<< std::endl;
}
```

affiche :

```
true
```

Dans ce cas, pas de problème, le résultat affiché est celui attendu. Si maintenant, vous testez avec un nombre entier :

### main.cpp

```
#include <iostream>
#include <complex>

int main() {
    using namespace std::literals;
    std::cout << std::boolalpha << ((1.0i * 1.0i) == -1) <<
std::endl;
}
```

affiche :

```
main.cpp:6:51: error: invalid operands to binary expression
('complex<double>' and 'int')
    std::cout << std::boolalpha << ((1.0i * 1.0i) == -1) <<
std::endl;
                                         ~~~~~ ^ ~
1 error generated.
```

Dans ce cas, le compilateur produit une erreur, indiquant qu'il ne sait pas comparer un nombre complexe et un nombre entier. Ce sont deux types différents pour lui.

Pour être plus précis, cela signifie que le compilateur connaît l'opérateur

d'égalité `==` entre un complexe et un réel, mais qu'il n'en connaît pas entre un complexe et un entier.

## La classe `std::complex`

Dans le message d'erreur précédent, vous pouvez remarquer que le compilateur interprète le calcul `1.0i * 1.0i` sous forme d'un type qui s'appelle `std::complex<double>`. Voyons plus en détail cela.

Une classe est un type, mais définie par un code C++ et non par le langage. Vous ne trouverez nulle part un fichier C++ qui définit `int` ou `float`, par contre le type `std::complex` est défini dans le fichier d'en-tête "`<complex>`".

Vous pourrez de la même façon créer vos propres types en créant des classes, mais plus tard dans ce cours. (La création de classes a une importance particulière en programmation, on parle de "programmation orientée objet".)

Cette classe est une **abstraction** représentant un nombre complexe :

- cela représente une version "manipulable par l'ordinateur" d'un concept mathématique et pas exactement ce concept mathématique (par exemple, comme vous l'avez vu, il n'est pas possible de comparer un entier avec un nombre complexe en C++, alors que cela peut avoir un sens en mathématique).
- vous n'avez pas besoin de savoir comment est écrit le code C++ définissant cette classe ou même comment l'ordinateur réalise les calculs. Tout ce qu'il vous faut connaître est l'interface publique, c'est à dire la partie de la classe accessible en dehors de la classe.

Cette notion d'abstraction est très importante à comprendre, puisque cela définit comment vous allez utiliser cette classe (interface publique) et ses limites (ce qui la différencie du modèle mathématique). En particulier pour les calculs numériques, n'oubliez pas que les nombres sur un ordinateur ont des limites (valeur minimale, valeur maximale,

nombre maximal de chiffres après la virgule, etc.).

Pour terminer avec la notion `std::complex<double>` : vous avez vu que `std::complex` correspond donc au nom de la classe représentant un nombre complexe. Les nombres complexes sont représentés par deux nombres réels " $x + y i$ ", donc la classe `std::complex` manipule également des nombres réels en interne. Pour le moment, vous n'avez pas vu à quel type correspondent les nombres réels que vous avez écrit dans vos codes C++, mais sachez en fait que le C++ peut utiliser plusieurs types différents pour représenter des nombres réels.

Le type `double` est un de ces types, mais il en existe d'autres (`float`, `long double`, etc.). Les chevrons dans la définition de la classe `std::complex` permettent de préciser le type qui sera manipulé en interne par cette classe. Dit autrement, cela signifie que `std::complex` utilise le type `double` en interne lorsque vous écrivez `std::complex<double>`, elle utilise `float` lorsque vous écrivez `std::complex<float>`, `MonType` lorsque vous écrivez `std::complex<MonType>`, et ainsi de suite.

(Notez bien que c'est toujours un type que vous devez mettre entre les chevrons et pas une valeur).

Pour créer une valeur de type `std::complex<double>`, vous avez vu que le plus simple est donc d'écrire une littérale numérique utilisant le suffixe `i`. Cependant, vous aurez besoin dans certains cas de créer un nombre complexe sans écrire de littérale. Par exemple, si vous souhaitez utiliser le résultat d'une expression pour calculer les parties réelle et imaginaire d'un nombre complexe :

`$$ (2 * 3) + (4 * 5) i $$`

Une première solution est de multiplier le résultat de l'expression de droite par le nombre imaginaire  $i$  (qui s'écrit donc `1.0i` en C++) :

`main.cpp`

```
#include <iostream>
#include <complex>
```

```
int main() {
    using namespace std::literals;
    std::cout << ((2.0 * 3.0) + (4.0 * 5.0) * 1.0i) << std::
endl;
}
```

affiche :

(6,20)

Une autre solution est d'appeler spécifiquement la classe `std::complex<double>` en passant les expressions entre parenthèses, sous la forme : `std::complex<double>(partie réelle, partie imaginaire)`. Concrètement, cela donne le code suivant :

main.cpp

```
#include <iostream>
#include <complex>

int main() {
    using namespace std::literals;
    std::cout << std::complex<double>(2.0 * 3.0, 4.0 * 5.0)
    << std::endl;
}
```

Ce qui affiche la même chose que précédemment.

Notez bien qu'il ne faut pas mettre dans cette écriture l'opérateur `+`, ni le nombre imaginaire `i`.

Cette syntaxe est nouvelle, donc pas forcément claire pour le moment. Mais pas d'inquiétude, vous verrez cela régulièrement, dans de nombreux codes. Retenez simplement l'idée générale :

Le code `std::complex<double>(2.0, 3.0)` signifie que `std::complex` manipule en interne des nombres réels de type `double` et représente le nombre complexe  $2 + 3i$ .

## Les opérations et fonctions sur std::complex

Pour terminer ce chapitre sur les nombres complexes, vous avez vu dans le chapitre sur les nombres réels que le C++ propose de nombreuses fonctions mathématiques pour les réels. C'est également le cas pour les nombres complexes. Cependant, toutes les fonctions sur les nombres réels ne sont pas forcément définies pour les nombres complexes.

Pour commencer, les nombres complexes définissent les opérations arithmétiques de base, comme l'addition et la soustraction entre complexes, ainsi que l'addition, la soustraction, la multiplication et la division entre un complexe et une nombre réel.

main.cpp

```
#include <iostream>
#include <complex>

int main() {
    using namespace std::literals;

    std::cout << ((2.0 + 3.0i) + (4.0 + 5.0i)) << std::endl;
// addition
    std::cout << ((2.0 + 3.0i) - (4.0 + 5.0i)) << std::endl;
// soustraction

    std::cout << ((2.0 + 3.0i) + 4.0) << std::endl;
// addition
    std::cout << ((2.0 + 3.0i) - 4.0) << std::endl;
// soustraction
    std::cout << ((2.0 + 3.0i) * 4.0) << std::endl;
// multiplication
    std::cout << ((2.0 + 3.0i) / 4.0) << std::endl;
// division
}
```

affiche :

```
(6,8)
(-2,-2)
(6,3)
```

```
(-2,3)  
(8,12)  
(0.5,0.75)
```

En complément de ces opérations de base, les nombres complexes peuvent être utilisés avec différentes fonctions mathématiques. La syntaxe à utiliser est similaire à celle que vous avez vu pour les nombres réels :

### main.cpp

```
#include <iostream>  
#include <complex>  
  
int main() {  
    using namespace std::literals;  
  
    std::cout << real(2.0 + 3.0i) << std::endl; // partie  
réelle  
    std::cout << imag(2.0 + 3.0i) << std::endl; // partie  
imaginaire  
    std::cout << abs(2.0 + 3.0i) << std::endl; // module  
(valeur absolue en anglais)  
    std::cout << arg(2.0 + 3.0i) << std::endl; // argument  
    std::cout << norm(2.0 + 3.0i) << std::endl; // norme  
    std::cout << conj(2.0 + 3.0i) << std::endl; // conjugué  
    std::cout << proj(2.0 + 3.0i) << std::endl; //  
projection  
    std::cout << polar(2.0 + 3.0i) << std::endl; //  
coordonnées polaires  
}
```

affiche :

```
2  
3  
3.60555  
0.982794  
13  
(2,-3)  
(2,3)  
((2,3),(0,0))
```

Il existe d'autres fonctions mathématiques sur les nombres complexes, qui s'utilisent de la même façon (voir la documentation pour la liste des fonctions : [Documentation de std::complex](#)) : fonctions exponentielles, puissances, trigonométriques et hyperboliques (voir la page de Wikipédia pour les explications sur ces fonctions mathématiques : [Nombre complexe](#)).

<a href="#">Chapitre précédent</a>	<a href="#">Sommaire principal</a>	<a href="#">Chapitre suivant</a>
------------------------------------	------------------------------------	----------------------------------

# Les nombres à virgule fixe

## Limitation des nombres réels

Les nombres réels sont intéressants, puisqu'ils permettent de représenter des nombres plus grands que les nombres entiers. Si par exemple, vous essayez d'exécuter le code suivant :

main.cpp

```
#include <iostream>

int main() {
    std::cout << 9223372036854775808 << std::endl;
}
```

vous obtiendrez une erreur (seules les premières erreurs sont copiées ici) :

```
main.cpp:4:26: warning: integer constant is so large that it
is unsigned
    std::cout << " " << 9223372036854775808 << std::endl;
                           ^
main.cpp: In function 'int main()':
main.cpp:4:15: error: ambiguous overload for 'operator<<'
(operand types are
'std::basic_ostream<char>' and '__int128')
    std::cout << 9223372036854775808 << std::endl;
                           ^
...
...
```

La première erreur “integer constant is so large that it is unsigned” indique que le nombre entré est tellement grand qu'il n'existe pas de type signé (*signed*, qui peut représenter de nombres positifs et négatifs) qui peut représenter ce nombre. Le compilateur est donc obligé d'utiliser un type non signé (*unsigned*, qui ne peut représenter que des nombres

positifs).

La seconde erreur “ambiguous overload” indique que le compilateur ne sait pas comment afficher ce nombre (plus précisément, il ne sait pas quel opérateur `<<` utiliser avec `std::cout` pour afficher ce nombre).

Note : ce nombre n'a pas été choisi au hasard. Il s'agit du plus grand nombre représentable avec les types de base du C++, plus un. Vous verrez par la suite comment obtenir des informations sur les types, comme par exemple la valeur maximale possible.

Si on modifie un tout petit peu ce code (en ajoutant une décimale), pour utiliser un nombre réel, le compilateur ne produit plus d'erreur :

main.cpp

```
#include <iostream>

int main() {
    std::cout << 9223372036854775808.0 << std::endl;
}
```

affiche :

```
9.22337e+18
```

Comme on peut s'y attendre, le programme affiche le nombre en utilisant la notation scientifique.

On retrouve ici un autre exemple du typage fort du C++. Le compilateur détecte bien que les nombres entiers ne conviennent pas pour représenter ce nombre. Mais comme vous avez choisi d'utiliser un entier, le compilateur ne fait pas la correction pour vous.

Une littérale est représentée par une valeur et un type. Le compilateur prend en compte les deux, pas uniquement la valeur.

Pour autant, les nombres réels ne sont pas parfaits non plus (sinon, on ne s'embêterait pas à faire la distinction entre entiers et réels). Prenons un

autre code d'exemple :

main.cpp

```
#include <iostream>

int main() {
    std::cout << " " << (9223372036854775807 -
9223372036854775806) << std::endl;
    std::cout << " " << (9223372036854775807.0 -
9223372036854775806.0) << std::endl;
}
```

affiche :

```
1
0
```

On a donc deux nombres, représentées dans le premier cas par des entiers et dans le second par des réels (notez la décimale). Ces deux nombres sont grands, mais peuvent être représentés sans problème par des entiers en C++ (plus précisément par le type `long long int`, que vous verrez ensuite). La différence entre ces deux nombres vaut un.

Dans tous les cas, les nombres en C++ sont représentés par un nombre fini d'octets dans la mémoire des ordinateurs. Il n'est donc pas possible de représenter tous les nombres possibles (ce qui n'aurait de toute façon aucun sens, puisque qu'il existe une infinité de nombres réels). Les nombres réels peuvent représenter des nombres plus grands que les nombres entiers parce qu'ils ne peuvent pas représenter les grands nombres avec la même précision que les nombres entiers.

Dans le code d'exemple précédent, les nombres entiers ne sont pas arrondis et le calcul est juste. Au contraire, les nombres réels sont arrondis et sont représentées par la même valeur en mémoire. Le calcul se fait donc sur la même valeur et le résultat est nul.

Encore une fois, il est important d'insister là-dessus : les types ont une grande importance en C++. Le choix du type peut modifier

complètement le résultat d'un calcul. Faites bien attention à cela, c'est une erreur qui revient souvent.

## Principe et arithmétique

Imaginons que vous souhaitez travailler sur des nombres décimaux, mais en conservant la précision des nombres entiers (vous ne voulez pas que les valeurs soient arrondies). Par exemple, dans une application bancaire, qui manipule des centimes (deux chiffres après la virgule).

Le principe des nombres à virgule fixe est relativement simple : au lieu de manipuler des nombres réels, on va multiplier les valeurs par un facteur constant (généralement un multiple de 10) et utiliser cette représentation pour faire les calculs. C'est uniquement lors de l'affichage que l'on va recalculer la valeur décimale exacte.

Par exemple, dans le cas d'une application bancaire, on peut utiliser un facteur 100 :

```
1      =    100  
12.34 =  1234  
123.4  = 12340
```

En C++, il est possible de créer un nouveau type de nombre, qui permet de représenter plus facilement des nombres à virgule fixe, mais vous verrez cela dans la partie sur la programmation objet. Dans ce chapitre, nous allons simplement utiliser un facteur directement dans le code. Ce n'est pas l'idéal en termes de conception, mais c'est suffisant pour le moment pour étudier le fonctionnement de cette représentation.

Définir une représentation n'est pas suffisante, il faut également définir les opérations arithmétiques de base. Pour l'addition et la soustraction, les opérateurs par défaut des entiers fonctionnent sans problème.

```
1.2 + 3.4 = 4.6  
120 + 340 = 460 // facteur 100
```

Pour la multiplication et la division, il faut faire une correction : il faut diviser le résultat par le facteur pour la multiplication et multiplier par le

facteur pour la division. Par exemple, pour la multiplication :

```
#include <iostream>

int main() {
    std::cout << (1.2 * 3.4) << std::endl;
    std::cout << (120 * 340) / 100 << std::endl;
}
```

affiche :

```
4.08
408
```

Remarque : pour la division, le résultat en utilisant des nombres réels peut avoir plus de chiffres après la virgule que les opérandes. En utilisant les nombres à virgule fixe, ces décimales supplémentaires seront perdues. Encore une fois, on doit faire un compromis avec la précision. Si cet arrondi est problématique, il faudra utiliser un facteur plus important ou utiliser des nombres réels.

**Exercice** : démontrer mathématiquement les propriétés précédentes pour la multiplication et la division.

## Facteurs non multiples de dix

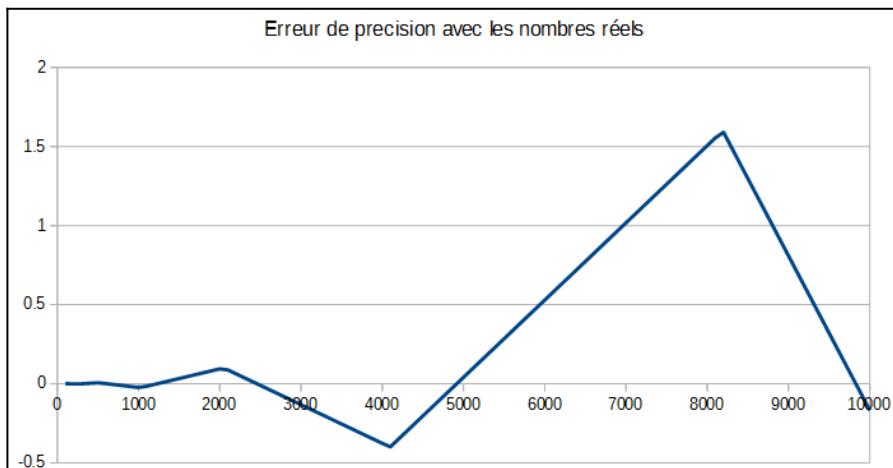
Supposons que vous souhaitez compter de 0 à 10 avec un pas de un tiers (cela signifie que vous incrémentez votre compteur de  $1/3$  à chaque fois). Pour cela, vous allez écrire un code similaire au code suivant (en pseudo-code, pas en C++) :

```
compteur = compteur + 1.0 / 3.0;
utiliser(compteur);
```

Remarque : un identifiant comme `compteur`, qui permet de se souvenir d'une valeur, est appelée une variable. Vous verrez cela dans les prochains cours.

Si vous répétez ce calcul, la différence entre la valeur calculée et la

valeur correcte attendue va progressivement augmenter. Le graphique suivant représente l'erreur de calcul en fonction de la valeur du compteur :



Note : pour mieux voir le phénomène, j'ai volontairement utilisé des nombres réels sur 32 bits (simple précision). Il existe des types de nombres réels plus précis, mais ce n'est pas important ici. Le problème existe avec tous les types de nombres réels, la seule différence est qu'il met plus de temps à apparaître.

Vous voyez que l'erreur devient assez vite importante. Selon vos besoins, vous pouvez accepter les valeurs en dessous de 1000, mais l'erreur devient trop importante au-delà.

La solution est assez simple (vous l'aurez deviné, vu que l'on est dans le chapitre sur les nombres à virgule fixe) : il suffit d'utiliser un compteur entier, qui sera incrémenté de 1 à chaque fois. La valeur du compteur réel sera obtenue en divisant par 3.

```
compteur = compteur + 1;  
utiliser(compteur / 3.0);
```

Avec un compteur réel, si on compte jusqu'à 10 000, il y aura 30 000 additions réelles, chaque addition ajoutant une erreur (presque infime)

de calcul. Ces erreurs vont s'accumuler, jusqu'au point de ne plus être négligeables.

Avec le compteur entier, comme chaque addition est exacte, les 30 000 additions ne produisent pas d'erreur de calcul. La seule opération réalisée sur des nombres réels (donc avec une erreur de précision) est la division par 3. L'erreur de calcul finale est donc l'erreur sur une seule opération réelle, ce qui reste négligeable.

Le code pour réaliser ce test est relativement simple. Cependant, vous imaginez bien qu'il n'est pas possible de faire cela en écrivant 30 000 fois la ligne de code C++ pour faire l'addition. Pour cela, on va utiliser une boucle qui sera vue par la suite. L'écriture de ce code sera proposée comme exercice.

## Lecture complémentaire

Voir l'article sur Wikipedia : [Fixed-point arithmetic](#).

[Chapitre précédent](#) [Sommaire principal](#) [Chapitre suivant](#)

Cours, C++

syntaxe full compile time, risque de confusion à voir tout de suite les 2 syntaxes aussi en détails

# Les nombres rationnels

## Rappel sur la représentation des nombres rationnels

Les nombres rationnels sont des nombres qui s'écrivent sous la forme d'une division de deux nombres entiers : un numérateur divisé par un dénominateur non nul.

```
$\$ \text{nombre rationnel} = \frac{\text{numérateur}}{\text{dénominateur}} \$\$
```

Comme pour les nombres complexes, le but de ce chapitre n'est pas d'entrer dans les détails mathématiques des nombres rationnels, mais de découvrir les outils fournis par le C++ pour les manipuler. Pour les détails, vous pouvez consulter la page de Wikipédia : [Nombre rationnel](#).

L'ensemble des nombres rationnels  $\mathbb{Q}$  est inclus dans l'ensemble des nombres réels  $\mathbb{R}$ , il est donc classique d'écrire un nombre rationnel sous une forme décimale limitée. Par exemple, le nombre  $\frac{1}{3}$  pourra être écrit :  $0,33333\dots$ . Cependant, cette représentation est problématique, puisque inexakte (il faudrait écrire une infinité de chiffre 3) : il n'existe de représentation décimale exacte pour les nombres rationnels.

Sur un ordinateur, la situation est encore plus problématique. Pour rappel, les nombres réels sont représentés en mémoire par un nombre fini d'octets, il n'est donc pas possible de représenter tous les nombres réels possibles, mais en général uniquement une valeur approchées (Il est donc tout à fait possible d'avoir deux nombres réels mathématiquement différents, mais qui auront la même représentation en mémoire et seront donc considérés égaux dans un programme C++).

Cela fait donc deux raisons qui limitent l'exactitude des calculs avec des nombres réels sur un ordinateur. Lorsque l'on réalise des calculs scientifiques, il est absolument nécessaire de prendre en compte ces erreurs.

Vous avez vu dans le chapitre précédent sur les nombres à virgule fixe une méthode pour représenter un nombre réels en utilisant un nombre entier et un diviseur fixe. Les nombres rationnels sont une généralisation de cette approche, mais au lieu d'utiliser un diviseur fixé dans le code, le diviseur pourra varier pour chaque nombre rationnel. Il ne faudra donc plus manipuler qu'un seul nombre, mais deux.

Remarque : n'oubliez pas que même si les nombres entiers n'ont pas de problème d'arrondi sur un ordinateur, ils sont quand même limités : ils ont une valeur maximale et une valeur minimale. Si vous utiliser des nombres trop grand ou trop petit, vos calculs seront faux.

## Créer un nombre rationnel avec `std::ratio`

Si vous vous souvenez du chapitre sur les nombres complexes, vous avez déjà manipulé en C++ une classe (`std::complex`) qui permet de manipuler deux nombres réels. Vous pouvez donc imaginer que la classe `std::ratio` sera similaire à `std::complex` et que l'étude de cette classe n'apporte pas grand chose.

En fait, ce n'est pas du tout le cas !

Vous avez vu dans le chapitre [Programme C++ minimal](#) qu'il y avait deux étapes pour obtenir le résultat d'un programme : une première phase de compilation (*compile-time*), qui permet de générer un programme à partir du code source C++, et une seconde phase d'exécution (*runtime*), qui exécute le programme. Cette distinction est intéressante, puisque la phase de compilation sera généralement réalisée une seule fois, pour ensuite exécuter plusieurs fois le programme. Donc tout ce qui est fait lors de cette première étape sera du temps de gagné sur l'exécution du programme.

## Optimisation à la compilation

En règle générale, le compilateur essayera au mieux d'optimiser les calculs à la compilation. Par exemple :

```
cout << (2+3) << endl;
```

Le compilateur peut calculer directement l'opération  $(2+3)$ , il va donc remplacer le code par le résultat et compilera en fait (mais vous ne le verrez pas, sauf à aller lire les fichiers binaires créés par le compilateur) :

```
cout << 5 << endl;
```

C'est très intéressant, puisque tout ce qui est fait lors de l'étape de compilation ne sera fait qu'une seule fois et l'application sera un peu plus rapide (mais cela ne sera pas forcément perceptible par les utilisateurs. Dans l'exemple avec l'addition qui est remplacé par sa valeur, la différence de performances est de l'ordre de la nanoseconde probablement, donc non perceptible).

La classe `std::ratio` est un cas particulier, puisqu'elle ne peut être utilisée que lors de la compilation. C'est une forme de limitation, puisque cela n'est pas utilisable tout le temps. Mais dans certains cas, il est préférable d'avoir une solution qui est performante et qui ne fonctionne qu'à la compilation ou qu'à l'exécution, plutôt que d'avoir une solution plus généraliste, mais moins performante.

Et c'est bien la distinction majeure entre `std::complex` et `std::ratio`. La première permet de réaliser des calculs lors de l'exécution, alors que la seconde travaille uniquement lors de la compilation. (Il est possible de créer une classe en C++ permettant de manipuler des nombres rationnels lors de l'exécution, vous ferez cela en exercice. Mais ce n'est pas l'approche utilisée par `std::ratio`).

La conséquence est que la syntaxe pour utiliser `std::ratio` est totalement différente de celle de `std::complex`. En particulier, il n'est pas possible d'utiliser les opérateurs que vous avez déjà utilisé pour les nombres et `std::complex` (comme `+` ou `==`).

Notez bien la distinction faite entre **l'opération** (calculer une addition, une soustraction, etc.) et les **opérateurs** (+, -, etc.).

En général, une opération représente un concept, qui est défini de façon unique, souvent par une définition mathématique (comme c'est le cas par exemple ici avec l'addition entre deux nombres rationnels).

Au contraire, un opérateur est un moyen dans le code C++ de réaliser cette opération. Une opération peut être définie pour une classe, sans que l'opérateur habituel correspondant ne soit défini (comme c'est le cas ici avec `std::ratio`, qui permet de calculer une addition, mais sans utiliser l'opérateur `+`). Il est également possible d'avoir plusieurs syntaxes permettant de réaliser une opération (par exemple que l'on puisse utiliser `+` et `std::add` pour calculer une addition).

Il est important que les fonctionnalités proposées par une classe soient cohérentes et quelque soit la méthode utilisée pour calculer une addition par exemple, le résultat soit toujours le même.

La classe `std::ratio` utilise la notation avec chevrons, que vous avez déjà rencontré rapidement. Mais pas d'inquiétude, la syntaxe est simple, il suffit d'écrire : `std::ratio<numérateur, dénominateur>`. Pour écrire la fraction  $\frac{1}{3}$  par exemple, il faudra donc écrire :

```
#include <ratio>

std::ratio<1, 3>
```

Il existe un certain nombre de `std::ratio` prédéfinie dans la norme C++. La liste complète est donnée dans la page de documentation : [std::ratio](#). Ces valeurs correspondent aux préfixes du système international d'unités (voir [Wikipédia](#) pour les détails). Par exemple :

- `nano = std::ratio<1, 1000000000>;`
- `micro = std::ratio<1, 1000000>;`
- `milli = std::ratio<1, 1000>;`
- `kilo = std::ratio<1000, 1>;`
- `mega = std::ratio<1000000, 1>;`
- `giga = std::ratio<1000000000, 1>;`

## Exercices

- Ecrire les fractions : 2/3, 1/2, 3/3.

## Afficher un std::ratio

Si vous compilez le code précédent, vous obtiendrez l'avertissement suivant :

```
main.cpp:4:5: warning: declaration does not declare anything
[-Wmissing-declarations]
    std::ratio<1, 3>;
    ^~~~~~
1 warning generated.
```

Ce message indique en fait que la ligne contenant `std::ratio` ne fait rien. En effet, c'est le cas, le code déclare simplement un `std::ratio`, et ne fait rien avec.

Essayons d'afficher un `std::ratio` avec `std::cout` :

```
#include <iostream>
#include <ratio>

int main() {
    std::cout << std::ratio<1, 3> << std::endl;
}
```

Malheureusement, ce code ne fonctionne pas directement et produit une erreur :

```
main.cpp:5:35: error: expected '(' for function-style cast
or type construction
    std::cout << std::ratio<1, 3> << std::endl;
                           ^~~~~~
1 error generated.
```

Le message est un peu plus complexe à comprendre, mais au final, cela veut dire qu'il ne comprend pas cette syntaxe. La raison est en fait très

simple : la classe `std::ratio` ne contient pas de fonctionnalités pour être affichée. Il faut récupérer directement les valeurs du numérateur et du dénominateur et les afficher.

Pour cela, il faut utiliser `num` et `den` avec la syntaxe suivante :

```
std::ratio<1, 3>::num // numérateur  
std::ratio<1, 3>::den // dénominateur
```

Au final, pour afficher un `std::ratio`, il faut donc écrire :

main.cpp

```
#include <iostream>  
#include <ratio>  
  
int main() {  
    std::cout << std::ratio<1, 3>::num << std::endl;  
    std::cout << std::ratio<1, 3>::den << std::endl;  
}
```

affiche :

```
1  
3
```

Une fraction est valide pour n'importe quelle paire d'entiers, avec un dénominateur non nul. Si on essaie de créer un `std::ratio` avec un dénominateur nul, on obtient une erreur assez explicite “denominator cannot be zero” :

main.cpp

```
#include <iostream>  
#include <ratio>  
  
int main() {  
    std::cout << std::ratio<1, 0>::num << std::endl;  
}
```

affiche :

```
In file included from main.cpp:2:  
/usr/local/bin/.../lib/gcc/x86_64-unknown-linux-gnu/5.2.0/.../  
./.../include/c++/5.2.0/ratio:265:7: error:  
static_assert failed "denominator cannot be zero"  
    static_assert(_Den != 0, "denominator cannot be  
zero");  
    ^~~~~~  
main.cpp:5:23: note: in instantiation of template class  
'std::ratio<1, 0>' requested here  
    std::cout << std::ratio<1, 0>::num << std::endl;  
    ^  
1 error generated.
```

Un nombre rationnel peut être représenté par une infinité de paire d'entiers, en multipliant le numérateur et le dénominateur par un même nombre non nul. Par exemple, toutes les fractions suivantes sont des représentations du même nombre rationnel :

$$\frac{2}{3} = \frac{4}{6} = \frac{6}{9} = \frac{12}{18} = \frac{22}{33} \dots$$

La représentation utilisant un numérateur et un dénominateur qui ne possèdent pas de diviseurs commun est appelée forme standardisée. Par exemple, “4/6” n'est pas une forme standardisée, puisque “4” et “6” ont un diviseur commun (on peut diviser 4 et 6 par 2). Par contre, “2/3” est une forme standardisée. Pour chaque nombre rationnel, il existe une et une seule forme standardisée.

La classe `std::ratio` simplifie les fractions et utilise la forme standardisée. Lorsque vous utiliser `num` et `den`, cela correspond au numérateur et dénominateur de la forme standardisée.

```
#include <iostream>  
#include <ratio>  
  
int main() {  
    std::cout << std::ratio<2, 3>::num << " / " <<  
    std::ratio<2, 3>::den << std::endl;  
    std::cout << std::ratio<4, 6>::num << " / " <<  
    std::ratio<4, 6>::den << std::endl;
```

```
}
```

affiche

```
2 / 3  
2 / 3
```

## Les opérateurs arithmétiques de std::ratio

Les quatre opérations de base sont fournies avec la classe `std::ratio` : addition, soustraction, multiplication et division. Cependant, il n'est pas possible ici d'utiliser les opérateurs habituels `+`, `-`, `*` et `/` (pour des raisons techniques concernant la syntaxe de ces opérateurs, mais ce n'est pas important pour le moment). Ces opérations sont donc implémentées dans des fonctionnalités :

- `ratio_add` pour l'addition ;
- `ratio_subtract` pour la soustraction ;
- `ratio_multiply` pour la multiplication ;
- `ratio_divide` pour la division.

Ces fonctionnalités utilisent aussi la syntaxe avec chevrons, il faut donc écrire :

```
std::ratio_add<première fraction, seconde fraction>
```

En remplaçant “première fraction” et “seconde fraction” par des appels à `std::ratio`. Donc, concrètement, si on veut additionner par exemple 2/3 et 3/4, il faudra écrire :

```
main.cpp
```

```
#include <iostream>  
#include <ratio>  
  
int main() {  
    std::cout << std::ratio_add<std::ratio<2, 3>, std::ratio<3, 4>>>::num << " / ";  
    std::cout << std::ratio_add<std::ratio<2, 3>, std::ratio<3, 4>>>::denom << std::endl;
```

```
<3, 4>>::den << std::endl;  
}
```

affiche :

17 / 12

Vous réalisez sans doute l'intérêt d'utiliser les opérateurs habituels `+`, `-`, `*` et `/` avec ce simple code. Même si le résultat est le même, l'utilisation des opérateurs habituels permet d'avoir une syntaxe plus simple et plus lisible.

Pour terminer, la classe `std::ratio` propose également les opérations de comparaison habituels, également avec des noms spécifiques au lieu des opérateurs :

- égalité = `ratio_equal` ;
- différence = `ratio_not_equal` ;
- infériorité = `ratio_less` ;
- infériorité ou égalité = `ratio_less_equal` ;
- supériorité = `ratio_greater` ;
- supériorité ou égalité = `ratio_greater_equal`.

Le résultat booléen est obtenu en utilisant `value` :

main.cpp

```
#include <iostream>  
#include <ratio>  
  
int main() {  
    std::cout << "2/3 == 3/4 ? " << std::boolalpha;  
    std::cout << std::ratio_equal<std::ratio<2, 3>, std::ratio<3, 4>>::value << std::endl;  
  
    std::cout << "2/3 == 4/6 ? " << std::boolalpha;  
    std::cout << std::ratio_equal<std::ratio<2, 3>, std::ratio<4, 6>>::value << std::endl;  
}
```

affiche :

```
2/3 == 3/4 ? false  
2/3 == 4/6 ? true
```

## Exercices

- écrivez les opérations suivantes :  $2/3+3/4$ ,  $2/3-3/4$ ,  $2/3*3/4$ ,  $(2/3) / (3/4)$ .

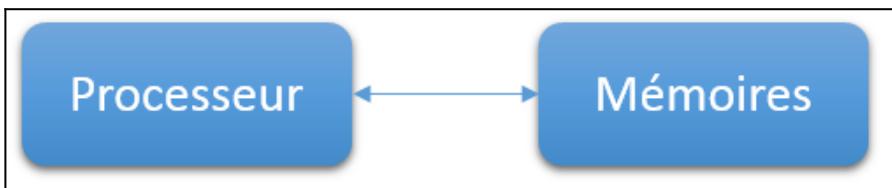
[Chapitre précédent](#) [Sommaire principal](#) [Chapitre suivant](#)

# Utiliser la mémoire avec les variables

## La mémoire et les variables

Très schématiquement, un ordinateur peut être décomposé en deux éléments :

- le processeur, qui réalise les calculs et les opérations logiques ;
- les mémoires, qui contiennent les informations (programmes à exécuter, données à traiter, résultats des calculs).



On distingue les mémoires de stockage sur le long terme (comme les disques durs) et les mémoires de travail à court terme (mémoire vive ou RAM, *Random Access Memory*). Cette dernière est utilisée pour conserver les données et les résultats des calculs des programmes.

En C++, les données sont manipulées en utilisant des variables. Ces variables peuvent être utilisées pour réaliser des calculs et diverses opérations logiques. Chaque variable dans un programme est désignée par un nom unique, appelé identifiant.

## Créer une variable

Pour commencer, voyons un exemple de code utilisant une variable :

main.cpp

```
#include <iostream>

int main() {
    int i { 123 };           // création de i
    std::cout << i << std::endl; // utilisation de i
}
```

affiche :

123

Ce code permet de créer une variable appelée `i`, qui peut contenir un nombre entier (`int` correspond à “integer”, qui signifie “entier”) et qui est initialisée avec la valeur `123`. Cette variable `i` est ensuite affichée en utilisant `std::cout`.

Lors de l'exécution de ce programme, `std::cout` ne va pas afficher le caractère `i`, mais la valeur contenue dans la variable `i`.

Faites bien attention à la syntaxe, même si un nom de variable est composé de caractères, ce n'est pas une littérale caractère ou une littérale chaîne de caractères :

```
std::cout << i << std::endl; // variable i
std::cout << 'i' << std::endl; // caractère 'i', entre
guillemets simples
std::cout << "i" << std::endl; // chaîne de caractères "i",
entre guillemets doubles
```

En pratique, la confusion est rare, la majorité des éditeurs de code utilisent des couleurs différentes pour les variables et les littérales.

La syntaxe générale pour créer une variable peut être résumée par la syntaxe suivante :

TYPE IDENTIFIANT { VALEUR };

Pour créer une variable, vous devez donc donner plusieurs informations, dans l'ordre :

- un **type** (par exemple `int` dans le code précédent) ;
- un **identifiant** (par exemple `i` dans le code précédent) ;
- une **valeur** (par exemple `123` dans le code précédent).

## Vocabulaire

- On **déclare** un identifiant.
- On **définit** une variable avec un type et un identifiant.
- On **initialise** une variable avec une valeur.

Vous pouvez créer autant de variables que vous le souhaitez dans vos programmes (en fonction des capacités de votre ordinateur). Mais même un ordinateur de bureau basique de nos jours peut contenir sans problème plusieurs milliards d'entiers en mémoire) :

main.cpp

```
#include <iostream>

int main() {
    int x { 123 };
    int y { 456 };
    int z { 789 };
    std::cout << x << std::endl;
    std::cout << y << std::endl;
    std::cout << z << std::endl;
}
```

affiche :

```
123
456
789
```

Nous allons voir en détail chaque élément de la définition et l'initialisation d'une variable.

## Syntaxes alternatives

Il existe en réalité plusieurs syntaxes possibles pour créer une variable. Voici quelques exemples :

```
int x;           // (1)
int x = 123;    // (2)
int x(123);    // (3)
auto x = 123;  // (4)
```

- (1) permet de créer une variable sans l'initialiser. Cette syntaxe est moins sûre que la syntaxe avec initialisation et ne sera pas utilisée dans ce cours.
- (2) et (3) sont des anciennes syntaxes qui sont encore très utilisées, mais n'apportent rien par rapport à la syntaxe utilisée dans ce cours (au contraire, dans certains cas, elles peuvent être ambiguës).
- (4) est appelée *inférence de type* et sera étudiée dans le prochain chapitre.

Vous rencontrez probablement ce type de syntaxe dans des codes existants, par exemple dans des tutoriels en ligne ou dans des livres. Ce cours se focalise sur les syntaxes recommandées en C++ moderne, ces syntaxes ne seront donc pas détaillées par la suite. Mais vous apprendrez sans problème ces syntaxes dans les exercices d'apprentissage que vous réaliserez.

Attention, il existe une variante de ces syntaxes, qui peut sembler correcte :

```
int x();
```

Mais cette syntaxe ne permet pas du tout de déclarer une variable (cela déclare une fonction). On voit régulièrement des débutants reproduire cette erreur.

## Modifier la valeur d'une variable

L'intérêt d'une variable est que vous allez pouvoir la réutiliser dans des expressions. A chaque fois qu'une expression contenant une variable est évaluée, la variable est remplacé par sa valeur lors du calcul.

main.cpp

```
#include <iostream>

int main() {
    int x { 123 };
    int y { 456 };
    std::cout << x * 2 << std::endl; // affiche le résultat
    du calcul 123 * 2
    std::cout << x + y << std::endl; // affiche le résultat
    du calcul 123 + 456
}
```

affiche :

```
246
579
```

Une expression peut également être utilisée pour initialiser une autre variable.

main.cpp

```
#include <iostream>

int main() {
    int x { 123 };
    int y { x * 2 };
    std::cout << y << std::endl; // affiche le résultat du
    calcul 123 * 2
}
```

affiche :

```
246
```

Une variable permet donc de retenir le résultat d'un calcul complexe, qui serait pénible de devoir réécrire plusieurs fois.

## Modifier une variable

Une variable est définie par un type, un identifiant et une valeur. Même si pour être rigoureux, il faut dire "modifier la valeur d'une variable", on simplifie souvent en disant "modifier une variable". Il n'y a pas de confusion possible, puisque le type et l'identifiant d'une variable ne peuvent être définis que lors de la création et ne plus être modifiés ensuite.

Il est également possible de modifier la valeur d'une variable, en utilisant l'opérateur d'affectation `=`. La syntaxe est la suivante :

```
IDENTIFIANT = VALEUR;
```

En pratique, cela donne :

```
main.cpp
#include <iostream>

int main() {
    int x { 123 };
    std::cout << x << std::endl;
    x = 456;
    std::cout << x << std::endl;
}
```

affiche :

```
123
456
```

## Confusion possible

Attention à ne pas confondre l'opérateur d'affectation pour modifier une variable `=` avec l'opérateur de comparaison d'égalité `==`.

De plus, dans les syntaxes alternatives, il est possible d'écrire le code suivant pour créer une variable.

```
int x = 123; // initialisation  
x = 456; // affectation
```

Notez la différence : une initialisation contient le type puis l'identifiant d'une variable, une affectation contient uniquement l'identifiant.

Pour éviter la confusion, il est recommandé d'utiliser la syntaxe avec des accolades pour l'initialisation et celle avec `=` pour l'affectation.

Dans de nombreux cas, vous n'aurez pas besoin de modifier la valeur d'une variable. Dans ce cas, on parle de constante. Pour indiquer cela dans le code, vous pouvez utiliser le mot-clé `const` (*constant*) devant le type de la variable lors de l'initialisation. De plus, cela permet au compilateur de vérifier que vous ne modifiez effectivement pas cette variable et de réaliser certaines optimisations.

```
main.cpp  
-----  
#include <iostream>  
  
int main() {  
    const int x { 123 };  
    x = 456; // erreur  
}
```

affiche :

```
main.cpp: In function 'int main()':  
main.cpp:5:7: error: assignment of read-only variable 'x'  
    x = 456; // erreur  
        ^
```

qui peut se traduire par “affectation sur une variable en lecture seule”.

Il est important d'utiliser `const` aussi souvent que possible, c'est-à-dire à chaque fois que vous ne modifiez pas une variable. Dans la suite de ce cours, nous utiliserons systématiquement `const` dans les codes d'exemple.

## Position de `const`

Le mot-clé `const` peut se placer avant ou après le type. Selon sa position, cela peut changer l'interprétation de ce qui est constant et de ce qui ne l'est pas. Mais dans les cas simples présentés ici, la position ne change rien à la signification de `const`. Les codes d'exemple utiliseront indifféremment les deux écritures.

```
const int i { 123 };
int const j { 456 };
```

La position de `const` sera importante dans certains cas, comme par exemple les pointeurs, qui seront vus à la fin de ce cours.

## Le type d'une variable

Vous avez déjà rencontré la notion de type dans les chapitres précédents :

```
2      // littérale entière
2.0    // littérale réelle
'2'    // littérale caractère
"2"   // littérale chaîne
```

Chaque littérale précédente possède un type défini ("entier", "réel", "caractère", "chaîne", mais il y en a beaucoup d'autres). C'est également le cas avec les variables, elles possèdent toutes un type défini, qui ne peut pas être changé (seule la valeur qu'elles contiennent peut changer).

En C++, les types de base s'écrivent avec des mots-clés définis dans le langage. Vous avez vu que le type `int` correspond aux entiers. Il existe beaucoup de types définis en C++ (et il est possible de définir ses propres types, il peut donc potentiellement exister une infinité de types différents), mais retenez pour le moment les types correspondants aux littérales que vous avez déjà manipulées :

- `int` (abréviation de *integer*, "entier" français) correspond à un nombre entier ;
- `double` correspond à un nombre réel (vous verrez par la suite

- pourquoi le C++ utilise ce terme) ;
- `std::string` correspond aux chaînes de caractères ;
- `char` correspond à un caractère ;
- `bool` correspond aux booléens.

Il existe également des mots-clés permettant de modifier un type de base. Par exemple, `signed` et `unsigned` permettent respectivement de spécifier un type **entier** signée (qui accepte des valeurs négatives et positives) et non-signée (qui acceptent uniquement des valeurs positives).

```
const int i { 123 };           // signed par défaut
const signed int j { 123 };    // signed explicite
const unsigned int k { 123 };   // unsigned
const unsigned int l { -123 }; // erreur, littérale signed
dans une variable unsigned
```

Pour rappel, voici comment s'écrivent les littérales correspondant à chaque type :

- pour un `int` : par exemple `123` ou `456` ;
- pour un `double` : par exemple `123.456` ou `123.456e789` ;
- pour un `std::string` : par exemple `"hello, world!"` ou `"bonjour!"` ;
- pour un `char` : par exemple `'a'` ou `'z'` ;
- pour un `bool` : uniquement `true` ou `false`.

## Conversion de types

Essayons de voir ce qui se passe si on utilise un type de littérale différent du type de variable (conversion).

main.cpp

```
int main() {
    int a { 1 };    // [1]
    int b { 1.2 }; // [2]
```

```
    int c { '1' }; // [3]
    int d { "1" }; // [4]
}
```

Ce code va produire les erreurs suivantes :

```
main.cpp:3:13: error: type 'double' cannot be narrowed to
'int' in initializer list [-Wc++11-narrowing]
    int b { 1.2 }; // [2]
               ^~~
main.cpp:3:13: note: insert an explicit cast to silence this
issue
    int b { 1.2 }; // [2]
               ^~~
                  static_cast<int>()
main.cpp:3:13: warning: implicit conversion from 'double' to
'int' changes value from 1.2 to 1 [-Wliteral-
conversion]
    int b { 1.2 }; // [2]
               ~ ^~~
main.cpp:5:13: error: cannot initialize a variable of type
'int' with an lvalue of type 'const char [2]'
    int d { "1" }; // [4]
               ^~~
1 warning and 2 errors generated.
```

Prenons chaque ligne en détail et les erreurs produites.

## Pas de conversion

La ligne [1] initialise une variable de type `int` à partir d'une littérale de type `int`. Dans ce cas, pas de problème, les types correspondent parfaitement.

## Conversion avec arrondi

Le ligne [2] produit plusieurs messages.

Un message d'erreur “type 'double' cannot be narrowed to 'int'” (“le type 'double' ne peut pas être restreint en type 'int'”) indique que la conversion de types peut produire une perte d'information, c'est à dire

que le type `double` peut contenir des valeurs que le type `int` ne peut pas contenir.

Une note permet d'aider le développeur à corriger ce problème : “insert an explicit cast to silence this issue” (“insérer une conversion explicite pour faire taire ce problème”).

Le message suivant est un avertissement : “implicit conversion (...) changes value from 1.2 to 1” (“la conversion implicite change la valeur 1.2 en 1”). Sans surprise, puisque les types ne sont pas directement convertibles sans perte potentielle d'information, les valeurs doivent être arrondies (dans ce cas “1.2” en “1”).

## Conversion implicite

La ligne [3] est beaucoup plus surprenante. Elle ne produit pas de message d'erreur ! Cependant, si on affiche la valeur de la variable `c`, le résultat est encore plus surprenant.

