

INFORMATIQUE ET C++

V. Chabannes, P. Helluy

- Mise à niveau en concepts informatiques: structures de données et algorithmes
- Conception, définition de structures de données et mise en œuvre informatique d'algorithmes classiques
- Maîtrise des principes de base de la programmation orientée objet
- Types abstraits de données : spécification, encapsulation objet, constructeurs, observateurs, destructeurs. Implantation des types classiques : piles, files, listes, tables, arbres et graphes.
- Programmation en C++

- 2 TP sur machine (2x0.3)
- Examen écrit (0.4)

1. ALGORITHMS ET PROGRAMMATIONS

Un algorithme est une suite d'instructions permettant de résoudre un problème : à partir de données d'entrée, il produit un résultat via des opérations élémentaires.

- Provient du nom du mathématicien persan Al-Khawarizmi (vers 820), le père de l'algèbre.

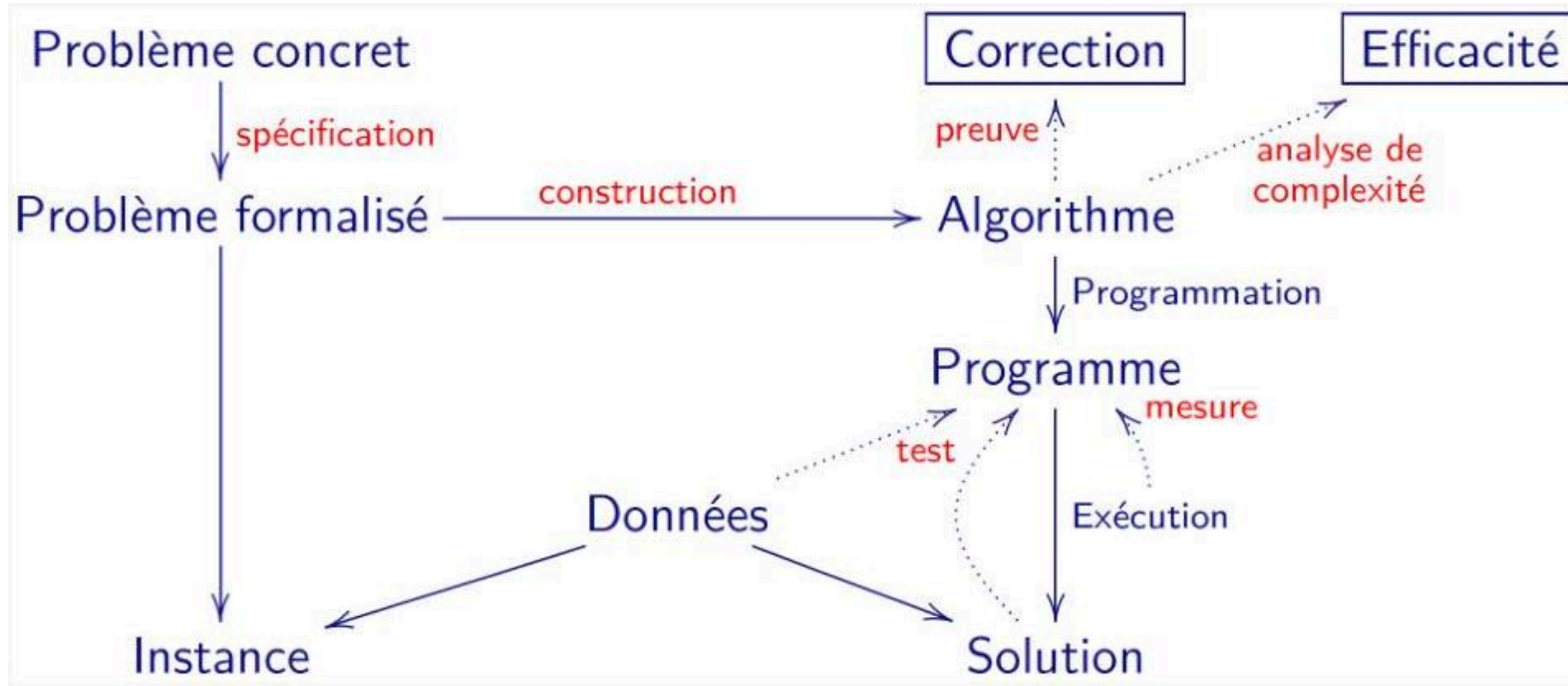


- Exemples d'algorithmes :
- Une recette de cuisine (ingrédients → plat préparé)
- La recherche dans un dictionnaire (mot → définition)

- La division entière (deux entiers → leur quotient)
- Le Rubik's Cube (séquence → séquence ordonnée)
- Dans chaque exemple, on s'intéresse à une classe de problèmes similaires, dont chaque instance est définie par des données.
- L'algorithmique est l'étude des algorithmes : conception, analyse de performance, etc.