```
main.cpp
#include <iostream>

int main() {
    int c { '1' }; // [3]
    std::cout << c << std::endl;
}
```

affiche :

49

En fait, pour des raisons historiques, le type `char`, qui représente un caractère, est considéré comme un type entier et peut donc être converti automatiquement par le compilateur (conversion implicite). La valeur 49 correspond au caractère `1` dans la norme [ASCII](#).

Généralement, la conversion de `char` en `int` ne posera pas de problème, mais dans d'autres cas, ce type de conversion implicite peut réellement produire des comportements non prévus par le développeur et être assez difficile à identifier et corriger.

## Conversion impossible

La ligne [4] produit un message d'erreur plus simple : "cannot initialize a variable of type 'int' with an lvalue of type 'const char [2]'" (impossible d'initialiser une variable de type `int` avec une lvalue de type `const char [2]`). Le compilateur ne sait pas convertir une chaîne de caractères en entier et le signale.

Cela peut sembler ennuyeux que le compilateur bloque le processus, on aimerait parfois qu'il se débrouille pour trouver une solution et réussisse toujours à créer le programme. Mais il faut bien comprendre que c'est en fait une aide, pas une punition. Il est préférable que le compilateur dise "je ne sais pas, aide moi", plutôt que suivre un comportement que le développeur n'a pas prévu.

## Typage fort

Avoir un contrôle sur les types par le compilateur permet de garantir leur utilisation correcte (*type safety*). Plus la prise en compte des types est importante, plus vous aurez de garantie sur le code. Ce typage fort est une des forces du C++ et il est intéressant de permettre au compilateur de faire un maximum de vérifications.

Dans tous les cas, il est important d'accorder une attention particulière aux types des variables et des données dans vos codes C++.

## L'identifiant

L'identifiant d'une variable est le nom de cette variable. Vous pouvez utiliser cet identifiant dans vos codes en remplacement d'une valeur dans un calcul par exemple. Si vous utilisez plusieurs variables, chaque identifiant doit être unique, vous ne pouvez pas définir plusieurs variables utilisant le même nom :

```
#include <iostream>

int main() {
    int const x { 123 }; // x correspond à un entier
```

```
int const x { 456 }; // erreur : l'identifiant x est
déjà utilisé
}
```

affiche :

```
main.cpp: In function 'int main()':
main.cpp:6:15: error: redeclaration of 'const int x'
    int const x { 456 }; // erreur : l'identifiant x est
déjà utilisé
          ^
main.cpp:5:15: note: 'const int x' previously declared here
    int const x { 123 }; // x correspond à un entier
          ^
```

Pour écrire un identifiant, vous pouvez utiliser les caractères alphanumériques minuscules et majuscules (a à z, A à Z et 0 à 9) et le tiret bas   (*underscore*, correspond à la touche 8 sur un clavier français). De plus, un identifiant doit obligatoirement commencer par une lettre.

Par exemple, les noms suivants sont des identifiants valides :

- x ;
- y ;
- unevariable ;
- uneVariable ;
- une\_variable ;
- UnEvArIaBle.

En revanche, les identifiants suivants ne sont pas valides ou sont déconseillés :

- \_une\_variable : commence par un tiret bas ;
- 123variable : commence par un chiffre ;
- variable\_réelle : contient un caractère interdit (é).

Un identifiant qui commence par un tiret bas, comme \_une\_variable, est en fait autorisé par la norme C++, mais son

usage est restreint. Pour éviter les problèmes, il est d'usage de ne pas utiliser ce type d'identifiant.

Comme vous le voyez, le langage C++ laisse de grandes libertés pour choisir un identifiant... ce qui peut poser des problèmes. Exemple de mauvais identifiant :

- `jjfnfsfkjgukzv` : ne veut rien dire, n'apporte pas d'information sur le rôle de cette variable ;
- `une_variable` : trop générique, ne dit pas quel est le rôle de cette variable ;
- `variable1, variable2`, etc. : trop générique aussi ;
- `UnEvAriAble` : le mélange de minuscule et majuscules rend ce nom peu lisible ;
- `une_variable_qui_contient_le_resultat_du_premier_calcul` : trop long.

Dans les projets réalisés en équipe, une bonne pratique est de définir des règles ("bonnes pratiques de codage") qui s'imposent à tous les développeurs du projet, pour faciliter la lecture du code. Le but est d'avoir des noms homogènes, simples et informatifs.

Il existe déjà des règles de bonnes pratiques de codage toutes faites, mais vous pouvez aussi utiliser vos propres règles. Les conventions de nommage les plus connues écrivent les noms en minuscules séparées par un tiret bas (`une_variable`, par exemple dans la bibliothèque standard ou Boost) ou en écrivant les noms avec des majuscule au début de chaque mot et sans séparateur (`uneVariable`, par exemple dans la bibliothèque Qt).

Le nom des variables participe à la qualité d'un code. Plus le nom d'une variable est informatif sur le rôle de cette variable, moins vous aurez de risque de mal utiliser cette variable.

## La valeur

Une variable contient obligatoirement une valeur. Il est possible de définir une variable sans l'initialiser, mais cette variable pourra alors

contenir une valeur aléatoire. Cependant, vous imaginez bien qu'un programme ne va pas forcément fonctionner correctement si certaines variables sont initialisées avec des valeurs non déterminées.

Une variable peut être initialisée avec une valeur par défaut (*value initialization*), avec une littérale (*direct initialization*) ou avec une expression (*copy initialization*).

- initialisation par défaut : `Type Identifiant {};`
- initialisation avec une littérale : `Type Identifiant { Valeur };`
- initialisation avec une expression : `Type Identifiant { Expression };`

Plus concrètement, avec du code :

main.cpp

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

int main() {
    // Initialisation par défaut
    const int      i_default {};
    const double   d_default {};
    const std::string s_default {};
    const char     c_default {};
    const bool     b_default {};

    // Initialisation avec une valeur
    const int      i_default { 123 };
    const double   d_default { 123.456 };
    const std::string s_default { "hello, world!" };
    const char     c_default { 'a' };
    const bool     b_default { true };

    // Initialisation avec une expression
    const int      i_default { 123 + 456 };
    const double   d_default { 12.34 + 56.78 };
    const std::string s_default { std::string{ "hello, " } +
"world!" };
    const char     c_default { 'a' };
}
```

```
    const bool      b_default { 123 > 456};  
}
```

Note : l'expression avec `std::string` est un peu plus complexe, du fait de certaines règles de manipulation des chaînes de caractères. Vous verrez cela plus en détail dans le chapitre sur les chaînes.

## Portée d'une variable

Une variable existe à partir du moment où elle est déclarée et jusqu'à la fin du bloc contenant sa déclaration. Cela s'appelle la portée de la variable. Utiliser une variable avant de l'avoir définie produit donc une erreur.

```
main.cpp  
#include <iostream>  
  
int main() {  
    std::cout << x << std::endl;  
    const int x { 123 };  
}
```

affiche :

```
main.cpp: In function 'int main()':  
main.cpp:4:18: error: 'x' was not declared in this scope  
        std::cout << x << std::endl;  
                  ^
```

Ce qui signifie "x n'est pas déclarée dans cette portée".

Il est également possible d'ajouter des blocs supplémentaires, pour limiter la portée des variables.

```
main.cpp  
#include <iostream>  
  
int main() {  
    const int x { 123 };           // début de portée de
```

```

x    //
{                                //
//      const int y { x + 456 };    //
// début de portée de y          std::cout << x << std::endl; // ok, x existe
//                                std::cout << y << std::endl; // //
// ok, y existe                  }
// fin de portée de y           std::cout << x << std::endl; // ok, x existe
//                                std::cout << y << std::endl; // //
// erreur, y est hors de portée }
}                                // fin de portée de x
//

```

Note : dans la partie sur les identifiants, il est dit qu'un identifiant devait être unique. En fait, la règle est plus précisément "il ne faut pas avoir deux identifiants identiques dans la même portée". Il est donc possible d'avoir plusieurs variables portant le même identifiant, si leur portée est différente.

### main.cpp

```

#include <iostream>

int main() {
{
    const int x { 123 };
    std::cout << x << std::endl;
}
{
    const int x { 456 };
    std::cout << x << std::endl;
}
}

```

Il faut bien comprendre ce qui se passe ici. Une première variable nommée `x` est créée dans le premier bloc et initialisée avec la valeur 123. Puis celle-ci est détruite à la fin du bloc et une nouvelle variable est créée.

Cette variable s'appelle aussi `x`, mais c'est bien une variable différente.

### Durée de vie (lifetime)

Une autre notion importante concernant les objets, c'est la durée de vie. Celle-ci est simplement le temps entre le moment où un objet est créé en mémoire et le moment où cet objet est libérée. Essayez d'utiliser un objet qui n'est pas disponible en mémoire va provoquer au mieux un crash du programme, voire pire. Dans tous les cas, cela sera un comportement indéfini (*Undefined Behavior*).

Il existe plusieurs types de variables (dynamique, statique, etc), mais pour le moment, vous n'avez vu que les variables locales, dans ce chapitre. Dans cette situation, la durée de vie peut être confondue avec la portée (ce n'est pas tout à fait vrai, mais vous verrez cela plus tard, dans le chapitre sur la Pile) :

**Un objet est créé en mémoire lorsqu'une variable locale est déclarée dans le code et il est détruit lorsque la variable sort de sa portée.**

On parle parfois de “variable automatique”, du fait que l'objet est détruit automatiquement. Mais cette appellation n'est pratiquement plus utilisée, on parle simplement de “variable locale”.

Notez bien la distinction entre le concept de “variable”, qui concerne le code avant compilation, et le concept “d'objet”, qui concerne la mémoire pendant l'exécution du programme.

[Chapitre précédent](#) [Sommaire principal](#) [Chapitre suivant](#)

# L'inférence de type

## Conversion implicite de types

Lorsque vous définissez une variable, vous pouvez l'initialiser avec une littérale de même type que le type de la variable ou avec un type différent. Si le type de la variable et de la littérale ne sont pas parfaitement identiques, le compilateur effectue une *conversion implicite* du type de la littérale vers le type de la variable. Lorsque cette conversion implicite ne pose pas de problème et que la littérale peut être convertie vers le type de la variable *sans perte d'information*, le compilateur ne va pas signaler cette conversion implicite.

```
main.cpp
#include <iostream>

int main() {
    int const i { 123 };
    float const c { 123 };
    std::cout << i << std::endl;
    std::cout << c << std::endl;
}
```

affiche :

```
123
123
```

En revanche, si la littérale ne peut pas être convertie vers le type de la variable sans perte d'information, le compilateur signalera cette conversion implicite (*narrowing*). Une perte d'information peut avoir lieu par exemple dans les conversions suivantes :

- une littérale de type réel dans une variable de type entier ;

- une littérale signée (qui peut être positive ou négative) dans une variable non signée (qui ne peut être que positive) ;
- une littérale trop grande dans une variable de type ne pouvant pas contenir cette valeur.

```
int i { 123.0 };           // double dans un int
unsigned int ui { -1 };   // signed dans un unsigned
char c { 123456789 };    // int dans un char
```

Le compilateur signalera une erreur ou un avertissement (selon les options de compilation) pour signaler la conversion implicite. Par exemple, dans le cas d'une conversion d'un `double` vers un `int` :

`main.cpp`

```
#include <iostream>

int main() {
    const int i { 123.456 };
    std::cout << i << std::endl;
}
```

affiche :

```
main.cpp:4:19: error: type 'double' cannot be narrowed to
'int' in initializer list [-Wc++11-narrowing]
    const int i { 123.456 };
                           ^
main.cpp:4:19: note: insert an explicit cast to silence this
issue
    const int i { 123.456 };
                           ^
                           static_cast<int>()
main.cpp:4:19: warning: implicit conversion from 'double' to
'int' changes value from 123.456 to
123 [-Wliteral-conversion]
    const int i { 123.456 };
                           ~ ^
1 warning and 1 error generated.
```

Le premier message donne le message d'erreur de conversion (« type

'double' cannot be narrowed to 'int' » signifie « le type 'double' ne peut pas être réduit dans 'int' »).

À la fin du message d'erreur, le compilateur indique l'option de compilation qui active la vérification des erreurs de conversion `-Wc++11-narrowing`. Vous avez déjà vu quelques options de compilation `-Wall -Wextra -pedantic`. Vous n'avez pas utilisé cette nouvelle option, mais pourtant le compilateur a signalé le problème. La raison est que les options de compilation que vous avez utilisées `-Wall -Wextra -pedantic` activent d'autres options de compilation, en particulier, `-Wc++11-narrowing`.

Le deuxième message propose de réaliser une conversion explicite au lieu d'une conversion implicite. La raison est que le compilateur ne peut deviner si la conversion est une erreur ou un choix volontaire du développeur. Pour lever cette ambiguïté, il demande donc au développeur d'écrire explicitement cette conversion. Vous verrez dans un prochain chapitre l'utilisation de `static_cast` pour ce cas d'utilisation.

Le dernier message d'avertissement prévient que du fait de la conversion implicite, la littérale 123.456 est arrondie et la variable est initialisée avec la valeur 123.

## Inférence de type

Utiliser des types différents pour une variable et une littérale lors de l'initialisation peut poser des problèmes. Il est donc logique d'appliquer (sauf cas particuliers) la bonne pratique qui consiste à utiliser le même type pour les deux. Mais dans ce cas, vous pouvez légitimement vous poser la question suivante : pourquoi devoir définir deux fois le type, au risque que cela pose problème ? Ne serait-il pas possible de le définir une seule fois ?

Cela est possible en C++, grâce à l'inférence de type, qui permet au compilateur de déduire automatiquement le type d'une variable à partir d'une expression. Il existe trois formes possibles pour définir une variable en C++ utilisant l'inférence de type, basées sur une même syntaxe :

```
<EXPRESSION> IDENTIFIANT = VALEUR;
```

Le type de la variable est remplacé par `<EXPRESSION>`, qui peut correspondre à l'un des trois mots-clés suivants :

- `auto` : déduit le type à partir de `VALEUR`, sans conserver les modificateurs de types (`const`, etc.) ;
- `decltype(auto)` : déduit le type à partir de `VALEUR`, en conservant les modificateurs de types ;
- `decltype(expression)` : déduit le type d'une expression.

Cette écriture présente de nombreux avantages par rapport à la définition d'une variable sans inférence. Premièrement, cela évite de devoir écrire le type. Avec des types simples, l'avantage est réduit, mais certains types complexes sont longs à écrire et source potentielle d'erreur lors de leur écriture. Vous verrez en particulier par la suite l'utilisation des itérateurs pour parcourir les collections, dont l'écriture est assez lourde.

Un deuxième avantage est d'avoir un code plus évolutif. Imaginez que vous réalisez plusieurs calculs numériques complexes. Si vous écrivez explicitement le type de toutes vos variables et que vous souhaitez ensuite utiliser un autre type, vous devrez modifier l'ensemble de votre code pour utiliser le nouveau type. En utilisant l'inférence, vous laissez le compilateur s'occuper de cette tâche.

Cependant, l'inférence de type n'est pas toujours utilisable, c'est pour cela qu'il faut connaître les deux syntaxes pour définir une variable.

## Le mot-clé `auto`

Commençons par détailler le mot-clé `auto`. Ce mot-clé indique au compilateur qu'il doit utiliser le type de `VALEUR` pour le type de la variable, sans conserver les modificateurs comme `const` (et les références, mais vous verrez cela dans le chapitre sur les fonctions). Par exemple :

main.cpp

```
int main() {
    const auto i = 123;      // 123 est une littérale de type
                            // int, donc auto déduit le type
                            // int.

                            // auto est précédé de const
                            // donc i est de type const int.

    const auto j = i;        // i est une variable de type
    const int,                // auto déduit le type int.
                            // auto est précédé de const
                            // donc j est de type const int.

    const auto x = 12.34;    // 12.34 est une littérale de type
                            // double, auto déduit le type
                            // double.

                            // auto est précédé de const
                            // donc x est de type const
                            // double

    const auto c = 'a';      // 'a' est une littérale de type
                            // char, auto déduit le type char.
                            // auto est précédé de const
                            // donc x est de type const char
}
```

## Le mot-clé decltype(auto)

La syntaxe avec `decltype(auto)` est similaire à celle avec `auto`, mais conserve les modificateurs de type comme `const`.

Par exemple, dans certains cas, vous souhaiterez définir une constante à partir d'une autre constante, dont la valeur n'est pas modifiée après initialisation. Et dans d'autres cas, vous souhaiterez créer une variable, initialisée avec la valeur d'une constante, que vous pourrez modifier.

Les deux syntaxes `auto` et `decltype(auto)` sont complémentaires et

vous permettent d'exprimer dans le code quelles sont vos intentions.

### main.cpp

```
auto i = ...;           // i récupère le type de  
l'expression, mais n'est pas const.  
                        // Vous souhaitez pourvoir modifier  
sa valeur.  
  
const auto j = ...;     // j est une constante, quel que  
soit le type de l'expression.  
  
decltype(auto) k = ...; // k sera exactement du même type  
que l'expression.
```

En pratique, pour le moment, ces différentes syntaxes peuvent vous sembler très proches. Lorsque vous déclarez une variable, vous savez si vous voulez qu'elle soit constante ou non, vous n'avez pas besoin de laisser le compilateur choisir s'il doit utiliser `const` ou pas. Vous n'aurez donc, dans un premier temps, besoin de n'utiliser que `auto` et `const`.

Cependant, n'oubliez pas qu'il existe d'autres modificateurs que `const`, que vous verrez par la suite (dans la partie de ce cours sur les fonctions). L'utilisation de `auto` et `decltype(auto)` prendra tout son sens à ce moment-là. Nous reviendrons sur ces syntaxes et leurs subtilités d'utilisation à ce moment-là.

Le tableau suivant résume les différentes syntaxes possibles et les types déduits correspondants (`T` représente un type fondamental, comme `int`, `double`, etc.)

Type	Déclaration	Type déduit
<code>T</code>	<code>auto</code>	<code>T</code>
<code>const T</code>	<code>auto</code>	<code>T</code>
<code>T</code>	<code>const auto</code>	<code>const T</code>
<code>const T</code>	<code>const auto</code>	<code>const T</code>
<code>T</code>	<code>decltype(auto)</code>	<code>T</code>
<code>const T</code>	<code>decltype(auto)</code>	<code>const T</code>

## Le mot-clé decltype

Alors que les mots-clés `auto` et `decltype(auto)` utilisent automatiquement l'expression qui se trouve à droite de l'opérateur d'affectation `=`, le mot-clé `decltype` permet d'écrire directement une expression qui sera évaluée pour déterminer le type. La syntaxe est la suivante :

```
decltype(EXPRESSION) IDENTIFIANT = VALEUR;
```

À la différence de `auto`, `decltype` conserve les modificateurs de type comme `const`.

main.cpp

```
int main() {
    const decltype(12) i = 34; // 12 est une littérale de
    // type int. // int, donc decltype déduit
    // const // donc i est de type const
    // int. // 34 n'est pas utilisé pour
    // évaluer le type de int.

    decltype(12) j = 34; // 12 est une littérale de
    // type int. // int, donc decltype déduit
    // le type int // et j est de type int.

    decltype(i) k = 34; // i est une variable de
    // type const int, // donc decltype déduit le
    // type const int // et k est de type const
    // int. // int.

    decltype(j) l = 34; // j est une variable de
```

```
type int,  
                                // donc decltype déduit le  
type int  
                                // et l est de type int.  
}
```

## Syntaxes alternatives

Dans le chapitre précédent, vous avez vu la syntaxe avec accolades pour initialiser une variable :

```
const int i { 123 };  
const int j {};
```

Cette syntaxe n'est pas utilisable directement avec `auto` (les accolades sont interprétées différemment dans ce cas), ce qui explique que la syntaxe avec affectation `=` est utilisée. Par contre, les accolades ne posent pas de problème avec `decltype`, vous pouvez donc utiliser la syntaxe sans problème :

```
const decltype(123) i { 123 };  
const decltype(i) j {};
```

Pour des raisons d'homogénéité des syntaxes, nous allons utiliser par défaut les accolades pour initialiser une variable sans inférence de type et l'opérateur d'affectation pour l'initialisation de variable avec inférence de type. Mais n'oubliez pas que d'autres syntaxes sont utilisables.

En particulier, pour initialiser une variable avec un type déduit par `decltype` et une valeur par défaut, on préférera la syntaxe avec accolades :

```
const decltype(i) j = 0; // n'a pas de sens si i n'est pas  
                        // un type entier  
const decltype(i) j {}; // syntaxe valide quel que soit le  
                        // type de i
```

## Erreurs de déduction des types

La déduction de type est une technique intéressante pour avoir un code plus évolutif, mais cela peut produire des surprises si les types déduits par le compilateur ne correspondent pas à ce que le développeur souhaitait. Dans des codes d'exemple simples comme présentés dans ce chapitre, il est facile de ne pas se tromper, mais dans des codes plus complexes, les erreurs arrivent très vite.

S'il fallait donner une règle pour éviter les problèmes, ce serait la suivante :

**Toujours s'assurer que le code est suffisamment explicite pour que l'on puisse comprendre les intentions de celui qui écrit le code. Si ce n'est pas le cas, il est préférable de ne pas utiliser l'inférence de type.**

Prenons quelques contre-exemples, dans lesquels le type de la variable n'est pas forcément explicitement donné par le contexte.

```
auto j = i;
```

La règle pour nommer une variable est que le nom doit être explicite, mais certains noms de variables sont tellement communs que leur utilisation ne pose pas de problème de compréhension. C'est en particulier le cas de `i`, `j`, `k`, qui sont utilisés en mathématiques et en programmation comme indices entiers. (La contrepartie est qu'il ne faut pas appeler une variable `i` si ce n'est pas un indice entier, sinon il y a un risque de confusion).

Donc ici, même si le contexte ne donne pas le type de `i`, par convention, ce sera un entier et de même pour `j`.

En cas d'erreur sur les types, dans le meilleur des cas, les opérations réalisées n'auront pas de sens et le compilateur produira un message d'erreur. Imaginons par exemple que vous souhaitez créer une variable entière, mais que vous utilisez une littérale chaîne de caractères :

```
main.cpp
```

```
#include <iostream>

int main() {
    const auto i = "hello";           // type entier ?
    std::cout << i * 10 << std::endl; // erreur
}
```

Bien sûr, le calcul n'a aucun sens et le compilateur signale qu'il y a un problème dans le code :

```
main.cpp:5:20: error: invalid operands to binary expression
('const char *' and 'int')
    std::cout << i * 10 << std::endl; // erreur
               ~ ^ ~~
1 error generated.
```

Le message d'erreur n'est pas forcément clair (et un compilateur C++ peut être très doué pour donner des messages d'erreur incompréhensibles). Il indique ici que le compilateur ne trouve pas d'opération entre un type `const char*` (une chaîne de caractères) et un type `int` (un entier), pas qu'il y a une erreur sur l'utilisation du mot-clé `auto` (comment le compilateur pourrait-il le savoir ?)

Mais la situation n'est pas forcément aussi simple. Si le code précédent est modifié pour utiliser l'opérateur `+`, le résultat est très différent :

```
main.cpp
#include <iostream>

int main() {
    const auto i = "hello";           // type entier ?
    std::cout << i + 10 << std::endl; // erreur
}
```

affiche par exemple (cela peut afficher n'importe quoi en fait) :

```
;0
```

Pour des raisons historiques, la chaîne est interprétée par le compilateur comme étant un entier dans ce cas, aucune erreur n'est détectée.

Comme vous l'avez déjà vu, ce code produit un comportement indéterminé (`Undefined Behavior`), ce qui est souvent très complexe à détecter et corriger. Il convient donc de faire très attention lors de l'utilisation de l'inférence de type.

<a href="#">Chapitre précédent</a>	<a href="#">Sommaire principal</a>	<a href="#">Chapitre suivant</a>
------------------------------------	------------------------------------	----------------------------------

# Manipuler les types

## Taille des variables en mémoire

Dans le chapitre [Logique binaire et calcul booléen](#), vous avez vu que lorsque vous affichez l'inverse d'un nombre binaire, la valeur affichée correspond à un nombre codé sur 32 bits (ou 64 bits selon le système). Pour rappel, 32 bits correspondent à 4 octets ou encore 8 chiffres hexadécimaux et 64 bits correspondent à 8 octets ou 16 chiffres hexadécimaux.

main.cpp

```
#include <iostream>

int main() {
    std::cout << std::hex << std::showbase;
    std::cout << ~0b1 << std::endl;
    std::cout << ~0b001 << std::endl;
    std::cout << ~0b00001 << std::endl;
}
```

affiche :

```
0xfffffffffe
0xfffffffffe
0xfffffffffe
```

## Quelques rappels

N'oubliez pas que dans les écritures précédentes, les préfixes `0b` et `0x` ne sont pas des chiffres, cela indique que les nombres affichés correspondent respectivement à une valeur écrite en binaire et une valeur écrite en hexadécimal.

Un autre point important à se souvenir est que la représentation d'un nombre dans un code peut être différente de la représentation de ce nombre en mémoire et de la représentation lors de l'affichage avec `std::cout`. Par exemple ici, les littérales sont écrites en binaire et l'affichage est en hexadécimal.

Le modificateur `std::hex` permet d'afficher les nombres en hexadécimal et le modificateur `std::showbase` permet d'afficher le préfixe `0x`.

Quel que soit le nombre de bits que vous utilisez pour écrire la littérale booléenne, la valeur inverse est toujours affichée sur 32 bits (dans cet exemple). La raison est que le compilateur crée une variable temporaire de 32 bits puis calcule l'inverse. Le résultat est donc toujours sur 32 bits.

Le nombre de bits utilisé pour représenter un nombre est important, puisque cela impacte le nombre de valeurs qui seront représentables. Prenons par exemple un nombre représenté par deux bits. Ce nombre pourra donc prendre quatre valeurs possibles :

```
00  
01  
10  
11
```

Si un nombre est représenté par quatre bits, il pourra alors prendre 16 valeurs possibles :

0000	0001	0010	0011
0100	0101	0110	0111
1000	1001	1010	1011
1100	1101	1110	1111

Pourquoi ne pas utiliser dans ce cas le nombre de bits le plus grand possible dans tous les cas ? Tout simplement parce que la mémoire est limitée. Et utiliser plus de bits que nécessaire diminuera la mémoire disponible et ralentira les programmes.

Dans ce cas, pourquoi un booléen est représenté par 32 bits et pas un simple bit ? Parce que les processeurs sont optimisés pour travailler avec certaines tailles, en général des multiples de 8, comme 8, 16, 32 ou 64.

bits. Dans l'exemple, 32 bits correspond au nombre de bits le plus naturel pour le processeur (et donc le plus efficace).

Il est possible de connaître le nombre de bits utilisé pour représenter un type ou une variable. Dans les deux cas, vous devez utiliser l'opérateur `sizeof`, qui prend en paramètre la variable ou le type. Par exemple, pour utiliser `sizeof` avec des types :

main.cpp

```
#include <iostream>

int main() {
    std::cout << "sizeof(int) = " << sizeof(int) << std::endl;
    std::cout << "sizeof(double) = " << sizeof(double) << std::endl;
    std::cout << "sizeof(bool) = " << sizeof(bool) << std::endl;
    std::cout << "sizeof(char) = " << sizeof(char) << std::endl;
}
```

Ce code affiche les tailles des types correspondants, en octets (pensez à multiplier par huit si vous souhaitez connaître le nombre de bits). La taille peut dépendre du système d'exploitation et du processeur.

```
sizeof(int) = 4
sizeof(double) = 8
sizeof(bool) = 1
sizeof(char) = 1
```

De la même façon, pour connaître la taille en mémoire d'une variable, vous pouvez écrire :

main.cpp

```
#include <iostream>

int main() {
    int const x { 123 };
    double const d { 12.34 };
```

```
bool const b { true };
char const c { 'a' };

std::cout << "sizeof(x) = " << sizeof(x) << std::endl;
std::cout << "sizeof(d) = " << sizeof(d) << std::endl;
std::cout << "sizeof(b) = " << sizeof(b) << std::endl;
std::cout << "sizeof(c) = " << sizeof(c) << std::endl;
}
```

Ce qui affichera (selon le contexte d'exécution) :

```
sizeof(x) = 4
sizeof(d) = 8
sizeof(b) = 1
sizeof(c) = 1
```

Pour les types non fondamentaux, en particulier pour les chaînes de caractères `std::string` (mais ça sera aussi le cas pour la majorité des classes de la bibliothèque standard que vous verrez), la valeur renournée par `sizeof` ne correspond pas au nombre d'éléments dans cette classe.

#### main.cpp

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

int main() {
    std::string const s1 { "hello, world!" };
    std::string const s2 { "Bonjour tout le monde !" };

    std::cout << "sizeof(s1) = " << sizeof(s1) << std::endl;
    std::cout << "sizeof(s2) = " << sizeof(s2) << std::endl;
}
```

affiche :

```
sizeof(s1) = 8
sizeof(s2) = 8
```

Pour comprendre pourquoi `sizeof` donne ce résultat, il faudra détailler le fonctionnement interne de la classe `string`. Vous verrez cela dans les chapitres sur la création de nouvelle classe. Pour connaître la taille de la

chaîne de caractères (c'est-à-dire le nombre de caractères), il faut utiliser la fonction membre `size` ;

main.cpp

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

int main() {
    std::string const s1 { "hello, world!" };
    std::string const s2 { "Bonjour tout le monde !" };

    std::cout << "s1.size() = " << s1.size() << std::endl;
    std::cout << "s2.size() = " << s2.size() << std::endl;
}
```

affiche :

```
s1.size() = 13
s2.size() = 23
```

Donc, pour résumer :

- avec un type fondamental (`int`, `double`, etc.), `sizeof` retourne la taille en mémoire ;
- avec un type complexe (par exemple les classes de la bibliothèque standard), `sizeof` ne retourne pas forcément la taille réellement occupée en mémoire par la classe.

## Les types entiers

## Les modificateurs de type

Dans la majorité des ordinateurs actuels, la mémoire disponible se compte en giga-octets et la taille mémoire des données ne sera pas critique (sauf dans des contextes particuliers, par exemple sur les systèmes embarqués ou dans les applications réalisant de nombreux

calculs numériques). Il sera alors possible, dans un grand nombre d'applications, d'utiliser les types par défaut présentés dans les chapitres précédents.

Dans d'autres situations, il pourra être intéressant d'optimiser la taille des données en fonction des besoins. En effet, si on regarde les valeurs retournées par `sizeof`, on peut remarquer qu'un booléen, qui peut être codé sur 1 bit, est en réalité codé sur 8 bits (1 octet). Donc un surcoût mémoire d'un facteur de 8. De même, si on souhaite créer une variable qui permet de compter de 0 à 3, on pourrait n'utiliser que 2 bits (qui pourraient alors prendre les valeurs : `0b00`, `0b01`, `0b10` et `0b11`). Une variable de type `int` prend 32 bits, soit un surcoût d'un facteur de 16.

Le C++ ne se limite pas aux types par défaut, vous avez beaucoup de liberté pour manipuler la mémoire. Il est par exemple possible de modifier la taille de la représentation en mémoire d'un type fondamental. De plus, pour les types entiers, il existe des types prédéfinis avec des tailles fixées.

La taille mémoire des types est paramétrable en utilisant des modificateurs de types, qui s'ajoutent au type de base `int`. Pour les entiers, les modificateurs disponibles sont : `short` ("court" en français), `long` et `long long`. Le modificateur de type se place devant le type qu'il modifie. Il est facile d'écrire un code pour vérifier que la taille des types est modifiée en conséquence :

```
main.cpp
#include <iostream>

int main() {
    std::cout << "sizeof(char) = " << sizeof(char) << std::endl;
    std::cout << "sizeof(short int) = " << sizeof(short int)
<< std::endl;
    std::cout << "sizeof(int) = " << sizeof(int) << std::endl;
    std::cout << "sizeof(long int) = " << sizeof(long int)
<< std::endl;
    std::cout << "sizeof(long long int) = " << sizeof(long
long int) << std::endl;
```

}

affiche :

```
sizeof(char) = 1
sizeof(short int) = 2
sizeof(int) = 4
sizeof(long int) = 8
sizeof(long long int) = 8
```

Vous pouvez remarquer que `long` et `long long int` ont la même taille en mémoire dans ce test. Mais il est possible que vous n'ayez pas les mêmes valeurs.

C'est une particularité des types du C++ : la norme ne définit pas une taille fixe pour ces types, simplement des contraintes par rapport à leur taille respective. Ces règles sont simples :

- la taille de `char` est 1 ;
- chaque type de la liste précédente a une taille supérieure ou égale au type précédent.

```
sizeof(char) == 1 <= sizeof(short int) <= sizeof(int) <=
sizeof(long int) <= sizeof(long long int)
```

## Type abrégé

Il est possible de ne pas indiquer `int` dans les types précédent. Si un modificateur est utilisé sans préciser le type, ça sera `int` par défaut. Il est donc possible d'écrire `short`, `long` et `long long`.

Dans ce cours, le `int` sera toujours indiqué.

Si vous souhaitez absolument des types avec une taille fixée, il existe également les types suivants : `int8_t`, `int16_t`, `int32_t`, `int64_t` et leur équivalent non signé (voir en dessous) `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t` ("u" pour "unsigned"). Voir [Fixed width integer types](#).

Ces types imposent des contraintes plus fortes, il est préférable de les utiliser uniquement si vous en avez besoin.

## Signature des nombres entiers

Par défaut, les nombres entiers peuvent être positifs ou négatifs (on dit qu'ils sont *signés*). Lorsque vous n'avez pas besoin de représenter des valeurs négatives, il est possible d'utiliser le mot-clé `unsigned` pour créer un type entier qui ne peut représenter que des valeurs positives.

### Signature de `char`

Il existe aussi le mot-clé `signed` pour définir explicitement un type signé, mais comme les types sont signés par défaut, ce mot-clé n'est pas très utilisé.

Le cas d'utilisation le plus important est avec `char` : la norme C++ ne définit pas la signature de `char`, il peut être signé ou non selon le système. Il est donc recommandé, lorsque vous utilisez `char` pour représenter un entier et non un caractère, de préciser systématiquement la signature avec `unsigned` ou `signed`.

Un entier non signé permet de représenter des nombres positifs plus grands qu'un entier signé. Pour comprendre cela, reprenons l'exemple d'un entier codé sur quatre bits.

Chaque bit peut prendre deux valeurs possibles (que l'on peut représenter par 0 ou 1 par exemple). Donc un nombre représenté par 1 bit pourra prendre 2 valeurs, un nombre représenté par 2 bits pourra prendre  $2 \times 2$  valeurs, un nombre de 3 bits  $2 \times 2 \times 2$  valeurs et ainsi de suite.

Pour un nombre représenté par N bits, il sera donc possible de représenter  $2^N$  valeurs possibles. Un nombre de 4 bits pourra représenter 16 valeurs.

Pour un entier signé, cela signifie que l'on va pouvoir par exemple représenter les nombres suivants (en décimal) :

-8	-7	-6	-5
-4	-3	-2	-1
0	1	2	3
4	5	6	7

Les valeurs possibles vont donc de  $-2^{N/2}$  à  $2^{N/2}-1$ . La valeur négative (8) est en valeur absolue plus grande que la valeur positive (7), du fait qu'il faut représenter la valeur 0. (On peut bien sûr représenter n'importe quelle plage de valeurs, ce n'est qu'une question de choix. Cette plage de valeurs est celle utilisée en général, pour des raisons d'efficacité).

Si on ne représente que des valeurs positives ou nulles, on va pouvoir représenter par exemple :

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

La plage de valeurs devient donc 0 à  $2^N-1$ . La valeur maximale des nombres non signés est donc bien supérieure à celle des nombres signés.

## Vérifier les erreurs de conversion

Un type `signed char` est codé sur 8 bits et acceptera des valeurs allant de -128 à +127, alors que le type `unsigned char`, codé aussi sur 8 bits, acceptera des valeurs allant de 0 à 255.

Si vous initialisez une variable avec une valeur hors limite, une erreur sera signalée.

```
main.cpp
#include <iostream>

int main() {
    const signed char x { 255 };
    const unsigned char y { 255 };
```

}

Ce code produit une erreur de conversion (*narrowing*) pour le `signed char`, mais pas pour le `unsigned char`.

```
main.cpp:4:27: error: constant expression evaluates to 255
which cannot be narrowed to
type 'signed char' [-Wc++11-narrowing]
    const signed char x { 255 };
                           ^~~
main.cpp:4:27: note: insert an explicit cast to silence this
issue
    const signed char x { 255 };
                           ^~~
                                  static_cast<signed char>( )
```

On peut également vérifier que la taille en mémoire ne change pas :

```
main.cpp
#include <iostream>

int main() {
    std::cout << "sizeof(char) = " << sizeof(char) << std::endl;
    std::cout << "sizeof(unsigned char) = " << sizeof(
unsigned char) << std::endl;
    std::cout << "sizeof(short int) = " << sizeof(short int)
<< std::endl;
    std::cout << "sizeof(unsigned short int) = " << sizeof(
unsigned short int) << std::endl;
    std::cout << "sizeof(int) = " << sizeof(int) << std::endl;
    std::cout << "sizeof(unsigned int) = " << sizeof(
unsigned int) << std::endl;
    std::cout << "sizeof(long int) = " << sizeof(long int)
<< std::endl;
    std::cout << "sizeof(unsigned long int) = " << sizeof(
unsigned long int) << std::endl;
    std::cout << "sizeof(long long int) = " << sizeof(long
long int) << std::endl;
    std::cout << "sizeof(unsigned long long int) = " <<
```

```
    sizeof(unsigned long long int) << std::endl;  
}
```

affiche

```
sizeof(char) = 1  
sizeof(unsigned char) = 1  
sizeof(short int) = 2  
sizeof(unsigned short int) = 2  
sizeof(int) = 4  
sizeof(unsigned int) = 4  
sizeof(long int) = 8  
sizeof(unsigned long int) = 8  
sizeof(long long int) = 8  
sizeof(unsigned long long int) = 8
```

Attention cependant, le compilateur ne vérifiera pas que les valeurs passées sont bien positives. Si vous entrez une valeur négative, le comportement sera différent :

main.cpp

```
#include <iostream>  
  
int main() {  
    unsigned int i {};  
    i = -1;  
    std::cout << "unsigned int i = " << i << std::endl;  
}
```

affiche :

```
unsigned int i = 4294967295
```

Une vérification est faite par le compilateur uniquement lorsque vous initialisez une variable `unsigned` avec une valeur négative et en utilisant les accolades :

main.cpp

```
#include <iostream>  
  
int main() {
```

```

    unsigned int const i { -1 }; // erreur
    unsigned int const j = -1;   // ok
    std::cout << "unsigned int i = " << i << std::endl;
    std::cout << "unsigned int j = " << j << std::endl;
}

```

affiche le message d'erreur :

```

main.cpp:4:22: error: constant expression evaluates to -1
which cannot be
narrowed to type 'unsigned int' [-Wc++11-narrowing]
    unsigned int i { -1 };
               ^
main.cpp:4:22: note: override this message by inserting an
explicit cast
    unsigned int i { -1 };
               ^
               static_cast<unsigned int>( )
1 error generated.

```

Plus généralement, lorsque vous initialisez une variable en utilisant une littérale, le compilateur vérifie que votre littérale est compatible avec le type donné.

```

main.cpp
#include <iostream>

int main() {
    char const c { 123456 };           // erreur, "123456" est
trop grand pour "char"
    int const i { 1234567890123 }; // erreur, "1234567890123"
est trop grand pour "int"
    float const c { 123.456e123 }; // erreur, "123.456e123"
est trop grand pour "float"
    unsigned int const i { -1 };    // erreur, "-1" est
négatif

```

## Les nombres réels

Il est possible de modifier également les types réels, mais avec une

syntaxe un peu différente. Vous avez vu le type de base `double` sur 64 bits. Pour créer un type réel sur 32 bits, vous pouvez utiliser le type `float` et pour un type réel sur 128 bits, le type `long double`. Il n'existe pas d'autres formats de nombres réels (8 ou 16 bits).

main.cpp

```
#include <iostream>

int main() {
    std::cout << "sizeof(float) = " << sizeof(float) << std
::endl;
    std::cout << "sizeof(double) = " << sizeof(double) <<
std::endl;
    std::cout << "sizeof(long double) = " << sizeof(long
double) << std::endl;
}
```

affiche :

```
sizeof(float) = 4
sizeof(double) = 8
sizeof(long double) = 16
```

Pour des raisons historiques, les nombres réels étaient calculés par défaut sur 32 bits. Ce type a donc été appelé `float` ("flottant") en rapport à "nombre à virgule flottante". Par la suite, lorsque les nombres réels ont été codés sur 64 bits, le nouveau type a été appelé `double` puisque c'était le double d'un type `float`.

Les nombres réels peuvent être positifs ou négatifs. Il n'existe pas de version `unsigned` pour les nombres réels.

## Les modificateurs de littérales

De la même manière que les variables, les littérales (c'est-à-dire les valeurs constantes entrées directement dans le code) possèdent également un type.

```
123456; // type int
```

```
123.456; // type double
'a';      // type char
true;     // type bool
```

Lorsque l'on initialise une variable d'un type donné avec une littérale d'un autre type, une conversion est réalisée si possible. Lorsque la valeur d'une littérale est trop grande et ne peut être attribuée à un type, le compilateur signale une erreur d'arrondi (*narrowing*). S'il n'y a pas de problème de conversion, le type de la littérale est converti dans le type de la variable.

### main.cpp

```
#include <iostream>

int main() {
    char const c { 12 };           // conversion de "int"
vers "char"
    unsigned int const i { 123 };   // conversion de "int"
vers "unsigned int"
    float const f { 123.456 };    // conversion de "double"
vers "float"
}
```

Il est possible d'ajouter un *suffixe* aux littérales pour modifier leur type. Pour les entiers, le suffixe `u` ou `U` indique une littérale non signée (`unsigned`). Pour la taille mémoire, le suffixe `l` ou `L` indique une littérale de type `long int` et le suffixe `ll` ou `LL` indique une littérale de type `long long int`. Il est possible de combiner le suffixe pour le signe avec un suffixe pour la taille, mais sans mélanger les minuscules et les majuscules (donc les suffixes suivants sont acceptés : `ul`, `UL`, `ull` ou `ULL`).

### main.cpp

```
#include <iostream>

int main() {
    int const i { 123 };
    unsigned int const ui { 123u };
    unsigned long int const uli { 123ul };
    unsigned long long int const ulli { 123ull };
}
```

```
}
```

Pour les nombres réels, le suffixe `f` ou `F` indique une littérale de type `float` et le suffixe `l` ou `L` indique une littérale `long double`.

```
main.cpp
```

```
#include <iostream>

int main() {
    float const f { 123.456f };
    double const d { 123.456 };
    long double const ld { 123.456l };
}
```

Les modificateurs de littérales prennent tout leur sens avec l'inférence de type. Lorsque vous écrivez :

```
main.cpp
```

```
const int i { 123 };
```

Vous indiquez deux fois que vous souhaitez manipuler un entier de type `int` : dans la littérale et dans la déclaration de la variable. Si vous modifiez le type de la littérale (par exemple pour utiliser un réel au lieu d'un entier), il faut penser à modifier aussi le type de la variable.

L'inférence de type permet d'éviter de spécifier deux fois la même information :

```
main.cpp
```

```
const auto i = 123;
```

Cette syntaxe est donc plus simple à maintenir, elle sera préférable. Cependant, il n'est pas possible de créer certains types de littérales. Par exemple, il n'est pas possible de créer une littérale de type `short int`. Dans ce cas, il est possible d'utiliser une conversion implicite avec `static_cast` (vous verrez plus en détail cela par la suite) :

```
main.cpp
```

```
const auto i = static_cast<short int>(123);
```

Mais cette syntaxe commence à être un peu lourde, il est dans ce cas probablement préférable de ne pas utiliser l'inférence de type :

main.cpp

```
const short int i { 123 };
```

(Il existe des arguments en faveur de la syntaxe avec `static_cast`, mais ce sont plus des considérations de style. Retenez simplement que certains développeurs C++ préfèrent une syntaxe ou l'autre... et que les développeurs C++ sont parfois des gens compliqués. 

[Chapitre précédent](#) [Sommaire principal](#) [Chapitre suivant](#)

# Obtenir des informations sur les types

Adapter le type des variables utilisées permet d'optimiser la mémoire en fonction des besoins, mais cela à un impact sur les valeurs que peut prendre un type donné. Il est donc indispensable d'avoir un moyen pour obtenir des informations détaillées sur les types numériques (valeur minimale, valeur maximale, nombre de chiffres après la virgule, etc.).

Pour cela, le C++ propose un ensemble de fonctionnalités dans le fichier d'en-tête `limits`. Ces fonctionnalités sont un peu particulières, ce sont des classes génériques, qui utilisent une syntaxe avec des chevrons `<>`. Une telle classe, permettant d'obtenir des informations sur un type, est appelée une classe de traits. Vous allez voir dans ce chapitre comment utiliser ce type de classes, leur création sera abordé dans la partie sur la programmation objet.

La syntaxe générale est la suivante :

```
std::numeric_limits<TYPE>::FONCTION()
```

Avec `TYPE`, qui correspond au type pour lequel vous souhaitez obtenir des informations (par exemple `int`, `double`, etc), et `FONCTION`, qui correspond à l'information que vous souhaitez (par exemple `max` pour la valeur maximale et `min` pour la valeur minimale).

## Valeur maximale

Par exemple, pour connaître la valeur maximale que peut prendre le type `int`, il faut utiliser la syntaxe suivante :

```
main.cpp
```

```
#include <iostream>
```

```
#include <limits> // N'oubliez pas d'inclure l'en-tête

int main() {
    std::cout << "Max(int) = " << std::numeric_limits<int>::
max() << std::endl;
    std::cout << "Max(int) = " << std::hex << std::showbase
        << std::numeric_limits<int>::max() << std::endl;
}
```

Le résultat affiché sera :

```
Max(int) = 2147483647
Max(int) = 0xffffffff
```

Cette valeur maximale signifie concrètement que lorsque vous utilisez une variable de type `int` pour compter, vous ne pourrez compter que jusqu'à 2147483647. Au delà, le compte ne sera plus correct (en pratique, il reviendra à une valeur négative) :

main.cpp

```
#include <iostream>

int main() {
    int i { 2147483645 };
    std::cout << i << std::endl;
    ++i;
    std::cout << i << std::endl;
}
```

affiche :

```
2147483645
2147483646
2147483647
-2147483648
```

-2147483647

La syntaxe `++i` permet d'incrémenter la variable `i` de `+1`. C'est équivalent à écrire : `i = i + 1;` ou `i += 1;`.

Comme vous le savez déjà, le C++ est permissif, il n'interdira pas ce code. C'est à vous de vérifier que vous ne dépassiez pas la valeur maximale possible pour un type.

On peut alors vérifier les valeurs maximales pour les différents types numériques :

main.cpp

```
#include <iostream>
#include <limits>

int main() {
    std::cout << "Types entiers :" << std::endl;
    std::cout << "Max(short int) = " << std::numeric_limits<
short int>::max() << std::endl;
    std::cout << "Max(unsigned short int) = " << std:::
numeric_limits<unsigned short int>::max() << std::endl;
    std::cout << "Max(int) = " << std::numeric_limits<int>::
max() << std::endl;
    std::cout << "Max(unsigned int) = " << std:::
numeric_limits<unsigned int>::max() << std::endl;
    std::cout << "Max(long int) = " << std::numeric_limits<
long int>::max() << std::endl;
    std::cout << "Max(unsigned long int) = " << std:::
numeric_limits<unsigned long int>::max() << std::endl;
    std::cout << "Max(long long int) = " << std:::
numeric_limits<long long int>::max() << std::endl;
    std::cout << "Max(unsigned long long int) = " <<
        std::numeric_limits<unsigned long long int>::max()
<< std::endl;

    std::cout << std::endl << "Types réels:" << std::endl;
    std::cout << "Max(float) = " << std::numeric_limits<
float>::max() << std::endl;
    std::cout << "Max(double) = " << std::numeric_limits<
double>::max() << std::endl;
```

```
    std::cout << "Max(long double) = " << std::  
numeric_limits<long double>::max() << std::endl;  
}
```

affiche :

```
Types entiers :  
Max(short int) = 32767  
Max(unsigned short int) = 65535  
Max(int) = 2147483647  
Max(unsigned int) = 4294967295  
Max(long int) = 9223372036854775807  
Max(unsigned long int) = 18446744073709551615  
Max(long long int) = 9223372036854775807  
Max(unsigned long long int) = 18446744073709551615  
Types réels:  
  
Max(float) = 3.40282e+38  
Max(double) = 1.79769e+308  
Max(long double) = 1.18973e+4932
```

Pour le type `char`, qui peut être interprété comme un nombre entier ou un caractère, si vous écrivez un code similaire au code précédent, cela n'affichera pas la valeur maximale (`std::cout` essaiera d'afficher cette valeur comme un caractère). Pour contourner ce problème, il suffit d'enregistrer cette valeur dans une variable de type entier (par exemple `int`) et d'afficher cette variable.

#### main.cpp

```
#include <iostream>  
#include <limits>  
  
int main() {  
    int const signed_char_max { std::numeric_limits<signed  
char>::max() };  
    std::cout << "Max(char) = " << signed_char_max << std::  
endl;  
  
    int const unsigned_char_max { std::numeric_limits<  
unsigned char>::max() };  
    std::cout << "Max(unsigned char) = " <<
```

```
unsigned_char_max << std::endl;  
}
```

affiche :

```
Max(char) = 127  
Max(unsigned char) = 255
```

Comme vous pouviez vous y attendre, plus un type est représenté par plus d'octets en mémoire, plus sa valeur maximale sera élevée. On peut remarquer aussi que les types `long int` et `long long int`, qui ont la même taille mémoire (8 octets), ont la même valeur maximale. Vous pouvez aussi noter que les types `unsigned` ont des valeurs maximales plus grandes que le type signé correspondant.

Comme indiqué précédemment, la signature du type `char` n'est pas définie par la norme et peut changer selon le système. De plus, ce type peut représenter un caractère et un entier.

Pour limiter les risques de confusion, dans ce cours, nous utiliserons `char` pour indiquer un caractère et `signed char` ou `unsigned char` lorsque l'on veut explicitement faire référence à la représentation entière.

Notez bien que ce n'est qu'une convention d'écriture dans ce cours. Si `char` est signé, il sera équivalent à `signed char` en termes de comportement. De même s'il n'est pas signé, il sera équivalent à `unsigned char`.

## Valeur minimale

Pour obtenir les valeurs minimales d'un type, il faut utiliser les fonctions `min` et `lowest` à la place de `max`. Cependant, ces fonctions ne retournent pas exactement la même chose, selon que vous l'utiliser avec un type entier ou un type réel.

Commençons par les types entiers. Dans ce cas, les deux fonctions retournent la même valeur, qui correspond à la plus petite valeur que

peu prendre un type. Pour les types non signées, cette valeur sera toujours 0 et pour les types signés, cette valeur sera `-max() - 1` :

main.cpp

```
#include <iostream>
#include <limits>

int main() {
    int const signed_char_min { std::numeric_limits<signed char>::min() };
    std::cout << "Min(char) = " << signed_char_min << std::endl;
    std::cout << "Min(short int) = " << std::numeric_limits<short int>::min() << std::endl;
    std::cout << "Min(int) = " << std::numeric_limits<int>::min() << std::endl;
    std::cout << "Min(long int) = " << std::numeric_limits<long int>::min() << std::endl;
}
```

affiche :

```
Min(char) = -128
Min(short int) = -32768
Min(int) = -2147483648
Min(long int) = -9223372036854775808
```

Concrètement, cela veut dire, pour une variable de type `int` par exemple, que vous pouvez compter de -2147483648 à 2147483647. Et pour un type `unsigned int`, que vous pouvez compter de 0 à 4294967295.

Dans le cas des nombres réels, les fonctions `min` et `lowest` ne retournent pas la même valeur :

- `min` retourne la plus petite valeur positive non nulle (la plus petite valeur proche de 0) ;
- `lowest` retourne la plus petite valeur représentable, donc généralement `-max`.

## Mathématiques discrète et continue

En mathématique, quelque soit  $x$  appartenant à l'ensemble des nombres réels positifs non nul  $\mathbb{R}^*$ , il existe toujours un nombre plus petit que  $x$  (on peut démontrer cela facilement, montrant que  $\frac{x}{2}$  est strictement compris entre 0 et  $x$ ). Ce qui revient à dire qu'il n'existe pas de plus petit nombre positif non nul.

En informatique, la situation est différente. Un nombre en mémoire est représenté par un nombre fini de bits, par exemple 32 pour `float` ou 64 bits pour `double`. Cela veut dire qu'il est possible représenter qu'un nombre limité de valeur possible sur un ordinateur et donc qu'il n'existe pas toujours un nombre réel  $\frac{x}{2}$ . Dit autrement, cela veut dire qu'il existe une valeur minimale positive.

Il est très important de garder en mémoire que les nombres réels dans un programme ne sont qu'une **REPRÉSENTATION** des nombres réels tels qu'ils sont conçus en mathématique. Cela aura une importance particulière lorsque vous réaliserez des calculs numériques.

Cette valeur minimale positive est celle obtenue avec la fonction `min`.

main.cpp

```
#include <iostream>
#include <limits>

int main() {
    std::cout << "Min(float) = " << std::numeric_limits<
float>::min() << std::endl;
    std::cout << "Min(double) = " << std::numeric_limits<
double>::min() << std::endl;
    std::cout << "Min(long double) = " << std::
numeric_limits<long double>::min() << std::endl;
}
```

affiche :

```
Min(float) = 1.17549e-38
Min(double) = 2.22507e-308
Min(long double) = 3.3621e-4932
```

Pour obtenir la plus petite valeur représentable , vous pouvez utiliser la fonction `lowest`.

main.cpp

```
#include <iostream>
#include <limits>

int main() {
    std::cout << "Lowest(float) = " << std::numeric_limits<
float>::lowest() << std::endl;
    std::cout << "Lowest(double) = " << std::numeric_limits<
double>::lowest() << std::endl;
    std::cout << "Lowest(long double) = " << std::
numeric_limits<long double>::lowest() << std::endl;
}
```

affiche :

```
Lowest(float) = -3.40282e+38
Lowest(double) = -1.79769e+308
Lowest(long double) = -1.18973e+4932
```

Il existe également des constantes définies dans l'en-tête `climits`, tel que `INT_MIN`, `INT_MAX`, `DBL_MIN`, `DBL_MAX`, mais elles n'offrent pas les mêmes garanties concernant les types. Elles sont fournies à titre de rétro-compatibilité avec le C et les anciens codes C++ et ne sont donc pas recommandées. Voir [C numeric limits interface](#) pour plus de détails.

## Informations sur les types

Il est également possible d'obtenir des informations sur les types, autre que les valeurs limites qu'elles peuvent prendre (`lowest`, `min` et `max`) ou la taille en mémoire (`sizeof`). Par exemple, pour savoir si un type est signé ou non, savoir si un type est entier ou réel, etc. L'ensemble de ces fonctionnalités est dans le fichier d'en-tête `type_traits`.

Fonction	Rôle	Exemple
is_arithmetic	Teste si un type représente un nombre	<code>std::is_arithmetic&lt;int&gt;::value</code>
is_integral	Teste si un type représente un nombre entier	<code>std::is_integral&lt;int&gt;::value</code>
is_floating_point	Teste si un type représente un nombre réel	<code>std::is_floating_point&lt;int&gt;::value</code>
is_signed	Teste si un type est signé	<code>std::is_signed&lt;int&gt;::value</code>
is_unsigned	Teste si un type n'est pas signé	<code>std::is_unsigned&lt;int&gt;::value</code>
is_const	Teste si un type est constant	<code>std::is_const&lt;int&gt;::value</code>

Ces différents code retournent un booléen, vous pouvez donc les afficher avec `std::cout` (en utilisant la directive `std::boolalpha` si vous voulez qu'il affiche `true` ou `false` en toutes lettres) et utiliser les opérateurs booléens déjà vus (inverse `!`, AND `&&` et OR `||`).

### main.cpp

```
#include <iostream>
#include <type_traits> // N'oubliez pas d'inclure l'en-tête

int main() {
    std::cout << std::boolalpha;
    std::cout << "is_arithmetic(int) = " << std:::
```

```

is_arithmetic<int>::value << std::endl;
    std::cout << "is_arithmetic(bool) = " << std::
is_arithmetic<bool>::value << std::endl;

    std::cout << std::endl;
    std::cout << "is_integral(int) = " << std::is_integral<
int>::value << std::endl;
    std::cout << "is_integral(float) = " << std::is_integral<
float>::value << std::endl;

    std::cout << std::endl;
    std::cout << "is_floating_point(float) = " << std::
is_floating_point<float>::value << std::endl;
    std::cout << "is_floating_point(int) = " << std::
is_floating_point<int>::value << std::endl;

    std::cout << std::endl;
    std::cout << "is_signed(int) = " << std::is_signed<int>
::value << std::endl;
    std::cout << "is_signed(unsigned int) = " << std::
is_signed<unsigned int>::value << std::endl;

    std::cout << std::endl;
    std::cout << "is_unsigned(unsigned int) = " << std::
is_unsigned<unsigned int>::value << std::endl;
    std::cout << "is_unsigned(int) = " << std::is_unsigned<
int>::value << std::endl;

    std::cout << std::endl;
    std::cout << "is_const(const int) = " << std::is_const<
const int>::value << std::endl;
    std::cout << "is_const(int) = " << std::is_const<int>::
value << std::endl;
}

```

affiche :

```

is_arithmetic(int) = true
is_arithmetic(bool) = true

is_integral(int) = true
is_integral(float) = false

```

```
is_floating_point(float) = true
is_floating_point(int) = false

is_signed(int) = true
is_signed(unsigned int) = false

is_unsigned(unsigned int) = true
is_unsigned(int) = false

is_const(const int) = true
is_const(int) = false
```

Pour terminer, il peut être intéressant de pouvoir comparer deux types, pour vérifier s'ils sont équivalents ou non. Pour cela, le C++ fournit la classe générique `std::is_same`, qui s'utilise avec la syntaxe suivante :

```
std::is_same<TYPE1, TYPE2>::value
```

Par exemple, pour comparer les types `char` :

main.cpp

```
#include <iostream>
#include <type_traits>

int main() {
    std::cout << std::boolalpha;
    std::cout << "is_signed(char) = " << std::is_signed<char>::value << std::endl;
    std::cout << "is_unsigned(char) = " << std::is_unsigned<char>::value << std::endl;
    std::cout << "is_same(char, signed char) = " << std::is_same<char, signed char>::value << std::endl;
    std::cout << "is_same(char, unsigned char) = " << std::is_same<char, unsigned char>::value << std::endl;
}
```

affiche :

```
is_signed(char) = true
is_unsigned(char) = false
is_same(char, signed char) = false
```

```
is_same(char, unsigned char) = false
```

Ce résultat peut paraître surprend, mais il faut bien comprendre que `char` est un cas particulier. Même s'il est signé ou non signé, c'est un type différent de `signed char` et `unsigned char`. Même s'il aura généralement le même comportement, en particulier lorsqu'il est utilisé avec `std::cout`.

Dans ce chapitre, vous avez vu une utilisation simple des classes `numeric_limits` et `is_same` (qui sont appelées “classes de traits”, d'où le nom du fichier d'en-tête). Mais en réalité, ces fonctionnalités sont beaucoup plus utiles et puissantes que ce qui est présenté ici. En effet, par la suite, vous verrez qu'il est possible de créer du code C++ qui produit du code C++ en utilisant ce type de fonctionnalités. Cela s'appelle de la métaprogrammation et c'est l'une des forces du C++.

## Exercices

- `epsilon`, `nextafter`, `nexttoward`
- `digits`, `digits10`, `max_digits10`, `radix`
- représentation des nombres entier négatif

[Chapitre précédent](#) [Sommaire principal](#) [Chapitre suivant](#)

# Créer des nouveaux types

## Intérêt et importance de définir ses types

Le C++ fournit de nombreux types fondamentaux, pour les besoins de base (et vous ne les avez pas encore tous vu). Les types sont des informations fondamentales pour le compilateur, pour générer un programme exécutable, mais également pour réaliser des optimisations et vérifier leur utilisation correcte (vous avez vu par exemple la vérification des dépassements de valeur - le *narrowing*).

Il est tout à fait possible d'écrire des programmes utilisant uniquement ces types fondamentaux, mais le C++ va plus loin et permet de définir ses propres types, plus ou moins complexes, en combinant des types fondamentaux et d'autres types complexes.

Les types que vous définirez seront utilisé à deux niveaux :

- par le compilateur, pour connaître la structure en mémoire (représentation binaire), les optimisations possibles, etc.
- par les “utilisateurs” de vos types (les autres développeurs, qui utiliseront votre code, pas les utilisateurs de votre programme), pour savoir comment les utiliser correctement (syntaxe) et ce qu'ils représentent (sémantique).

**Avant de chercher à obtenir les meilleures performances possibles, la première chose à faire sera toujours de définir correctement ses types en termes de syntaxe et de sémantique.**

L'une des forces du C++ est de permet de définir des types parfois très complexes, avec de nombreuses contraintes, mais qui seront validés lors de la phase de compilation. Ce qui signifie que quelque soit la complexité de vos types, cela n'aura pas d'impact sur les performances.

On dit parfois que le C++ est un langage performant. Ce n'est pas tout à fait vrai : le C++ est avant tout un langage qui vous donne le contrôle de ces performances. Mais cela nécessite d'écrire du code correcte et demande souvent du travail supplémentaire pour obtenir ces performances optimales.

## Alias de type

La première façon de définir un nouveau type est de créer un alias de type. Cela consiste simplement à donner un nouveau nom à un type. L'intérêt est de pouvoir donner un nom court à un type complexe, de donner un nom plus expressif à un type ou de faciliter l'évolution du code.

La syntaxe utilise le mot-clé `using` :

```
using NOM = TYPE;
```

`NOM` correspond au nouveau type que vous souhaitez créer et `TYPE` au type correspondant. Plus concrètement, vous pouvez écrire par exemple :

```
using mon_type_entier = int;
```

Pour les noms de type, les règles sont les mêmes que pour les noms de variables (caractères alphanumériques et tiret bas et doit commencer par une lettre). Il existe également des règles de codage spécifique pour le nommage des types. Par exemple, la bibliothèque standard ou la bibliothèque `Boost` ajoutent le suffixe `_t` ou `_type` dans certains cas, pour nommer un type, la bibliothèque `Qt` nomme les types avec un `Q` comme premier caractère.

## `typedef`

Il existait une ancienne syntaxe pour déclarer un alias de type, utilisant le mot-clé `typedef`. La syntaxe est la suivante :

```
typedef TYPE NOM;
```

La syntaxe avec `using` est plus puissante (elle est utilisable avec les templates, que vous verrez par la suite) et est donc préférée en C++ moderne. Mais vous pouvez encore rencontrer la syntaxe avec `typedef` dans les anciens codes.

Un alias est simplement une façon de donner un autre nom à la même "chose". Pour le compilateur, un type et son alias sont exactement le même type. Vous pouvez le vérifier avec `is_same` que vous avez déjà vu :

main.cpp

```
#include <iostream>

int main() {
    using my_type = int;
    std::cout << std::boolalpha;
    std::cout << std::is_same<my_type, int>::value << std::endl;
}
```

affiche :

```
true
```

Attention, vous avez déjà rencontré le mot-clé `using`, pour déclarer l'utilisation d'un espace de noms. Il s'agit bien de deux utilisation différentes et donc deux syntaxes différentes. Faites bien attention à bien les distinguer. Pour rappel :

```
using namespace std; // déclaration d'un espace de noms
using std::cout; // déclaration d'une fonctionnalité d'un espace de noms

using nouveau_type = ancien_type; // déclaration d'un nouveau type
```

## Donner un nom plus simple

Un alias de type peut être utilisé pour simplifier le code, en donnant un nom court à des types. Par exemple :

```
using ull_int = unsigned long long int;
```

Cette technique est particulièrement intéressante avec les types complexes, par exemple ceux que l'on trouve dans la bibliothèque standard, et avec la métaprogrammation. Par exemple, les types à taille fixe `int8_t`, `int16_t`, etc. sont des alias de type.

```
// "Exact-width integer types" définis dans MingW 4.9.2
using int8_t = signed char;
using int16_t = short;
```

## Donner un nom plus explicite

Les noms des types apportent des informations sur leur représentation, ce qu'ils signifient. Par exemple, vous savez que si vous voyez le type `int`, vous savez que cela représente un entier signé, généralement codé sur 32 bits, avec une valeur maximale et minimale que vous pouvez obtenir avec `std::numeric_limits`, etc.

Par contre, vous ne savez pas forcément ce que représente une variable de ce type. Cette information peut être apportée par le nom des variables, mais il est également possible d'utiliser les alias de type pour renforcement l'expressivité des types.

Par exemple, si vous créez deux variables représentant des temps en heures et en minutes :

```
int temps_1 { 12 }; // heure
int temps_2 { 40 }; // minutes
```

Les commentaires ne sont pas suffisants, ils seront vite oublié plus loin dans le code. Un alias de type permet d'avoir des noms plus explicites.

Par exemple :

```
using heure = int;
using minute = int;

heure temps_1 { 12 };
minute temps_2 { 40 };
```

Cette correction peut sembler mineure, mais elle correspond à un principe important en programmation. Un code est plus souvent lu qu'il n'est écrit. Il ne faut donc pas écrire vos codes pour qu'ils soient plus faciles à écrire, mais pour qu'ils soient les plus faciles à lire. L'expérience montre que l'on perd beaucoup de temps à comprendre un ancien code pour corriger les erreurs de programmation ou le faire évoluer.

## Avoir un code plus évolutif

Imaginez que vous écrivez le code suivant :

```
int x {};
int y {};
int z {};
int a {};
int b {};
int c {};
```

Quelque temps après, une analyse des contraintes sur les données montre que les valeurs prises par ces variables seront comprises entre 0 et 100. Vous souhaitez donc changer les types pour utiliser un type entier non signé sur 8 bits `std::uint8_t`, au lieu d'un type entier signé `int`. Vous faites donc le remplacement de tous les types des variables :

```
std::uint8_t x {};
std::uint8_t y {};
std::uint8_t z {};
std::uint8_t a {};
std::uint8_t b {};
```

```
std::uint8_t c {};
```

Dans cet exemple, le changement est facile, il suffit de copier-coller le type. Imaginez maintenant la même chose sur 1000 fichiers. Vous allez perdre du temps pour un changement aussi simple.

Deux solutions sont possibles : soit interdire ce type de changement, soit faire en sorte que vous n'ayez qu'une seule ligne de code à changer.

La première solution peut paraître surprenante, mais c'est une situation qui arrive (trop souvent) dans les projets professionnels (en particulier lorsque la question de l'évolutivité du code n'a pas été prise en compte dès le début). Il est en effet possible de discuter de la pertinence d'un tel changement (quel est le gain **réel** de passer de `int` à `int8_t` ?) ou que ce changement soit facturé au client (et il peut refuser dans ce cas). Dans ce cas, les changements non critiques peuvent être repoussés ou annulés (ce qui peut contribuer à la [dette technique](#)).

L'autre solution est de prendre en compte dès le début que le code puisse évoluer. Dans ce cours, et plus généralement dans les approches modernes de développement logiciel, on supposera toujours que le code peut évoluer (c'est l'un des principes de base des [méthodes Agile](#)).

Avec un alias de type, il est possible d'avoir une seule ligne de code à modifier si le code évolue. Par exemple :

```
using my_int = int;

my_int x {};
my_int y {};
my_int z {};
my_int a {};
my_int b {};
my_int c {};
```

La modification du type ne nécessite plus que de modifier la ligne avec `using`.

```
using my_int = int8_t;
```

## Évolutivité du code

Vous voyez dans cet exemple que l'évolutivité du code n'est pas uniquement une question de syntaxe. Le C++ offre différentes fonctionnalités pour écrire (plus ou moins facilement) du code évolutif, comme par exemple les alias de type ou l'inférence de type (vous verrez d'autres fonctionnalités plus avancées par la suite).

**Mais dans tous les cas, n'oubliez pas qu'un code est évolutif ou réutilisable avant tout parce que vous avez penser a ces problématiques en écrivant votre code.**

## Composition de données

### Déclaration d'une structure

L'alias de type n'est pas suffisant pour créer des structures de données complexes. Pour cela, un première approche (limitée, ce point sera détaillé à la fin de ce chapitre) est la composition de données. Cela consiste simplement à créer un nouveau type contenant plusieurs variables de type quelconque.

Cette notion est assez intuitive à comprendre, il est possible de la rapprocher d'exemple dans la vie de tous les jours :

- une date composée d'un jour, d'un mois et d'une année ;
- une voiture est composée d'une carrosserie, d'un moteur, de sièges, etc. ;
- un humain est composé d'un corps, d'une tête, de membres.

La syntaxe générale pour créer une composition de données est la suivante :

```
struct NOM {  
    LISTE DES MEMBRES  
};
```

Cette syntaxe permet de créer un nouveau type nommé **NOM** et qui contient les variables détaillées dans **LISTE DES MEMBRES**. La liste de membres est une suite de déclaration de variables, d'alias de type, d'énumération, comme vous avez vu dans les chapitres précédents.

```
struct NOM {  
    // variables  
    TYPE IDENTIFIANT {};           // variable initialisée  
    par défaut  
    TYPE IDENTIFIANT { VALEUR };   // variable initialisée  
    avec une valeur  
  
    // alias de type  
    using NOM = TYPE;  
};
```

Cette liste peut être vide, contenir qu'une seul membre ou autant de membres que vous souhaitez.

En fait, presque tout ce que vous pouvez déclarer en dehors d'une composition de données, vous pouvez également le déclarer comme membre d'une composition de données (variable, alias de type, énumération, fonction, etc). Mais vous ne pouvez pas, par exemple, déclarer un espace de noms (*namespace*) ou utiliser l'inférence de type (**auto**, **decltype**).

Si vous reprenez l'exemple de la date, cela donnera par exemple :

```
struct Date {  
    uint8_t day {};  
    uint8_t month {};  
    int16_t year {};  
};
```

Quelques remarques sur ce code :

- dans ce cours, les noms de types seront écrit avec une majuscule initiale et les variables avec une minuscule ;
- prenez l'habitude d'écrire vos codes en anglais ;
- prenez également l'habitude de choisir les types les plus

adaptées aux données que vous souhaitez manipuler. Un jour est compris entre 0 et 31, c'est donc un nombre positif qui peut être représenté par un entier sur 8 bits, d'où le choix de `uint8_t`. L'année peut être positive ou négative et peut prendre des valeurs plus grande que le jour, d'où le choix de `int16_t`.

## Structure et classes

Le mot-clé `struct` signifie “structure”. Certains langages de programmation orientés objet (POO) distinguent les composition de données simple sans sémantique particulière (structure) et les composition de données complexes (classes).

En C++, la distinction entre `struct` et `class` n'est pas pertinente, ces deux mots-clés sont équivalents (à quelques détails syntaxiques). Tout ce que vous pouvez faire avec `struct`, vous pouvez également le faire avec `class` et vice-versa.

La distinction entre un composition simple et une classe complexe est purement pédagogique dans ce cours et ne traduit pas une réalité dans les syntaxes C++. Le terme “composition de données” est utilisé pour bien montrer la différence avec “structure” et “classe”, mais dans la suite de ce cours, le terme “classe” sera plus souvent utilisé.

La création de classes complexes sera détaillée dans la partie sur la programmation objet.

## Utilisation

Une structure ou classe sont des types à part entière. Vous pouvez donc les utiliser dans toutes les syntaxes que vous avez déjà vu. Par exemple pour déclarer une variable :

```
Date dog_birthday{};  
Date nez_year_day{};
```

Vous pouvez également l'utiliser avec `sizeof` ou les classes de traits. Vous pouvez utiliser par exemple `std::is_integral` ou `std::numeric_limits`, mais cela n'aura pas beaucoup d'intérêt par défaut (ces fonctionnalités retourneront des valeurs par défaut, sauf si vous définissez un comportement spécifique pour vos types. Vous verrez cela dans la partie programmation orientée objet). Par contre, vous pouvez utiliser des traits tel que `is_class` sans problème (voir [Type support](#) pour les détails).

### main.cpp

```
#include <iostream>
#include <type_traits>
#include <limits>

struct Date {
    uint8_t day {};
    uint8_t month {};
    int16_t year {};
};

int main() {
    std::cout << sizeof(Date) << std::endl;
    std::cout << std::boolalpha;
    std::cout << std::is_integral<Date>::value << std::endl;
    std::cout << std::is_class<Date>::value << std::endl;
    return 0;
}
```

Une fois que vous avez déclarer une classe, il faut pouvoir accéder à ses membres. Il existe deux opérateurs pour accéder aux membres d'une classe : le point `.` et le double deux-points. Le premier s'utilise avec un nom de variable et le second avec un nom de type, selon les syntaxes :

```
VARIABLE.MEMBRE  
TYPE::MEMBRE
```

Plus concretement :

```
struct Date {
    // déclaration d'un alias de type
```

```
using day_t = uint8_t;

// déclaration d'une variable membre
day_t day{};

// déclaration d'un variable de type Date
Date today{};

// utilisation de la variable membre "day" de la variable
// "today"
today.day = 10;

// utilisation du type membre "day_t" du type "Date"
Date::day_t birthday{};
```

Comme vous n'avez pas encore vu les fonctions membres, vous pourriez penser que pour utiliser une variable membre, il faut utiliser l'opérateur `.` sur une variable et un type membre (*nested type*), il faut utiliser l'opérateur `::` sur un type.

En fait, ce n'est pas le cas, il est possible d'utiliser l'opérateur `::` avec une variable membre, lorsque vous utiliserez des fonctions membres.

La seule règle à retenir est que l'opérateur `.` est précédé d'une variable et l'opérateur `::` est précédé d'un type.

## Portée et instantiation

L'opérateur `::` est appelée opérateur de portée. Il permet d'indiquer qu'un identifiant (nom de variable, de fonction, de type, etc) appartient à une portée spécifique. Vous avez déjà souvent rencontré cet opérateur, avec l'espace de noms `std`, par exemple pour écrire du texte avec `std::cout`. Les espaces de noms et le fonctionnement détaillé de la portée seront vu en détail dans la suite de ce cours.

Les types et variables membres s'utilisent comme des types et variables

habituelles, sauf qu'ils sont précédé de l'identifiant du type ou de la variable.

L'opérateur `.` permet d'accéder à une variable membre d'une variable. Il faut bien comprendre que deux variables auront leurs variables membres différentes :

```
Date first_day {};  
Date last_day {};  
  
first_day.day = 1;  
last_day.day = 31;
```

Dans ce code, même si `day` est utilisée deux fois, elle concerne deux variables différentes (`first_day` et `last_day`) et peut donc avoir des valeurs différentes.

## Sémantiques

Ecrire un code réutilisable nécessite donc de garder en tête la question : “comment pourra-t-on réutiliser ce code ?”

Cette problématique est complexe et ne peut être résumée en quelques lignes. Mais il est possible de définir quelques règles de base, non exhaustives :

- **Présentez correctement vos codes.** Passez à la ligne, indentez vos lignes (c'est-à-dire ajoutez des espaces en début de ligne pour que vos codes soient alignés), aérez votre code (utilisez des espaces pour faciliter la lecture) ;
- **Regroupez vos codes en termes de fonctionnalités.** Vous verrez par la suite comment regrouper vos codes dans des fichiers pour créer des fonctions, classes et modules.
- **Donnez un sens aux choses.** Donnez des noms de variable et de type qui ont un sens (sémantique).

Un code sera réutilisable si il est facile de comprendre ce qu'un code fait.

Cela passe par la documentation (qu'il ne fait pas négliger), mais également par le choix des identifiants que vous donnerez à vos variables et types (puis, plus tard, à vos fonctions et classes).

### Limitation des alias de type

Créer un alias de type permet de donner un identifiant explicite à un type. Cependant, cette approche ne sera pas suffisante. Donner un sens aux choses est intéressant, faire que le compilateur comprenne le sens donné et valide leur bonne utilisation, c'est encore mieux. Ainsi, même en créant des types `Heure` et `Minute`, le compilateur ne fait aucune validation (pour lui, ce sont des entiers) et ne posera pas de problèmes à les additionner.

```
using Heure = int;
using Minute = int;

Heure h { 8 };
Minute m { 15 };

TYPE temps_total { h + m };
// Quel est le type résultant ?
// Quel est le sens de cet addition ?
```

Il faudrait au minimum que le compilateur prévienne que vous essayez d'additionner deux concepts distincts. Ou encore mieux, qu'il comprennent que 1 heure vaut 60 minutes et qu'il fasse l'addition en faisant la conversion, pour donner le résultat correct. Les alias de type ne permettent pas de faire cela.

### Limitation des composition de données

Il est possible d'aller un peu plus loin, en créant une classe pour les heures et les minutes, de la façon suivante :

```
class Heure { int h {} };
class Minute { int m {} };

Heure h { 8 };
Minute m { 15 };
```

```
TYPE temps_total { h + m }; // Erreur
```

Dans ce cas, le compilateur reconnaît que les types ne sont pas compatibles et produit une erreur (ce qui est mieux que de donner un résultat incorrect, puisque vous êtes avertie du problème et que vous pouvez modifier votre code pour le corriger).

Mais ça serait encore mieux que l'addition soit autorisée ET que le résultat soit correct.

Il manque donc “quelque chose” à une simple “composition de données” pour en faire un type à part entière. Ce qu'il manque c'est une sémantique, c'est à dire un “sens” à ce type, c'est-a-dire l'ensemble des opérations que vous pouvez faire et ce qu'elles signifient.

Mais heureusement, le C++ propose des techniques plus avancées pour proposer une sémantique complète, validée par le compilateur. Vous verrez dans la suite du cours comment créer ce type de sémantique, en utilisant des classes (programmation orientée objet) et la métaprogrammation (création de langages spécifiques d'un domaine - DSL).

<a href="#">Chapitre précédent</a>	<a href="#">Sommaire principal</a>	<a href="#">Chapitre suivant</a>
------------------------------------	------------------------------------	----------------------------------

# Les nombres pseudo-aléatoires

Un nombre aléatoire est un nombre dont la valeur est choisie au hasard. Ils sont utilisés dans de nombreux contextes, par exemple dans les jeux vidéo (pour avoir un comportement non prédictible), en cryptographie (pour encoder des informations), ou dans les tests automatisés de code (pour vérifier le comportement d'un programme avec des valeurs indéterminées). Vous verrez dans différents projets de ce cours des cas d'utilisation des nombres aléatoires.

## Qualité des nombres aléatoires

Tirer des nombres aléatoires peut sembler être une tâche facile. On peut par exemple prendre un dé (non pipé) à six faces et le lancer plusieurs fois pour obtenir une série de nombres aléatoires compris entre un et six. On peut également lancer une pièce de monnaie pour obtenir des nombres aléatoires valant zéro (pile) ou un (face).

La difficulté vient du fait qu'il est facile d'obtenir des suites de nombres dont on n'arrive pas à prédire la prochaine valeur mais qui présentent quand même des propriétés mathématiques prédictibles. Si on additionne par exemple la valeur de deux dés, on aura un nombre aléatoire compris entre 2 et 12. Mais la probabilité d'obtenir un 12 est bien inférieure (1 chance sur 36) à la probabilité d'obtenir un 7 (12 chances sur 36).

Quand vous jouez à un jeu de société, ce n'est pas très important que les nombres obtenus ne soient pas parfaitement aléatoires (et c'est même intégré comme composante du jeu). Mais lorsque les nombres aléatoires permettent de crypter des données importantes ou de sécuriser un numéro de carte bancaire, la moindre propriété mathématique prédictible sur une série de nombres aléatoires peut permettre de contourner ces sécurités.

Il ne faut donc pas simplement que les nombres semblent aléatoires. Il faut le prouver mathématiquement. Et bien sûr, cela sort complètement du cadre de ce cours, ce sont des travaux de recherches mathématique très pointus.

Pour en revenir à la programmation, on peut se poser une question : comment générer des nombres parfaitement aléatoires sur un ordinateur ? Ce n'est pas si simple, puisqu'un ordinateur est prévu pour être parfaitement prédictible (la même suite d'instructions avec les mêmes données produira le même résultat).

Heureusement, il existe plusieurs approches possibles.

## Nombres aléatoires en C++

La bibliothèque standard fournit plusieurs fonctionnalités pour générer des nombres aléatoires, en utilisant différents algorithmes. L'ensemble de ces fonctionnalités sont accessibles en utilisant le fichier d'en-tête `random` ("aléatoire" en anglais).

```
#include <random>
```

La documentation est accessible en ligne sur [cppreference.com](http://cppreference.com): [Pseudo-random number generation](#), n'hésitez pas à vous familiariser un peu avec elle avant de continuer ce chapitre.

Les fonctionnalités de génération de nombres aléatoires se décomposent en trois catégories :

- le générateur de nombres aléatoires non déterministe ;
- les générateurs de nombres aléatoires déterministes (nombres pseudo-aléatoires) ;
- les générateurs de distributions statistiques.

Si vous travaillez sur du code qui a besoin d'un générateur le plus aléatoire possible, il faudra vérifier le fonctionnement du générateur ou utiliser une bibliothèque dédiée à la génération de nombres aléatoires pour la cryptographie (ces générateurs sont plus lents que les

générateurs utilisés dans la bibliothèque standard, ce qui explique pourquoi ils ne sont pas utilisés par défaut).

## Générateurs non déterministes

L'utilisation d'un générateur aléatoire est relativement simple, même si cela utilise une syntaxe que vous n'avez pas encore vu. Il faut créer une variable utilisant le générateur comme type, puis appeler cette variable comme une fonction. On parle d'"objet appelable" (*callable* en anglais). En pratique, la syntaxe est la suivante :

```
CallableType myVariable[];           // créé une variable
de type CallableType
auto result = myVariable(arg1, arg2); // appelle cette
variable comme une fonction
```

Le type `CallableType` représente n'importe quel type qui permet de créer un objet appelable. Comme n'importe quelle fonction, un objet appelable peut prendre des arguments en entrée et retourner une valeur.

Le générateur non déterministe est la classe `std::random_device`. Cet objet sera appelé sans utiliser d'argument, et retourne un entier aléatoire à chaque fois qu'il est appelé.

```
main.cpp
#include <iostream>
#include <random>

int main() {
    std::random_device rd[];           // création du
générateur

    std::cout << rd() << std::endl;   // génération d'un
nombre aléatoire
    std::cout << rd() << std::endl;   // génération d'un
nombre aléatoire
    std::cout << rd() << std::endl;   // génération d'un
nombre aléatoire
```

{}

Ce code permet de générer trois nombres aléatoires, qui seront différents à chaque fois que vous relancez le programme. Par exemple, cela peut donner la série suivante :

```
3555021123  
4089866385  
3885819577
```

En pratique, l'implémentation des générateurs n'est pas définie dans la norme C++, chaque implémentation de la bibliothèque standard peut avoir un comportement différent. Il n'est donc pas possible de prévoir le comportement réel de ce générateur.

Par exemple, sur [Coliru](#), à chaque fois que le programme est relancé, il va générer la même suite de nombres aléatoires. Le même code sur [Ideone](#) va produire des nombres aléatoires différents à chaque fois que le programme est lancé.

Pour savoir si le générateur non déterministe est effectivement non déterministe ou pas, il faut afficher l'entropie en utilisant la fonction `entropy` :

main.cpp

```
#include <iostream>  
#include <random>  
  
int main() {  
    std::random_device rd;  
    std::cout << "Entropy: " << rd.entropy() << std::endl;  
}
```

Si la valeur affichée est nulle, alors le générateur est déterministe.

Le problème de ce générateur est qu'il n'est pas possible de prouver qu'il a un comportement parfaitement aléatoire, qu'on ne peut prédire aucune caractéristique mathématique sur la série de nombres générés. On préférera donc généralement les générateurs de nombres aléatoires

déterministes.

## Générateurs déterministes

Un générateur déterministe produit des nombres aléatoires selon un algorithme complexe donné. Ce type de générateur prend en argument une graine (*seed* en anglais), qui définit la série de nombres aléatoires à générer. Utiliser deux fois la même graine produit la même série de nombres.

## Graine d'un générateur

Dans de nombreux cas, on ne souhaite pas utiliser une graine en particulier, on voudra avoir une série différente de nombres aléatoires, à chaque fois qu'on lance une application. Par exemple :

- dans un jeu, pour que les monstres aient un comportement différent à chaque partie, sans que l'on puisse prédire ce qu'ils vont faire ;
- dans un programme testant la qualité d'un code (test unitaire), pour avoir des valeurs qui changent à chaque fois que l'on relance les tests ;
- dans un programme d'encodage de données, pour que l'on ne puisse pas prédire l'encodage utilisé.

Mais dans certains cas, il sera intéressant d'utiliser une graine connue. Un exemple classique est un programme de test de code, qui détecte une erreur en utilisant une série spécifique de nombres aléatoires. Pour reproduire l'erreur, on pourra générer la même série de nombres en utilisant la même graine.

Pour créer un générateur en utilisant une graine, il suffit de mettre cette graine comme argument lors de la création du générateur. (Note : il existe plusieurs générateurs déterministes, qui seront vu par la suite. Pour le début de ce cours, nous allons utiliser le générateur

```
std::default_random_engine).
```

```
    std::default_random_engine engin { seed };
```

Ce code crée une variable nommée `engin`, de type `std::default_random_engine` et initialisé avec la valeur `seed`.

Pour générer les nombres aléatoires, il faut ensuite appeler la variable `engin` comme une fonction, comme vous avez fait pour `random_device` :

main.cpp

```
#include <iostream>
#include <random>

int main() {
    std::default_random_engine engin { 123 }; // création
    du générateur

    std::cout << engin() << std::endl; // génération
    d'un nombre aléatoire
    std::cout << engin() << std::endl; // génération
    d'un nombre aléatoire
    std::cout << engin() << std::endl; // génération
    d'un nombre aléatoire
}
```

affichera :

```
2067261
384717275
2017463455
```

Remarque : cette série de nombres est déterministe, ce qui veut dire que si vous utiliser le même générateur et la même graine que dans le code, vous devriez avoir exactement la même série de nombres aléatoires (contrairement au `random_device`, pour lequel vous pouvez avoir un résultat différent que celui donné dans le cours).

Pour générer une série en utilisant une graine qui change à chaque fois que le programme est lancé, il est classique d'initialiser le générateur

avec le temps (ce qui garantit que la graine change à chaque fois). Par exemple :

main.cpp

```
#include <iostream>
#include <random>
#include <chrono>

int main() {
    auto const seed = std::time(nullptr);
    std::cout << seed << std::endl;
    std::default_random_engine engin { seed };
}
```

La fonction `std::time` retourne le temps passé depuis une date fixée (généralement le 1er janvier 1970), en secondes.

Encore une fois, on peut remarquer des “failles” dans le système de génération des nombres aléatoires. Du fait que la graine change toutes les secondes, cela veut dire que si on génère deux graines trop rapidement, elles seront identiques.