Questions classiques lors de la résolution d'un problème algorithmique :

- Est-ce que je peux construire un programme qui résout mon problème (sur une machine conforme à mon modèle) ?
 - Spécification du problème et construction d'un algorithme
- Est-ce que l'exécution de mon programme donne bien le résultat souhaité ?
 - Vérification de propriétés qualitatives par test et preuve
- Est-ce que mon programme fournit le résultat en un temps acceptable ?
 - Validation de propriétés quantitatives par mesure et analyse



La résolution de problèmes algorithmiques combine souvent des algorithmes avec des structures de données adaptées, pour optimiser les accès et modifications selon les besoins de l'algorithme.

Structures de données

- Types atomiques fournis par défaut : `int`, `float`, `char`, `bool`, ...
- Structures de données : définies par l'utilisateur ou fournies dans une bibliothèque. Elles permettent de stocker et d'organiser des données de manière spécifique.
- Exemples : piles, files, listes, tables, arbres, graphes, ...

La programmation consiste à traduire un algorithme en un programme écrit dans un langage de programmation, puis exécuté sur un ordinateur à partir de données concrètes.

- Tous les langages usuels sont Turing-complets : ils ont le même pouvoir d'expression
- Le choix du langage a cependant un impact sur la facilité d'écriture et l'efficacité du programme

Un algorithme peut être spécifié de différentes manières :

- en langage naturel
- graphiquement
- algorithme
- en pseudo-code
- par un programme écrit dans un langage informatique

La seule condition est que la description soit précise.

Problème du tri

On possède une collection d'éléments comparables. On veut les obtenir triés : une collection contenant les mêmes éléments, ordonnée du plus petit au plus grand.

Exemple avec des entiers :

- Entrée : 31, 41, 59, 26, 41, 58

Sortie : 26, 31, 41, 41, 58, 59

Exemple avec des dates :

- Entrée : 10/03, 21/02, 2/12, 30/08

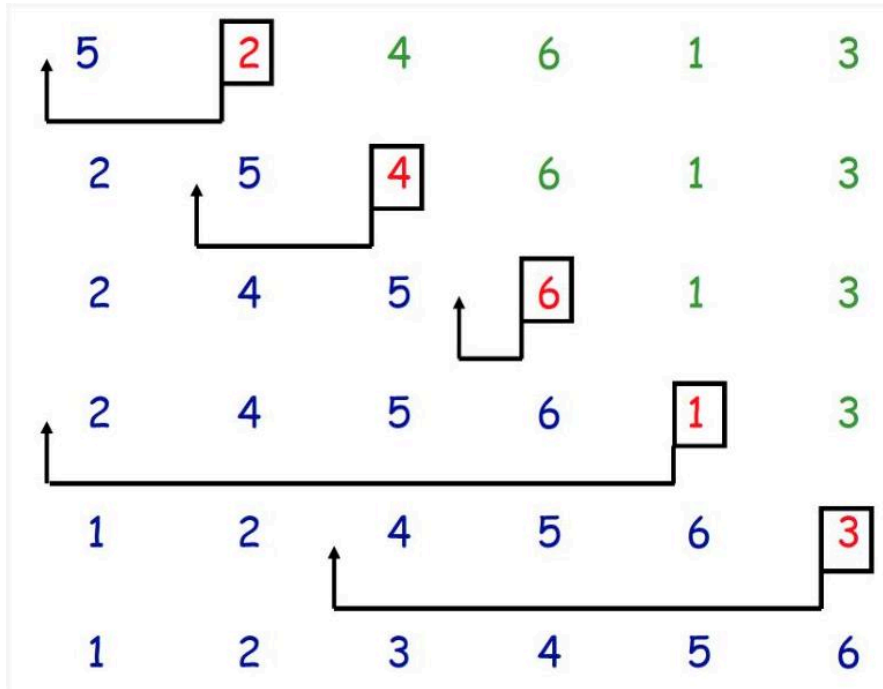
Sortie : 21/02, 10/03, 30/08, 2/12



- On s'arrête dès que le dernier élément a été inséré

Description en langage naturel :

- On prend le premier élément de la séquence non ordonnée : cela forme une séquence ordonnée d'un seul élément.
- On parcourt ensuite le reste de la séquence non ordonnée de gauche à droite et pour chaque élément :
 - On parcourt la séquence ordonnée de droite à gauche
 - On l'insère dès que l'on trouve sa position

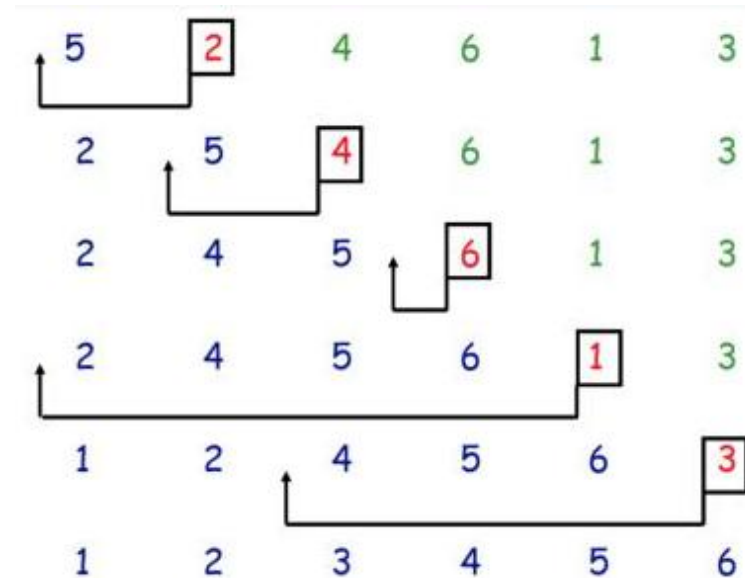


En programmation, le pseudo-code (ou LDA, Langage de Description d'Algorithmes) décrit un algorithme en langage quasi naturel, sans référence à un langage particulier.

Pseudo-code

Require: A séquence à trier

```
for j = 2 to A.length do
  key = A[j]
  i = j - 1
  while i > 0 and A[i] > key do
    A[i + 1] = A[i]
    i = i - 1
  end while
  A[i + 1] = key
end for
```



Pseudo-code

Require: A séquence à trier

```
for j = 2 to A.length do
  key = A[j]
  i = j - 1
  while i > 0 and A[i] > key do
    A[i + 1] = A[i]
    i = i - 1
  end while
  A[i + 1] = key
end for
```

Programme C/C++

```
int key, i;
for(int j = 1; j < length; j++)
{
  key = a[j];
  i = j-1;
  while ( i>=0 && a[i]>key )
  {
    a[i+1] = a[i];
    i = i-1;
  }
  a[i+1] = key;
}
```

2. LA PROGRAMMATION ET LE C++

Un programme informatique est une suite d'instructions exécutables par l'ordinateur. L'ordinateur ne sait manipuler que du binaire : une succession de 0 et de 1.

Pour simplifier, les informaticiens ont créé des langages intermédiaires, compréhensibles par l'homme : les langages de programmation. Il en existe aujourd'hui des centaines.

Pour tous les langages de programmation, la création d'un programme s'effectue en 2 étapes :

- Écrire des instructions dans un langage de programmation
- Générer le code binaire grâce à un programme de traduction



- bas niveau : gestion mémoire, types, pointeurs, ...
- haut niveau : STL, nombreuses bibliothèques externes, C++20, ...

Langages de haut niveau

Orientés autour du problème à résoudre. Ils utilisent des mots usuels et des symboles mathématiques familiers.

Langages de bas niveau

Proches du fonctionnement de la machine, ils en exploitent toutes les capacités avec une grande rapidité d'exécution.

Le langage C++ combine à la fois ces deux aspects :



Langages interprétés

Le code source est exécuté par un interpréteur.
Le même code source fonctionne directement sur tout ordinateur. On parle aussi de langages de scripts.