De même, si on sait à peu près quand un nombre aléatoire a été généré, on peut retrouver plus facilement la graine qui a été utilisée et la série de nombres aléatoires.

Dans un programme critique, par exemple en programme d'encodage de données importantes, cela constitue des failles de sécurité. On voit ici qu'il est finalement assez facile, si on ne fait pas attention, de rendre prédictible un générateur et donc de créer des failles de sécurité dans un programme.

**L'utilisation d'un générateur aléatoire de qualité n'est pas suffisant, il faut aussi l'utiliser correctement.**

Pour des applications qui nécessite un niveau de sécurité élevé, on utilisera à la place des générateurs dédiés à la cryptographie.

## Les différents générateurs

Plusieurs générateurs déterministes sont proposés dans la bibliothèque standard du C++. Ces générateurs sont définies par l'algorithme utilisé et par les valeurs utilisées pour le configurer. Pour comprendre cela, regardons un peu plus en détail la documentation : [Pseudo-random number generation](#).

Trois algorithmes de génération sont utilisé. La documentation donne des liens vers les pages de Wikipédia décrivant ces algorithmes.

- `std::linear_congruential_engine` : algorithme congruentiel linéaire ;
- `std::mersenne_twister_engine` : algorithme de Mersenne Twister ;
- `std::subtract_with_carry_engine` : algorithme de soustraction avec retenue.

Il est très probable que vous ne connaissiez pas ces algorithmes... et cela ne va pas changer tout de suite :) Le but de ce cours est de vous expliquer comment utiliser les générateurs aléatoires et lire la documentation, pas d'expliquer chaque algorithme en détail. Pour cela, il vous faudra lire les documentations, voire lire un cours de cryptographie.

Il existe ensuite trois adaptateurs, qui permettent de modifier un autre algorithme. Les adaptateurs de la bibliothèques standard sont :

- `std::discard_block_engine` ;
- `std::independent_bits_engine` ;
- `std::shuffle_order_engine`.

Et pour terminer, les différents générateurs proposés par défaut, à partir de ces algorithmes et adaptateurs :

- `std::minstd_rand0` ;
- `std::minstd_rand` ;

- `std::mt19937` ;
- `std::mt19937_64` ;
- `std::ranlux24_base` ;
- `std::ranlux48_base` ;
- `std::ranlux24` ;
- `std::ranlux48` ;
- `std::knuth_b`.

Ces générateurs s'utilisent de la même façon que le générateur `std::default_random_engine` cité précédemment. Par exemple, pour `std::ranlux48_base`, avec une graine fixe :

`main.cpp`

```
#include <iostream>
#include <random>

int main() {
    std::ranlux48_base engin { 123 };           // création
du générateur

    std::cout << engin() << std::endl;          // génération
d'un nombre aléatoire
    std::cout << engin() << std::endl;          // génération
d'un nombre aléatoire
    std::cout << engin() << std::endl;          // génération
d'un nombre aléatoire
}
```

affiche :

```
52617863149155
181669238551779
228994550894105
```

### Comment générer un nombre aléatoire en pratique ?

Au final, cela fait beaucoup d'algorithmes à utiliser, vous ne savez probablement pas quel algorithme utiliser parmi tous les générateurs proposés. La réponse est simple :

- si cela n'a pas d'importance, uniquement  
`std::default_random_engine` ;
- si c'est important : uniquement les algorithmes que vous avez étudié et que vous comprenez.

## Générateurs de distributions statistiques

### Rappel sur les distributions statistiques

Dans les générateurs précédents, chaque valeur possible a la même chance d'apparaître (équiprobable). Dans de nombreuses situations (probablement dans la majorité des situations), on a besoin de restreindre le domaine de définition des nombres générés (par exemple, générer des nombres entre 0 et 10) ou d'avoir des probabilités différentes pour chaque valeur (par exemple, pour des nombres compris entre 0 et 1, plus une valeur est proche de 0, plus elle aura de chance d'apparaître).

Par exemple, si vous avez un jeu de 32 cartes et que vous souhaitez en tirer une au hasard, il faudra générer un nombre compris en 1 et 32. Si vous avez un personnage dans un jeu vidéo contrôlé aléatoirement, qui ne peut se déplacer que dans quatre directions, vous devrez générer un nombre aléatoire qui ne peut prendre par exemple que les valeurs 0 (monter), 1 (descendre), 2 (aller à droite) ou 3 (aller à gauche). Ce type de distribution est uniforme, chaque valeur à la même probabilité d'apparaître.

D'autres exemples. Si vous créer un jeu de simulation de courses de voiture et que le joueur tourne de  $10^\circ$ , dans le monde réel, ça ne sera pas exactement  $10^\circ$ , mais  $10^\circ$  plus ou moins une erreur (due à l'imprécision du volant, des imperfections de la route, etc). Dans le jeu, pour avoir plus de réalisme, on peut modifier l'angle avec une erreur, pour obtenir par exemple  $10.2^\circ$  ou  $9.9^\circ$ . On pourra avoir des valeurs très différentes de  $10^\circ$ , par exemple  $12^\circ$  ou  $8.5^\circ$ , mais cela sera beaucoup plus rare. Pour simuler cela, on va utiliser une distribution normale.

## Les générateurs standard de distribution

La bibliothèque standard du C++ contient d'autres distributions que les distributions uniformes et normale. Vous pouvez avoir le détail dans la [documentation](#) pour les fonctions à utiliser et sur [Wikipédia](#) pour les explications sur les propriétés des distributions. Dans la majorité des cas, les deux distributions citées seront suffisantes, nous verrons cela dans la suite du cours.

Il existe deux distributions uniformes et une normale :

- `std::uniform_int_distribution` pour générer des nombres entiers selon une loi uniforme ;
- `std::uniform_real_distribution` pour générer des nombres réels selon une loi uniforme ;
- `std::normal_distribution` pour générer des nombres réels selon une loi normale.

Ces générateurs ne sont pas à proprement parlé des générateurs de nombres aléatoires. Ils prennent en fait l'un des générateurs aléatoires vu précédemment pour générer un nombre respectant une loi donnée. Ce ne sont pas non plus des adaptateurs, comme ceux vu précédemment, puisqu'ils ne modifient pas le comportement des générateurs aléatoires. Ces générateurs de distribution prennent simplement un générateur en paramètre et l'utilise en interne pour générer des nombres.

## Initialisation

La création d'un générateur de distribution se fait en deux étapes :

- l'initialisation, permet de définir les paramètres de la distribution ;
- la génération, en utilisant un générateur aléatoire.

Pour initialiser un générateur de distribution, il faut créer une variable en utilisant le type de générateur que vous souhaitez, et en donnant les paramètres de la distribution. Ces paramètres varient selon le type de générateur :

- valeurs minimale et maximale pour les générateurs de loi uniforme ;
- moyenne et déviation standard pour le générateur de loi normale.

Le code C++ correspond est relativement similaire à ce que vous avez déjà vu.

main.cpp

```
#include <random>
#include <iostream>

int main() {
    std::uniform_int_distribution<> uniform_int(0, 10);
// loi uniforme sur des entiers
    std::uniform_real_distribution<> uniform_real(-1, 1);
// loi uniforme sur des réels
    std::normal_distribution<> normal(5, 1);
// loi normale
}
```

Vous avez déjà rencontré la notation avec chevrons < et > dans ce cours. Dans les codes précédents, un paramètre était placé entre ces chevrons, mais ce n'est pas une obligation ici. Si les chevrons ne contiennent rien, cela signifie que le paramètre par défaut est utilisé (voir la documentation pour connaître les paramètres par défaut. Ici, les paramètres par défaut sont `int` pour la loi uniforme sur les entiers et `double` pour les deux autres lois). Vous pouvez utiliser un autre paramètre :

main.cpp

```
#include <random>
#include <iostream>

int main() {
    std::normal_distribution<> double_normal(5, 1); //
```

```
loi normale utilisant des double
    std::normal_distribution<float> float_normal(5, 1); // 
loi normale utilisant des float
}
```

Pour rappel, `double` est un type de nombres réels double précision (généralement 64 bits) et `float` est un type de nombres réels simple précision (généralement 32 bits).

## Note importante sur les types

Il faut bien faire la différence entre les différents types d'arguments utilisés pour initialiser une variable de type générateur de distribution.

main.cpp

```
std::normal_distribution<float> normal(5, 1);
```

Le type `float` permet de préciser le type `std::normal_distribution`. Les nombres générés seront de type `float`, cette information est prise en compte lors de la compilation :

main.cpp

```
std::normal_distribution<float> normal(5, 1);
const std::string value { normal(engine) }; // erreur,
impossible de convertir un float en std::string
```

Le type passé en paramètre fait partie intégrante du type final. Le type de `normal` n'est pas `std::normal_distribution`, mais bien `std::normal_distribution<float>`. Cela signifie par exemple que les types `std::normal_distribution<>` (dont le type par défaut est `double`) et `std::normal_distribution<float>` ne sont pas compatibles entre eux :

main.cpp

```
std::normal_distribution<float> d1(5, 1);
std::normal_distribution<float> d2 { d1 }; // ok, d1 et d2
sont de même type
std::normal_distribution<double> d3 { d1 }; // erreur, d1 et
```

*d3 ne sont pas compatibles*

Les valeurs 5 et 1 correspondent aux paramètres de la distribution. Elles ne sont pas utilisées lors de la compilation, mais uniquement lors de l'exécution.

La séparation de la phase de compilation (*compile-time*) et de la phase d'exécution (*runtime*) est une notion importante en C++. Il est nécessaire de bien comprendre ce qui est fait à chaque étape. Vous verrez cette notion régulièrement.

## Génération

La dernière étape est la génération d'un nombre aléatoire selon une distribution. Les générateurs de distribution sont des objets-fonctions, comme les générateurs aléatoires. Pour rappel, cela signifie que l'on peut appeler la variable comme une fonction :

main.cpp

```
std::normal_distribution<float> normal(0, 1); //  
initialisation  
auto const value = normal(engine); // génération
```

Contrairement aux générateurs aléatoires qui ne prennent pas d'argument, les générateurs de distribution prennent en argument le générateur aléatoire à utiliser. À chaque fois que le générateur de distribution est appelé, il appelle en interne le générateur aléatoire pour générer un nombre aléatoire, puis l'adapte en fonction de la distribution.

Pour résumer l'utilisation des générateurs de distributions, ceux-ci utilisent trois sources d'information pour générer un nombre :

- le type de valeur générée (le type passé en argument entre chevrons, qui sera le même type que la variable `value` avec `auto`) ;
- le type de distribution et ses paramètres (définis lors de

- l'initialisation du générateur de distribution) ;
- le générateur aléatoire utilisé (défini lors de la génération des nombres).

## Pour résumer

Au final, vous devriez avoir un code similaire à cela pour générer un nombre aléatoire (à adapter selon le type de distribution) :

main.cpp

```
#include <iostream>
#include <random>
#include <chrono>

int main() {
    auto const seed = std::time(nullptr); // génération de la graine
    std::default_random_engine engin { seed }; // générateur aléatoire
    std::normal_distribution<float> normal(0, 1); // générateur de distribution
    std::cout << normal(engin) << std::endl; // génération
}
```

## Travaux pratiques

1. Donner une série de nombres aléatoires, produit avec un algo/graine (donné ou non), une graine comprise entre 0 et 100. Essayer de retrouver la graine/algo probablement à faire plus tard, après les boucles et tests
2. exos : proposer des exos avec fonctions de calculs et affiche déjà écrit

Ce chapitre est un peu spéciale. Le but est de vous présenter des techniques... qu'il ne faut pas utiliser ! Ou tout au moins, qu'il faut fortement limiter leur utilisation. La raison est que cela produit généralement une dette technique et diminue la qualité du code (moins maintenable, moins évolutif et plus difficilement testable).

Une dette technique est le surcoût de travail nécessaire lorsque vous faites des mauvais choix dans votre code. La conséquence, par exemple, est qu'au lieu d'implémenter une fonctionnalité en une semaine, vous l'implémenterez en deux semaines. Plus le temps passe et plus les problèmes augmenteront dans la dette technique, jusqu'à ce que vous corriger ces problèmes.

Donner des explications détaillées sur pourquoi les globaux et assimilés posent problème est assez difficile, puisque cela demande de l'expérience et du recul, de préférence sur des projets conséquents. Il va falloir dans un premier temps accepter cette règle : "interdit d'utiliser les globaux !"

# Variables globales et statiques

## Variables globales

Dans les chapitres precedants, vous avez vu uniquement les variables locales, qui etait definie a partir de leur declaraiton et n'etait plus valide a la fin du bloc.

```
int main() {
    int i { 123 };                                // déclaration de i
    {
        int j { 456 };                            // déclaration de j
        std::cout << j << std::endl;             // j est valide
    }                                              // destruction de j
    std::cout << i << std::endl;                 // i est valide
```

```
}
```

```
// destruction de i
```

Une variable *globale* est une variable dont la portée n'est pas limitée à un bloc, mais peut s'étendre à tout le programme.

Un premier exemple simple est la déclaration d'une variable en dehors de la fonction `main`.

```
int i {};  
  
int main() {  
    ++i;  
    std::cout << i << std::endl;  
    ++i;  
    std::cout << i << std::endl;  
}
```

affiche :

```
1  
2
```

## Variables statiques

Une variable *statique* est une variable dont la valeur est conservée.

```
void f() {  
    static int i {};  
    ++i;  
    cout << i << endl;  
}  
  
int main() {  
    f();  
    f();  
    f();  
}
```

affiche :

1  
2  
3

**[Chapitre précédent](#)** **[Sommaire principal](#)** **[Chapitre suivant](#)**

# Les collections de données

L'une des difficultés principales que vous aurez à résoudre en tant que développeur est de définir comment les données doivent être organisées en mémoire (structures de données) et comment ces données doivent être traitées (algorithmes). Il est intéressant, dès que cela est possible, de séparer ces deux aspects du problème dans des classes différentes, pour renforcer la réutilisabilité du code que vous écrivez. La bibliothèque standard du C++ est organisée de cette façon, avec d'un côté des classes de structures de données (`std::string` pour les chaînes, `std::vector` pour les tableaux, etc.) et des fonctions libres pour le traitement des données (regroupées dans le fichier d'en-tête `<algorithms>`).

Pour qu'un code soit réutilisable au maximum, l'idéal serait que n'importe quelle structure de données soit compatible avec n'importe quel algorithme. Dit autrement, cela veut dire que si vous créez une structure de données, elle doit être utilisable avec n'importe quel algorithme de la bibliothèque standard et que si vous créez un algorithme, il doit être utilisable avec n'importe quelle structure de données de la bibliothèque standard. Pour cela, les structures de données sont conçues autour du concept de collection, que vous allez voir dans ce chapitre.

## Les chaînes comme collection de caractères

Les chaînes de caractères `string`, que vous avez manipulées dans les chapitres précédents, sont un exemple de collection. Une collection est un ensemble d'éléments (`string` est un ensemble de caractères) qui respecte les propriétés suivantes :

- accéder au début de la collection en utilisant la fonction `begin` (en fonction libre ou en fonction membre) ;
- accéder à la fin de la collection en utilisant la fonction `end`

- (également en fonction libre ou en fonction membre) ;
- respecter la notion “élément suivant”, c'est-à-dire que chaque élément d'une collection possède un et un seul élément suivant. Cette élément est accessible en utilisant l'opérateur `++` ou la fonction libre `next`.

Il est alors possible de parcourir une collection du début à la fin en passant d'un élément au suivant. On parle d'accès séquentiel dans ce cas.

Pour rappel, une fonction membre s'écrit en utilisant l'opérateur `.` entre le nom d'une variable et la fonction membre. Une fonction libre s'applique sur une variable en la donnant en argument entre parenthèses. Les deux syntaxes sont identiques en termes de comportement du programme.

```
std::string const s {};  
  
// début de la collection  
s.begin(); // fonction membre  
std::begin(s); // fonction libre  
  
// fin de la collection  
s.end(); // fonction membre  
std::end(s); // fonction libre
```

Les algorithmes de la bibliothèque standard s'appliquent sur une collection en donnant en argument le premier et le dernier élément de la collection sur laquelle on souhaite appliquer l'algorithme. Par exemple, pour appliquer l'algorithme `std::sort` (qui permet de trier les éléments d'une collection), on écrira :

```
#include <iostream>  
#include <string>  
#include <algorithm>  
  
int main() {  
    std::string s { "azerty" };  
    std::sort(std::begin(s), std::end(s));  
    std::cout << s << std::endl;
```

```
}
```

affiche :

```
aertyz
```

L'inclusion de l'en-tête `<algorithm>` permet d'utiliser les algorithmes de la bibliothèques standard.

On peut se demander pourquoi passer en argument des fonctions le début et la fin d'une collection, au lieu de passer la collection complète en argument (en écrivant par exemple `std::sort(s)`). La raison est que cela permet d'utiliser les algorithmes également sur une partie d'une collection au lieu de la collection complète. En effet, les algorithmes prennent en paramètre le début et la fin sur lesquels s'appliquent l'algorithme, qui ne sont pas forcément les premier et dernier éléments de la collection. Il est ainsi possible de trier une chaîne à partir du troisième élément en écrivant `std::sort(std::begin(s)+3, std::end(s))`.

Attention, cette syntaxe nécessite que la chaîne contienne au moins trois caractères, sous peine d'obtenir un comportement indéterminé. La manipulation des collections élément par élément sera vu dans un prochain chapitre.

Il est également possible de parcourir une collection du dernier élément au premier élément en utilisant les fonctions `rbegin` (*reverse begin*) et `rend` (*reverse end*). Ces fonctions sont utilisables uniquement en fonctions membres.

Par exemple, il est possible de trier dans l'ordre inverse de cette manière :

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

int main() {
```

```
    std::string s { "azerty" };

    std::sort(std::begin(s), std::end(s));
    std::cout << s << std::endl;

    std::sort(s.rbegin(), s.rend());
    std::cout << s << std::endl;
}
```

affiche :

```
aertyz
zytrea
```

## Templates et généricité

Les données que vous aurez à manipuler ne se limiteront pas aux chaînes de caractères. Heureusement, la bibliothèque standard fournit d'autres structures de données, qui permettent de manipuler n'importe quel type de données dans une collection.

La [page de documentation des conteneurs de la bibliothèque standard](#) liste une quinzaine de types différents de structures de données. Nous n'allons voir dans ce chapitre que les tableaux de type `std::vector` et `std::array`, les autres conteneurs seront vus dans des prochains chapitres. La différence entre ces deux types de tableau est que `std::array` représente un tableau dont le nombre d'éléments (on parle de la "taille du tableau") est fixé à la compilation, tandis que le nombre d'éléments de `std::vector` peut être modifié durant l'exécution.

Les conteneurs de données sont des classes, au même titre que `std::string`. Pour déclarer un tableau, on utilisera donc la même syntaxe que pour déclarer une chaîne. Par exemple, pour créer un tableau vide d'entiers, on écrira :

```
std::vector<int> const integers {};
```

Dans ce code, `integers` est le nom de la variable qui contient le tableau d'entiers (`integers` signifie "des entiers" - prenez l'habitude d'écrire vos

codes en anglais, en particulier pour nommer vos variables, fonctions et classes). Le type de tableau est `vector<int>`, que l'on peut lire directement comme étant un tableau (`vector`) d'entiers (`int`).

La syntaxe de `std::vector` est un peu particulière. C'est ce que l'on appelle une classe template (ou classe générique). Vous avez déjà rencontré cette syntaxe dans le chapitre [Obtenir des informations sur les types](#), pour la classe `std::numeric_limits`.

Imaginons que l'on vous demande de créer un telle classe, qui permette de manipuler des tableaux de données. Une première approche serait d'écrire une classe représentant un tableau d'entier `vector_int`, puis de la dupliquer et de modifier cette classe pour manipuler un tableau de doubles `vector_double`. Et ainsi de suite pour chaque type que l'on souhaite manipuler.

Cependant, ce type d'approche est très problématique. En premier lieu, cela implique de copier-coller plusieurs fois le code et de le modifier selon le type de données contenu dans le tableau. Si vous corrigez un problème dans l'un des types de tableau, il ne faut pas oublier de corriger également les autres tableaux. C'est un risque d'erreur, le code est moins maintenable.

Le second problème est qu'il ne sera pas possible de prendre en charge tous les types. Un utilisateur de votre code qui crée ses propres types devra copier votre code et le modifier pour prendre en charge ses propres types. C'est une autre source d'erreur importante, le code n'est pas évolutif.

Les templates permettent d'écrire des classes et fonctions dont le comportement dépendra d'informations données à la compilation. Les informations données peuvent être un type ou une valeur entière. Les classes `vector` et `std::array` sont des exemples de template. Ces classes sont conçues pour accepter n'importe quel type de données, en particulier les types que vous créerez. Ce type d'approche présente de nombreux avantages :

- maintenabilité : le code n'est pas dupliqué. S'il doit être corrigé, il suffit de corriger une seule implémentation et tous les codes

utilisant cette classe seront corrigés en même temps.

- évolutivité : si vous créez un nouveau type, vous pouvez l'utiliser dans un template (à partir du moment où vous respectez les conditions d'utilisation de ce template).

Les arguments template sont donnés entre chevrons `<>` après le nom de la classe ou de la fonction template. S'il y a plusieurs arguments, ils sont séparés par des virgules.

```
template_class<int>           // classe template avec 1
argument
another_template_class<int, float> // classe template avec 2
arguments

template_function<int>()        // une fonction template
```

Attention aux arguments passés dans les appels de classes et fonctions. Vous avez donc deux types d'arguments (par exemple pour une fonction) :

```
foo<arg1, arg2>(arg3, arg4);
```

Arguments	Type	Valeurs possible	Évaluation
<code>arg1</code> et <code>arg2</code>	arguments de template	Types et valeurs entières	Lors de la compilation
<code>arg3</code> et <code>arg4</code>	arguments de fonction	Variables et littérales	Lors de l'exécution

On voit bien que ces deux types d'arguments répondent à des besoins différents, acceptent des valeurs différentes et ne sont pas évalués en même temps. Il faut faire attention de ne pas les confondre. L'utilisation de deux types d'arguments peut sembler être une complexité inutile, mais en fait, c'est l'une des forces du C++. Cela permet de penser les problèmes à résoudre en termes de ce qui peut être évalué à la compilation ou ce qui peut l'être à l'exécution. Et donc de pouvoir écrire des abstractions très complexes (mais très simples à utiliser), sans perte de performance lors de l'exécution.

Vous apprendrez par la suite différentes approches pour écrire du code générique performant, mais dans un premier temps, voyons comment utiliser de telles classes template.

## Les tableaux comme collections

Les tableaux étant des collections, il est possible d'utiliser les fonctions `begin` et `end` pour obtenir respectivement le début et la fin d'un tableau. Ces fonctions sont également utilisables comme fonctions membres ou fonctions libres. Par exemple, avec la fonction de tri `sort` :

```
std::sort(a.begin(), a.end());           // fonctions membres  
std::sort(std::begin(a), std::end(a)); // fonctions libres
```

Au final, vous n'avez besoin que de connaître ces deux fonctions, `begin` et `end`, pour utiliser une collection avec les algorithmes de la bibliothèque standard. Il existe d'autres types de collection que `std::array` et `std::list`, n'hésitez pas à parcourir la page de documentation correspondante aux [conteneurs de données](#) et à en tester quelques uns.

## Déclarer et initialiser des tableaux

La classe `std::vector` prend un seul argument template, qui est le type de données que le tableau doit contenir. La classe `std::array` prend deux arguments : le type de données et le nombre d'éléments que doit contenir le tableau. Par exemple, pour créer des tableaux :

```
std::vector<int> const integers {};  
std::array<float, 5> const floats {};
```

Par défaut, `std::vector` ne contient pas d'éléments lors de l'initialisation de la variable `integers`. Comme `std::vector` est un tableau de taille redimensionnable, vous pourrez ajouter des éléments par la suite.

Au contraire, `std::array` est initialisé avec cinq éléments dans le code précédent. Il est possible de créer une `std::array` avec aucun élément, mais comme il n'est pas possible d'ajouter des éléments, l'intérêt est limité. Cela i

Remarque : pour rappel, le but de ce cours n'est pas de vous présenter toutes les syntaxes possibles, mais celles qui sont utiles à connaître pour comprendre les bases du C++. Il est possible d'utiliser d'autres syntaxes pour les classes `std::vector` et `std::array`, mais la compréhension de ces syntaxes nécessite des connaissances plus avancées en C++.

Il est possible d'initialiser `std::vector` avec un nombre déterminé d'éléments, comme pour `std::array`. Pour cela, il faut donner ce nombre entre parenthèses :

```
std::vector<int> const integers(5); // crée un tableau  
contenant 5 éléments
```

On retrouve ici la différence de syntaxe entre argument template et argument de fonction. Pour `std::array`, la taille du tableau (nombre d'éléments) est fixée à la compilation, c'est donc un argument template (entre chevrons). Pour `std::vector`, la taille est variable durant l'exécution, c'est donc un argument de fonction (entre parenthèses).

Dans les deux cas, il est possible de connaître la taille d'un tableau en utilisant la fonction membre `size`.

main.cpp

```
#include <iostream>  
#include <vector>  
#include <array>  
  
int main() {  
    std::vector<int> const integers(5);  
    std::array<float, 3> const floats {};  
    std::cout << "Size of vector is " << integers.size() <<  
    std::endl;
```

```
    std::cout << "Size of array is " << floats.size() << std  
    ::endl;  
}
```

affiche :

```
Size of vector is 5  
Size of array is 3
```

Vous pouvez donner des valeurs entre les crochets pour initialiser le tableau. Une liste de valeurs (*initializer-list*) s'écrit entre crochets, avec des virgules comme séparateurs.

```
std::vector<int> const integers { 0, 1, 2, 3 };  
std::array<int, 5> const floats { 0, 1, 2, 3 };
```

Remarque : pour rappel, l'utilisation des espaces est libre en C++. Le critère principal doit être la lisibilité du code. Il est tout à fait possible d'écrire les listes de valeurs sous forme compacte `{0,1,2,3}`. Peu importe comment vous souhaitez présenter vos codes, mais essayez de respecter des conventions d'écriture (même informelles) dans vos codes, pour que les syntaxes soient homogènes.

Dans ce code, les deux tableaux sont initialisés avec les valeurs 0 à 3. `std::vector` étant un tableau dynamique, sa taille est initialisée en fonction du nombre d'éléments donné dans la liste. Ce qui signifie que `integers` contient 4 éléments dans ce code. Au contraire, la taille de `std::array` est fixée par le second argument template et non par le nombre d'éléments dans la liste. `floats` contient donc 5 éléments, les 4 premiers correspondant aux valeurs données dans la liste, le dernier est initialisé par défaut (avec la valeur 0 donc).

## Modifier la taille d'un tableau

Comme `std::array` est un tableau de taille fixée à la compilation, il n'est pas possible d'ajouter ou de supprimer des éléments lors de l'exécution du programme. Cette partie ne s'applique donc qu'aux

tableaux de type `vector`.

Les tableaux de type `std::vector` peuvent être manipulés comme des piles. Une pile est une collection qui permet de lire, supprimer (*pop*) ou ajouter (*push*) des éléments à la fin de la collection (*back*). Les noms des fonctions membres sont assez simples à comprendre (même si vous ne comprenez pas très bien l'anglais) :

- `back` permet d'accéder au dernier élément, pour le lire ou le modifier ;
- `push_back` permet d'ajouter (*push*) un élément à la fin de la pile ;
- `pop_back` permet de retirer (*pop*) l'élément qui se trouve à la fin de la pile.

Attention, il est nécessaire que la collection ne soit pas vide pour appeler `pop_back`, sinon cela provoque un comportement indéfini (*undefined behavior*). Aucun message d'erreur (à la compilation ou à l'exécution) n'est affiché si vous utilisez `pop_back` sur un tableau vide.

Notez aussi qu'il est possible de lire et modifier le premier élément d'un tableau en utilisant la fonction membre `front`.

#### main.cpp

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

int main() {
    std::vector<int> v { 1, 2, 3 };
    // v contient { 1, 2, 3 }

    v.push_back(4);
    std::cout << v.back() << std::endl;
    // v contient { 1, 2, 3, 4 }

    v.back() = 5;
    std::cout << v.back() << std::endl;
    // v contient { 1, 2, 3, 5 }

    v.pop_back();
    v.pop_back();
```

```
    std::cout << v.back() << std::endl;
    // v contient { 1, 2 }
}
```

affiche :

```
4
5
2
```

Il existe également une collection qui permet d'être agrandie ou rétrécie depuis le premier et le dernier élément : `std::deque`. Son utilisation est assez similaire à `std::vector`, vous verrez dans les exercices d'application comment l'utiliser.

## Afficher le contenu d'une collection

Pour accéder à chaque élément individuellement, il est nécessaire de faire intervenir la notion d'itérateur et de boucles, ce qui sera vu dans la suite ce cours. Cependant, il est intéressant de voir rapidement la syntaxe permettant d'afficher le contenu d'une collection. Pour cela, vous pouvez utiliser la syntaxe suivante :

```
for (auto const value: integers )
    std::cout << value << std::endl;
```

Ce code se lit de la façon suivante : "pour (for) chaque élément d'une collection (`integers` dans ce code), afficher la valeur avec `cout`". Ce code est relativement générique, vous pouvez l'utiliser avec n'importe quelle collection de données dont le type peut être affiché avec `std::cout`.

main.cpp

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

```
int main() {
    std::vector<int> const i { 1, 2, 3, 4 };
    for (auto const value: i)
        std::cout << value << std::endl;
    std::cout << std::endl;

    std::vector<double> const d { 1.2, 3.4, 5.6, 7.8 };
    for (auto const value: d)
        std::cout << value << std::endl;
    std::cout << std::endl;

    std::vector<std::string> const s { "un", "deux", "trois",
"quatre" };
    for (auto const value: s)
        std::cout << value << std::endl;
    std::cout << std::endl;
}
```

affiche :

```
1
2
3
4

1.2
3.4
5.6
7.8

un
deux
trois
quatre
```

Bien sûr, comme une chaîne de caractères `std::string` peut être vue comme une collection, il est également possible d'utiliser cette syntaxe pour afficher les caractères individuellement.

main.cpp

```
#include <iostream>
```

```
#include <string>

int main() {
    std::string const s { "hello, wolrd!" };
    for (auto const value: s)
        std::cout << value << std::endl;
}
```

affiche :

```
h  
e  
l  
l  
o  
,  
w  
o  
l  
r  
d  
!
```

[Chapitre précédent](#) [Sommaire principal](#) [Chapitre suivant](#)

# Les foncteurs

foncteur ? fonction objet ? opérateur ? autre ?

## Les foncteurs par défaut

Vous avez vu dans le chapitre précédent comment utiliser la fonction `std::sort` pour trier une collection :

```
std::vector<int> v { 1, 5, 2, 4, 3 };
std::sort(begin(v), end(v));

std::string s { "hello, world!" };
std::sort(begin(s), end(s));
```

Cet algorithme est dit “modifiant” puisqu'il modifie directement le conteneur sur lequel on applique la fonction.

Fondamentalement, cet algorithme fonctionne de la façon suivante : il parcourt les éléments de la collection et réalise des tests de comparaison par paire d'éléments. Pour faire cette comparaison, l'algorithme utilise l'opérateur de comparaison `<` sur les éléments. Par exemple, pour faire le tri d'un tableau d'entiers (`vector<int>`), l'algorithme réalise des comparaisons d'entiers (`valeur 1 < valeur 2`).

Dit autrement, cela veut dire que si on utilise un `vector<un_type>`, il faut que la comparaison `<` ait un sens pour ce type `un-type` (ce qui sera le cas avec la majorité des types de base du C++).

On dit que l'opérateur `<` est le prédictat utilisé par l'algorithme de tri `std::sort`. Plus généralement, un prédictat est une expression qui retourne un booléen (`true` ou `false`). Les différents algorithmes de la bibliothèque standard n'utilisent pas tous l'opérateur `<`, certains utilisent l'opérateur d'égalité `==`, d'autres n'utilisent pas de prédictat.

Imaginons maintenant que l'on souhaite trier une collection dans l'ordre inverse, c'est-à-dire du plus grand au plus petit. Une première solution serait de trier dans l'ordre par défaut (plus petit au plus grand), puis d'inverser l'ordre des éléments. Une autre solution serait de réécrire un algorithme de tri (appelé `reverse_sort` par exemple) et qui trie dans l'ordre inverse (du plus grand au plus petit).

Ces deux solutions ne sont pas correctes en termes de C++ moderne. La première est inutilement plus compliquée (il faut écrire deux lignes au lieu d'une seule), la seconde demande de réécrire l'algorithme de tri.

## Les foncteurs de la bibliothèque standard

Heureusement, la bibliothèque standard a été conçue pour être le plus générique possible, suivant les principes de la programmation moderne. Pour cela, la majorité des algorithmes de la bibliothèque standard existe en deux versions. La première utilise les foncteurs par défaut, la seconde admet un argument supplémentaire permettant de fournir un foncteur personnalisé. Par exemple, la fonction `sort` peut s'utiliser avec le prédictat par défaut (utilisation de `<`) ou un foncteur personnalisé :

```
std::sort(begin(v), end(v)); // foncteur par
détail
std::sort(begin(v), end(v), un_foncteur); // foncteur
personnalisé
```

Les cas les plus génériques, comme trier du plus grand au plus petit, sont déjà implémentés dans la bibliothèque standard. Ces prédictats sont définis dans le fichier d'en-tête `<functional>`. Par exemple, pour trier du plus grand au plus petit, il est possible d'utiliser le prédictat `greater` ("plus grand que") de la façon suivante :

main.cpp

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
```

```
int main() {
    std::vector<int> v { 1, 5, 2, 4, 3 };
    std::sort(begin(v), end(v));
    for (auto const value: v)
        std::cout << value << std::endl;
    std::cout << std::endl;

    std::sort(begin(v), end(v), std::greater<int>());
    for (auto const value: v)
        std::cout << value << std::endl;
}
```

affiche :

```
1
2
3
4
5

5
4
3
2
1
```

Le prédictat `greater` est une classe template, il faut préciser le type que l'on souhaite comparer comme argument template (donc entre chevrons). Les parenthèses permettent d'instancier un objet à partir de la classe `greater`. Vous n'avez pas encore vu les classes, vous verrez cela en détail dans la partie sur la programmation objet. Pour le moment, le plus important est de se souvenir de la syntaxe.

Il est possible d'utiliser les foncteurs directement, sans passer par un algorithme. Pour cela, il faut créer un objet du type `std::greater`, puis l'appeler comme une fonction (d'où leur nom : foncteur = "function object") :

main.cpp

```
#include <iostream>
#include <functional>
```

```
int main() {
    std::greater<int> greater_operator {};
    std::cout << std::boolalpha;
    std::cout << greater_operator(1, 2) << std::endl;
    std::cout << greater_operator(3, 2) << std::endl;
}
```

Pour bien comprendre la différence entre les divers types de fonction, revoyons les multiples syntaxes pour créer une variable (nommée `object`) d'un type donné (`MyObject`) et sur laquelle nous appliquons une fonction :

```
MyObject object {};

foo(object); // fonction libre
object.foo(); // fonction membre
object(); // foncteur
```

Pour le moment, vous avez seulement vu comment utiliser ces types de fonction. Vous verrez dans les chapitres sur les fonctions et sur la programmation objet comment les créer. En pratique, il vous suffit de lire la documentation pour savoir quelle est la syntaxe correcte d'une fonction (sa signature, ses paramètres, savoir si c'est une fonction libre ou membre, etc.) Avec l'expérience, vous vous souviendrez des fonctions les plus utilisées et vous n'aurez plus besoin de lire la documentation pour vérifier la syntaxe. Un éditeur de code avec l'auto-complétion (qui affiche la signature des fonctions) peut également être très utile pour gagner du temps.

Vous pouvez consulter la liste des différents foncteurs de la bibliothèque standard dans la page de documentation [Function objects](#). Vous voyez dans cette page que les foncteurs sont rangés en plusieurs catégories :

- les opérations arithmétiques (*arithmetic operations*) : `plus`, `minus`, `multiplies`, `divides`, `modulus` et `negate` ;
- les comparaisons (*comparisons*) : `equal_to`, `not_equal_to`, `greater`, `less`, `greater_equal` et `less_equal` ;
- les opérations logiques (*Logical operations*) : `logical_and`,

`logical_or` et `logical_not` ;

- les opérations sur les bits (*Bitwise operations*) : `bit_and`, `bit_or`, `bit_xor` et `bit_not`.

composition de fonction, not1, not2, bind ?

Plus généralement, il est possible d'utiliser n'importe quelle fonction de la bibliothèque standard, à partir du moment où celle-ci respecte la signature attendue par un algorithme donné. En particulier, il est possible d'utiliser les fonctions de manipulations de caractères, définies dans le fichier d'en-tête `<cctype>` : **Null-terminated byte strings** (ou leur équivalent pour `wstring` : **Null-terminated wide strings**). Ces fonctions se divisent en deux catégories :

- les fonctions de test (*Character classification*) :

- `isalnum` (alphanumérique) ;
- `isalpha` (alphabétique) ;
- `islower` (minuscule) ;
- `isupper` (majuscule) ;
- `isdigit` (chiffre décimal) ;
- `isxdigit` (chiffre hexadécimal) ;
- `iscntrl` (caractère de contrôle) ;
- `isgraph` (caractère graphique) ;
- `isspace` (espace) ;
- `isblank` (caractère blanc) ;
- `isprint` (caractère affichable) ;
- `ispunct` (ponctuation).

- les fonctions de modification (*Character manipulation*) :

- `tolower` (convertie en minuscule) ;
- `toupper` (converti en majuscule).

La syntaxe pour utiliser les fonctions dans un algorithme est différente de celle pour les objets-fonctions. Il faut simplement mettre la nom de la fonction. Par exemple, pour convertir une chaîne de caractères en majuscule, il est possible d'utiliser l'algorithme `transform` (que vous verrez par la suite) et la fonction `toupper` :

### main.cpp

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

int main() {
    std::string s { "abcdef" };
    std::transform(begin(s), end(s), begin(s), toupper);
    std::cout << s << std::endl;
}
```

affiche :

```
ABCDEF
```

## Exercices

- trier avec d'autres prédictats
- combiner des prédictats

## Les fonctions lambdas

Si vous avez regardé la documentation des foncteurs de la bibliothèque standard, vous avez peut être compris que ce sont de simples classes. Vous apprendrez dans la partie sur la programmation orientée objet comment créer vos propres foncteurs (que l'on appelle aussi "function object"). Mais il est possible de créer des foncteurs plus simplement, en utilisant des fonctions lambdas.

Il est important que vous sachiez créer des fonctions, c'est un point fondamental en C++, vous les utiliserez dans tous vos codes. Et plus important, ce qui sera fondamental est de savoir découper correctement les problèmes complexes en fonctions plus simples. Ce chapitre ne sera pas suffisant pour étudier toutes les possibilités offertes par les fonctions, nous reviendrons dessus en détail par la suite. Cette partie se focalise sur

l'utilisation simple des fonctions lambdas avec les algorithmes de la bibliothèque standard.

Les fonctions lambdas sont une technique issue de la programmation fonctionnelle. Vous avez déjà utiliser des fonctions (membres ou libres) et vous avez déjà défini une fonction : la fonction `main`. Pour rappel, une fonction permet de réaliser une tâche particulière et est constituée de :

- un nom de fonction (`main`) ;
- des valeurs optionnelles (arguments) en entrée et sortie ;
- un corps de la fonction, entre crochets, qui contient les instructions à exécuter lorsque l'on appelle la fonction ;
- il n'est pas possible de définir une fonction dans une autre fonction.

La fonction `main` est une fonction particulière :

- elle est obligatoire ;
- elle ne peut être appelée que par le système (jamais par l'utilisateur) ;
- elle doit être unique ;
- sa signature (c'est-à-dire sa façon d'être écrite) est définie par la norme C++.

```
int main() {  
    // corps de la fonction main  
}
```

### Programmation fonctionnelle

Une fonction lambda est une fonction particulière. Elle n'a pas de nom (elle est dite anonyme) et peut être déclarée dans le corps d'une autre fonction. A part cela, elle se comporte comme une fonction classique et peut recevoir des arguments, retourner une valeur. Il existe plusieurs façon d'écrire des fonctions lambdas, mais comme nous les utilisons ici dans les algorithmes de la bibliothèques standard, il est nécessaire de respecter la signature imposée par les algorithmes.

La définition d'une fonction lambda se décompose en trois parties :

```
[capture](paramètres){corps de la fonction}
```

La capture et les paramètres permettent de passer des informations dans la fonction. Pour le moment, nous n'utiliserons pas la capture, uniquement les paramètres. Le corps de la fonction contient le code à exécuter lorsque la fonction est appelée. Les différents algorithmes n'attendent pas la même signature pour les foncteurs, vous verrez dans les prochains chapitres chaque algorithme et la signature de foncteur en détail. Nous verrons ici que l'utilisation de `std::sort` comme exemple de fonction lambda.

Vous connaissez déjà la signature que doit avoir une fonction lambda avec `sort`. En effet, vous savez que cet algorithme peut s'utiliser avec les opérateurs de comparaison `<` ou `>` (`greater`) par exemple. Et vous savez que ces opérateurs prennent deux valeurs, les comparent et retourne une valeur booléenne.

Les valeurs en entrée (paramètres) s'écrivent entre les parenthèses dans la fonction lambda. Chaque paramètre s'écrit en indiquant un type et un nom et les paramètres sont séparés par des virgules. Par exemple, pour écrire une fonction lambda qui prend deux entiers en entrée, on écrit :

```
[](int a, int b){}
```

Pour simplifier l'écriture des fonctions lambdas et avoir un code plus générique, il est possible d'utiliser l'inférence de type et le mot-clé `auto` (voir le chapitre [L'inférence de type](#) pour rappel). Il est classique de nommer les paramètres `rhs` (*right hand side*, "côté droit") et `lhs` (*left hand side*, "côté gauche"). La fonction devient donc :

```
[](auto lhs, auto rhs){}
```

Il faut également que cette fonction lambda retourne une valeur booléenne. Pour cela, il faut utiliser le mot-clé `return`, suivi d'une valeur ou d'une expression. Le type renvoyée est déduit automatiquement en fonction de l'expression écrite après `return`. Par exemple, si on écrit :

```
[]{() { return true; } // retourne une valeur booléenne
[]{() { return 123; } // retourne une valeur entière
[]{() { return 12.4 * 45.6; } // retourne une valeur réelle
(double)}
```

Le corps de la fonction lambda (entre les crochets) peut contenir plusieurs lignes (séparées par un point-virgule), déclarer des variables, appeler d'autres fonctions, etc. Bref, vous pouvez mettre dans le corps d'une fonction lambda toutes les instructions que vous souhaitez.

Pour écrire une fonction lambda qui prend deux paramètres, compare les valeurs et retourne un booléen, on peut donc écrire :

```
[](auto lhs, auto rhs){ return lhs < rhs; }
```

La valeur renvoyée correspond à l'expression à droite du mot-clé `return`, donc `lhs < rhs`. Le résultat de cette expression est `true` ou `false`, c'est donc bien un booléen. Le code complet avec la fonction `sort` s'écrit :

```
std::sort(begin(v), end(v), [](auto lhs, auto rhs){ return
lhs < rhs; });
```

Ce code contient beaucoup d'informations sur une seule ligne, il vous faudra probablement un peu de temps pour réussir à repérer en un coup d'œil les différents éléments qui la compose. Avec l'expérience, cela ne posera plus de problèmes.