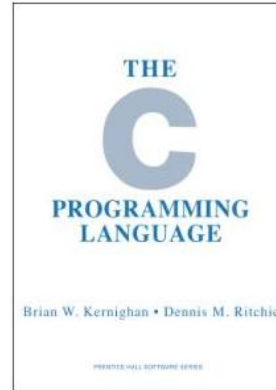
Langages compilés

Le code source est d'abord compilé en code binaire par un compilateur. Ils sont en général plus rapides que les langages interprétés.

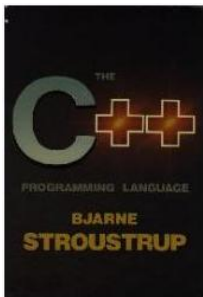
- Interprétés : Python, Matlab, Mathematica, Ruby, PHP, ...

- Compilés : C, C++, Fortran, Pascal, ADA, ...
- Intermédiaire : Java
- Un interpréteur pour le C++ est maintenant disponible : CLING

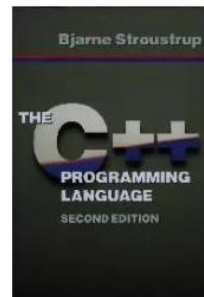
Le langage C a été inventé en 1972 dans les Laboratoires Bell, développé en parallèle d'UNIX par Dennis Ritchie et Ken Thompson. Ken Thompson avait créé le langage B, inspiré de BCPL. Dennis Ritchie a fait évoluer B en ajoutant notamment les types, créant ainsi le C. Brian Kernighan aida à populariser le langage et fut en 1978 le principal auteur du livre **The C Programming Language**.



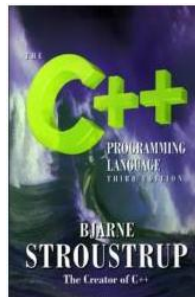
Bjarne Stroustrup a développé C++ dans les années 1980 aux laboratoires Bell. Son but : améliorer le C en lui ajoutant l'approche objet et les classes. Il baptisa d'abord le langage **C with classes**.



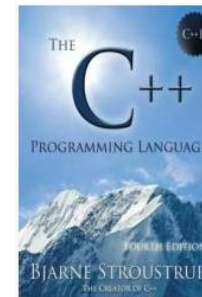
1986 (326 p.)



1991 (696 p.)