On peut alors facilement écrire une fonction lambda pour trier un tableau du plus petit au plus grand ou l'inverse :

main.cpp

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

int main() {
    std::vector<int> v { 1, 5, 2, 4, 3 };

    std::sort(begin(v), end(v), [](auto lhs, auto rhs){
```

```
return lhs < rhs; });
    for (auto const value: v)
        std::cout << value << std::endl;
    std::cout << std::endl;

    std::sort(begin(v), end(v), [](auto lhs, auto rhs){
return lhs > rhs; });
    for (auto const value: v)
        std::cout << value << std::endl;
}
```

affiche :

```
1
2
3
4
5

5
4
3
2
1
```

Remarque : ce code est bien sûr un code d'exemple pour illustrer l'utilisation des fonctions lambdas avec la fonction `sort`. En pratique, la première fonction lambda ne sert à rien, puisque cela reproduit le comportement par défaut de `sort`. Et la seconde reproduit le comportement du foncteur `greater`, elle n'est pas très utile non plus. Vous verrez dans les exercices et les chapitres suivants des exemples d'utilisation plus intéressants des fonctions lambdas avec les prédictats.

## Exercices

- trier selon la valeur absolue

## Cours, C++

# Introduction aux algorithmes standards

Même si les collections offrent plus de fonctionnalités que les concepts de base `begin` et `end`, il est déjà possible d'utiliser plusieurs types d'algorithmes de la bibliothèque standard. Les différents concepts plus avancés, tels que les itérateurs, vont être introduit en même temps que des exemples d'algorithmes (recherche d'un élément, trier les éléments d'une collection, etc). Et pour commencer, ce chapitre détaille les algorithmes de comparaison.

## L'opérateurs d'égalité

Dans les chapitres précédents, vous avez vu l'utilisation de l'opérateur de comparaison `==` (*equal-to operator*). Les collections de la bibliothèque standard, comme `std::vector`, `std::array` ou `std::string`, fournissent également cet opérateur. Vous pouvez donc écrire :

main.cpp

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

int main() {
    std::string const s1 { "salut" };
    std::string const s2 { "salut" };
    std::string const s3 { "hello" };
    std::cout << std::boolalpha;
    std::cout << (s1 == s2) << std::endl;
    std::cout << (s1 == s3) << std::endl;
}
```

affiche :

```
true
```

```
false
```

Il ne faut pas oublier les parenthèses autour du test d'égalité pour `std::string`. L'opérateur `<<` ayant un sens pour cette classe, le compilateur ne pourra pas savoir si vous souhaitez écrire :

```
std::cout << (s1 == s2) << std::endl;
std::cout << s1 == (s2 << std::endl);
```

De la même façon pour les collections, il est possible d'utiliser l'opérateur `==`.

main.cpp

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

int main() {
    std::vector<int> const v1 { 1, 2, 3, 4 };
    std::vector<int> const v2 { 1, 2, 3, 4 };
    std::vector<int> const v3 { 4, 3, 2, 1 };
    std::cout << std::boolalpha;
    std::cout << (v1 == v2) << std::endl;
    std::cout << (v1 == v3) << std::endl;
}
```

affiche :

```
true
false
```

Avec les collections, l'opérateur d'égalité compare les éléments de la collection un par un. S'il trouve une différence, il retourne `false`. S'il arrive à la fin de la collection sans trouver de différence, il retourne `true`.

Pour vérifier si deux chaînes sont différentes, il est possible d'utiliser l'opérateur booléen de négation `!` : `!(s1 == s2)`. Plus simplement, il est possible d'utiliser l'opérateur de comparaison `!=` (*not-equal-to operator*).

main.cpp

```
#include <iostream>
```

```
#include <string>

int main() {
    std::string const s1 { "salut" };
    std::string const s2 { "salut" };
    std::string const s3 { "hello" };
    std::cout << std::boolalpha << (s1 != s2) << std::endl;
    std::cout << (s1 != s3) << std::endl;
}
```

affiche :

```
false  
true
```

On voit ici que les opérateurs `==` et `!=` sont étroitement liés. Lorsque l'un de ces deux opérateurs est utilisable, il est habituel de pouvoir utiliser aussi l'autre opérateur.

Une classe qui propose l'opérateur d'égalité `==` respecte le concept de “comparable par égalité” ([EqualityComparable](#)). Ce concept précise que l'opérateur d'égalité doit suivre les propriétés suivantes :

- réflexivité : quelque soit `a`, `a == a` est toujours vrai ;
- commutativité : si `a == b`, alors `b == a` ;
- transitivité : si `a == b` et `b == c`, alors `a == c`.

Ce concept est assez classique, vous le retrouvez en mathématique dans la théorie des ensembles. vous voyez ici un point important : lorsqu'une classe définit un opérateur `==`, vous pouvez vous attendre à ce qu'elle suive un certain nombre de règles : elle suit une **sémantique**.

Du point de vue de l'utilisateur de cette classe, il pourra l'utiliser de la même façon qu'il utilise n'importe quelle classe respectant cette sémantique. Du point de vue du concepteur de la classe (ce que vous apprendrez à faire dans la suite de ce cours), il suffit de définir les sémantiques que vous souhaitez donner à notre classe, l'écriture de la classe en découlera.

Au contraire, le non respect d'une sémantique sera très perturbant pour

l'utilisateur - et une source d'erreur sans fin. Imaginez que l'opérateur `==` ne réalise pas un test d'égalité, mais permet de faire la concaténation de deux chaînes ? Ou n'importe quoi d'autres, selon la classe ? La cohérence et l'homogénéité des syntaxes sont des notions importantes pour faciliter la lecture d'un code (et donc éviter les erreurs).

Bien sûr, ces considérations s'appliquent à l'ensemble des sémantiques usuelles, en particulier celle que vous connaissez en mathématique (addition avec `+`, soustraction avec `-`, etc.)

## Comparer les éléments un par un

Si vous souhaitez comparer les éléments de deux collections un par un, deux sous-ensembles de collections ou si vous souhaitez utiliser un prédicat différent, il ne sera pas possible d'utiliser l'opérateur d'égalité `==`. Dans ce cas, la bibliothèque standard fournit l'algorithme `std::equal` pour comparer si les éléments de deux collections.

Lorsque vous découvrez une nouvelle fonctionnalité, la première chose à faire est de regarder la documentation : [std::equal](#). Cette page peut vous apprendre plusieurs choses :

- cet algorithme est défini dans le fichier d'en-tête `<algorithm>`, il faudra donc l'inclure dans votre code pour utiliser `std::equal` ;
- il existe quatre versions de cet algorithme :
  - le premier prend en argument le début et la fin d'une première collection et le début d'une seconde collection ;
  - le second est similaire au premier, avec un prédicat personnalisé ;
  - la troisième prend en argument le début et la fin de la première collection, puis de la seconde ;
  - le quatrième est similaire au troisième, avec un prédicat personnalisé.
- le prédicat doit respecter la signature suivante : `bool pred(const Type1 &a, const Type2 &b);`. C'est donc une fonction binaire - une fonction qui prend deux arguments - et retourne un booléen.

La page de documentation donne également des codes d'exemple d'utilisation de ces fonctions.

La première et deuxième version de `std::equal` doit être utilisé avec précaution. Ces deux fonctions comparent les éléments un par un et s'arrête à la fin de la premier collection. Si la seconde collection est plus grande que la première (par exemple les chaînes "abcd" et "abcdEF"), la comparaison se terminera sans prendre en compte les derniers éléments et `std::equal` retournera vrai, alors que ce n'est pas forcément le cas.

Si la seconde collection est plus petite que la première, `std::equal` continuera de travailler après la fin de la seconde collection, ce qui produira un crash, voire un comportement indéfini.

Dans les deux cas, il est possible de résoudre le problème en comparant les tailles respectives de deux collections avant d'utiliser `std::equal`. Si les tailles sont différentes, alors il n'est pas nécessaire d'utiliser `std::equal` dans ce cas.

La troisième (et quatrième) version de `std::equal` permet de tester si l'algorithme est arrivé à la fin des deux collections et donc qu'elles sont parfaitement identiques. Voici un code d'exemple pour illustrer ce problème :

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

int main() {
    std::string const s1 { "abcd" };
    std::string const s2 { "abcdEF" };
    std::cout << std::boolalpha;
    std::cout << std::equal(begin(s1), end(s1), begin(s2))
<< std::endl;
    std::cout << std::equal(begin(s2), end(s2), begin(s1))
<< std::endl;
    std::cout << std::endl;
    std::cout << std::equal(begin(s1), end(s1), begin(s2),
end(s2)) << std::endl;
```

```
    std::cout << std::equal(begin(s2), end(s2), begin(s1),
end(s1)) << std::endl;
}
```

affiche :

```
true  
false
```

```
false  
false
```

Par défaut, préférez l'utilisation de la version prenant le début et la fin des deux collections (les versions 3 et 4 de `std::equal`).

Attention aux éléments que vous passez en argument dans les fonctions. Le compilateur vérifie que vous passer des collections de types identiques en argument, pas que les informations passées ont un sens. Par exemple, si vous passer en argument un élément d'une collection puis un élément d'une seconde collection, cela n'a pas de sens de parcourir une collection entre ce deux éléments.

```
std::equal(begin(s1), end(s2), begin(s2)); // problème, les
deux premiers arguments
                                              // ne proviennent
pas de la même collection
```

Dans ce cas, ce code ne produit pas d'erreur de compilation, mais un comportement indéterminé (*undefined behavior*, UB). Ce type d'erreur est assez complexe à détecter et à corriger, il faut être très attentif pour les éviter.

## Comparer des collections différentes

Comme cela a été dit auparavant, la classe `std::string` peut être vu comme une collection de caractères (`char`). Cependant, si on essaie de

comparer une variable de type `string` à un tableau de caractères, on obtient une erreur :

main.cpp

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

int main() {
    std::string const s { "abcdef" };
    std::vector<char> const v { 'a', 'b', 'c', 'd', 'e', 'f' };
    std::cout << std::boolalpha;
    std::cout << (s == v) << std::endl;
}
```

affiche l'erreur à la compilation suivante :

```
main.cpp:9:21: error: invalid operands to binary expression
('const std::string'
(aka 'const basic_string<char, char_traits<char>,
allocator<char> >') and
'const std::vector<char>')
    std::cout << (s == v) << std::endl;
               ^   ~
1 error generated.
```

Cette erreur signifie que le compilateur ne trouve pas d'opérateur d'égalité `==` permettant de comparer un type `std::string` avec un type `std::vector<char>`. En effet, les opérateurs de comparaison sont définis pour accepter deux arguments de même type, par exemple deux `string` ou deux `vector<char>`, mais pas deux collection de type différents, même si ces deux types correspondent tous deux à des collections de `char` (on parle parfois de typage “fort” du C++) .

Dans cette situation, il est possible d'utiliser la fonction `std::equal` pour tester l'égalité de deux collections. Cet algorithme fonctionne sur des collections de types différents, à partir du moment où les éléments sont comparable par égalité. Par exemple, il sera possible de comparer des `std::string` et des `std::vector<char>` (les éléments sont dans les

deux cas des `char`) ou des `std::vector<int>` et `std::vector<float>` (il est possible de comparer un entier et un nombre réel), mais pas `std::vector<int>` et `std::vector<string>` (`int` et `string` ne sont pas comparable par égalité).

En utilisant `std::equal`, la comparaison devient :

main.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>

int main() {
    std::string const s { "abcdef" };
    std::vector<char> const v1 { 'a', 'b', 'c', 'd', 'e',
'f' };
    std::vector<char> const v2 { 'a', 'z', 'e', 'r', 't',
'y' };
    std::cout << std::boolalpha;
    std::cout << std::equal(begin(s), end(s), begin(v1), end
(v1)) << std::endl;
    std::cout << std::equal(begin(s), end(s), begin(v2), end
(v2)) << std::endl;
}
```

affiche :

```
true
false
```

Ce qui correspond bien à une comparaison des éléments un par un.

## Comparer des parties de collections

Avec `std::equal`, il est possible de comparer une partie d'une collection. Pour cela, il suffit de fournir des positions différentes que le début et la fin d'une collection. Par exemple, vous pouvez incrémenter ou décrémenter les positions de début (avec `begin(s)+n`) et de fin (avec

`end(s)-n`) en faisant attention de ne pas donner des valeurs en dehors de la collection ou en utilisant des fonctions de recherche (`std::find`, que vous verrez dans un prochain chapitre).

Par exemple, pour comparer les quatre premiers éléments de deux collections, vous pouvez écrire :

#### main.cpp

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

int main() {
    std::string const s1 { "abcdef" };
    std::string const s2 { "abcdEF" };
    std::cout << std::boolalpha;
    std::cout << std::equal(begin(s1), end(s1), begin(s2),
end(s2)) << std::endl;
    std::cout << std::equal(begin(s1), begin(s1)+4, begin(s2),
begin(s2)+4) << std::endl;
}
```

affiche :

```
false
true
```

La première version compare la totalité des deux chaînes ("abcdef" et "abcdEF") et retourne faux, puisque la chaîne `s2` contient des majuscules. La seconde version compare le début de la première chaîne (à partir de `begin(s1)` donc "a" jusqu'au quatrième caractère `begin(s1)+4` donc "d") avec le début de la seconde (également "abcd") et retourne vrai.

En utilisant la notation `begin(s)+n`, si vous sortez de la collection, cela produit un comportement indéterminé (*undefined behavior*). Il n'y a aucun message d'erreur vous prévenant qu'il y a un problème.

Faites bien attention de ne pas perdre de vue ce que vous comparer. Ce n'est pas parce que vous utiliser `std::equal` ("égal" en français) que vos collections sont "égales". Seuls les éléments que vous comparer sont "égaux" (donc potentiellement des collections de tailles ou de types différents, voire des éléments de type différents). Ce n'est pas une "égalité" stricte.

## Utiliser un prédictat personnalisé

Par défaut, l'algorithme `std::equal` compare chaque élément de deux collections en utilisant l'opérateur `==` de chaque élément. En pseudo-code, cela donnerait :

```
si premier élément de collection 1 est différent du premier
élément de collection 2
    alors retourner "les collections sont différentes"

si deuxième élément de collection 1 est différent du
deuxième élément de collection 2
    alors retourner "les collections sont différentes"

si troisième élément de collection 1 est différent du
troisième élément de collection 2
    alors retourner "les collections sont différentes"

...
Si tous les éléments sont identiques
    alors retourner "les collections sont identiques"
```

Lorsque les collections contiennent des éléments de types non comparables ou lorsque la comparaison d'égalité par défaut ne convient pas, il est possible de fournir un prédictat personnalisé dans `std::equal`.

Un prédictat est "quelque chose" qui prend des arguments et retourne un booléen. Vous utiliserez principalement des prédictats avec un argument (prédictat unaire) ou deux arguments (prédictat binaire) avec les algorithmes de la bibliothèque standard.

```
bool result_1 = predicat_unaire(argument_1);
bool result_2 = predicat_binaire(argument_1, argument_2);
```

Ce “quelque chose” n'est volontairement pas défini pour le moment, cela peut correspondre à différentes syntaxes que vous verrez plus tard dans ce cours (une fonction libre, une fonction lambda, un foncteur). Le plus important est donc de retenir que c'est “quelque chose” qui peut s'utiliser de la façon donnée dans le code.

Vous reconnaîtrez la syntaxe pour appeler une fonction. Plus généralement, il est possible d'utiliser n'importe quel objet qui peut s'utiliser comme une fonction (*callable-object*, “objet appelable”).

Ce prédicat sera utilisée à la place de l'opérateur `==` pour la comparaison des éléments un par un. En pseudo-code, cela donnerait :

```
si prédicat(premier élément de collection 1, premier élément
de collection 2)
    alors retourner "les collections sont différentes"

si prédicat(deuxième élément de collection 1, deuxième
élément de collection 2)
    alors retourner "les collections sont différentes"

si prédicat(troisième élément de collection 1, troisième
élément de collection 2)
    alors retourner "les collections sont différentes"

...
Si tous les éléments sont identiques
    alors retourner "les collections sont identiques"
```

## Les objets-fonctions

La bibliothèque standard fournit quelques prédicats de base, décrits dans

la documentation ([Function objects](#)) et inclus dans le fichier d'en-tête `<functional>`. Ces prédictats sont simples à comprendre, ils correspondent aux opérateurs logiques (`logical_and`, `logical_or` et `logical_not`) et de comparaison (`equal_to`, `not_equal_to`, `greater`, `less`, `greater_equal` et `less_equal`) que vous avez déjà vu.

Attention de ne pas confondre l'algorithme `std::equal` et le prédictat `std::equal_to`.

Les autres objets-fonctions seront utiles pour les autres algorithmes de la bibliothèque standard, que vous verrez par la suite.

Les objets-fonctions peuvent être appelés comme des fonctions, mais sont avant tout des objets. Il est donc nécessaire dans un premier temps de créer l'objet avant de pouvoir l'utiliser. La création d'un objet-fonction est similaire à n'importe quelle création d'objet.

```
std::equal_to<TYPE>()
std::equal_to<>()
```

Les objets-fonctions de la bibliothèque standard sont des classes template et s'écrivent donc avec des chevrons `<>`. Il est possible de spécifier le type du prédictat entre les chevrons (ce qui produira une erreur si vous essayez d'utiliser le prédictat avec un type non compatible) ou de laisser les chevrons vides pour accepter n'importe quel type comparable.

L'objet créé peut ensuite être appelée comme une fonction.

```
const auto predicat = std::equal_to<>();
std::cout << predicat(1, 2) << std::endl;
```

Le code suivant :

```
main.cpp
#include <iostream>
#include <functional>

int main() {
```

```
const auto predicat = std::equal_to<>();
std::cout << std::boolalpha;
std::cout << predicat(1, 2) << std::endl;
std::cout << predicat(1, 1.0) << std::endl;
std::cout << predicat('a', 'b') << std::endl;
std::cout << predicat('a', 'a') << std::endl;
std::cout << predicat("azerty", "abcdef") << std::endl;
std::cout << predicat("azerty", "azerty") << std::endl;
}
```

affiche :

```
false
true
false
true
false
true
```

Il n'est pas nécessaire de créer une variable intermédiaire pour créer un objet-fonction, mais cela permet d'avoir un code plus lisible.

Notez bien l'utilisation des parenthèses :

- la première paire pour créer l'objet-fonction ;
- la seconde pour appeler la fonction.

Si vous ne souhaitez pas créer de variable intermédiaire, il faut bien mettre les deux paires de parenthèses (cette syntaxe trouble parfois les débutants).

main.cpp

```
#include <iostream>
#include <functional>

int main() {
    std::cout << std::boolalpha;
    std::cout << std::equal_to<>()(1, 2) << std::endl;
    std::cout << std::equal_to<>()(1, 1.0) << std::endl;
    std::cout << std::equal_to<>()(‘a’, ‘b’) << std::endl;
    std::cout << std::equal_to<>()(‘a’, ‘a’) << std::endl;
```

```
    std::cout << std::equal_to<>() ("azerty", "abcdef") <<
std::endl;
    std::cout << std::equal_to<>() ("azerty", "azerty") <<
std::endl;
}
```

## Les algorithmes avec prédictats personnalisés

Les prédictats peuvent être utilisé avec les algorithmes de la bibliothèque standard. Dans ce cas, les algorithmes appellent directement le prédictat sur les éléments d'une collection. Cela implique qu'il faut donner un objet directement appelleable, donc qu'il faut instancier l'objet-fonction avant de la passer en argument d'un algorithme.

Pour tous les algorithmes de la bibliothèque standard qui acceptent un prédictat, celui-ci est donné en dernier argument. Par exemple, avec `std::equal`, la syntaxe devient :

```
std::equal(itérateur, itérateur, itérateur, itérateur)
std::equal(itérateur, itérateur, itérateur, itérateur,
PRÉDICAT)
```

Il est possible de créer une variable intermédiaire pour le prédictat ou de l'instancier directement dans l'appel de l'algorithme.

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

int main() {
    const std::vector<int> v { 1, 2, 3, 4, 5 };
    const std::vector<int> w { 2, 4, 3, 1, 5 };

    std::cout << std::boolalpha;
    std::cout << std::equal(begin(v), end(v), begin(v), end(
v), std::equal_to<>()) << std::endl;
    std::cout << std::equal(begin(v), end(v), begin(w), end(
w), std::equal_to<>()) << std::endl;
```

```
    std::cout << std::equal(begin(v), end(v), begin(v), end(v),
    std::not_equal_to<>()) << std::endl;
    std::cout << std::equal(begin(v), end(v), begin(w), end(w),
    std::not_equal_to<>()) << std::endl;
}
```

affiche :

```
true
false
false
false
```

## Exercice

Est-ce que les deux syntaxes suivantes donnent le même résultat ? C'est à dire de comparer si deux collections sont différentes. (Notez bien l'opérateur de négation logique `!` dans la seconde ligne).

```
bool result_1 = std::equal(begin(v), end(v), begin(v), end(v),
std::not_equal_to<>());
bool result_2 = ! std::equal(begin(v), end(v), begin(v), end(v));
}
```

## L'ordre lexicographique

L'égalité et l'inégalité de deux collections est simple à définir : deux collections sont égales si elles contiennent les mêmes éléments (même nombre d'éléments, dans le même ordre). Elles sont différentes dans le cas contraire.

Mais quel sens donner à la phrase “une collection est inférieure à une autre collection” ?

L'ordre lexicographique est une méthode qui permet de comparer deux collections. Même si vous ne connaissez pas le terme, vous connaissez obligatoirement cette méthode : c'est celle qui est utilisée pour trier des mots, en particulier dans un dictionnaire. (N'oubliez pas qu'une chaîne de

caractères peut être vue comme une collection de caractères).

Pour rappel, voici comment appliquer cette méthode :

- Commencez par prendre le premier élément de chaque collection et comparez les.
- Si l'un des éléments d'une des collections est inférieur à l'élément de l'autre collection, la collection correspondante est inférieure à l'autre.
- Si les deux éléments sont égaux, comparez les éléments suivants de chaque collection.
- Si une collection se termine avant l'autre, elle est inférieure à l'autre.
- Si tous les éléments sont identiques, les collections sont égales.

Voici un exemple pour bien comprendre, avec les chaînes "abc" et "acd". Les premiers caractères sont "a" et "a". Ils sont identiques, passez aux deuxièmes caractères. Ceux-ci sont "b" et "c". Comme "b" est plus petit que "c", la chaîne "abc" est inférieure à la chaîne "acd".

Un autre exemple, comparez "abc" et "ab". Les premiers et deuxièmes caractères sont identiques, il faudrait donc comparer les troisièmes caractères. Cependant, la seconde chaîne ne possède pas de troisième caractère, elle est donc inférieure à la première.

De même avec une collection d'entiers. Par exemple, la collection `{ 1, 2, 3 }` sera inférieure à la collection `{ 1, 2, 4 }` et supérieure à la collection `{ 1, 2 }`.

### main.cpp

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

int main() {
    std::cout << std::boolalpha;
    std::cout << (std::string { "abc" } < std::string { "acd" })
} << std::endl;
    std::cout << (std::string { "abc" } < std::string { "ab" })
```

```
}) << std::endl;
    std::cout << (std::vector<int> { 1, 2, 3 } < std::vector
<int> { 1, 2, 4 }) << std::endl;
    std::cout << (std::vector<int> { 1, 2, 3 } < std::vector
<int> { 1, 2 }) << std::endl;
}
```

affiche :

```
true
false
true
false
```

La comparaison “plus petit que” est également un concept ([LessThanComparable](#)), ce qui implique que différentes propriétés doivent être respectées :

- **identité** : une collection n'est pas inférieure à elle-même (elle est égale à elle-même) ;
- **inverse** : si une collection est inférieur à une seconde collection, la seconde collection n'est pas inférieure à la première (elle est supérieure ou égale) ;
- **transitivité** : si une collection a est inférieur à une collection b et que cette collection b est inférieure à une collection c, alors la collection a est inférieure à la collection c.

Ce concept “LessThanComparable” est valable pour d'autres types du C++ (`int`, `float`, etc.) et de la bibliothèque standard (`std::string`, `std::complex`, `std::vector`, etc.).

Lorsqu'une classe définit un opérateur de comparaison, il est logique que les autres opérateurs soient aussi définis (il faudrait que cela ait un sens de ne pas les définir). Vous apprendrez dans la partie sur la programmation orientée objet (en particulier dans la partie sur la sémantique de valeur) comment définir ces opérateurs dans une classe que vous créez.

## L'algorithme std::lexicographical\_compare

L'algorithme `std::equal` permet de comparer si deux collections sont égales ou différentes. L'équivalent pour la comparaison est l'algorithme `std::lexicographical_compare`. Tout comme `std::equal` est une version plus générique de l'opérateur `==` (et indirectement de `!=`), l'algorithme `std::lexicographical_compare` est une version plus générique de l'opérateur `<` (et donc indirectement des opérateurs `>`, `<=` et `>=`). En particulier, `std::lexicographical_compare` pourra être utilisé sur des sous-collections ou des collections de types différents.

La fonction `std::lexicographical_compare` ([documentation](#)) prend comme arguments le début et la fin des deux collections à comparer. Elle existe en deux versions, avec ou sans prédictat personnalisé, et retourne vrai si la première collection est inférieure à la seconde.

main.cpp

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

int main() {
    std::string const s1 { "azerty" };
    std::string const s2 { "abcdef" };
    std::string const s3 { "qwerty" };
    std::cout << std::boolalpha;
    std::cout << std::lexicographical_compare(begin(s1), end(s1),
                                                begin(s2), end(s2)) << std::endl;
    std::cout << std::lexicographical_compare(begin(s1), end(s1),
                                                begin(s3), end(s3)) << std::endl;
}
```

affiche :

```
false
true
```

## Exercices

- comparer vector<int> et vector<float>
- comparer vector<int> et vector<string> (Aide : utilisez `std::stoi` dans une fonction lambda)
- comparer vector<int> et vector<string> (qui contient “un”, “deux”, etc.)
- tester si une chaîne est un palindrome (un palindrome est un mot qui peut être lu de droite à gauche ou de gauche à droite, comme par exemple “kayak” ou “radar”).

main.cpp

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

int main() {
    std::string const s1 { "azerty" }; // non palindromique
    std::string const s2 { "abccba" }; // palindromique
    std::cout << std::boolalpha;
    std::cout << std::equal(begin(s1), end(s1), s1.rbegin())
<< std::endl;
    std::cout << std::equal(begin(s2), end(s2), s2.rbegin())
<< std::endl;
}
```

affiche :

```
false
true
```

- Tester des collections différentes

main.cpp

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
```

```
int main() {
    std::string const s { "abcdef" };
    std::vector<char> const a { 'a', 'z', 'e', 'r', 't', 'y' };
    std::cout << std::boolalpha;
    std::cout << std::lexicographical_compare(begin(s), end(s),
        begin(a), end(a)) << std::endl;
}
```

affiche :

```
true
```

- Comparer des parties de collections

main.cpp

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

int main() {
    std::string const s1 { "abcdef" };
    std::string const s2 { "abcdg" };
    std::cout << std::boolalpha;
    std::cout << std::lexicographical_compare(begin(s1), end(s1),
        begin(s2), end(s2)) << std::endl;
    std::cout << std::lexicographical_compare(begin(s1),
        begin(s1)+4, begin(s2), begin(s2)+4) << std::endl;
}
```

affiche :

```
true
false
```

- Utiliser un prédictat personnalisé

main.cpp

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

```
#include <cctype>

int main() {
    std::string const s1 { "ABCDEF" };
    std::string const s2 { "azerty" };
    std::cout << std::boolalpha;
    std::cout << std::lexicographical_compare(begin(s1), end(s1), begin(s2), end(s2)) << std::endl;
    std::cout << std::equal(begin(s1), end(s1), begin(s2), end(s2),
        [] (auto lhs, auto rhs){ return std::toupper(lhs) <
        std::toupper(rhs); })
        << std::endl;
}
```

affiche :

```
true
false
```

<a href="#">Chapitre précédent</a>	<a href="#">Sommaire principal</a>	<a href="#">Chapitre suivant</a>
------------------------------------	------------------------------------	----------------------------------

# Les catégories d'algorithmes standards

La [page de documentation](#) sur les algorithmes de la bibliothèque standard fournit une liste d'une centaine de fonctions différentes. À cela, il faut ajouter le fait que chaque fonction peut avoir plusieurs signatures (listes de paramètres différents, en particulier des versions avec et sans prédictat personnalisé).

Ce cours ne va pas détailler tous ces algorithmes. Cela serait purement descriptif et vous n'apprendrez pas grand chose de plus qu'en lisant la documentation. Ce qui est important est d'avoir une vision d'ensemble des algorithmes proposés, savoir où trouver l'algorithme qui vous intéresse lorsque vous êtes face à une problématique.

Pour organiser un peu les algorithmes de la bibliothèque standard, la [page de documentation sur cppreference.com](#) sépare les algorithmes en plusieurs catégories. Vous allez voir dans ce chapitre quelques algorithmes notables, ce qui va permettre d'introduire quelques notions importantes, vous apprendre à utiliser au mieux les algorithmes et à lire la page de documentation.

Ce chapitre détaille plus particulièrement les algorithmes qui ne retournent pas d'itérateur et qui ne nécessitent donc pas de détailler ce concept. Les itérateurs seront détaillés dans le chapitre suivant, ainsi que d'autres algorithmes de la bibliothèques standard.

Note : des exercices seront ajoutés par la suite à ce cours. Ils vous permettront d'étudier et pratiquer plus en détail ces algorithmes.

## Les algorithmes non modifiants

Les algorithmes non modifiants (*non-modifying sequence operations*). Ce

sont des algorithmes qui ne modifient pas les collections sur lesquelles ils sont utilisés. Vous allez trouver dans cette catégorie par exemple l'algorithme d'égalité `std::equal` que vous avez déjà vu, ainsi que les algorithmes de recherche (en premier lieu `std::find`), les algorithmes de comptage (`std::count`) et l'algorithme `std::for_each` (qui permet d'appeler une fonction sur chaque élément d'une collection).

Un algorithme non modifiant va donc prendre une collection sans la modifier et retourner un résultat. La valeur renournée peut être utiliser directement ou enregistrée dans une variable, comme n'importe quelle valeur. Le type de la valeur renournée dépend de l'algorithme et du type de collection, le plus simple est d'utiliser l'inférence de type pour créer une variable (consulter la documentation pour connaître le type exact).

Les exemples de code suivants utilisent une chaîne, pour simplifier l'écriture du code et l'affichage du résultat. N'oubliez pas qu'une chaîne est une collection de caractères.

## **std::all\_of, std::any\_of et std::none\_of**

Les algorithmes `std::all_of`, `std::any_of` et `std::none_of` permettent de tester respectivement : si tous les éléments d'une collection respectent un prédictat, si au moins un élément respecte un prédictat ou si aucun élément ne respecte un prédictat. Ces trois algorithmes retournent un booléen.

Par exemple, pour tester si une chaîne possède des majuscules, vous pouvez utiliser les fonctions définies dans [l'en-tête <cctype>](#), en particulier la fonction `isupper` ("est une majuscule").

`main.cpp`

```
#include <iostream>
#include <string>
#include <cctype>
#include <algorithm>
```

```
int main() {
    const std::string s { "abcDEFF" };
    std::cout << std::boolalpha;
    std::cout << std::all_of(begin(s), end(s), isupper) <<
std::endl;
    std::cout << std::any_of(begin(s), end(s), isupper) <<
std::endl;
    std::cout << std::none_of(begin(s), end(s), isupper) <<
std::endl;
}
```

affiche :

```
false
true
false
```

## std::all\_of, std::any\_of et std::none\_of

Les algorithmes `std::count` et `std::count_if` permettent le nombre d'éléments d'une collection correspondant respectivement à une valeur et un prédictat. Ces algorithmes retournent une valeur entière signée.

main.cpp

```
#include <iostream>
#include <string>
#include <cctype>
#include <algorithm>

int main() {
    const std::string s {
"f9c02b6c9da8943feaee4966ba7417d65de2fe7e" };
    std::cout << std::count(begin(s), end(s), '7') << std::endl;
    std::cout << std::count_if(begin(s), end(s), isdigit) <<
std::endl;
}
```

affiche :

## std::equal

Et bien sur, vous avez déjà vu l'algorithme `std::equal`.

main.cpp

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

int main() {
    const std::string s1 {
        "f9c02b6c9da8943fea4966ba7417d65de2fe7e" };
    const std::string s2 {
        "8cc52221d9bd6a3701c90969bcee91be4810c8d5" };
    std::cout << std::boolalpha;
    std::cout << std::equal(begin(s1), end(s1), begin(s2),
end(s2)) << std::endl;
}
```

affiche :

false

## Les algorithmes modifiants

Les algorithmes modifiants (*modifying sequence operations*), vous l'avez sûrement deviné, modifient les collections sur lesquelles ils sont utilisés. Vous trouverez dans cette catégorie les algorithmes pour ajouter (`std::fill`), supprimer (`std::remove`), remplacer (`std::replace`), copier (`std::copy`), échanger (`std::swap`) ou mélanger (`std::shuffle`) des éléments.

Les algorithmes modifiants sont de deux types : les algorithmes qui modifient directement une collection et les algorithmes qui utilisent une

collection et modifient une autre collection.

Par exemple :

```
std::fill(begin(in), end(in), value);
// seulement in
std::transform(begin(in), end(in), begin(out), operation);
// in et out
```

Mais il est généralement possible d'utiliser le second type d'algo modifiant avec la même collection en entrée et sortie :

```
std::transform(begin(in), end(in), begin(in), operation);
```

Dans le premier cas, transforme les éléments de `in` et met le résultat dans `out`. Dans le second cas, transforme les éléments de `in` et met le résultat dans `in`. (les valeurs initiales sont donc perdues).

## **std::copy, std::copy\_if et std::copy\_n**

Note : pas de contrôle de dépassement de collection.

Equivalent : `std::move`, mais sera vu plus tard.

`main.cpp`

```
#include <iostream>
#include <string>
#include <cctype>
#include <algorithm>

int main() {
    const std::string s1 {
"f9c02b6c9da8943feaee4966ba7417d65de2fe7e" };
    std::string s2 {
"....." };
    std::string s3 {
"....." };
    std::string s4 {
"....." };
}
```

```

    std::copy(begin(s1), end(s1), begin(s2));
    std::cout << s2 << std::endl;

    std::copy_if(begin(s1), end(s1), begin(s3), isalpha);
    std::cout << s3 << std::endl;

    std::copy_n(begin(s1), 10, begin(s4));
    std::cout << s4 << std::endl;
}

}

```

affiche :

```
f9c02b6c9da8943feaea4966ba7417d65de2fe7e
fcbcdafaeaebaddefee.....
f9c02b6c9d.....
```

## **std::copy\_backward**

Note : pas de contrôle de dépassement de collection.

Equivalent : `std::move_backward`, mais sera vu plus tard.

### main.cpp

```

#include <iostream>
#include <string>
#include <algorithm>

int main() {
    const std::string s1 {
" f9c02b6c9da8943feaea4966ba7417d65de2fe7e" };
    std::string s2 {
"....." };
    std::string s3 {
"....." };

    std::copy(begin(s1) + 10, begin(s1) + 15, begin(s2) + 10);
    std::cout << s2 << std::endl;
}

```

```
    std::copy_backward(begin(s1) + 10, begin(s1) + 15, begin  
(s3) + 10);  
    std::cout << s3 << std::endl;  
}
```

affiche :

```
.....a8943.....  
.....a8943.....
```

## std::fill et std::fill\_n

main.cpp

```
#include <iostream>  
#include <string>  
#include <algorithm>  
  
int main() {  
    std::string s { "1234567890" };  
    std::fill(begin(s), end(s), 'A');  
    std::cout << s << std::endl;  
    std::fill_n(begin(s), 5, '1');  
    std::cout << s << std::endl;  
}
```

affiche :

```
AAAAAAAAAA  
11111AAAAA
```

## std::transform

Différent des autres algos, ne prend pas un prédictat (objet appelleable qui retourne un booléen), mais prend en paramètre un opérateur unaire ou binaire (selon la version de `transform` appelée).

La documentation précise la signature des fonctions (in = input = entrée, out = output = sortie) :

```
// unaire
TypeOut fun1(const Type &a);
std::transform(std::begin(in), std::end(in), std::begin(out),
fun1);

// binaire
TypeOut fun2(const Type1 &a, const Type2 &b);
std::transform(std::begin(in1), std::end(in1), std::begin(in2),
std::begin(out), fun2);
```

Par exemple :

Version unaire :

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

int main() {
    std::string s { "azerty" };
    std::transform(begin(s), end(s), begin(s), toupper);
    std::cout << s << std::endl;
}
```

affiche :

```
AZERTY
```

Version binaire :

```
#include <iostream>
#include <functional>
#include <algorithm>

int main() {
    std::vector<int> v1 { 1, 2, 3, 4 };
    const std::vector<int> v2 { -1, 2, -1, 2 };
    std::transform(begin(v1), end(v1), begin(v2), begin(v1),
```

```
std::multiplies<int>());
    for (auto i: v1) std::cout << i << ' ';
    std::cout << std::endl;
}
```

affiche :

```
-1 4 -3 8
```

## std::generate et std::generate\_n

avec rand ?

## std::remove et std::replace

main.cpp

```
#include <iostream>
#include <string>
#include <cctype>
#include <algorithm>

int main() {
    std::string s1 {
        "f9c02b6c9da8943fea4966ba7417d65de2fe7e"
    };
    std::remove(begin(s1), end(s1), '7');
    std::cout << s1 << std::endl;

    std::string s2 {
        "f9c02b6c9da8943fea4966ba7417d65de2fe7e"
    };
    std::remove_if(begin(s2), end(s2), isdigit);
    std::cout << s2 << std::endl;

    const std::string s3 {
        "f9c02b6c9da8943fea4966ba7417d65de2fe7e"
    };
    std::string s4 {
        "....."
    };
    std::remove_copy(begin(s3), end(s3), begin(s4), '7');
```

```

    std::cout << s4 << std::endl;

    const std::string s5 {
"f9c02b6c9da8943feaee4966ba7417d65de2fe7e" };
    std::string s6      {
"....." };
    std::remove_copy_if(begin(s5), end(s5), begin(s6),
isdigit);
    std::cout << s6<< std::endl;
}

```

affiche :

```

f9c02b6c9da8943feaee4966ba41d65de2fee7e
fcbcdafaeaebaddefea4966ba7417d65de2fe7e
f9c02b6c9da8943feaee4966ba41d65de2fee...
fcbcdafaeaebaddefe.....

```

main.cpp

```

#include <iostream>
#include <string>
#include <cctype>
#include <algorithm>

int main() {
    std::string s1 {
"f9c02b6c9da8943feaee4966ba7417d65de2fe7e" };
    std::replace(begin(s1), end(s1), '7', '.');
    std::cout << s1 << std::endl;

    std::string s2 {
"f9c02b6c9da8943feaee4966ba7417d65de2fe7e" };
    std::replace_if(begin(s2), end(s2), isdigit, '.');
    std::cout << s2 << std::endl;

    const std::string s3 {
"f9c02b6c9da8943feaee4966ba7417d65de2fe7e" };
    std::string s4      {
"....." };
    std::replace_copy(begin(s3), end(s3), begin(s4), '7',
'.');

```

```

    std::cout << s4 << std::endl;

    const std::string s5 {
"f9c02b6c9da8943fea4966ba7417d65de2fe7e" };
    std::string s6      {
"....." };
    std::replace_copy_if(begin(s5), end(s5), begin(s6),
isdigit, '.');
    std::cout << s6<< std::endl;
}

```

affiche :

```

f9c02b6c9da8943fea4966ba.41.d65de2fe.e
f.c..b.c.da....fea4ea....ba....d..de.fe.e
f9c02b6c9da8943fea4966ba.41.d65de2fe.e
f.c..b.c.da....fea4ea....ba....d..de.fe.e

```

## **std::swap et std::swap\_ranges**

std::swap ne travaille pas sur des collections en particulier.

main.cpp

```

#include <iostream>
#include <string>
#include <algorithm>

int main() {
    std::string s1 { "azerty" };
    std::string s2 { "123456" };
    std::swap(s1, s2);
    std::cout << s1 << std::endl;
    std::cout << s2 << std::endl;

    int x { 123 };
    int y { 456 };
    std::swap(x, y);
    std::cout << x << std::endl;
    std::cout << y << std::endl;
}

```

```
}
```

affiche :

```
123456  
azerty  
456  
123
```

```
main.cpp
```

```
#include <iostream>  
#include <string>  
#include <algorithm>  
  
int main() {  
    std::string s1 { "azerty" };  
    std::string s2 { "123456" };  
    std::swap_ranges(begin(s1), begin(s1) + 3, begin(s2));  
    std::cout << s1 << std::endl;  
    std::cout << s2 << std::endl;  
}
```

affiche :

```
123rty  
aze456
```

iter\_swap ?

## reverse et reverse\_copy

```
main.cpp
```

```
#include <iostream>  
#include <string>  
#include <algorithm>  
  
int main() {  
    std::string s1 { "azerty" };  
    std::reverse(begin(s1), end(s1));
```

```
    std::cout << s1 << std::endl;

    const std::string s2 { "azerty" };
    std::string s3      { "....." };
    std::reverse_copy(begin(s2), end(s2), begin(s3));
    std::cout << s3 << std::endl;
}
```

affiche :

```
ytreza  
ytreza
```

## rotate

main.cpp

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

int main() {
    std::string s1 { "azerty" };
    std::rotate(begin(s1), begin(s1) + 2, end(s1));
    std::cout << s1 << std::endl;

    const std::string s2 { "azerty" };
    std::string s3      { "....." };
    std::rotate_copy(begin(s2), begin(s2) + 2, end(s2),
begin(s3));
    std::cout << s3 << std::endl;
}
```

affiche :

```
ertyaz  
ertyaz
```

## shuffle

```
main.cpp
#include <iostream>
#include <string>
#include <algorithm>
#include <random>

int main() {
    std::string s { "azerty" };
    std::random_device rd;
    std::mt19937 g(rd());
    std::shuffle(begin(s), end(s), g);
    std::cout << s << std::endl;
}
```

affiche par exemple :

```
yazert
```

## unique

## Les algorithmes de partitionnement

Les algorithmes de partitionnement (*partitioning operations*) permettent séparer une collection en deux sous-collections.

## Les algorithmes de tri

Les algorithmes de tri (*sorting operations*) permettent de trier les éléments d'une collection.

## Les algorithmes binaires de recherche

Les algorithmes binaires de recherche (*binary search operations*) permettent de rechercher un élément dans une collection triée.

## Les algorithmes sur les ensembles

Les algorithmes sur les ensembles (*set operations*) permettent de manipuler une collection comme un ensemble d'éléments (donc sans élément en double).

## Les algorithmes sur les Tas

Les algorithmes sur les Tas (*heap operations*) permettent de manipuler une collection comme un Tas d'éléments.

## Les algorithmes minimum/maximum

Les **algorithmes minimum/maximum** permettent de recherche des éléments minimum ou maximum et réaliser des permutations.

## Les algorithmes numériques

Les **algorithmes numériques** (*numeric operations*) permettent de travailler sur des collections de nombres.

[Chapitre précédent](#) [Sommaire principal](#) [Chapitre suivant](#)

# Les itérateurs

Pour appeler les algorithmes sur des collections, vous avez vu les fonctions `begin` et `end` pour parcourir du début à la fin (et leur équivalent “reverse”, pour lire de la fin au début, `rbegin` et `rend`). Il est possible d'enregistrer ces positions dans des variables, par exemple en utilisant l'inférence de type avec `auto` :

```
std::vector<int> v { 1, 5, 2, 4, 3 };
auto b = std::begin(v);
auto e = std::end(v);
std::sort(b, e);
```

Bien sûr, il est également possible d'écrire explicitement le type des variables `b` et `e`, au lieu d'utiliser l'inférence de type. La syntaxe est dans ce cas :

```
std::vector<int>::iterator b = std::begin(v);
std::vector<int>::iterator e = std::end(v);
```

Vous verrez par la suite la signification exacte de cette syntaxe, un peu compliquée (c'est la raison pour laquelle on utilise l'inférence de type, c'est plus simple à écrire). Ce qui est important de comprendre, c'est que la position dans une collection est gérée en C++ par le concept d'itérateur. Peu importe pour le moment de savoir à quoi correspondent exactement les itérateurs, retenez qu'ils représentent une position et qu'ils permettent de parcourir les éléments d'un conteneur.

La paire d'itérateurs `b` et `e` définit ce que l'on appelle un “range”, un intervalle. Plus généralement, un *range* est une paire d'itérateurs :

- qui proviennent de la même collection ;
- qui sont ordonnés (la première position est plus petite ou égale à la seconde).

Ainsi, les paires d'itérateurs `(begin(v), end(v))` ou `(end(v), begin(v))`

sont des *ranges*, tandis que les paires `(begin(v1), end(v2))` et `(end(v), begin(v))` ne le sont pas. Cette notion de *range* est importante, puisque les algorithmes de la bibliothèque standard n'acceptent en argument que des *ranges*.

Attention, si vous n'utilisez pas un *range* valide, cela ne produit pas de message d'erreur, mais un comportement indéfini (*undefined behavior*), ce qui est une erreur très difficile à identifier.

## Relation entre itérateur et élément d'une collection

### Créer une sous-collection

Il est possible de créer une nouvelle collection à partir d'un *range*, en passant une paire d'itérateurs en argument (donc entre parenthèses) lors de la définition d'une variable.

```
std::vector<int> v1 { 1, 2, 3, 4, 5 };

// définition d'une nouvelle variable
std::vector<int> v2(std::begin(v1), std::end(v1));

// avec auto
auto v3 = std::vector<int>(std::begin(v1), std::end(v1));
```

Il est également possible de créer un nouvel objet et de l'affecter à une variable existante.

```
std::vector<int> v4 {}; // vide
v4 = std::vector<int>(std::begin(v1), std::end(v1));
```

Bien sûr, en utilisant `begin` et `end` comme *range*, cela est peu intéressant, on fait l'équivalent d'une copie là où on peut utiliser l'opérateur d'affection :

```
std::vector<int> v2 { v1 };
```

```
auto v3 = v1;  
v4 = v1;
```

Contrairement à la copie, il est possible de changer le type de conteneur.  
Par exemple, pour passer à un `vector<float>`

```
std::vector<int> v1 { 1, 2, 3, 4, 5 };  
  
std::vector<float> v2 { v1 }; //  
erreur  
std::vector<float> v2(std::begin(v1), std::end(v1)); // ok
```

On va également pouvoir créer une sous-collection. Par exemple, pour créer une collection contenant la première moitié des éléments d'une collection :

```
std::vector<int> v1 { 1, 2, 3, 4, 5 };  
std::vector<float> v2(std::begin(v1), std::end(v1) - v1.size()  
() / 2);
```

ou les 2 premiers éléments :

```
std::vector<int> v1 { 1, 2, 3, 4, 5 };  
std::vector<float> v2(std::begin(v1), std::begin(v1) + 2);
```

## Itérateur retourné par une recherche

Pourquoi cette façon de faire ? Avec `find`, si aucun élément n'est trouvé, comment indiquer que la recherche a échoué ? Une solution serait que `find` retourne un booléen en plus. Mais ce serait compliqué à gérer. La solution choisie est qu'elle retourne `end()` si la recherche échoue.

Donc tester le résultat de `find` :

```
auto p = std::find(begin(v), end(v), 3);  
std::cout << boolalpha << (p == end(v)) << endl;  
p = std::find(begin(v), end(v), 10);  
std::cout << boolalpha << (p == end(v)) << endl;
```

Comme `end(v)` ne représente pas un élément dans une collection, il est interdit de le manipuler comme les autres positions. Ainsi, il est légal d'écrire `begin(v)+1`, puisque `begin(v)` est une position correspondante à un élément dans la collection (s'il y a au moins un élément dans la collection), on a le droit de le manipuler. Au contraire, `end(v)+1` est interdit. C'est pour cela que la notion `begin(v)+n` est dangereuse, puisque l'on peut accéder à une position invalide, si `n > size()`.

### advance ? next ?

Supposons que l'on ait plusieurs fois la même valeur dans une collection. Comment toutes les trouver avec `find` ?

On recherche d'abord sur l'intervalle qui nous intéresse avec `begin` et `end`. Puis, si on trouve un élément (donc si `p != end`), alors on peut refaire la recherche entre `p+1` et `end` pour trouver un deuxième élément. Et ainsi de suite, jusqu'à trouver tous les éléments.

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

int main() {
    std::string const s { "azertyazerty" };
    auto position = std::find(std::begin(s), std::end(s),
    'e');
    auto const first = std::string(std::begin(s), position);
    auto const second = std::string(position, std::end(s));
    std::cout << first << std::endl;
    std::cout << second << std::endl;

    position = std::find(position+1, std::end(s), 'e');
    std::cout << std::string(std::begin(s), position) << std::endl;
    std::cout << std::string(position, std::end(s)) << std::endl;
}
```

affiche :

```
az  
ertyazerty  
azertyaz  
erty
```

second `find` : recherche à partir du 'e' trouvé

Pour une collection vide (`{}` ou `" "`), il n'y a pas d'élément et `begin == end`.

## Ce que représente `std::end`

Pour effectuer une recherche dans une collection, on peut utiliser `find`. Si on regarde la documentation, elle indique que cette fonction prend un `range` et un élément à chercher, elle retourne l'itérateur (donc une position) de l'élément dans la collection s'il a été trouvé, sinon l'itérateur `end`.

```
std::vector<int> v1 { 1, 2, 3, 4, 5 };  
auto position = std::find(std::begin(v1), std::end(v1), 3);
```

On peut alors utiliser cette position pour créer deux sous-collections :

main.cpp

```
#include <iostream>  
#include <vector>  
#include <algorithm>  
  
int main() {  
    std::vector<int> v1 { 1, 2, 3, 4, 5 };  
    auto p = std::find(std::begin(v1), std::end(v1), 3);  
  
    std::vector<float> v2(std::begin(v1), p);  
    for (auto i: v2) std::cout << i << ' ';  
    std::cout << std::endl;  
  
    std::vector<float> v3(p, std::end(v1));
```

```
    for (auto i: v3) std::cout << i << ' ';
    std::cout << endl;
}
```

affiche :

```
1 2
3 4 5
```

On voit ici une particularité importante des *ranges*. La variable `p` correspond à la position de la valeur `3` dans la collection. On pourrait croire que la collection `v2` qui est créée à partir de `begin(v1)` et `p` allait contenir les éléments `{ 1, 2, 3 }`, mais ce n'est pas le cas.

En effet, un *range* est une paire d'itérateurs dont le premier est inclus et le second exclu.

On écrira souvent `[first, last)` pour désigner un *range*. Cette notation mathématique permet d'écrire un ensemble, avec les crochets droits pour inclure les bornes et les parenthèses pour les exclure. Ainsi, l'ensemble  $[0, 1)$  correspond à l'ensemble des réels compris entre 0 inclus et 1 exclu,  $(0, 1)$  exclu les valeurs 0 et 1,  $(0, 1]$  exclu 0 et inclus 1, etc.

Revenons sur un code précédent. Lorsque l'on écrit :

```
std::vector<int> v1 { 1, 2, 3, 4, 5 };
std::vector<int> v2(std::begin(v1), std::end(v1));
```

On a bien une copie complète de `v1`. Est-ce que cela est compatible avec l'exclusion de la borne supérieure du *range* ? En fait, oui, tout simplement parce que `end(v1)` correspond à la fin de la collection, et non au dernier élément de cette collection. Si on crée des sous-collections depuis les premiers et derniers éléments :

```
std::vector<int> v1 { 1, 2, 3, 4, 5 };

// premier élément ?
std::vector<int> v2(std::begin(v1), std::begin(v1));
```

```
std::cout << boolalpha << v2.empty() << std::endl; // vide

// premier élément
std::vector<int> v3(std::begin(v1), std::begin(v1)+1);
for (auto i: v3) std::cout << i << ' '; // ok, contient {1}

// dernier élément ?
std::vector<int> v4(std::end(v1), std::end(v1));
std::cout << boolalpha << v4.empty() << std::endl; // vide

// dernier élément
std::vector<int> v5(std::end(v1)-1, std::end(v1));
for (auto i: v5) std::cout << i << ' '; // ok, contient {5}
```

En pratique, cela signifie que `end(v)` ne représente pas le dernier élément d'une collection, mais un élément supplémentaire et imaginaire qui se trouverait après le dernier élément.

## faire une figure

## Notion d'indirection

## Accéder à un élément d'une collection

Lorsque un itérateur correspond à une position valide dans une collection (donc pas `end`), il est possible d'accéder à l'élément en utilisant l'opérateur `*` devant l'itérateur :

```
vector<int> v { 1, 2, 3, 4, 5 };
auto p = begin(v);
cout << (*p) << endl;
```

affiche 1

Egalement accessible en modification (si non `const`) :

```
auto p = begin(v) + 3;
```

```
*p = 0;  
for (auto value: v)  
    cout << value << endl;
```

Si fonction template :

```
template<class Iterator>  
void foo(Iterator it) {  
    auto value = *it;  
}
```

it est un itérateur sur un élément d'un conteneur, donc \*it correspond à un élément d'un conteneur. Le type déduit par auto est le type de cet élément

```
std::vector<int> v {};  
foo(begin(v)); // dans foo, it est de type  
vector<int>::iterator et *it est de type int
```

Il est possible d'expliciter auto. Pour cela, il faut pouvoir récupérer le type d'un itérateur et le type de valeur pointé. Pour cela, il faut utiliser la classe de trait iterator\_traits :

```
template<class Iterator>  
void foo(Iterator it) {  
    typename iterator_traits<Iterator>::value_type value = *  
it;  
}
```

explication sur typename ? sur classe de traits ?

On comprend l'intérêt de auto dans ce code.

## Catégories d'itérateurs

Iterateur, bidirectional, random. Possibilité de manipuler directement les éléments d'une collection. Via itérateur (le plus générique) et pour certains types de collection par [] ou at(). next, prev, advance, distance

Vu comment accéder séquentiellement dans un conteneur (du début à la fin). Méthode la plus simple et la plus performante. Autre méthode : accéder directement à un élément dans le conteneur. Faisable sur certain type de conteneur, dont array et vector.

Utilisation de at() et []

Vérification des limites d'accès

## **Adaptateurs d'itérateurs**

Adaptateur : back\_inserter, front\_inserter, inserter etc

[Chapitre précédent](#) [Sommaire principal](#) [Chapitre suivant](#)

## D'autres collections

Notions de structure de données

Données contigues, liste, arbre, etc.

# Les tableaux de bits

Manipuler

## Notion de tableau

Plusieurs fois des données de même type. Accès avec un indice, partant de 0. Taille fixé à la compilation ou à l'exécution : bitset à la compilation (vector<bool> à l'exécution).

### Représentation

Accès a un élément : [] Validation taille : assert connaitre la taille : size

## **bitset**

créer un bitset

afficher un bitset

tester un bit : mask, flag, opérateur ET bit à bit, test() forcer un bit : OU bit a bit

tester plusieurs bit : count, all, any, none (cf algo)

## **Exercices**

A partir de la représentation binaire d'un nombre (42 = 0b0000000000101010)

- Compter le nombre de 1 dans la représentation
- trouver la plus longue chaîne de 1 dans la représentation

<a href="#">Chapitre précédent</a>	<a href="#">Sommaire principal</a>	<a href="#">Chapitre suivant</a>
------------------------------------	------------------------------------	----------------------------------

<a href="#">Chapitre précédent</a>	<a href="#" style="color: blue;">Sommaire principal</a>	<a href="#">Chapitre suivant</a>
------------------------------------	---	----------------------------------

# Valarray

<a href="#">Chapitre précédent</a>	<a href="#" style="color: blue;">Sommaire principal</a>	<a href="#">Chapitre suivant</a>
------------------------------------	---	----------------------------------

# Gérer des intervalles de valeurs

En mathématiques, un [intervalle](#) est l'ensemble des variables comprises entre deux valeurs. Par exemple, l'intervalle des entiers  $[1, 5[$  (fermé à gauche et ouvert à droite) correspond aux valeurs 1, 2, 3, 4.

Le but de ces exercices va être d'écrire une représentation d'ensemble d'intervalles. Par exemple l'ensemble  $[0, 5[ \cup [10, 15[$  correspond aux valeurs 0, 1, 2, 3, 4, 10, 11, 12, 13, 14.

Note : par symétrie avec les conteneurs standards, nous allons travailler sur des intervalles fermés à gauche et ouverts à droite.

En particulier, il faudra pouvoir gérer les unions et intersections d'intervalles :

- $[0, 5[ \cup [2, 7[ = [0, 7[$
- $[0, 5[ \cap [2, 7[ = [2, 5[$

## Intervalle simple

### Première version

Écrire une classe `Interval_1` permettant de gérer un intervalle simple d'entiers (`int`).

Construction et affectation

- construction à partir de deux valeurs ;
- assertion que les deux valeurs sont ordonnées ;
- sémantique de valeur : copiable, déplaçable (move).

Opérations de base

- intersection d'intervalle ;

- tester si une valeur appartient à un intervalle.

```
class Interval_1 {  
    ...  
private:  
    int low_value{};  
    int hight_value{};  
};
```

## Seconde version

Idem, avec template au lieu de `int`.

```
template<class T>  
class Interval_1_bis {  
    ...  
private:  
    T low_value{};  
    T hight_value{};  
};
```

## Intervalle composé

### Première version

Idem, écrire une classe `Interval_2`, mais ajout de l'union d'intervalles. Il n'est plus possible de conserver uniquement deux valeurs correspondant aux bornes.

Utiliser un vector pouvant recevoir plusieurs `Interval_1` défini précédemment. Il faut en particulier s'assurer de la cohérence et la simplification des données (l'union de deux intervalles peut produire deux intervalles ou un seul).

```
class Interval_2 {  
    ...  
private:  
    std::vector<Interval_1> m_intervals{};  
};
```

## Seconde version

Utiliser un conteneur associatif (std::set) pour trier les intervalles. Note : deux intervalles disjoints sont ordonnables, deux intervalles non disjoints doivent être fusionnés pour former un seul intervalle.

```
class Interval_2_bis {  
    ...  
private:  
    std::set<Interval_1> m_intervals{};  
};
```

## Ensemble de valeurs

### Première version

Écrire une classe `Interval_3`, en changeant de représentation interne. Au lieu de conserver des paires de valeurs pour représenter des intervalles, utiliser un tableau de valeurs avec l'état courant.

- l'intervalle  $[-\infty, \infty[$  sera représenté en interne par un tableau vide ;
- l'intervalle  $[0, \infty[$  sera représenté en interne par la valeur `{0, true}` ;
- l'intervalle  $[0, 1[$  sera représenté en interne par les valeurs `{0, true}, {1, false}` ;
- l'ensemble  $[0, 1[ \cup [2, 3[$  sera représenté en interne par les valeurs `{0, true}, {1, false}, {2, true}, {3, false}`.

```
class Interval_3 {  
    ...  
private:  
    std::vector<pair<int, bool>> m_intervals{};  
};
```

### Deuxième version

Idem, mais utiliser d'autres types que `bool`. Par exemple avec `char` :

```
interval<char> i;           // [-inf, +inf[ = ''
i[0] = 'A';                 // [-inf, 0[ = '', [0, +inf[ =
'A'
i[10] = 'B';                // [-inf, 0[ = '', [0, 10[ = 'A',
[10, +inf[ = 'B'
cout << i[5] << std::endl; // 'A'
```

Écrire la classe correspondante :

- sémantique de valeur ;
- ajout de valeurs ;
- union et intersection ;
- tester une valeur.

```
class Interval_3_bis {
    ...
private:
    std::vector<pair<int, char>> m_intervals{};
};
```

## Deuxième version

Idem, mais avec templates

```
template<class T, class U>
class Interval_3_bis {
    ...
private:
    std::vector<pair<T, U>> m_intervals{};
};
```

<a href="#">Chapitre précédent</a>	<a href="#">Sommaire principal</a>	<a href="#">Chapitre suivant</a>
------------------------------------	------------------------------------	----------------------------------

## Zip et unzip

Intentionnalités inspirées de Python (et autres langages, probablement). Cf <https://docs.python.org/3.3/library/functions.html#zip>). Permet de manipuler des listes.

Zip : prend plusieurs listes et en crée une seule.

```
const auto x = { 1, 2, 3 };
const auto y = { 'a', 'b', 'c' };
const auto z = zip(x, y);
std::cout << z << std::endl; // affiche ((1,a), (2,b),
(3,c))
```

Unzip : prend une liste et la sépare en plusieurs listes.

```
const Zip z = {{ 1, 'a' }, { 2, 'b' }, { 3, 'c' } };
std::tie(x, y) = unzip(z);
std::cout << x << std::endl; // affiche (1, 2, 3)
std::cout << y << std::endl; // affiche (a, b, c)
```

## Vérifier les données

Écrire une fonction qui vérifie que deux vectors ont le même nombre d'éléments, en utilisant un algorithme de la bibliothèque standard.

```
bool check_zip(vector<int> const& x, vector<char> const& y);
```

## Avec un algorithme standard

Écrire une fonction qui zip deux vectors, en utilisant un algorithme de la bibliothèque standard.

```
vector<pair<int, char>> zip1(vector<int> const& x, vector<
```

```
char> const& y);
```

## Avec des boucles

Idem, en utilisant une boucle for sur itérateur (zip2a), une boucle for avec indices (zip2b). Idem avec while (zip2c et zip2d).

Quelle est la syntaxe que vous préférez et pourquoi ?

Est-il possible d'utiliser un “range-based for” ?

## Générique sur les types

Écrire une fonction qui est générique pour les types.

```
vector<pair<T, U>> zip3(vector<T> const& x, vector<U> const& y);
```

## Générique sur les collections

Écrire une fonction qui est générique pour les collections (en entrée uniquement ?). S'inspirer des algorithmes de la bibliothèque standard.

```
void zip3(InputIterA first_x, InputIterA last_x,
          InputIterB first_y, InputIterB last_y,
          OutputIter first_z);
```

## Adaptateur

La fonction de la question précédente n'a pas la même syntaxe que les fonctions des questions avant (vector en paramètre mutable plutôt qu'en retour de fonction), par homologie avec les algorithmes standards (et pour éviter la copie).

Écrire un **adaptateur** permettant de convertir le zip de la question

précédente (donc sans réécrire le zip).

```
vector<pair<T, U>> z = (...) zip3(x1, x2, y1, y2, z1) (...);
```

## Paire d'itérateurs (Range)

Utiliser des paires d'itérateurs sur chaque collection en paramètre, plutôt que des itérateurs (sauf pour la sortie).

```
void zip5(std::pair<InputIterA, InputIterA> x,
          std::pair<InputIterB, InputIterB> y,
          OutputIter first_z);
```

## Paire d'itérateurs

Idem, mais chaque paire représente le début ou la fin dans les deux collections.

```
void zip5b(std::pair<InputIterA, InputIterB> first,
           std::pair<InputIterA, InputIterB> last,
           OutputIter first_z);
```

Quelle syntaxe vous semble la plus pratique ?

Est-il possible d'écrire zip5 et zip5b en utilisant zip3 et des adaptateurs ?

## Meta-programmation

Prendre un nombre indéfini de collections en entrée. Plusieurs approches sont possibles :

- réécrire zip5 avec std::tuple ;
- réécrire zip5b avec std::tuple ;
- prendre un seul std::tuple en paramètre, contenant successivement le début et la fin de chaque collection ;
- en utilisant un variadic template.

Écrire chaque approche. Laquelle préférez-vous ?

Reference : [Boost Zip Iterator](#)

Note : cette question sort des limites de ce cours.

[Chapitre précédent](#) [Sommaire principal](#) [Chapitre suivant](#)

[Cours, C++](#)

# Jouons avec les chiffres

Quelques problèmes mathématiques.

## TWO + TWO = FOUR

Substitution de lettres par des chiffres. trouver à quoi correspond chaque lettre. Attention au 0 en début de nombre.

Cf <http://nrich.maths.org/781/note>

Autres exemples de chaînes :

- ONE + ONE = TWO
- ONE + TWO = THREE
- ONE + THREE = FOUR
- FOUR + FIVE = NINE

faire plusieurs chapitres sur l'internationalisation. Séparer les chaines, la localisation, le formatage. Peut être faire un chapitre sur ICU et sur Qt::tr ?

# Les chaînes de caractères internationales

De nos jours, avec les progrès des moyens de communication, en particulier d'internet, il est très facile de partager ses programmes à l'international. Des outils de partage du code, comme par exemple GitHub que vous verrez par la suite, permettent de mettre en place des équipes de développement collaboratif, même sur des projets lancés par une seule personne. Il est donc conseillé, dès le début d'un projet, de penser en termes de travail en équipe et d'écrire ses programmes en anglais (code, commentaires, documentation).

Le corollaire à cela est qu'un programme pourra facilement être utilisé par des personnes qui ne parlent pas l'anglais (en premier lieu vous peut-être) et il sera intéressant de pouvoir afficher des chaînes de caractères dans d'autres langages.

Et c'est à ce niveau que la gestion des chaînes de caractères peut devenir complexe. Par défaut, les chaînes en C++ sont basées sur le système anglais, qui ne contient aucun caractère accentué. En français par exemple, nous utilisons des accents aigu, grave ou circonflexe, des trémas. D'autres langues utilisent plus de types d'accents (Suédois). D'autres encore utilisent des alphabets complètement différents (Russe, Chinois).

En plus des alphabets différents, il faut également gérer d'autres problématiques. Par exemple, en anglais, le point est utilisé comme séparateur décimal. En français, nous utilisons la virgule. En anglais américain, une date s'écrit sous la forme mois-jour-année. En français, nous écrivons jour-mois-année.

Certaines langages s'écrivent de gauche à droite, d'autres de droite à gauche, d'autres encore de haut en bas.

On voit bien, par ces quelques exemples, que la gestion des langages est quelque chose de complexe. Dans ce chapitre, nous allons commencer par étudier les principaux types d'encodage des caractères et les chaînes utilisables dans ce contexte en C++.

Pour une gestion complète des langues ayant des alphabets complexes, il sera préférable d'utiliser des bibliothèques dédiées, comme [ICU](#). Ce chapitre est une simple introduction aux problématiques posées lorsque l'on souhaite prendre en compte l'internationalisation des programmes.

## La norme ASCII

Vous avez vu dans les chapitres précédents que le type `char` est un type d'entier un peu particulier. Il est possible de manipuler une variable de ce type comme un nombre (initialisation avec un nombre, addition, soustraction, etc.), mais lorsque l'on affiche une variable de ce type, cela affiche un caractère. Il est également possible d'afficher la valeur numérique d'un caractère de la façon suivante :

main.cpp

```
#include <iostream>

int main() {
    int const i { 'a' };
    std::cout << std::showbase << std::hex << i << ' ' <<
    std::dec << i << std::endl;
}
```

Dans ce code, le littérale caractère 'a' est de type `char`. Il est converti en type `int` lors de l'initialisation de la variable `i`. Ce code ne produit pas d'erreur de conversion implicite puisque le type `char` est plus petit que le type `int`, il n'y a pas de risque de perte d'information.

Ce code affiche :

0x61 97

En d'autres termes, cela veut dire que la valeur hexadécimale 0x61 (97 en décimal) sera interprétée comme étant le caractère 'a' pour le type `char`. Cette correspondance entre la valeur en mémoire et le caractère qui est affiché s'appelle l'encodage des caractères.

L'opération inverse est également possible : on peut initialiser une variable de type `char` avec une valeur entière et afficher le caractère correspondant à cette valeur. Par exemple :

```
main.cpp
#include <iostream>

int main() {
    char const c { 0x61 };
    std::cout << c << std::endl;
}
```

affiche :

a

Naturellement, en utilisant la même valeur numérique, le caractère affiché est le même.

L'encodage des caractères est une simple convention, dans l'absolu rien n'empêche pour chaque ordinateur et système d'exploitation d'utiliser son propre encodage. Mais cela voudrait dire qu'une chaîne affichée sur un ordinateur ne sera pas la même sur un autre ordinateur. Cela serait compliqué de créer des programmes dans cette situation.

Heureusement, la situation n'est pas aussi catastrophique. Pour permettre d'afficher une chaîne correctement sur différents systèmes, différents systèmes d'encodage ont été normalisés. Le plus connu est l'ASCII ([American Standard Code for Information Interchange](#)), qui est l'encodage par défaut en C++ et donc celui que l'on a utilisé depuis le début sans le savoir.

Le tableau suivant permet de retrouver la correspondance entre valeur numérique et caractère affiché. Pour le lire, il faut regarder les en-têtes de ligne et de colonne pour trouver la valeur. Par exemple, le caractère 'a' se trouve dans la ligne "0x6." et la colonne "0x.1", ce qui correspond donc à la valeur "0x61", ce que l'on a vu dans les codes précédents.

	<b>0x.0</b>	<b>0x.1</b>	<b>0x.2</b>	<b>0x.3</b>	<b>0x.4</b>	<b>0x.5</b>	<b>0x.6</b>	<b>0x.7</b>	<b>0x.8</b>	<b>0x.9</b>	<b>0x.A</b>	<b>0x.B</b>	<b>0x.C</b>	<b>0x.D</b>	<b>0x.E</b>	<b>0x.F</b>
<b>0x2.</b>	espace	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
<b>0x3.</b>	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
<b>0x4.</b>	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
<b>0x5.</b>	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
<b>0x6.</b>	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
<b>0x7.</b>	p	q	r	s	t	u	v	w	x	y	z	{		}	~	

Il est possible d'écrire directement des littérales caractère ou chaîne en utilisant les valeurs hexadécimales des caractères en utilisant la syntaxe suivante : "\x" suivi du code hexadécimal. Par exemple :

main.cpp

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

int main() {
    std::cout << '\x24' << '\x58' << std::endl; // 
littérales caractère
    std::cout << "\x64\x34" << std::endl;           // littérale
chaîne
}
```

affiche :

\$X  
d4

Les valeurs comprises entre 0x00 et 0x1F correspondent à des caractères spéciaux, dont l'utilisation est réservée.

## Les normes d'encodage sur 8 bits

	ASCII	Accent		Arabe		Chinois		
Caractères affichés (UTF-8)	a	é	ç			漢		
Représentation en mémoire	0x61	0xC3	0xA9	0xD8	0xB9	0xE6	0xBC	0xA2
OEM 850 (console Windows)	a	ㅏ	®	ї	ۼ	ߤ	ۼ	ó
Windows-1252 (Qt Creator)	a	Ã	©	Ø	¹	æ	¼	¢

Les plus attentifs auront peut-être remarqué un détail sur la norme ASCII. Si on vérifie les valeurs limites du type `char`, on peut remarquer que celui-ci permet de coder 256 valeurs possibles (de -128 à +127). Or, dans la norme ASCII, seules les valeurs de 0x00 (0 ou 0b00000000) à 0x7F (127 ou 0b01111111) sont définies, les valeurs de 0x80 (0b10000000) à 0xFF (0b11111111) ne sont pas définies dans la norme. Cela revient à dire que la norme ASCII encode les caractères sur 7 bits dans un type sur 8 bits.

L'autre problème avec la norme ASCII est qu'elle ne permet pas d'encoder les accents ou les caractères différents de l'alphabet latin. Historiquement, cela s'explique du fait que l'anglais s'est imposé dans le passé comme langue par défaut de l'informatique. Cependant, avec le temps et la démocratisation de l'informatique, il a été nécessaire de pouvoir afficher d'autres langages que l'anglais, ce qui a abouti à l'apparition de nombreuses normes d'encodage.

La première solution pour ajouter ces caractères supplémentaires a donc été d'utiliser le huitième bit qui est inutilisé dans la norme ASCII.

[http://fr.wikipedia.org/wiki/Page\\_de\\_code\\_850](http://fr.wikipedia.org/wiki/Page_de_code_850) EOM850, par défaut dans la console windows

[http://fr.wikipedia.org/wiki/ISO\\_8859](http://fr.wikipedia.org/wiki/ISO_8859) ISO 8859, codage internationale, selon la version (ISO 8859-1 à ISO 8859-16 pour les autres alphabets)

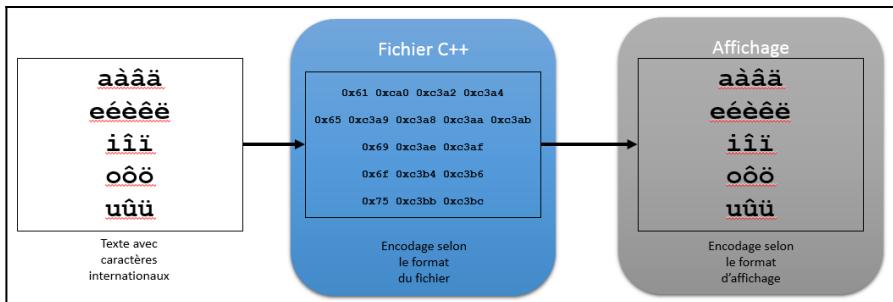
<http://fr.wikipedia.org/wiki/Windows-1252> spécifique à Windows, dérivé de ISO 8859-1 (pas international)

## Les normes d'encodage multi-octets

Représentation des caractères internationaux sur plusieurs octets.

[http://fr.wikipedia.org/wiki/Table\\_des\\_caract%C3%A8res\\_Unicode\\_%2800-0FFF%29](http://fr.wikipedia.org/wiki/Table_des_caract%C3%A8res_Unicode_%2800-0FFF%29) et <http://fr.wikipedia.org/wiki/UTF-8> sur 8 bits, respecte ASCII.  
Si bit0 = 1, alors encodage sur plusieurs octets

UTF-16 et UTF-32 : encodage de base sur 16 ou 32



Screenshot of an online C++ editor:

- Code Area:**

```
Donate                                     Coliru                                         Restore defaults   Help   Feedback
€                                              Editor   Command   Q&A   Read Write
1 #include <iostream>
2
3 int main() {
4     std::cout << "Texte en français aaâä eeèêë iiî ooô uuû" << std::endl;
5     std::cout << "En japonais : ああかカササタナナハハママヤヤララワワん" << std::endl;
6 }
```
- Output Area:** Displays the text "Texte en français aaâä eeèêë iiî ooô uuû" and "En japonais : ああかカササタナナハハママヤヤララワワん".
- Compiler Area:** Shows the command "clang++ main.cpp -Wall -Wextra -pedantic -std=c++1y && ./a.out".
- Buttons:** "Compile, link and run..." and "Share!".

✓edit ⌂fork ↴download ⌂copy

```
1. #include <iostream>
2.
3. int main() {
4.     std::cout << "Texte en français ààââ èéèè iiii ôôûû" << std::endl;
5.     std::cout << "En japonais : ああかかさたななははまややらうわわん" << std::endl;
6. }
```

Success comments (0)

stdin copy  
Standard input is empty

stdout copy  
Texte en français ààââ èéèè iiii ôôûû  
En japonais : ああかかさたななははまややらうわわん

main.cpp main(): int

```
1. #include <iostream>
2.
3. int main()
4. {
5.     std::cout << "Texte en français ààââ èéèè iiii ôôûû" << std::endl;
6.     std::cout << "En japonais : ああかかさたななははまややらうわわん" << std::endl;
7. }
```

G:\Qt\Tools\QtCreator\bin\qtcreator\_process\_stub.exe

Texte en français ààââ èéèè iiii ôôûû  
En japonais : ああかかさたななははまややらうわわん

Appuyez sur <ENTRÉE> pour fermer cette fenêtre...

ConsoleApplication1.cpp

```
1 #include <iostream>
2
3 int main() {
4     std::cout << "Texte en français ààââ èéêê iîî ôôö üûû" << std::endl;
5     std::cout << "En japonais : あアカカササタタナナハハママヤヤララワワン" << std::endl;
6 }
7
```

C:\Windows\system32\cmd.exe

```
Texte en français ààââ èéêê iîî ôôö üûû
En japonais : ??????????????????????
Appuyez sur une touche pour continuer... -
```

main.cpp

```
1 #include <iostream>
2
3 int main() {
4     std::cout << "Texte en français ààââ èéêê iîî ôôö üûû" << std::endl;
5     std::cout << "En japonais : あアカカササタタナナハハママヤヤララワワン" << std::endl;
6 }
7
```

C:\Users\Guillaume\Desktop\testsss\bin\Debug\testsss.exe

```
Texte en français ààââ èéêê iîî ôôö üûû
En japonais : ??????????????????????

Process returned 0 <0x0> execution time : 0.017 s
Press any key to continue.
-
```

## wstring, u32string, u12string

caractère de 16 ou 32 bits (wchar\_t), préfixe L'a' et L"bla bla"

également wcout, wcerr, wclog, wofstream, etc

- char : 8 bits, UTF-8

- wchar : implémentation spécifique
- char16\_t : 16 bits, UTF-16
- char32\_t : 32 bits, UTF-32

Distinguer affichage et données en mémoire

## Utilisation des expressions régulières

main.cpp

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

int main()
{
    std::regex pattern { "□" };
    std::cout << std::boolalpha << std::regex_search("□□□□□", pattern) << std::endl;
    std::cout << std::boolalpha << std::regex_search("□□□□□□□□", pattern) << std::endl;
}
```

Conversion :

main.cpp

```
#include <iostream>
#include <string>
#include <regex>
#include <map>

int main()
{
    std::wstring source { L"□□□□" }; // bonjour
    std::map<wchar_t, std::string> katakana = {
        {L'□', "ko"}, {L'□', "n"}, {L'□', "ni"}, {L'□', "chi"}, {L'□', "ha"}}
```

```
    };
    std::string result {};
    for (auto c: source) {
        result += katakana[c];
    }
    std::cout << result << std::endl;
}
```

## Localisation (pour plus tard)

Affichage d'une chaîne dans une autre langue. Par défaut, écrire en anglais dans ses codes et prévoir une fonctionnalité de traduction. Lib dispo

Éviter d'écrire : `cout << "bla bla" << i << "bla bla" << endl;` parce que dans une autre langue, l'ordre peut être différent. Par exemple : `cout << "bla bla bla bla" << i << endl.` Écrire une chaîne avec placeholder : `cout << tr("bla bla %1 bla bla, i) << endl;`.

Exercices : implémentation simple `tr()` avec `find/replace`. Implémenter `tr()` avec `regex`.

[Chapitre précédent](#) [Sommaire principal](#) [Chapitre suivant](#)

[Cours, C++](#)

Remplir une chaîne avec des caractères

Il est possible d'initialiser une chaîne en la remplissant avec un caractère répété n fois. Dans ce cas, la syntaxe pour initialiser la chaîne est différente de la syntaxe que vous avez déjà utilisée. Lorsque l'on initialise une variable avec une littérale ou avec une expression, vous avez utilisé la syntaxe avec des crochets :

```
string s { "hello, world!" };
```

Dans le cas d'une initialisation avec des caractères, il faut initialiser la variable comme si c'était une fonction, en utilisant des parenthèses pour passer les arguments. Les arguments à passer sont :

- le nombre de caractères à placer dans la chaîne ;
- le caractère à utiliser.

Par exemple :

main.cpp

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

int main()
{
    std::string const s1 (10, 'a');
    std::cout << s1 << std::endl;
    std::string const s2 (5, 'b');
    std::cout << s2 << std::endl;
}
```

affiche :

```
aaaaaaaaaa
bbbb
```

Fondamentalement, cela revient à initialiser une chaîne en donnant une littérale contenant une suite identique de caractères :

```
string const s1 { "aaaaaaaaaa" };
```

```
string const s2 { "bbbbbb" };
```

L'intérêt d'initialiser en donnant un nombre de caractères à générer est de permettre de créer une chaîne contenant un nombre très important de caractères (imaginez par exemple si vous devez initialiser une chaîne avec une centaine de caractères).

Pour faire la même chose avec une chaîne existante, il faut appeler la fonction membre `assign`. Les arguments à utiliser sont les mêmes :

main.cpp

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

int main()
{
    std::string s {};
    s.assign(7, 'c');
    std::cout << s << std::endl;
}
```

affiche :

ccccccc

[Chapitre précédent](#) [Sommaire principal](#) [Chapitre suivant](#)

# Les expressions régulières 1

Lorsque nous lisons un texte, nous sommes capable de reconnaître la signification (ou sémantique) de certains motifs dans le texte. Par exemple, si on écrit “25/12/2014”, beaucoup de personnes reconnaîtront une date, correspondant au 25 décembre 2014. Si on écrit “18:30”, on reconnaît une heure : dix-huit heure trente. Ou encore, on reconnaît que “<http://www.google.fr>” est une URL internet.

Nous sommes capable de trouver la sémantique d'une chaîne parce que l'on connaît le motif qui caractérise cette chaîne. On a l'habitude d'écrire les dates en indiquant le jour, le mois et l'année (en français). On a l'habitude aussi de voir des URL écrites sous la forme “`http:/ /`” suivi de plusieurs mots séparés par des points ou des barres obliques.

Les expressions régulières sont un moyen efficace d'écrire de tels motifs. Avec ces motifs, il sera ensuite possible de vérifier qu'une chaîne respecte ce motif ou encore d'identifier les sous-chaînes qui respectent ce motif.

## Création et initialisation

### Origine du terme “expression régulière” ?

Les expressions régulières sont une fonctionnalité que l'on trouve dans beaucoup de langages de programmation modernes. En C++, une expression régulière correspond à la classe `regex` de la bibliothèque standard (dans le fichier d'en-tête `regex`). Il est possible de créer une expression régulière directement à partir d'une littérale chaîne de caractères ou d'une variable chaîne de type `string`.

main.cpp

```
#include <regex>
#include <string>
```

```
int main()
{
    std::regex pattern1 { "bla bla bla" }; // création à
partir d'une littérale

    std::string s { "bla bla bla" };
    std::regex pattern2 { s }; // création à
partir d'une string
}
```

## Utilisation des raw string

Pour bien comprendre les expressions régulières, il faut donner quelques définitions :

- une **séquence cible** (*target sequence*) est la chaîne de caractères sur laquelle est appliquée l'expression régulière.
- un **motif** (*pattern*) est la séquence de caractères représentant ce que l'on cherche à identifier.
- une **correspondance** (*match*) est une sous-chaîne de la séquence cible qui correspond au motif.

Plus concrètement, si l'on prend la chaîne suivante : “La date du 25/12/2014 est un jeudi” et que l'on demande d'écrire une expression régulière pour trouver la date dans cette chaîne, alors la chaîne “La date du 25/15/2014 est un jeudi” est la séquence cible et la correspondance est “25/12/2014”. Le motif est une chaîne qui signifie “trouver une date au format jour/mois/année”. Bien sûr, il n'est pas possible d'écrire un motif de cette façon, il faut utiliser une syntaxe spécifique, qui sera décrite dans la suite de ce chapitre.

Avec une expression régulière, on va donc pouvoir réaliser principalement trois opérations, chaque opération correspondant à une fonction. Ces différentes fonctions seront détaillées dans les prochains chapitres, la suite de ce chapitre sera consacrée à la syntaxe utilisable pour écrire un motif. Mais pour vous permettre de pratiquer et apprendre correctement les expressions régulières, nous allons voir rapidement une syntaxe possible de ces fonctions (il est possible d'utiliser ces fonctions

de différentes façons, nous n'en verrons qu'une seule pour le moment).

La première fonctionnalité des expressions régulières est la **validation** d'une chaîne, c'est-à-dire vérifier qu'une chaîne respecte un motif. La fonction correspondante est la fonction `regex_match`. Une version simple de cette fonction prend en arguments la séquence cible et l'expression régulière et retourne une valeur booléenne (vrai si la séquence cible correspond au motif, faux sinon).

main.cpp

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

int main()
{
    std::regex pattern { "abc" }; // on recherche le motif
"abc"
    std::string target { "abcdef" };
    bool result = std::regex_match(target, pattern);
    std::cout << std::boolalpha << result << std::endl;

    target = "abc";
    result = std::regex_match(target, pattern);
    std::cout << std::boolalpha << result << std::endl;
}
```

affiche :

```
false
true
```

Le motif “abc” correspond donc à la séquence cible “abc”, mais pas à la séquence “abcdef”. Il permet donc de vérifier que la séquence cible correspond exactement à la chaîne “abc”. Ce n'est pas très utile, le but est surtout pour montrer la syntaxe de la fonction.

La deuxième utilité des expressions régulières est de **rechercher** les sous-chaînes de la séquence cible correspondant au motif. La fonction correspondante est `regex_search`. Celle-ci prend les mêmes arguments

que la fonction `regex_match` et retourne vraie si la fonction trouve au moins une sous-chaîne correspond au motif.

main.cpp

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

int main()
{
    std::regex pattern { "abc" }; // on recherche le motif
"abc"
    std::string target { "abcdef" };
    bool result = std::regex_search(target, pattern);
    std::cout << std::boolalpha << result << std::endl;

    target = "abc";
    result = std::regex_search(target, pattern);
    std::cout << std::boolalpha << result << std::endl;
}
```

affiche :

```
true
true
```

Le code est le même que précédemment, en remplaçant `regex_match` par `regex_search`. Par contre, le résultat est très différent : la séquence cible “abcdef” ne correspondait pas au motif lorsque l'on utilise `regex_match`, mais il correspond en utilisant `regex_search` (il existe bien le motif “abc” dans la séquence “abcdef”).

Pour terminer, la troisième utilisation principales des expressions est le remplacement des sous-chaînes correspondant à un motif par d'autres chaînes. La fonction correspondante est `regex_replace`, qui prend en argument la séquence cible, l'expression régulière (comme pour les précédentes fonctions) et la chaîne de remplacement, puis retourne la chaîne après modifications.

main.cpp

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

int main()
{
    std::regex pattern { "abc" }; // on recherche le motif
"abc"
    std::string target { "abcdefabc" };
    std::string replacement { "123" };
    std::string result = std::regex_replace(target, pattern,
replacement);
    std::cout << result << std::endl;
}
```

affiche :

```
123def123
```

Ce qui correspond bien à la chaîne “abcdefabc”, en remplaçant les sous-chaînes “abc” par “123”.

## Les différentes syntaxes possibles

Maintenant que vous avez une idée générale du fonctionnement des expressions régulières, il va falloir apprendre à écrire des motifs. La syntaxe à utiliser n'est pas très compliquée quand on a l'habitude, mais elle est écrite sous forme assez compacte dans une chaîne, ce qui ne facilite pas la lecture. Il ne faut donc pas hésiter à s'attarder un peu sur ce chapitre et pratiquer un maximum d'exercices, le temps de bien assimiler la syntaxe des expressions régulières.

Même si les expressions régulières sont utilisables dans de nombreux langages, cela ne veut pas dire pour autant que la syntaxe pour écrire des motifs soit la même dans tous ces langages. Il existe en pratique plusieurs syntaxes possibles, par exemple :

- ECMAScript, issu de l'éditeur de texte Emacs ;
- basic posix et extended posix, définis dans la norme POSIX pour Linux ;
- grep et egrep, issus du programme en ligne de commande grep (recherche de chaînes sous Linux) ;
- awk, issu du programme en ligne de commande du même nom.

La classe `std::regex` du C++ prend en charge différentes syntaxes, que l'on peut spécifier comme second argument lors de la création d'une expression régulière. Par exemple :

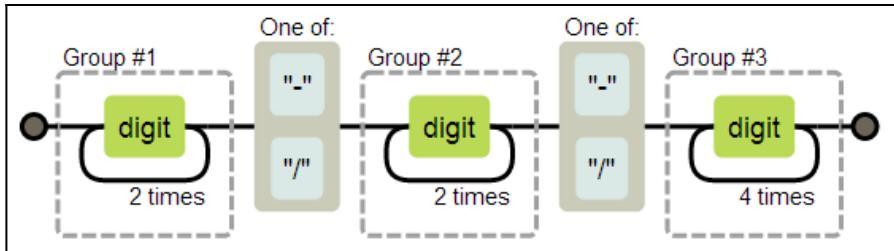
```
std::regex("abc", std::regex::ECMAScript);
```

La liste des syntaxes acceptées par `std::regex` est disponible dans la [documentation de cette classe](#). Par défaut, c'est la syntaxe ECMAScript qui est utilisée, nous utiliserons donc uniquement cette syntaxe dans ce cours. Mais sachez que d'autres syntaxes sont possibles.

## Visualiser les expressions régulières

Lorsque l'on débute avec les expressions régulières, il est parfois plus facile de visualiser le motif sous forme graphique. Si vous êtes plus à l'aise pour apprendre de cette façon, il existe des outils qui permettent de générer une représentation graphique à partir d'une expression régulière.

Par exemple, l'expression régulière `"(\d{2})[-/](\d{2})[-/](\d{4})"`, qui permet de valider une date, peut être représentée sous la forme suivante :



Ce graphique se lit de gauche vers la droite, il suffit de suivre les chemins possibles pour lire ce motif. Ce motif est donc constitué de trois groupes séparés par les caractères `-` ou `/`. Chaque groupe est constitué de chiffres (“digit”) répété 2 fois pour les deux premiers groupes et 4 fois pour le dernier groupe.

L'ensemble des graphiques de ce cours pour les expressions régulières sont générés automatiquement par le site <http://www.regexper.com/> et sont sous licence Creative Common (CC BY 3.0).

## Les caractères de base

La forme la plus simple de motif est constitué de caractères alphanumériques de base (a, b, E, G, 3, 7, etc). Chaque caractère du motif est recherché dans la séquence cible à l'identique. Ainsi, le motif “abc” permettra de rechercher dans une chaîne cette séquence exacte.

`main.cpp`

```
#include <iostream>
#include <regex>

int main()
{
    std::regex const pattern { "a" };
    std::cout << "'a' match with ': '" << std::boolalpha <<
        std::regex_match("", pattern) << std::endl;

    std::cout << "'a' match with 'a': '" << std::boolalpha <<
        std::regex_match("a", pattern) << std::endl;
```

```

        std::cout << "'a' match with 'b': " << std::boolalpha <<
            std::regex_match("b", pattern) << std::endl;

        std::cout << "'a' match with 'ab': " << std::boolalpha
<<
            std::regex_match("ab", pattern) << std::endl;
}

```

affiche :

```

'a' match with '' : false
'a' match with 'a' : true
'a' match with 'b' : false
'a' match with 'ab' : false

```

Ainsi, le motif “a” ne peut correspondre que si la séquence cible correspond exactement à “a”, les chaînes “”, “b” et “ab” ne correspondent pas.

Attention de bien faire attention à la fonction que l'on utilise. Si on réalise une recherche de sous-chaînes (avec `regex_search`) au lieu d'une validation de chaîne (avec `regex_match`), le résultat obtenu n'est pas identique.

`main.cpp`

```

#include <iostream>
#include <string>
#include <regex>

int main()
{
    std::regex pattern { "a" };
    std::cout << "'a' match with 'ab': " << std::boolalpha
<<
        std::regex_match("ab", pattern) << std::endl;

    std::cout << "search 'a' in 'ab': " << std::boolalpha <<
        std::regex_search("ab", pattern) << std::endl;
}

```

affiche :

```
'a' match with 'ab': false
search 'a' in 'ab': true
```

Dans le premier cas, la chaîne “ab” ne correspond pas au motif “a” (elles ne sont pas identiques). Dans le second cas, le motif “a” est retrouvé dans la chaîne “ab” (la séquence cible contient le motif).

Il est possible d'utiliser la grande majorité des caractères dans un motif (même des caractères étrangers, ce point sera détaillé dans la partie “Internationalisation”). Cependant, certains caractères ont une signification particulière (que vous verrez par la suite) dans un motif et ne peuvent être utilisés directement.

Imaginons que vous souhaitez par exemple vérifier qu'une chaîne contient un point. Vous pourriez écrire le code suivant par exemple :

main.cpp

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

int main()
{
    std::regex const pattern { "." };
    std::string target { "cette phrase contient un point" };
    std::cout << target << " : " << std::boolalpha <<
        std::regex_search(target, pattern) << std::endl;

    target = "cette phrase contient un point.";
    std::cout << target << " : " << std::boolalpha <<
        std::regex_search(target, pattern) << std::endl;
}
```

On pourrait s'attendre à ce que, la recherche échoue dans le premier cas et réussit dans le second. Cependant, le résultat affiché ne correspond pas à ce que l'on attend.

```
cette phrase contient un point : true
cette phrase contient un point. : true
```

il y a déjà beaucoup à dire sur les regex, parler des caractères d'échappement et des raw string plus tôt

pas clair, à réécrire

Donc la recherche réussit dans les deux, alors que la première séquence cible ne contient pas de point. La raison est que le point est un caractère spécial et ne permet pas de rechercher un point dans la séquence cible. Pour rechercher le caractère point ., il faut le faire précédé du caractère barre oblique inversée \. Le motif pour rechercher un point dans une chaîne est donc "\." et non pas simplement ".".

Cependant, le caractère barre oblique inversée possède également un signification particulière en C++. On l'appelle le caractère d'échappement. Il permet d'entrer un caractère spéciale, comme les tabulations ou les retours à la ligne (comme vu dans les chapitres précédents). Ce caractère d'échappement s'associe avec le caractère qui le suit pour ne former qu'un seul caractère. Ainsi, '\n' ou '\n' ne sont pas deux caractères (\ puis n) mais bien un seul (si vous essayez d'écrire '/n', vous obtiendrez une erreur, puisqu'il n'est pas possible de mettre deux caractères dans une littérale caractère. Par contre '\n' ne pose pas de problème puisque c'est considéré comme un seul caractère).

Pour écrire le caractère ., il faut donc écrire \\ en C++, ce qui fait que le motif "\.." devient "\\.." en C++. Cette chaîne doit être lue de la façon suivante : le premier "\\" correspond au caractère d'échappement, donc "\\.." correspond au caractère "\.." dans le motif, et donc le motif "\.." permet de rechercher un point.

Une autre solution en C++, pour éviter de devoir utiliser les caractères d'échappement, est d'utiliser les littérales chaînes brutes (*raw string*). Dans ce cas, les caractères spéciaux du C++ (\ ou " par exemple) sont interprétés comme des caractères normaux. Pour écrire une littérale chaîne brute, il faut remplacer "..." par R"(...)"

Ainsi, au lieu d'écrire "\\..", il est possible d'écrire R"(\..)". le code devient alors :

main.cpp

```

#include <iostream>
#include <string>
#include <regex>

int main()
{
    std::regex const pattern { R"(\.)" };
    std::string target { "cette phrase contient un point" };
    std::cout << target << " : " << std::boolalpha <<
        std::regex_search(target, pattern) << std::endl;

    target = "cette phrase contient un point.";
    std::cout << target << " : " << std::boolalpha <<
        std::regex_search(target, pattern) << std::endl;
}

```

Ce code affiche le résultat attendu :

```

cette phrase contient un point : false
cette phrase contient un point. : true

```

Les caractères spéciaux des expressions régulières, qu'il faut ajouter \ sont les suivants :

<b>Caractère utilisé dans les motifs</b>	<b>Caractère recherché dans la séquence cible</b>
\^	^
\\$	\$
\\	\
\.	.
\*	*
\+	+
\?	?
\(	(
\)	)
\[	[
\]	]
\{	{
\}	}

VI

I

[Chapitre précédent](#) [Sommaire principal](#) [Chapitre suivant](#)

Cours, C++

# Les expressions régulières 2

## Caractère générique, classe de caractères et ensemble de caractères

Chaque caractère de base correspond à un seul caractère dans la séquence terminale, mais il est parfois nécessaire de pouvoir exprimer dans un motif que l'on souhaite n'importe quel caractère issu d'un ensemble. Par exemple, pour créer un motif permettant de valider une date, il faut pouvoir écrire que l'on souhaite avoir n'importe quelle chiffre. Peu importe le chiffre, du moment que c'est un chiffre. Les caractères génériques, classes de caractères et ensembles de caractères permettent d'écrire cela.

Le caractère générique correspond au point dans une expression régulière. Il signifie "n'importe quel caractère", mais uniquement un seul caractère. Par exemple, le motif "ab.de" peut correspondre aux chaînes "abcde" ou "ab\$de", mais pas à la chaîne "ab\$\$de".

Ceci explique pourquoi, dans le code précédent, le motif ". ." était retrouvé dans n'importe quelle chaîne, puisque cela signifie simple "trouve un caractère quelconque dans la chaîne". Donc seule la chaîne vide "" retourne faux avec ce motif.

main.cpp

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

int main()
{
    std::regex pattern { "." };
    std::cout << '.' ' match with ': ' << std::boolalpha <<
```

```

        std::regex_match("", pattern) << std::endl;

    std::cout << "'.' match with 'a': " << std::boolalpha <<
        std::regex_match("a", pattern) << std::endl;

    std::cout << "'.' match with 'ab': " << std::boolalpha
<<
        std::regex_match("ab", pattern) << std::endl;

    pattern = "abc";
    std::cout << "'abc' match with 'abc': " << std::
boolalpha <<
        std::regex_match("abc", pattern) << std::endl;

    std::cout << "'abc' match with 'a$c': " << std::
boolalpha <<
        std::regex_match("a$c", pattern) << std::endl;

    pattern = "a.c";
    std::cout << "'a.c' match with 'abc': " << std::
boolalpha <<
        std::regex_match("abc", pattern) << std::endl;

    std::cout << "'a.c' match with 'a$c': " << std::
boolalpha <<
        std::regex_match("a$c", pattern) << std::endl;
}

```

affiche :

```

'.' match with '' : false
'.' match with 'a' : true
'.' match with 'ab' : false
'abc' match with 'abc' : true
'abc' match with 'a$c' : false
'a.c' match with 'abc' : true
'a.c' match with 'a$c' : true

```

On voit donc bien que la chaîne correspond au motif uniquement si elle ne contient un et un seul caractère, pas plus, pas moins.

Les ensembles de caractères (*character set*) permettent de représenter

une liste de caractères entre crochets droits `[]`. Chaque ensemble permet de remplacer un et un seul caractère. Par exemple, le motif "`[abc]`" représente un seul caractère, qui peut être a, b ou c.

### main.cpp

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

int main()
{
    std::regex const pattern { "[abc]" };
    std::cout << "'[abc]' match with ''': " " << std::boolalpha
<<
    std::regex_match("", pattern) << std::endl;

    std::cout << "'[abc]' match with 'a': " " << std::boolalpha <<
        std::regex_match("a", pattern) << std::endl;

    std::cout << "'[abc]' match with 'b': " " << std::boolalpha <<
        std::regex_match("b", pattern) << std::endl;

    std::cout << "'[abc]' match with 'z': " " << std::boolalpha <<
        std::regex_match("z", pattern) << std::endl;

    std::cout << "'[abc]' match with 'ab': " " << std::boolalpha <<
        std::regex_match("ab", pattern) << std::endl;
}
```

affiche :

```
'[abc]' match with ''': false
'[abc]' match with 'a': true
'[abc]' match with 'b': true
'[abc]' match with 'z': false
'[abc]' match with 'ab': false
```

Si l'on souhaite écrire un motif qui permet de valider n'importe quelle lettre minuscule ou n'importe quel chiffre, il est fastidieux d'écrire tous les caractères correspondant : "[abcdefghijklmnopqrstuvwxyz]" et "[0123456789]". Pour éviter cela, il est possible de spécifier une plage de valeur en indiquant le premier et le dernier caractère de la plage, séparés par un tiret. Par exemple : "[a-z]" ou "[0-9]".

### main.cpp

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

int main()
{
    std::regex pattern { "[a-e]" };
    std::cout << "'[a-e]' match with ''': " << std::boolalpha
<<
    std::regex_match("", pattern) << std::endl;

    std::cout << "'[a-e]' match with 'a': " << std::boolalpha <<
        std::regex_match("a", pattern) << std::endl;

    std::cout << "'[a-e]' match with 'e': " << std::boolalpha <<
        std::regex_match("e", pattern) << std::endl;

    std::cout << "'[a-e]' match with 'z': " << std::boolalpha <<
        std::regex_match("z", pattern) << std::endl;

    std::cout << "'[a-e]' match with 'ab': " << std::boolalpha <<
        std::regex_match("ab", pattern) << std::endl;

    pattern = "ab[c-f]";
    std::cout << "'ab[c-f]' match with ''': " << std::boolalpha <<
        std::regex_match("", pattern) << std::endl;

    std::cout << "'ab[c-f]' match with 'abc': " << std::boolalpha <<
```

```

boolalpha <<
    std::regex_match("abc", pattern) << std::endl;

    std::cout << "'ab[c-f]' match with 'abz': " << std::endl;
boolalpha <<
    std::regex_match("abz", pattern) << std::endl;
}

```

affiche :

```

'[a-e]' match with '' : false
'[a-e]' match with 'a' : true
'[a-e]' match with 'e' : true
'[a-e]' match with 'z' : false
'[a-e]' match with 'ab' : false
'ab[c-f]' match with '' : false
'ab[c-f]' match with 'abc' : true
'ab[c-f]' match with 'abz' : false

```

Il est possible de spécifier plusieurs plages dans un ensemble, par exemple "[a-er-z]" correspond à un caractère qui peut être a, b, c, d, e, r, s, t, u, v, w, x, y ou z.

Certains ensembles de classes sont souvent utilisées, il existe donc des raccourcis pour éviter de les réécrire à chaque fois. Ce sont les classes de caractères (*character class*). Par exemple, le motif `[:alpha:]` correspond à n'importe quel caractère alphabétique, donc est équivalent à `[a-zA-Z]`. Le tableau suivant présente l'ensemble des classes de caractères possible :

<b>Classe de caractères</b>	<b>Description</b>	<b>Ensemble équivalent</b>
<code>[:alnum:]</code>	Caractères alphanumériques	<code>[a-zA-Z0-9]</code>
<code>[:alpha:]</code>	Caractères alphabétiques	<code>[a-zA-Z]</code>
<code>[:ascii:]</code>	Caractères ASCII	<code>[\x00-\x7F]</code>
<code>[:blank:]</code>	Espace et tabulation	<code>[ \t]</code>

[:cntrl:]	Caractères de contrôle	[\x00-\x1F\x7F]
[:digit:]	Chiffres	[0-9]
[:graph:]	Caractères visibles	[\x21-\x7E]
[:lower:]	Caractères minuscules	[a-z]
[:print:]	Caractères visibles et espace	[\x20-\x7E]
[:punct:]	Ponctuation et symboles	[ !#\$%&'()*)+, \-. /:;?@[\\\\]^_`{ }~]
[:space:]	Caractères blancs	[ \t\r\n\v\f]
[:upper:]	Caractères majuscules	[A-Z]
[:word:]	Lettre, chiffre ou tiret bas	[A-Za-z0-9_]
[:xdigit:]	Chiffres hexadécimaux	[A-Fa-f0-9]

Ces classes de caractères permettent de simplifier l'écriture des motifs.

### main.cpp

```
#include <iostream>
#include <regex>

int main()
{
    std::regex pattern { "[[:alpha:]]" };
    std::cout << "'[[:alpha:]]' match with 'a': " << std::boolalpha <<
        std::regex_match("a", pattern) << std::endl;

    std::cout << "'[[:alpha:]]' match with '1': " << std::boolalpha <<
        std::regex_match("1", pattern) << std::endl;

    std::cout << "'[[:alpha:]]' match with '&': " << std::boolalpha <<
        std::regex_match("&", pattern) << std::endl;
}
```

```

boolalpha <<
    std::regex_match("&", pattern) << std::endl;

    pattern = "[[:alnum:]]";
    std::cout << "'[[:alnum:]]' match with 'a': " << std::endl;
boolalpha <<
    std::regex_match("a", pattern) << std::endl;

    std::cout << "'[[:alnum:]]' match with '1': " << std::endl;
boolalpha <<
    std::regex_match("1", pattern) << std::endl;

    std::cout << "'[[:alnum:]]' match with '&': " << std::endl;
boolalpha <<
    std::regex_match("&", pattern) << std::endl;
}

```

affiche :

```

'[[:alpha:]]' match with 'a': true
'[[:alpha:]]' match with '1': false
'[[:alpha:]]' match with '&': false
'[[:alnum:]]' match with 'a': true
'[[:alnum:]]' match with '1': true
'[[:alnum:]]' match with '&': false

```

Il est possible de prendre la négation d'une classe de caractères, c'est-à-dire de pouvoir accepter tous les caractères sauf ceux de la classe, en utilisant le symbole `^`. Par exemple, `[^[:alpha:]]` signifie “tous les caractères non alphabétiques” :

`main.cpp`

```

#include <iostream>
#include <regex>

int main()
{
    std::regex const pattern { "[^[:alpha:]]" };
    std::cout << "'[[:alpha:]]' match with 'a': " << std::endl;
boolalpha <<
    std::regex_match("a", pattern) << std::endl;

```

```

        std::cout << "[[:alpha:]]' match with '1': " << std::
boolalpha <<
        std::regex_match("1", pattern) << std::endl;

        std::cout << "[[:alpha:]]' match with '&': " << std::
boolalpha <<
        std::regex_match("&", pattern) << std::endl;
}

```

affiche :

```
'[[:alpha:]]' match with 'a': false
'[[:alpha:]]' match with '1': true
'[[:alpha:]]' match with '&': true
```

Il est possible d'ajouter des caractères à une classe de caractères, en les spécifiant entre les premiers crochets droits. Par exemple, l'ensemble `[123[:alpha:]]` permet de représenter tous les caractères alphabétiques, ainsi que les caractères 1, 2 et 3.

### main.cpp

```

#include <iostream>
#include <regex>

int main()
{
    std::regex const pattern { "[123[:alpha:]]" };
    std::cout << "[123[:alpha:]]' match with 'a': " << std
::boolalpha <<
    std::regex_match("a", pattern) << std::endl;

    std::cout << "[123[:alpha:]]' match with '1': " << std
::boolalpha <<
    std::regex_match("1", pattern) << std::endl;

    std::cout << "[123[:alpha:]]' match with '4': " << std
::boolalpha <<
    std::regex_match("4", pattern) << std::endl;

    std::cout << "[123[:alpha:]]' match with '&': " << std

```

```
    ::boolalpha <<
        std::regex_match("&", pattern) << std::endl;
}
```

affiche :

```
'[123[:alpha:]]' match with 'a': true
'[123[:alpha:]]' match with '1': true
'[123[:alpha:]]' match with '4': false
'[123[:alpha:]]' match with '&': false
```

Pour terminer avec les classes de caractères, il existe une écriture simplifiée pour certaines classes. Ces écritures simplifiées s'écrivent avec la barre oblique inversée \ suivie d'un caractère. Lorsque le caractère est minuscule, cela correspond à une classe de caractères. Lorsqu'il est en majuscule, cela correspond à l'inverse de la classe de caractères correspondante. Le tableau suivant liste l'ensemble des écritures simplifiées :

Écriture simplifiée	Classe de caractères
\d	[ [:digit:]]
\D	[^[:digit:]]
\s	[ [:space:]]
\S	[^[:space:]]
\w	[_[:alnum:]]
\W	[^_[:alnum:]]

N'oubliez pas qu'il faut ajouter une barre oblique inversée en C++ pour échapper le caractère \ ou utiliser les littérales chaînes brutes. Par exemple, pour écrire \d, il faut écrire en C++ :

```
std::regex const pattern { "\\\d" };
// ou
std::regex const pattern { R"(\d)" };
```

main.cpp

```
#include <iostream>
#include <regex>

int main()
```

```

{
    std::regex const pattern { R"(\w)" };
    std::cout << R"('w' match with 'a': )" << std::boolalpha <<
        std::regex_match("a", pattern) << std::endl;

    std::cout << R"('w' match with '1': )" << std::boolalpha <<
        std::regex_match("1", pattern) << std::endl;

    std::cout << R"('w' match with '&': )" << std::boolalpha <<
        std::regex_match("&", pattern) << std::endl;
}

```

affiche :

```

'\w' match with 'a': true
'\w' match with '1': true
'\w' match with '&': false

```

## Les répétitions

Jusqu'à maintenant, les syntaxes que l'on a vu ne permettent de remplacer qu'un seul caractère. Si l'on veut écrire un motif correspondant à trois lettres, il faudra écrire : `[[:alpha:]][[:alpha:]][[:alpha:]]`. C'est un peu lourd à écrire (surtout si l'on veut 20 lettres par exemple) et cela ne permet pas de spécifier des motifs comme "5 à 10 lettres" ou "tous les caractères jusqu'au premier point".

Une répétition permet de répéter un caractères ou un groupe de caractères. Il existe plusieurs répétitions :

Symbole	Écriture	Répétition
*	a*	zéro ou plus
+	a+	un ou plus
?	a?	zéro ou un

{n}	a{3}	n répétitions
{n,}	a{3,}	n ou plus répétitions
{n,m}	a{3,5}	entre n et m répétitions

La première répétition `*` permet de répéter zéro ou plusieurs fois un caractère. Par exemple, le motif `a*` correspond à une chaîne vide `""` ou une chaîne contenant un nombre quelconque de caractère `a` (par exemple `"aaa"` ou `"aaaaaaaa"`), mais aucun autre caractère.

### main.cpp

```
#include <iostream>
#include <regex>

int main()
{
    std::regex const pattern { "a*" };
    std::cout << "'a*' match with ''': " " << std::boolalpha <<
        std::regex_match("", pattern) << std::endl;

    std::cout << "'a*' match with 'a': " " << std::boolalpha
<<
        std::regex_match("a", pattern) << std::endl;

    std::cout << "'a*' match with 'aaaa': " " << std::boolalpha <<
        std::regex_match("aaaa", pattern) << std::endl;

    std::cout << "'a*' match with 'abcd': " " << std::boolalpha <<
        std::regex_match("abcd", pattern) << std::endl;
}
```

affiche :

```
'a*' match with ''': true
'a*' match with 'a': true
'a*' match with 'aaaa': true
'a*' match with 'abcd': false
```

La répétition `+` permet de répéter une ou plusieurs fois un caractère. Le motif `a+` est similaire au motif précédent, mais interdit la chaîne vide :

### main.cpp

```
#include <iostream>
#include <regex>

int main()
{
    std::regex const pattern { "a+" };
    std::cout << "'a+' match with ''': " << std::boolalpha <<
        std::regex_match("", pattern) << std::endl;

    std::cout << "'a+' match with 'a': " << std::boolalpha
<<
        std::regex_match("a", pattern) << std::endl;

    std::cout << "'a+' match with 'aaaa': " << std::boolalpha <<
        std::regex_match("aaaa", pattern) << std::endl;

    std::cout << "'a+' match with 'abcd': " << std::boolalpha <<
        std::regex_match("abcd", pattern) << std::endl;
}
```

affiche :

```
'a+' match with ''': false
'a+' match with 'a': true
'a+' match with 'aaaa': true
'a+' match with 'abcd': false
```

La répétition `?` permet de répéter zéro ou une fois un caractère. Le motif `"a?"` accepte donc uniquement la chaîne vide et la chaîne `"a"`.

### main.cpp

```
#include <iostream>
#include <regex>

int main()
{
    std::regex const pattern { "a?" };
    std::cout << "'a?' match with ''': " << std::boolalpha <<
```

```

        std::regex_match("", pattern) << std::endl;

    std::cout << "'a?' match with 'a': " << std::boolalpha
<<
        std::regex_match("a", pattern) << std::endl;

    std::cout << "'a?' match with 'aaaa': " << std::boolalpha <<
        std::regex_match("aaaa", pattern) << std::endl;
}

```

affiche :

```
'a?' match with '' : true
'a?' match with 'a' : true
'a?' match with 'aaaa' : false
```

Les répétitions avec les crochets s'appellent des répétitions bornées (*bounded repeat*). La première permet de spécifier le nombre exact de répétitions du caractère, la seconde le nombre minimal de répétitions et la dernière les nombres minimal et maximal de répétitions.

pourquoi je m'amuse à recopier le std::boolalpha à chaque cout ??

main.cpp

```
#include <iostream>
#include <regex>

int main()
{
    std::regex pattern { "a{3}" };
    std::cout << "'a{3}' match with '' : " << std::boolalpha
<<
    std::regex_match("a", pattern) << std::endl;

    std::cout << "'a{3}' match with 'aaaa': " << std::boolalpha <<
    std::regex_match("aaaa", pattern) << std::endl;

    std::cout << "'a{3}' match with 'aaaaaaaa': " << std::boolalpha <<
```

```

        std::regex_match("aaaaaaaa", pattern) << std::endl
<< std::endl;

    pattern = "a{3,}";
    std::cout << "'a{3,}' match with ''': " << std::boolalpha
<<
        std::regex_match("a", pattern) << std::endl;

    std::cout << "'a{3,}' match with 'aaaa': " << std::boolalpha <<
        std::regex_match("aaaa", pattern) << std::endl;

    std::cout << "'a{3,}' match with 'aaaaaaaa': " << std::boolalpha <<
        std::regex_match("aaaaaaaa", pattern) << std::endl
<< std::endl;

    pattern = "a{3,5}";
    std::cout << "'a{3,5}' match with ''': " << std::boolalpha <<
        std::regex_match("a", pattern) << std::endl;

    std::cout << "'a{3,5}' match with 'aaaa': " << std::boolalpha <<
        std::regex_match("aaaa", pattern) << std::endl;

    std::cout << "'a{3,5}' match with 'aaaaaaaa': " << std::boolalpha <<
        std::regex_match("aaaaaaaa", pattern) << std::endl;
}

```

affiche :

```

'a{3}' match with ''': false
'a{3}' match with 'aaaa': false
'a{3}' match with 'aaaaaaaa': false

'a{3,}' match with ''': false
'a{3,}' match with 'aaaa': true
'a{3,}' match with 'aaaaaaaa': true

'a{3,5}' match with ''': false

```

```
'a{3,5}' match with 'aaaa': true  
'a{3,5}' match with 'aaaaaaaa': false
```

## Les ancrés

Les ancrés (*anchor*) permettent d'attacher un motif au début (avec `^`) ou à la fin (avec `$`) de la séquence cible. Elles servent uniquement (principalement ?) pour les recherches. Ainsi, le motif `^a` recherche le caractère `a` au début d'une chaîne, alors que le motif `a$` le recherche à la fin.

main.cpp

```
#include <iostream>  
#include <regex>  
  
int main()  
{  
    std::regex pattern { "^a" };  
    std::cout << "'^a' match with ''": " << std::boolalpha <<  
        std::regex_search("", pattern) << std::endl;  
  
    std::cout << "'^a' match with 'abcd': " << std::boolalpha <<  
        std::regex_search("abcd", pattern) << std::endl;  
  
    std::cout << "'^a' match with 'dcba': " << std::boolalpha <<  
        std::regex_search("dcba", pattern) << std::endl <<  
    std::endl;  
  
    pattern = "a$";  
    std::cout << "'a'$ match with ''": " << std::boolalpha <<  
        std::regex_search("", pattern) << std::endl;  
  
    std::cout << "'a'$ match with 'abcd': " << std::boolalpha <<  
        std::regex_search("abcd", pattern) << std::endl;  
  
    std::cout << "'a'$ match with 'dcba': " << std::boolalpha <<
```

```
    std::regex_search("dcba", pattern) << std::endl;
}
```

affiche :

```
'^a' match with '' : false
'^a' match with 'abcd' : true
'^a' match with 'dcba' : false

'a$' match with '' : false
'a$' match with 'abcd' : false
'a$' match with 'dcba' : true
```

[Chapitre précédent](#) [Sommaire principal](#) [Chapitre suivant](#)  
Cours, C++

# Les expressions régulières 3

## Les groupes de capture

Les groupes de capture permettent de réutiliser une sous-chaîne de la séquence cible, identifiée par un motif. La sous-chaîne capturée peut être réutilisée dans le motif ou être récupérée dans le code C++. Pour créer un groupe de capture, vous devez simplement écrire le motif correspondant à ce groupe entre parenthèses. Pour que cela soit plus clair, voyons quelques exemples.

## Répéter un groupe de capture

Commençons par un exemple simple de groupe, sans utiliser la capture (donc sans réutiliser la sous-chaîne identifiée par le groupe). Le but est d'écrire un motif qui permet de valider une chaîne constituée de répétition la chaîne "ab", comme par exemple "ab" ou "ababab". En première intention, on pourrait être tenté d'écrire le motif suivant : "ab\*".

Cependant, le caractère de répétition `*` s'applique que sur le caractère `b`. Pour indiquer que l'on souhaite répéter le motif "ab", il faut donc le mettre entre parenthèses : "(ab)\*".

main.cpp

```
#include <iostream>
#include <regex>

int main()
{
    std::regex pattern { "ab*" };
    std::cout << "'ab*' match with '" << std::boolalpha
<<
```

```

        std::regex_match("", pattern) << std::endl;

    std::cout << "'ab*' match with 'ab': " << std::boolalpha
<<
        std::regex_match("ab", pattern) << std::endl;

    std::cout << "'ab*' match with 'abab': " << std::boolalpha
<<
        std::regex_match("abab", pattern) << std::endl <<
std::endl;

pattern = "(ab)*";
std::cout << "'(ab)*' match with ''": " << std::boolalpha
<<
        std::regex_match("", pattern) << std::endl;

std::cout << "'(ab)*' match with 'ab': " << std::boolalpha
<<
        std::regex_match("ab", pattern) << std::endl;

std::cout << "'(ab)*' match with 'abab': " << std::boolalpha
<<
        std::regex_match("abab", pattern) << std::endl;
}

```

affiche :

```

'ab*' match with '' : false
'ab*' match with 'ab' : true
'ab*' match with 'abab' : false

'(ab)*' match with '' : true
'(ab)*' match with 'ab' : true
'(ab)*' match with 'abab' : true

```

## Réutiliser un groupe de capture

Les chaînes identifiées par un groupe de capture peut être réutilisées dans le motif. Chaque groupe est identifié par un numéro, dans l'ordre de

leur déclaration. Pour réutiliser une chaîne, il faut indiquer le numéro de groupe précédé d'une barre oblique inversée : `\1`, `\2`, `\3`, etc.

Par exemple, pour écrire un motif qui permet d'identifier n'importe quelle chaîne commençant et terminant par le même caractère, on pourra écrire le motif suivant :

- pour récupérer le premier caractère : `(.)` ;
- pour les caractères de la chaîne autre que le premier et le dernier caractère : `.*` ;
- pour le dernier caractère, qui doit être identique au premier : `\1`.

Donc le motif final est `(.).*\1`. Pour écrire ce motif en C++, n'oubliez pas qu'il faut ajouter un caractère d'échappement devant la barre oblique inversée (le motif s'écrit alors `"(.).*\\"1"`) ou il faut écrire une littérale chaîne brute (le motif s'écrit alors `R"((.).*\1)"`).

### main.cpp

```
#include <iostream>
#include <regex>

int main()
{
    std::regex const pattern { R"((.).*\1)" };
    std::cout << "'(.).*\1' match with ''": " << std::boolalpha <<
        std::regex_match("", pattern) << std::endl;

    std::cout << "'(.).*\1' match with 'abc': " << std::boolalpha <<
        std::regex_match("abc", pattern) << std::endl;

    std::cout << "'(.).*\1' match with 'aba': " << std::boolalpha <<
        std::regex_match("aba", pattern) << std::endl;

    std::cout << "'(.).*\1' match with 'abcdefa': " << std::boolalpha <<
        std::regex_match("abcdefa", pattern) << std::endl;
}
```

affiche :

```
'(.)'*' match with '' : false  
'(.)'*' match with 'abc' : false  
'(.)'*' match with 'aba' : true  
'(.)'*' match with 'abcdefa' : true
```

De la même façon, si on veut pouvoir identifier une chaîne qui commence par deux caractères et qui se termine par ces deux caractères dans l'ordre inverse, on va pouvoir utiliser deux groupes de capture. Le motif s'écrit alors : `(.)(.)).*\2\1`.

Pour terminer avec les groupes de capture, ils vont permettre de récupérer dans le code C++ une sous-chaîne correspondant à un motif complet de la chaîne. En effet, imaginons que l'on souhaite écrire un motif qui permet de reconnaître une date au format `jj/mm/aa` (jour-mois-année, chaque élément étant écrit avec deux chiffres) et qui permet de récupérer le jour. On pourrait écrire simplement le motif `^[:digit:]{2}[:digit:]{2}[:digit:]{2}` (ou `^\d{2}\d{2}\d{2}`). Le problème est que ce motif peut correspondre à n'importe quelle chaîne commençant par deux chiffre (pour rappel, le caractère `^` est une ancre indiquant le début de la séquence cible), par exemple "12abcde" ou "123456".

Pour éviter cela, on va simplement écrire un motif qui correspond à une date, selon le format indiqué : `\d{2}/\d{2}/\d{2}`. On ajoute ensuite un groupe de capture pour les sous-chaînes que l'on souhaite récupérer dans le code C++. Par exemple, pour récupérer les jours, on écrit : `(\d{2})/\d{2}/\d{2}`. Cela permet de récupérer la date du jour, en garantissant que le format de date est respecté.

Nous verrons par la suite le code C++ utilisé pour récupérer les groupes capturés.

[Chapitre précédent](#) [Sommaire principal](#) [Chapitre suivant](#)  
Cours, C++

## Regex - suite

### Valider qu'une chaîne correspond à un motif

Pour terminer ce chapitre sur la comparaison de chaînes, voyons l'utilisation des motifs pour valider une chaîne. La fonction correspondante, `std::regex_match`, a déjà été utilisée dans les chapitres sur les expressions régulières pour présenter leur syntaxe. Pour rappel, la syntaxe de base prend simplement en arguments la séquence cible et le motif et retourne vrai si la séquence correspond au motif.

```
std::regex pattern { "abc" };
std::string target { "abcdef" };
bool result = std::regex_match(target, pattern);
```

Il est également possible d'utiliser cette fonction en passant en arguments les premier et dernier éléments de la séquence cible, comme pour les algorithmes.

```
std::regex_match(begin(target), end(target), pattern);
```

match flags ? [http://en.cppreference.com/w/cpp/regex/match\\_flag\\_type](http://en.cppreference.com/w/cpp/regex/match_flag_type)  
exemples : <https://support.google.com/a/answer/1371417?hl=fr>

main.cpp

```
{
    std::regex pattern("(ab)cd(eF)");      // Find double
word.
    std::string replacement = "le premier groupe est $1
et le second groupe est $2";
    std::string target = "abcdef";
```

```

        std::string output_str = regex_replace(target,
pattern, replacement);
        std::cout << output_str << std::endl;
    }
}

std::regex pattern(R
"((\d{2})[-/](\d{2})[-/](\d{4}))");
std::smatch match;
std::regex_search(std::string("12-03-2014"), match,
pattern);
for (size_t i = 0; i < match.size(); ++i)
{
    std::cout << i << ":" << match[i].str() << '\n';
}
}

std::cout << "Traduction" << std::endl;
{
    std::regex pattern("[a-zA-Z]+ \\\1");
    std::string replacement = "$1";
    std::string target = "The cat cat bites the dog
dog.";
    std::string output_str = regex_replace(target,
pattern, replacement);
    std::cout << output_str << std::endl;
}

std::cout << tr("bla bla $1 bla bla", 123) << std::endl;
std::cout << tr("bli bli bli bli $1", 123) << std::endl;
std::cout << std::endl;

std::cout << "Groupe" << std::endl;
match("", R"(ab)*");
match("a", R"(ab)*");
match("b", R"(ab)*");
match("ab", R"(ab)*");
match("abc", R"(ab)*");
match("ababab", R"(ab)*");
std::cout << std::endl;

```

```

match("cat", R"(c[a-zA-Z]*t)");
std::cout << std::endl;

std::cout << "Exemples de regex" << std::endl;
std::cout << "Date" << std::endl;
match("12-03-2014", R"(\d{2}[-/]\d{2}[-/]\d{4})");
std::cout << "Time" << std::endl;
match("15:17", R"(\d{2}:\d{2})");
std::cout << std::endl;

// vérifier qu'un identifiant C++ est valide
// [a-zA-Z_][a-zA-Z_0-9]*
// fichier windows
// [a-zA-Z_][a-zA-Z_0-9]*\.[a-zA-Z0-9]+

std::string regex_str = "[a-zA-Z_][a-zA-Z_0-9]*\\.[a-zA-Z0-9]+";
std::regex reg1(regex_str, std::regex_constants::icase);
std::string str = "File names are readme.txt and
my.cmd.";
std::sregex_iterator it(str.begin(), str.end(), reg1);
std::sregex_iterator it_end;
while(it != it_end) {
    std::cout << it->str() << std::endl;
    ++it;
}
}

```

### Groups

le premier groupe est ab et le second groupe est ef

0: 12-03-2014

1: 12

2: 03

3: 2014

### Traduction

The cat bites the dog.

bla bla 123 bla bla

bli bli bli bli 123

### Groupe

"(ab)\*" match with "" = true

"(ab)\*" match with "a" = false

```
"(ab)*" match with "b" = false
"(ab)*" match with "ab" = true
"(ab)*" match with "abc" = false
"(ab)*" match with "ababab" = true

"c[a-z]*t" match with "cat" = true
```

Exemples de regex

Date

```
"\d{2}[-/]\d{2}[-/]\d{4}" match with "12-03-2014" = true
```

Time

```
"\d{2}:\d{2}" match with "15:17" = true
```

readme.txt

my.cmd

## Manque : vorace

A déplacer dans le chapitre sur les recherches ?

main.cpp

```
// non-greedy (non vorace)
search("aaaaa", R"(a{3})");      // -> aaaaa = plus
longue correspondance
search("aaaaa", R"(a{3}?)");      // -> aaa = plus courte
correspondance
std::cout << std::endl;
}
```

affiche :

```
search "a{3}" in "aaaaa" = true
search "a{3}?" in "aaaaa" = true
```

[Chapitre précédent](#) [Sommaire principal](#) [Chapitre suivant](#)

Cours, C++

# Analyse lexicale et syntaxique

Prendre une grammaire et vérifier avec des regex qu'un texte respecte cette syntaxe.

Analyser un texte et extraire les informations syntaxiques

AST : [http://en.wikipedia.org/wiki/Abstract\\_syntax\\_tree](http://en.wikipedia.org/wiki/Abstract_syntax_tree)

BNF : [http://fr.wikipedia.org/wiki/Forme\\_de\\_Backus-Naur](http://fr.wikipedia.org/wiki/Forme_de_Backus-Naur)

[http://en.wikipedia.org/wiki/Lexical\\_analysis](http://en.wikipedia.org/wiki/Lexical_analysis) et  
<http://en.wikipedia.org/wiki/Parsing>

# Évaluation d'expressions

Exemple : "123 + 456" retourne "579"

Avec ou sans analyse syntaxique

Analyse directe : lire caractère par caractère

Analyse avec regex

Analyse avec AST

# Créer un modèle de document

Exemple : "hello {{name}}" + "toi" → "hello toi"

Avec ou sans analyse syntaxique

Analyse directe : lire caractère par caractère

Analyse avec regex

Analyse avec AST

# La ligne de commande

Lorsque l'on lance une application il est possible de lui envoyer des informations :

```
myapp.exe myfile.txt
```

Comment lire ces informations ?

## Utilisation de argc et argv

La fonction `main` est obligatoire et doit respecter une signature imposée par la norme C++. Pour le moment, on utilise :

```
int main() {  
}
```

Elle correspond à une fonction qui ne reçoit aucun paramètre et retourne un entier. Souvent, on ne mets pas l'entier (0 par défaut) ou on retourne un code d'erreur :

```
int main() {  
    return 123456;  
}
```

Il existe une autre signature, avec des paramètres :

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

`argc` contient le nombre de paramètres passé lors de l'appel de la fonction. Chaque paramètre est séparé par une espace et le premier paramètre est le nom du programme. Par exemple :

```
main.cpp
```

```
#include <iostream>

int main(int argc, char* argv[]) {
    std::cout << argc << std::endl;
}
```

affiche, selon la ligne de commande :

1

pour `./a.out`,

2

pour `./a.out int.txt`,

3

pour `./a.out int.txt out.txt`.

pas encore vu les tableaux (mais vu les string et accès aux caractères ?)

`argv` permet d'accéder aux chaînes de caractères correspondant à chaque paramètre. On reconnaît dans la déclaration le type `char*` correspondant aux littérales chaînes de caractères. Les crochets droits `[]` permettent de déclarer un tableau. Le type de `argv` peut donc se lire comme étant un tableau de chaînes de caractères.

Pour accéder aux éléments d'un tableau, on utilise également les crochets droits, en indiquant la position dans le tableau (en commençant à 0).

`main.cpp`

```
#include <iostream>

int main(int argc, char* argv[]) {
    std::cout << argc << std::endl;
    std::cout << argv[0] << std::endl;
    std::cout << argv[1] << std::endl;
    std::cout << argv[2] << std::endl;
```

}

affiche si on l'appelle avec `./a.out in.txt out.txt` :

```
3  
./a.out  
in.txt  
out.txt
```

Si on essaie d'afficher une valeur en dehors du tableau, le comportement est indéterminé.

L'utilisation des tableaux sera détaillée par la suite, en particulier comment les parcourir automatiquement avec une boucle.

**[Chapitre précédent](#)** **[Sommaire principal](#)** **[Chapitre suivant](#)**

[Cours, C++](#)

# Entrée console

On connaît déjà cout, qui signifie “output C console”, permet d'envoyer des messages sur la console. Egalement possible d'entrée des valeurs depuis la console avec `cin`

## entrée standard

Permet d'entrer des informations. Lecture du texte en entrée, conversion selon le type de variable :

```
int i {};  
cin >> i;  
cout << i*2 << endl;
```

## Entrer une ligne de texte

getline j'ai écrit sur le bac à sable, mais je préfère une approbation d'un programmeur plus expérimenté d3m0t3p

## Gestion des erreurs

filtrer le texte entré, vérifier qu'il est conforme

pas encore vu les boucles...

# Les fichiers

## Représentation binaire et texte

Permettre la lecture et l'écriture de fichiers. 2 types de fichiers : binaire et texte. Déjà vu les différentes représentation d'un nombre, en binaire :

main.cpp

```
#include <iostream>
#include <bitset>

int main() {
    int const i { 123456 };
    std::cout << "représentation numérique: " << i << std::endl;
    std::cout << "représentation binaire: " << std::bitset<
sizeof(i)*8>(i) << std::endl;
    std::cout << sizeof(123) << std::endl;
    std::cout << sizeof("123") << std::endl;
    std::cout << sizeof(123456) << std::endl;
    std::cout << sizeof("123456") << std::endl;
    std::cout << sizeof(123456789) << std::endl;
    std::cout << sizeof("123456789") << std::endl;
}
```

affiche :

```
représentation numérique: 123456
représentation binaire: 000000000000000011110001001000000
4
4
4
7
4
10
```

En mode texte, chaque chiffre prend 1 octet (+ un octet de fin de chaîne), en binaire, toujours la même taille pour un int. Donc fichier texte permet d'être lu avec un simple éditeur, mais plus gros en taille.

## Création et ouverture de fichier

Utilisation de stream, comme pour `cout` et `cin`, avec `<<` pour écrire et `>>` pour lire.

Utiliser `fstream` pour lecture et/ou écriture ("file stream") ou classes plus spécialisées `ofstream` ("output file stream") et `ifstream` ("input file stream").

On peut simplement ouvrir un fichier en créant un `ifstream` :

```
#include <fstream>

int main() {
    std::ifstream text_file("in.txt"); // mode texte
    std::ifstream binary_file("in.bin", std::ios::binary);
// mode binaire
}
```

Le fichier est automatiquement enregistré et fermée lorsque les variables `text_file` et `binary_file` sont détruite (donc à la fin du bloc). En utilisant la portée des variables, on peut donc décider quand les fichiers sont ouvert et fermé :

```
#include <fstream>

int main() {
    cout << "pas encore ouvert" << endl;
{
    cout << "ouverture du fichier" << endl;
    std::ifstream file("in.txt");
} // fermeture du fichier
cout << "le fichier est fermé" << endl;
}
```

De la même façon, pour ouvrir un fichier en écriture :

```
#include <fstream>

int main() {
    std::ofstream file("out.txt");
}
```

Si le fichier n'existe pas, il est créé.

Le premier paramètre est le nom du fichier. Attention, le chemin par défaut est celui de l'application, pas celui du fichier .cpp (s'ils ne sont pas au même endroit). Possibilité de mettre chemin relatif ou absolu :

```
std::ofstream file("out.txt");
std::ofstream file("../..\\out.txt");
std::ofstream file("C:/myapp/out.txt");
```

Remarque : normalement, sous Windows, le séparateur de chemin est la barre oblique inversée \. Vous pouvez l'utiliser, mais n'oublier pas que ce caractère correspond au caractère d'échappement en C++, il faut donc écrire \\ ou utiliser une chaîne brute :

```
std::ofstream file("C:\\myapp\\\\out.txt"); // double barre
oblique inversée
std::ofstream file(R"(C:\\myapp\\out.txt)"); // chaîne brute
```

Le second paramètre permet de spécifier les options d'ouverture :

- `std::ios::binary` : ouverture en mode binaire ;
- `std::ios::in` : en lecture (pour fstream) ;
- `std::ios::out` : en écriture (pour fstream) ;
- `std::ios::app` (`append = "ajouter"`) : ajoute à la fin du fichier ;
- `std::ios::trunc` (`truncate = "tronquer"`) : supprimer le contenu ;
- `std::ios::ate` (`at the end = "à la fin"`) : ajoute à la fin lors de l'ouverture.

## **Ecriture et lecture**

En utilisant les opérateurs de flux `<<` pour l'écriture et `>>` pour la lecture.

Par exemple, pour écrire une liste de valeur, séparé par un espace :

```
file << i << ' ' << j << ' ' << k << endl;
```

Remarque : avec mode texte, les chiffres sont écrit les uns à la suite des autres. Si on écrit :

```
file << 1;  
file << 2;  
file << 3;
```

cela produit dans le fichier “123”. Si on essaie de lire, on lira qu'un seul nombre (123). Donc penser à ajouter des séparateurs.

Pour la lecture :

```
file >> i;
```

Problème de séparation des caractères. De lecture d'une ligne

lecture complet dans un string ?

Caractère spéciaux : tabulation \t retour à la ligne \n

Problème lecture windows/linux : \n et \r\n

ouverture et fermeture manuelle ? (open, is\_open, close)

Se positionner dans un fichier ? (eof, fpos, etc.)

Gestion des erreurs, exceptions

## **Pratiquer**

- fichier csv (comma separated values) : tableau avec séparation

par ,

- fichier texte excel : tabulation + retour à la ligne
- génération xml
- génération json
- lecture et écriture image ?

<a href="#">Chapitre précédent</a>	<a href="#">Sommaire principal</a>	<a href="#">Chapitre suivant</a>
------------------------------------	------------------------------------	----------------------------------

[Cours, C++](#)

# Lire et écrire des feuilles de calculs

Lisible avec Excel ou OpenOffice/LibreOffice Calc.

Format texte simple, avec extension spécifique. Chaque cellule est séparée par un caractère séparateur, retour à la ligne pour une nouvelle ligne.

- `.csv` : [http://en.wikipedia.org/wiki/Comma-separated\\_values](http://en.wikipedia.org/wiki/Comma-separated_values)
- `.xls` : séparation par tabulation

# Lire et enregistrer des images

Travaux pratiques. Lire la doc et implémenter la lecture de formats d'images simples.

- `.ppm` : format ASCII ou binaire.

[http://en.wikipedia.org/wiki/Netpbm\\_format](http://en.wikipedia.org/wiki/Netpbm_format)

- `.bmp` : [http://en.wikipedia.org/wiki/BMP\\_file\\_format](http://en.wikipedia.org/wiki/BMP_file_format)

- `.svg` : image vectorielle.

[http://en.wikipedia.org/wiki/Scalable\\_Vector\\_Graphics](http://en.wikipedia.org/wiki/Scalable_Vector_Graphics)

Utilisation de libs :

- `.png` : png++ ? <http://www.nongnu.org/pngpp/> (ou libpng ?)

- `.jpeg` : libjpg

- autre : CImg (<http://cimg.sourceforge.net/>),

[https://www.opengl.org/wiki/Image\\_Libraries](https://www.opengl.org/wiki/Image_Libraries), ImageMagik (

<http://www.imagemagick.org/script/index.php>), GLI (

<http://www.g-truc.net/project-0024.html#menu>)

# Créer des fonctions

## Syntaxe de base

Déjà utilisé des fonctions libres. Le prototype, c'est les algo de la STL. Le but est de mettre un bloc d'instruction que l'on va pouvoir appeler plusieurs fois

Par exemple, si on veut écrire un message deux fois :

```
std::cout << "hello, world!" << std::endl;
std::cout << "hello, world!" << std::endl;
```

On peut mettre cette ligne de code dans une fonction et appeler la fonction 2 fois.

On a déjà vu qu'une fonction était définie par :

- des informations en entrée et sortie
- un nom
- une bloc d'instructions

La syntaxe de base est similaire à la fonction main. Dans la version la plus simple, pas d'infos en entrée et sortie :

```
void nom_fonction() {
    instructions
}
```

Pour appeler cette fonction, on fait comme déjà vu :

```
nom_fonction();
```

## Portée des variables

pas ici, mettre dans un chapitre avec les bloc d'instruction

Variable sont locale à un bloc. Dès que l'on sort du bloc, la variable est détruite. Mais variable accessibles dans bloc inclus :

```
int main() {
    int i {};
    {
        std::cout << i << std::endl; // ok
    }
    std::cout << i << std::endl; // ok
}
```

Au contraire, une variable définie dans un bloc n'est pas accessible en dehors :

```
int main() {
{
    int i {};
    std::cout << i << std::endl; // ok
}
std::cout << i << std::endl; // erreur, i n'existe plus
}
```

variable globale, en dehors des fonctions : fu..

Idem entre bloc :

```
int main() {
{
    int i {};
}
{
    std::cout << i << std::endl; // erreur, i n'existe
pas
}
}
```

Dans ce code, même si les 2 blocs sont des sous-bloc du bloc de la

fonction main, ce sont deux bloc différents

Plus généralement :

```
{  
    int i {};  
    ... // i existe ici  
}      // i est détruit ici
```

## Arguments de fonctions

Même situation avec fonction :

```
void f() {  
    std::cout << i << std::endl; // erreur, i n'existe pas  
dans ce bloc  
}  
  
int main() {  
    int i {};  
    f(); // i existe lors que l'on appelle f()  
}
```

Dans ce code, même si la variable i existe lorsque l'on appelle la fonction f, le bloc d'instructions de f n'est pas sous-bloc de main, donc pas accessible.

Pour envoyer des informations dans la fonction, on a déjà vu avec les lambda : utilisation de paramètres de fonction. Déclarés entre les parenthèses :

```
void f(paramètres de fonction) {  
}  
  
int main() {  
    f(arguments de fonction);  
}
```

Paramètres = liste de 0, 1 ou plusieurs (type + nom). Par exemple, pour

envoyer 1 valeur entière :

```
void f(int i) { // création de i
} // destruction de i
```

Les paramètres ont même portée que variables locales, depuis le début du bloc jusqu'à la fin

Pour appeler la fonction, donne une liste d'argument, qui peut être une littérale, une variable ou une fonction qui retourne un valeur de même type. Liste des arguments doit correspondre à la liste des paramètres :

```
void f() {
    std::cout << "f()" << std::endl;
}

void g(int i) {
    std::cout << "g() avec i=" << i << std::endl;
}

int main() {
    f(); // ok
    f(123); // erreur, trop d'argument

    g(); // erreur, pas assez d'argument
    g(123); // ok
}
```

Exemple de message d'erreur pour f (clang)

```
main.cpp:13:5: error: no matching function for call to 'f'
    f(123);
    ^
main.cpp:3:6: note: candidate function not viable: requires
0 arguments, but 1 was provided
void f() {
    ^
```

Le compilateur indique qu'il ne trouve pas de fonction f correspondant à l'appel f(123) ("no matching function"). Il indique à la ligne suivante qu'il connaît une fonction f ("candidate function"), mais qui prend 0

arguments (“requires 0 arguments”) alors que l’appel donne 1 argument (“but 1 was provided”)

Pour l’appel de g, le message :

```
main.cpp:15:5: error: no matching function for call to 'g'  
    g();  
    ^  
main.cpp:7:6: note: candidate function not viable: requires  
single  
argument 'i', but no arguments were provided  
void g(int i) {  
    ^
```

De la même manière, le compilateur indique qu'il ne trouve pas de fonction correspondant à g(), mais qu'il trouve une fonction g qui prend 1 argument.

On peut utiliser n'importe quel type copiable :

```
void f(int i, double d, string s, vector<int> v) {  
}
```

Au besoin, le compilateur réalise une conversion pour adapter les arguments. Par exemple si on écrit une fonction qui prend un long int et qu'on passe un int :

```
void f(long int i) {  
}  
  
int main() {  
    f(123); // 123 est une littérale de type int  
             // peut être converti implicitement en long int  
}
```

## Surcharge de fonction

(ou *overloading* ou polymorphisme ad-hoc)

Polymorphisme : plusieurs fonctions de même nom. Des fonctions peuvent avoir le même nom, tant que les paramètres sont différents :

```
void f(int i) {
    std::cout << "f(int) avec i=" << i << std::endl;
}

void f(string s) {
    std::cout << "f(string) avec s=" << s << std::endl;
}
```

Le compilateur choisit la fonction correspondante, selon le type que l'on donne en argument :

```
void f(int i) {
    std::cout << "f(int) avec i=" << i << std::endl;
}

void f(long int i) {
    std::cout << "f(long int) avec i=" << i << std::endl;
}

int main() {
    f(1); // 1 est une littérale de type int
    f(2L); // 2L est une littérale de type long int

    int i { 1 };
    f(i);

    long int l { 2 };
    f(l);
}
```

affiche :

```
f(int) avec i=1
f(long int) avec i=2
f(int) avec i=1
f(long int) avec i=2
```

Le compilateur commence par rechercher s'il connaît une fonction avec

le nom correspondant. Par exemple pour f(1), il trouve 2 fonctions : f(int) et f(long int). Ensuite il regarde si l'un des types en paramètre correspondant au type en argument. Ici, c'est le cas, il appelle donc f(int).

Si on écrit :

```
#include <iostream>

void f(long int i) {
    std::cout << "f(long int) avec i=" << i << std::endl;
}

int main() {
    f(1); // 1 est une littérale de type int
}
```

le compilateur trouve la fonction f, mais le paramètre ne correspond pas. Il regarde s'il peut faire une conversion. Ici, oui, on peut convertir implicitement un int en long int. Il convertie donc 1 en 1L et appelle f(long int).

S'il en trouve pas de conversion possible, il lance un message d'erreur. Par exemple, si on appelle f("du texte"), le compilateur donne :

```
main.cpp:19:5: error: no matching function for call to 'f'
    f("une chaine");
    ^
main.cpp:3:6: note: candidate function not viable: no known
conversion
from 'const char [11]' to 'int' for 1st argument
void f(int i)
    ^
main.cpp:7:6: note: candidate function not viable: no known
conversion
from 'const char [11]' to 'long' for 1st argument
void f(long int i)
    ^
1 error generated.
```

Ce qui signifie qu'il ne trouve aucune fonction correspond à l'appel de f("une chaine"), mais qu'il a 2 candidat (2 fonction qui ont le même nom)

mais sans conversion possible (“no known conversion”).

Au contraire, dans certain cas, il aura plusieurs possibles, soit parce que vous déclarez par erreur 2 fonctions avec les mêmes paramètres, soit parce que le compilateur peut faire 2 conversions pour 2 types. Dans le premier cas :

```
void f() {
    std::cout << "première fonction f" << std::endl;
}

void f() {
    std::cout << "seconde fonction f" << std::endl;
}
```

produit le message :

```
main.cpp:7:6: error: redefinition of 'f'
void f(int i) {
^
main.cpp:3:6: note: previous definition is here
void f(int i) {
^
```

Quand le compilateur arrive à la ligne 7 et rencontre la seconde fonction f (qu'il connaît déjà), il prévient qu'il connaît déjà (“redefinition of 'f'”) et que la première version (“previous definition is here”) se trouve à la ligne 3.

L'autre cas est si plusieurs fonctions peuvent correspondre, l'appel est ambigu. Par exemple :

```
#include <iostream>

void f(int i) {
    std::cout << "f(int) avec i=" << i << std::endl;
}

void f(long int i) {
    std::cout << "f(long int) avec i=" << i << std::endl;
}
```

```
int main() {
    f(1u); // 1 est une littérale de type unsigned int
}
```

affiche le message d'erreur :

```
main.cpp:12:5: error: call to 'f' is ambiguous
    f(1u); // 1 est une littérale de type int
          ^
main.cpp:3:6: note: candidate function
void f(int i) {
          ^
main.cpp:7:6: note: candidate function
void f(long int i) {
          ^
```

Il existe une conversion de unsigned int vers int et vers long int. Il n'y a pas de priorité dans les conversions, le compilateur ne sait pas quelle conversion choisir et donc quelle fonction appeler. L'appel est ambigu ("call to 'f' is ambiguous"), il trouve deux fonctions candidate ("candidate function").

La méthode qui permet au compilateur de trouver la fonction correspondant à une appel s'appelle la résolution des noms (name lookup)

## Note sur bool

Comme cela a déjà été expliqué, certains types, dont les littérales chaînes (et plus généralement les pointeurs), sont convertissables automatiquement en booléen. Si on écrit la surcharge suivante :

```
void foo(bool) { std::cout << "f(bool)" << std::endl; }
void foo(string const&) { std::cout << "f(string)" << std::endl; }

foo("abc");
```

Ce code ne va pas afficher `f(string)`, mais `f(bool)`. Si on ajoute une fonction `f(const char*)` et sera appelée en premier. La raison est que

la littérale chaîne est de type `const char*`, les fonctions seront appelée dans l'ordre suivant :

- `f(const char*)` : par de conversion entre l'argument et le paramètre ;
- `f(bool)` : conversion automatique ;
- `f(string)` : conversion passant par une classe.

Donc attention lorsque vous écrivez une fonction qui prend `bool`, elle peut prendre aussi n'importe quel pointeur.

Solution C++14 : écrire “abc”s pour créer une littérale de type `string`.

Détailler le name lookup

## Valeur par défaut

On peut souhaiter pouvoir appeler une fonction avec et sans un argument. Par exemple `f` qui prend un entier ou 0 si on en donne aucune valeur

Première solution, surcharger la fonction :

```
void f() {
    std::cout << "f()" << std::endl;
}

void f(int i) {
    std::cout << "f(int) avec i=" << i << std::endl;
}

int main() {
    f();      // ok, appel de f()
    f(123); // ok, appel de f(int i)
}
```

Le compilateur trouve à chaque fois deux fonctions avec le même nom, mais pas d'ambiguïté pour savoir laquelle appeler.

Possibilité de simplifier en donnant une valeur par défaut à un paramètre :

```
void f(int i = 0) {
    std::cout << "f(int) avec i=" << i << std::endl;
}
```

Dans ce cas, on indique que `f` peut prendre un entier. Si on ne donne pas de valeur, le compilateur peut utiliser la valeur par défaut :

```
int main() {
    f();      // ok, appel de f(int i) avec i = 0
    f(123); // ok, appel de f(int i) avec i = 123
}
```

Bien sûr, il ne faut pas laisser les 2 fonctions, pour éviter les ambiguïté :

```
void f() {
    std::cout << "f()" << std::endl;
}

void f(int i = 0) {
    std::cout << "f(int) avec i=" << i << std::endl;
}

int main() {
    f();      // erreur, appel de f() ou de f(int i) avec i =
0 ?
}
```

## Retour de fonction

Idem dans l'autre sens :

```
void f() {
    int i {};
}

int main() {
    f(); // i existe dans f()
```

```
    std::cout << i << std::endl; // erreur, i n'existe pas  
dans ce bloc  
}
```

Une variable déclarée localement dans une fonction ne sera pas accessible dans le code qui appelle cette fonction. Utilisation de retour de fonction, permet de retourner 1 seule valeur. Utilisation du mot-clé return pour indiquer la valeur que la fonction doit retourner et remplacer void par le type de la valeur que l'on veut retourner.

```
int f() {  
    int const i { 123 };  
    return i;  
}  
  
int main() {  
    int const j = f();  
    std::cout << j << std::endl;  
}
```

Lorsque l'on appelle f, la variable i dans f est créée et initialisée avec la littérale 123. après le return, la valeur de i est retournée au code appelant, la variable j est créée et initialisée en copiant la valeur retournée par la fonction f (elle copie i) tandis que la variable i est détruite.

retourner directement une valeur :

```
int f() {  
    return 123;  
}  
  
int main() {  
    int const j = f();  
    std::cout << j << std::endl;  
}
```

Portée de variable fait que l'on peut utiliser 2 variables de même noms si portée différentes. Par exemple :

```
int f() {
    int const i { 123 };
    return i;
}

int main() {
    int const i = f();
    std::cout << i << std::endl;
}
```

Il faut bien comprendre ici que même si les 2 variables dans la fonction et dans main s'appellent toutes les 2 "i", ce sont 2 variables différentes.

Mot clé return retourne immédiatement de la fonction. Si on écrit :

```
int f() {
    int const i { 123 };
    return i;
    std::cout << "on est après le return" << std::endl; // n'est jamais exécuté
}

int main() {
    int const i = f();
    std::cout << i << std::endl;
}
```

le cout après le return n'est pas exécuté.

## Exos

<http://cpp.developpez.com/tutoriels/Explicit-C++/ecrire-algorithme-standard/>

<a href="#">Chapitre précédent</a>	<a href="#">Sommaire principal</a>	<a href="#">Chapitre suivant</a>
Cours, C++		

# Les conditions

## Créer ses algorithmes

Dans de nombreuses situations, vous pourrez utiliser des combinaisons d'un ou plusieurs algorithmes avec des prédictats spécifiques pour résoudre un grand nombre de problématique. Mais malgré la diversité et la généréricité des algorithmes de la bibliothèque standard, ceux-ci ne peuvent pas non plus répondre à tous les besoins possibles et imaginables. Il vous faudra alors écrire vos propres algorithmes.

Jusqu'à maintenant, on a utilisé le terme "algorithme" pour désigner les fonctions fournies dans le fichier d'en-tête `<algorithm>` de la bibliothèque standard. Mais plus généralement, un algorithme est une suite d'instruction permettant de réaliser une tâche précise. C'est donc un concept plus générique que les fonctions que vous avez vu et il n'est pas nécessaire qu'un algorithme s'applique que sur des collections en passant par des itérateurs. Cependant, le respect de cette approche facilitera la réutilisation du code, puisque les nouveaux algorithmes que vous créerez seront utilisable avec les collections existantes.

### création multi fichier. Création de modules et libs

De la même façon, pour améliorer la réutilisabilité du code, il est préférable, quand c'est possible, d'implémenter les algorithmes pour qu'ils réalisent une seule tâche, le mieux possible. Par exemple, ne pas créer une algorithme `recherche_replacer`, mais écrire deux algorithmes (sauf raison de performances ou implémentation spécifique)

### Notion de SRP ? SOLID ?

Dans les chapitres précédents, les codes que vous avez écrit étaient des suites séquentielles d'instructions, séparées par des points-virgules. Cela signifie que les instructions sont exécutées dans l'ordre où elles

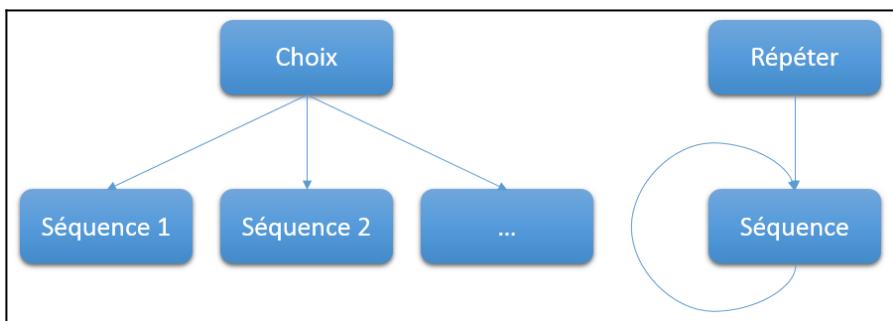
apparaissent dans le code et qu'une instruction est exécutée uniquement lorsque l'instruction précédente est terminée.

Pour des raisons de lisibilités, il est d'usage d'écrire une seule instruction par ligne. Mais il est possible d'écrire plusieurs instructions par ligne, comme par exemple :

```
int i { 1 }; int j { 2 }; int k { 3 };
```

Il existe deux méthodes pour écrire des instructions qui ne s'exécutent pas de façon séquentielle :

- les conditions (qui sera vu dans ce chapitre), qui permettent de choisir entre plusieurs séquences d'instructions.
- les boucles (dans le chapitre suivant), qui permettent de répéter une séquence d'instructions.



## Les conditions

Les conditions (*selection statement*) permettent de prendre des décisions, selon une valeur donnée. Le type de valeur permet de distinguer deux types de conditions.

- les tests `if-else` prennent une valeur booléenne et permettent de choisir entre deux chemins (`true` ou `false`) ;

- les tests `switch` prennent une valeur entière et permettent de choisir entre plusieurs valeurs possibles.

## Les tests if-else

La clause `else` étant optionnelle, il est possible d'écrire les tests `if-else` selon deux syntaxe :

```
if (valeur booléenne) { bloc d'instructions si vrai } else {  
    bloc d'instructions si fausse  
if (valeur booléenne) { bloc d'instructions si vrai }
```

Dans ce code, si la valeur booléenne est vraie, seul le premier bloc d'instructions est exécuté. Si la valeur est fausse, seul le second bloc d'instructions est exécuté dans la première syntaxe et rien n'est exécuté dans la seconde syntaxe.

```
#include <iostream>  
  
int main() {  
    int i { 123 };  
    std::cout << "On va faire un test..." << std::endl;  
// ligne 1  
    if (i < 5) {  
// ligne 2  
        std::cout << "i est inférieur à 5" << std::endl;  
    } else {  
        std::cout << "i est supérieur ou égal à 5" << std::endl;  
    } // ligne 3  
    std::cout << "On a fait un test !" << std::endl;  
// ligne 4  
}
```

affiche

```
On va faire un test...  
i est supérieur ou égal à 5  
On a fait un test !
```

Ce code suit donc la séquence d'instructions suivantes :

- la ligne 1 affiche un premier message ;
- la ligne 2 teste la valeur booléenne de l'expression. Le résultat est `false`, le premier bloc d'instructions n'est pas exécuté ;
- le second bloc de code (après le mot-clé `else`) est exécuté ;
- le programme a terminé d'exécuté le test `if-else`, l'exécution reprend son cours normal et passe à la ligne suivant le test (ligne 4) et affiche le dernier message.

La valeur booléenne peut être une variable de type booléen, le résultat d'une expression avec un opérateur de comparaison (`==`, `<`, `>`, etc.), une fonction qui retourne un booléen ou une combinaison des trois.

```
if (test) ...           // variable booléenne
if (i == j) ...         // i est égal à j
if (v.empty()) ...      // si le vector est vide
if (!v.empty()) ...     // si le vector n'est pas vide
```

La dernière ligne signifie que l'on prend le résultat de fonction `empty` (un booléen) et on applique dessus l'opérateur “inverse” `!`.

utiliser les opérateurs booléen vu avant ! && ||

non évaluation des expressions après && et ||

plusieurs if

```
if () {
    if () {
        if () {
        }
    }
}
```

## switch

en fonction de valeur prise par un entier

```
switch (value) {  
    case 1: ...  
        break;  
    case 2: ...  
        break;  
    default: ...  
}
```

équivalent à plusieurs if

```
if (value == 1) {  
    ...  
} else if (value == 2) {  
    ...  
} else {  
    ...  
}
```

## L'opérateur ternaire

ternaire = qui prend 3 arguments (&& et || binaire, ! unary). Peut être utilisé dans une expression, par exemple pour initialiser une valeur

yntaxe :

```
booléen ? instruction 1 : instruction 2;
```

1 et 2 doivent retourner le même type

Par exemple :

```
int i { 123 };  
int j = (i < 6) ? 0 : 10;
```

(parenthèse pour la lisibilité). Se lit : si  $i < 6$ , retourne 0, sinon retourne 10.

[Chapitre précédent](#) [Sommaire principal](#) [Chapitre suivant](#)

[Cours, C++](#)

# Les boucles

seconde partie des structures de contrôle : les boucles. Permet de répéter un bloc d'instruction.

Similarité avec algos (transform, foreach, etc) dans le sens où cela permet de parcourir un tableau

## **while et do while**

Permet de répéter un bloc d'instructions tant qu'une condition est vérifiée. Syntaxe :

```
while (condition) {  
    ...  
}  
  
do {  
    ...  
} while (condition);
```

Dans les 2 cas, exécute le bloc tant que la condition est vraie. Différence entre les 2 : avec do while, bloc exécuté au moins 1 fois avant de tester la condition ; avec while, commence par tester

## **for**

3 éléments, tous optionnels :

- initialisation
- test de continuation
- incrémentation

yntaxe :

```
for (initialisation; test de continuation; boucle) {  
}
```

Par exemple, pour compter de 1 à 10, on utilise un compteur (variable entière) :

- initialisation à 1
- s'arrête lorsque == à 10
- incrémenté à chaque boucle

devient :

```
for (int i { 1 }; i <= 10; ++i) {  
    std::cout << i << std::endl;  
}
```

### Exercice :

1. Ecrire le code permettant de réaliser le test présenté à la fin du chapitre [Les nombres à virgule fixe](#). Pour dessiner le graphique, vous pouvez simplement afficher la valeur du compteur dans la console avec `std::cout`, puis copier le contenu dans un tableur (par exemple LibreOffice Calc).

2. L'exercice précédent présente un problème : afficher toutes les valeurs prend du temps à afficher et à manipuler dans le tableur. De plus, ce n'est pas nécessaire d'afficher toutes les valeurs, il n'est pas possible de toutes les représenter dans un même graphique.

Utiliser un test `if` et l'opérateur modulo `%` pour afficher une valeur pour 100 valeurs du compteur.

3. Idem, mais en utilisant deux boucles `for` au lieu de `if`.

[Chapitre précédent](#) [Sommaire principal](#) [Chapitre suivant](#)

[Cours, C++](#)

# Fonctions génériques

## Problématiques

On souhaite écrire une fonction qui fait une addition. On peut écrire par exemple :

```
int add(int lhs, int rhs) {
    return lhs + rhs;
}
```

Ce code fonctionne et donne le résultat attendu :

```
int main() {
    cout << add(3, 4) << endl;
}
```

affiche :

7

Maintenant, si on écrit :

```
int main() {
    cout << add(1.2, 3.4) << endl;
}
```

Ce code affiche :

4

au lieu de “4.6”. La raison est qu'il n'existe pas de fonction add qui prend en arguments des types réels. Le compilateur ne trouve que la fonction `add` pour des entiers. Il regarde donc s'il peut faire une conversion, ce qui est le cas, et donc le fait. Code équivalent à :

```
int lhs = 1.2; // converti en 1
int rhs = 3.4; // converti en 3
add(lhs, rhs); // calcul 1 + 3
```

Une première solution est d'utiliser une surcharge et d'écrire deux fonctions :

```
int add(int lhs, int rhs) {
    return lhs + rhs;
}

double add(double lhs, double rhs) {
    return lhs + rhs;
}
```

Dans ce cas, le compilateur n'a pas besoin de faire de conversion. Il trouve les 2 fonctions, la première a besoin d'une conversion, l'autre non. Il choisit donc celle qui ne nécessite pas de conversion.

On comprend vite le problème de cette approche. Si on veut utiliser cette fonction avec 10 types différents, il faudra écrire 10 fonctions surchargées différentes.

Si on met ce code dans une bibliothèque et que l'on veut l'utiliser sur un 11<sup>e</sup> type que l'on avait pas prévu, il faudra modifier le code existant, c'est à dire modifier la bibliothèque que l'on n'a pas écrite.

Ce code n'est donc pas générique, il est compliqué de le faire évoluer.

Si on regarde les algorithmes de la bibliothèque standard, il voit qu'ils sont conçus pour pouvoir être utilisés sur n'importe quel type de conteneur. Il est donc possible d'écrire un code générique, qui s'adapte au type que l'on utilise.

## L'inférence de type

En fait, vous avez déjà vu une solution possible dans les chapitres précédents. Lors que l'on a vu les fonctions lambdas, vous avez utilisé l'inférence de type et le mot clé `auto` pour laisser le compilateur choisir

les types des paramètres de fonction.

Cette approche est également utilisable avec les fonctions : uniquement gcc 4.9 pour le moment, cf TS concept

```
#include <iostream>

auto add(auto lhs, auto rhs) {
    return lhs + rhs;
}

int main() {
    std::cout << add(1, 2) << std::endl;
    std::cout << add(1.2, 3.4) << std::endl;
}
```

Dans le premier cas, le compilateur déduit que l'on souhaite utiliser des entiers et détermine que `auto` est équivalent à `int`. Dans le second cas, il utilise `double`.

De la même manière, il détermine que le type de retour de la fonction est de même type que l'expression `lhs + rhs`. Comme ces deux variables sont de même type dans le code d'exemple, le compilateur détermine sans problème que le résultat doit être de même type.

Que se passe-t-il lorsque les deux types ne sont pas identiques ? Par exemple, si on force l'un des deux paramètres :

```
#include <iostream>

auto add(int lhs, auto rhs) {
    return lhs + rhs;
}

int main() {
    std::cout << add(1, 2) << std::endl;
    std::cout << add(2, 3.4) << std::endl;
}
```

affiche :

Le compilateur réussit à déterminer le type correct de l'expression. Lorsque l'on utilise deux entiers, le résultat est un entier et lorsque l'on utilise un entier et un réel, le résultat est un réel.

std::common\_type ??

## Les fonctions template

`auto` n'est pas pris en charge par tous les compilateurs. Possibilité d'expliciter le type générique en utilisant des fonctions template.

Une fonction classique permet de passer des données en paramètre. Les fonctions template vont plus loin, elles permettent de passer des types comme paramètre. C'est à dire que les types manipulés par un template ne sont pas fixés (`int`, `double`, etc), mais sont des paramètres.

Vous avez déjà vu des template dans ce cours, le meilleur exemple est `std::vector` et `std::array`. Ces classes template représentent des collections pouvant contenir n'importe quel type de données. Le type manipulé dans la collection est indiqué dans les chevrons :

```
std::vector<int> ints {} ; // tableau de int
std::vector<double> doubles {} ; // tableau de double
```

Pour définir une fonction template, la syntaxe :

```
template<paramètres template>
paramètre_retour nom_fonction(paramètres de fonction) {  
}
```

On voit ici qu'une fonction template prend deux types de paramètre :

- les paramètres template, qui sont des types et sont évalués à la compilation. Mot-clé “typename” ou “class” suivi d'un nom de paramètre ;

- les paramètres de fonction, qui sont des valeurs et sont évaluées à l'exécution (sauf constexpr...)

Les paramètres template déclarés entre les chevrons peuvent ensuite être utilisés dans la fonction (même dans les paramètres de fonction et le type de retour de fonction).

Par exemple, pour la fonction `add`, on peut écrire :

```
template<typename T>
T add(T lhs, T rhs) {
    return lhs + rhs;
}
```

On déclare ici un paramètre template qui se nomme “T”, que l'on utilise comme retour de fonction et comme type pour les paramètres de fonction (n'oubliez pas que `T` représente un type, pas une variable).

Pour appeler une fonction template, en spécifiant les arguments :

```
nom_fonction<arguments template>(arguments de fonction);
```

Les arguments template sont les types qui seront utilisés pour appeler la fonction. Par exemple, écrire `add<int>` permet au compilateur de remplacer `T` par `int` dans le code précédent, qui devient :

```
int add(int lhs, int rhs) {
    return lhs + rhs;
}
```

De même si on écrit `add<double>` ou n'importe quoi d'autre.

On peut donc appeler cette fonction `add` avec différents types d'arguments :

```
int i = add<int>(1, 2);
double d = add<double>(1.2, 3.4);
```

Le type `T` est défini uniquement dans la fonction template, donc il n'est pas possible d'écrire :

```
T i = add<int>(1, 2);  
T d = add<double>(1.2, 3.4);
```

(cela n'aurait pas de sens, le compilateur ne sait pas si `T` = `int` ou `double`). Il est possible d'utiliser l'inférence de type :

```
auto i = add<int>(1, 2);  
auto d = add<double>(1.2, 3.4);
```

Dans ce cas, le compilateur sait déterminer le type.

déduction automatique des arguments template

différence avec `auto` → un seul type pour `lhs` et `rhs`, `add(1, 1.2)` pose problème. Possible écrire :

```
template<typename T1, typename T2, typename T3>  
T1 add(T2 lhs, T3 rhs);
```

Nécessite de mettre `T1` au moins (ne peut pas être déduit)

Possible aussi de ne pas mettre `common_type` :

```
template<typename T1, typename T2>  
std::common_type<T1, T2> add(T1 lhs, T2 rhs);
```

plus besoin de spécifier le type de retour

## Les algorithmes de la bibliothèque standard

Idem, avec Itérateur en paramètre template. Prototype :

```
template<typename Iterator>  
void sort(Iterator begin, Iterator end);
```

Lorsque l'on appelle cette fonction, le compilateur détermine quel est le type de `Iterator` en fonction des arguments passés :

```
std::vector<int> v {};
```

```
std::sort(std::begin(v), std::end(v); // on sait que  
Iterator correspond à un itérateur sur un vector<int>
```

[Chapitre précédent](#) [Sommaire principal](#) [Chapitre suivant](#)

[Cours, C++](#)

# Les références

parler de &, const& et && ?

[Cours, C++](#)

<a href="#">Chapitre précédent</a>	<a href="#">Sommaire principal</a>	<a href="#">Chapitre suivant</a>
rethrow exception		

# Gérer les erreurs dans les fonctions

Bon code : facile à maintenir. Veut dire en particulier qu'il vérifie les erreurs et produit des messages lisibles.

## Les assertions

`static_assert` et `assert`

Fonction template → message pas clair, mettre un assert

## pré-conditions et post-conditions

introduction à la programmation par contrat

## Les exceptions

Comment créer une exception ? Dans quel cas les utiliser ?

try catch ?

types de garanties (pas de garantie, basique, forte, nothrow) (exemple : update 2 vector en même temps, cohérence)

<a href="#">Chapitre précédent</a>	<a href="#">Sommaire principal</a>	<a href="#">Chapitre suivant</a>
<a href="#">Cours, C++</a>		

# Les paires, tuples et structures

Conteneur = collection d'objets de même type. Par exemple, dans bibliothèque, livre, mais également des dvd

Pair = 2 éléments (first et second), de type différent. Création avec make\_pair

Tuple = N éléments, création avec make\_tuple.

Inconvénient : non nommé, pas de sens sémantique. Par exemple pair<string, string> peut représenter un nom et un prénom, un titre de livre et un auteur, une marque de voiture et un modèle, etc. Cela dit juste que l'on a 2 string, mais ne donne pas de sens à ces string.

Structure = N éléments de type différent, avec des noms. Agrégation de variables (avec nom, type et valeur). Nom de structure.

Par exemple :

```
struct Personne {  
    string nom {};  
    string prenom {};  
};  
  
struct Livre {  
    string titre {};  
    string auteur {};  
};  
  
struct Voiture {  
    string marque {};  
    string modele {};  
};
```

N'importe quel type :

```
struct Personne {  
    string nom {};  
    int age {}  
    double taille {};  
    bool masculin { true };  
};
```

Voir même contenir d'autre structure :

```
struct Identite {  
    string nom {};  
    string prenom {};  
};  
  
struct Personne {  
    Identite identite {};  
    int age {}  
    double taille {};  
    bool masculin { true };  
};  
  
Personne personne {};  
personne.age = 24;  
personne.identite.nom = "toto";
```

Permet de créer de nouveaux types.

Sémantique liée = mieux que tuple/pair (moins de risque d'erreur, meilleur compréhension du code)

Syntaxe pour déclarer : struct, bloc de déclaration, ne pas oublier le ;

Initialisation des membres : par défaut ou avec une valeur (si cela à un sens)

Accéder aux variable membres. En lecture et en écriture

Généricité ? Ajouter un type template, permet de modifier la structure de données. Par exemple :

```
template<typename T>
```

```
struct Personne {  
    T identifiant {};  
    string nom {};  
    string prenom {};  
};
```

Utilisation :

```
Personne<int> personne_avec_numero_secu {};  
personne_avec_numero_secu.identifiant = 12234567897;
```

```
Personne<string> personne_avec_autre {};  
personne_avec_autre.identifiant = "1a4slns";
```

```
// utilisation possible avec aussi des structures  
Personne<int> personne_avec_identifiant {};  
personne_avec_identifiant.identifiant.nom = "toto";  
personne_avec_identifiant.identifiant.prenom = "dupont";
```

Remarque : déjà rencontré des types génériques : vector et array. Même principe, mais avec plus de fonctionnalités. On verra par la suite en détail comment ajouter encore plus de fonctionnalités.

[Chapitre précédent](#) [Sommaire principal](#) [Chapitre suivant](#)

[Cours, C++](#)

# Mesurer les performances

## Complexité algorithmique

Dans la page de documentation de `std::equal`, il y a un point dont on n'a pas parlé. Il y a une partie appelée "Complexity", qui décrit la complexité algorithmique de `equal`. La complexité est une forme de mesure de la performance d'un algorithme. C'est une notion importante à connaître, mais il faut aussi connaître ses limites.

notation big O

mesure empirique et théorique de la complexité

impact des données de test (données constante, aléatoire, etc) pire cas et meilleur cas. Comment créer des scénarios de tests réalistes

exemple de vector et list

## Mesurer les temps d'exécution

`std::chrono`

## Les gros problèmes à éviter

Dans le tome 1 ? cache, données contiguës, copie inutile, inline, pool d'objets

# Séparer définition et implémentation

Séparation en plusieurs fichiers

header guard. Le processus de pré-compilation

cas particulier des templates

comment compiler (include des .h, compilation des .cpp puis link des .o)

# Les classes et fonctions variadiques

[http://en.cppreference.com/w/cpp/language/parameter\\_pack](http://en.cppreference.com/w/cpp/language/parameter_pack) et  
<http://en.cppreference.com/w/cpp/language/sizeof...>

## Fonctions avec nombre de paramètres non fixés

Permet d'écrire une fonction avec un nombre variable de paramètres de fonction. Exemple simple : on veut afficher un tableau avec des variables. Première approche, avec fonctions classiques et paramètre par défaut :

```
void print_array (int a, int b = 0, int c = 0, int d = 0) {
    std::cout << a << '\t' << b << '\t' << c << '\t' << d <<
'\n';
}

// ou avec surcharge de fonction
void print_array (int a) {
    std::cout << a << '\n';
}

void print_array (int a, int b) {
    std::cout << a << '\t' << b << '\n';
}

void print_array (int a, int b, int c) {
    std::cout << a << '\t' << b << '\t' << c << '\n';
}

void print_array (int a, int b, int c, int d) {
    std::cout << a << '\t' << b << '\t' << c << '\t' << d <<
'\n';
}
```

La première affiche toujours 4 colonnes (rempli avec des 0 si

nécessaire). La seconde évite les 0 inutiles, mais nécessite d'écrire une fonction pour chaque possibilité.

Améliorable pour ne pas prendre que des int :

```
template<class A>
void print_array (A a) {
    std::cout << a << '\n';
}

template<class A, class B>
void print_array (A a, B b) {
    std::cout << a << '\t' << b << '\n';
}

template<class A, class B, class C>
void print_array (A a, B b, C c) {
    std::cout << a << '\t' << b << '\t' << c << '\n';
}

template<class A, class B, class C, class D>
void print_array (A a, B b, C c, D d) {
    std::cout << a << '\t' << b << '\t' << c << '\t' << d <<
'\n';
}
```

Très lourd à écrire, nombre maximal de paramètres fixés

## Les fonctions variadiques

Possibilité d'indiquer qu'une fonction prend un nombre indéterminé de paramètres. A la compilation, le compilateur déterminer lui même le nombre de paramètre.

Comme on ne connaît pas le nombre de paramètres, le plus simple est d'utiliser une fonction récursive, avec une condition d'arrêt. L'idée est que l'on définit une fonction qui prend N paramètres et utilise la fonction N-1 paramètres (fonction récursive) et une fonction qui prend 1 paramètre et s'appelle pas de fonction N (condition d'arrêt).

En pseudo code :

```
f(N paramètres) {  
    appel de f(N-1 paramètres)  
}  
  
f(1 paramètre) {  
}
```

Si on appelle ce code avec 5 paramètres par exemple, le code précédent est "déroulé" (ie convertie en une suite de fonctions non récursives) de la façon suivante :

```
f(5 paramètres) {  
    appel de f(4 paramètres)  
}  
  
f(4 paramètres) {  
    appel de f(3 paramètres)  
}  
  
f(3 paramètres) {  
    appel de f(2 paramètres)  
}  
  
f(2 paramètres) {  
    appel de f(1 paramètre)  
}  
  
f(1 paramètre) {  
}
```

On voit que chaque fonction appelle la fonction N-1 sauf pour N=2, qui appelle la fonction déjà définie f1.

## Syntaxe

Utilisation de `...` dans la liste des paramètres template et la liste des paramètres de fonction.

```
template<class ... T>
void f(T ... values) {
}
```

Appel récursif : extraire le premier paramètre et les autres paramètres,  
appel récursif sur les autres paramètres.

```
template<class T>
void f(T value) {                                // condition d'arrêt
}

template<class T, class ...Args>
void f(T value, Args ... args) {
    f(args);                                     // appel récursif
}
```

Avec le code de print\_array :

```
#include <iostream>

template<class T>
void print_array (T value) {
    std::cout << value << '\n';    // affichage du dernier
élément
}

template<class T, class ...Args>
void print_array (T value, Args ... args) {
    std::cout << value << '\t';    // affichage de l'élément
courant
    print_array(args...);                // affichage des autres
éléments
}

int main() {
    print_array(1, 2, 3);
    print_array(1.2, "hello", 2.3, "world");
}
```

Template spécialisation ?

## Classes template variadiques

Idem, avec des classes.

```
template<class ... Args>
struct A {
    std::tuple<Args...> args;
};
```

## Exercices

- réécrire std::tuple
- écrire make\_tuple

[Chapitre précédent](#) [Sommaire principal](#) [Chapitre suivant](#)

[Cours, C++](#)

# La Pile d'appel des fonctions

Pile stocke les informations lorsque l'on entre dans une fonction, dans *Stack frames*. Permet de stocker

- les arguments de fonction
- les variables locales
- divers informations utiles (retour d'appel de fonction)

Pile “dernier entré, premier sorti” (*Last In, First Out*, LIFO). Similaire à une pile de pièces empilées. On peut ajouter une pièce sur le dessus de la pile et retirer la pièce qui se trouve au dessus.

Déroulement d'un programme :

```
void f() {  
}  
  
void g() {  
    f();  
}  
  
int main() {  
    g();  
}
```

1. entrée dans main. Pile = main
2. appel de g() - Pile = main-g
3. dans g, appel de f. Pile = main-g-f
4. sortie de f, retour dans g. Pile = main-g
5. sortie de g, retour dans main. Pile = main

Avec des variables locales

```
void f() {  
    int j { 456 };
```

```
}

void g() {
    int i { 123 };
    f();
}

int main() {
    g();
}
```

Pile :

1. main
2. main - g+i
3. main - g+i - f+j
4. main - g+i
5. main

Il est possible de manipuler directement les informations dans la Pile, mais le fonctionnement exact dépend du système, du compilateur et des options de compilation. Nécessite d'écrire un code spécifique pour chaque cas possibles et de bien connaître la mécanique interne du fonctionnement des programmes, cela sort du code de ce cours.

[Chapitre précédent](#) [Sommaire principal](#) [Chapitre suivant](#)

[Cours, C++](#)

# Sampler

Écrire une fonction qui prend un tableau et génère une partition (découpe le tableau en sous tableau, répartition aléatoire des éléments, de façon à ce que tous les éléments sont retrouvé dans la partition et deux à deux disjoint)

[R sample](#)

[Fisher-Yates shuffle](#)

# Jouer au Poker

Evaluer une main de poker, trouver la meilleure main parmi plusieurs mains, trouver la meilleure main de 5 cartes parmi 7 cartes

Fonctions à créer :

- `poker(hands) # return best hand`
- `hand_rank(hand) # return rank of hand`
- `card_rank(hand) # return ranks of cards`
- `straight(ranks), flush(ranks), kind(ranks), two_pair(ranks)`. cf [List of poker hands](#)
- `deal(players) # deal hands for player`

# Algorithmes d'extraction de sous-chaînes

Comparaison des performances entre lire séquentielle, regex, Knuth-Morris-Pratt, etc

# Résoudre des intégrammes

Résoudre un [Intégramme](#)

[Cours, C++](#)

# Jouer au Scrabble

Trouver le meilleur mot pour un scrabble. Sans le plateau. Avec un plateau vide. Avec un plateau plein

[Cours, C++](#)

# Analyseur simple d'expressions régulières

Ecrire un parseur simple pour regex simplifié. Parcours des caractères de la regex et fonction récursive

[Cours](#), [C++](#)

# Concevoir une bibliothèque

Idée générale : penser son code en terme de composants réutilisables =  
création de libs