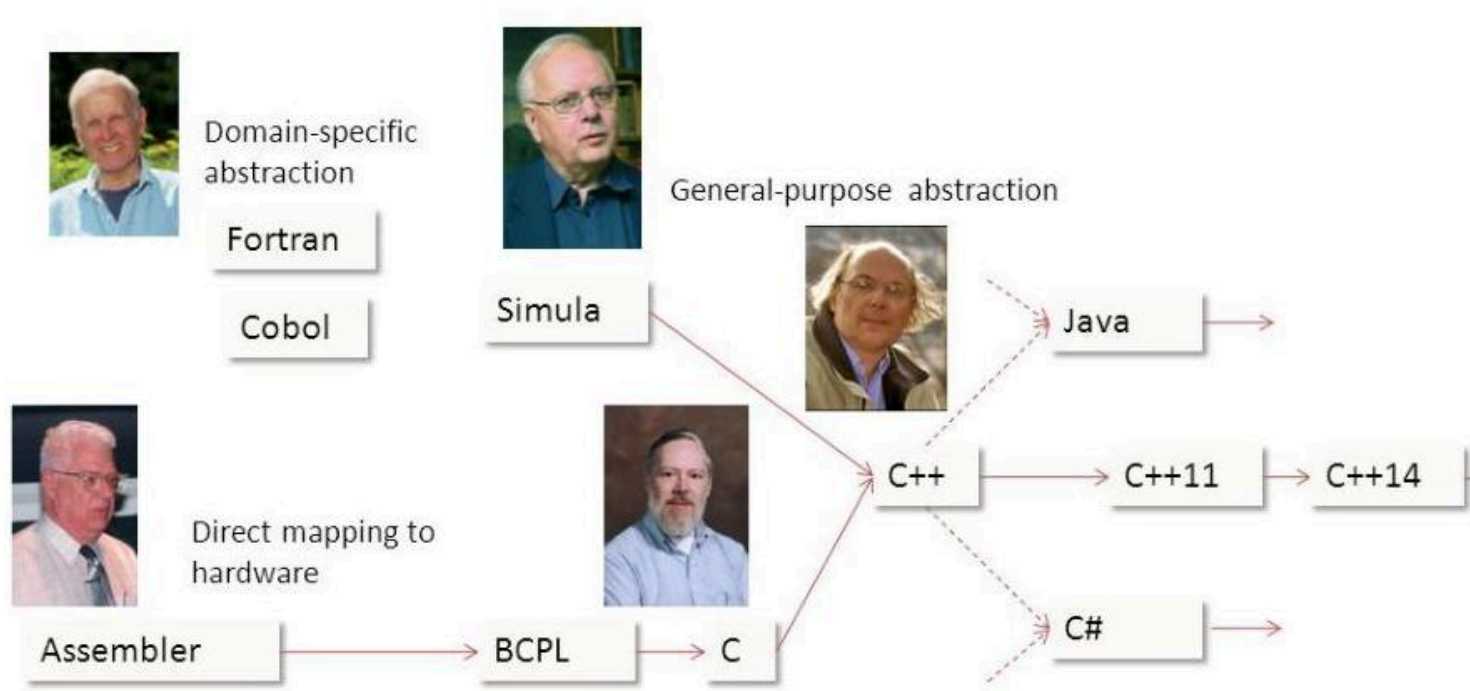


1997 (920 p.)



2013 (1360 p.)

The roots of C++ — Stroustrup, Lang.Next 2014



Le C++ est un langage normalisé par l'ISO qui a évolué en plusieurs standards :

1998:C++98

2003:C++03

2011:C++11

2014:C++14

2017:C++17

2020:C++20

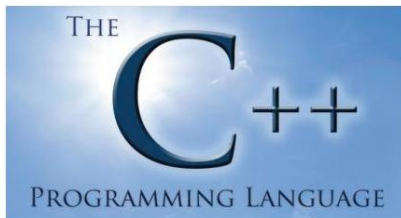
Le prochain prévu est C++23



International Organization for Standardization

Plus d'informations sur <https://isocpp.org/>

Le standard est un document de 1400 pages !





- Polyvalent : dispose de nombreux concepts de programmation : procédurale, orienté objet, programmation générique, STL, itérateurs, ...
- Performant : il est souvent utilisé pour développer des moteurs de jeux vidéo, des applications de bureau, des navigateurs web, du calcul scientifique, ...
- Évolutif : le langage et la bibliothèque standard ne cessent de s'enrichir



Un compilateur C++ traduit le code source en instructions exécutables par le processeur. Cela se passe en plusieurs étapes :

1. Le préprocesseur
2. La compilation
3. L'assemblage
4. L'édition des liens

Compilateurs C++ (liste complète)

- GCC (GNU Compiler Collection, Windows, Linux, MacOS)
- Clang (Windows, Linux, MacOS)
- Microsoft Visual C++ (Windows)
- Borland C++ Builder (Windows)
- Intel C++ Compiler (Windows, Linux, MacOS)

3. EXEMPLES DE PROGRAMME C++

Cet exemple très simple consiste à afficher à l'écran le message Hello, world
hello.cpp

```
#include <iostream>
int main()
{
    std::cout << "Hello, world" << std::endl;
    return 0;
}
```

- ligne 1 : inclusion d'une bibliothèque permettant l'affichage à l'écran
- ligne 2 : fonction de départ d'un programme
- ligne 4 : affichage du message à l'écran
- ligne 5 : fin de la fonction main avec renvoi d'un code d'erreur

Pour compiler ce programme :

```
g++ hello.cpp -o hello
# Pour exécuter ce programme :
./hello
# Hello, world
```

1. Le préprocesseur
2. La compilation
3. L'assemblage
4. L'édition des liens

Pour voir le résultat du préprocesseur (étape 1) :

```
g++ -E hello.cpp > hello.E  
# Pour construire le code assembleur hello.s (étape 1-2) :  
g++ -S hello.cpp -o hello.s
```

1. Le préprocesseur
2. La compilation
3. L'assemblage
4. L'édition des liens

Pour construire le fichier objet hello.o (étapes 1-2-3) :

```
g++ -c hello.cpp -o hello.o
# ou depuis le code assembleur (étape 3) :
as hello.s -o hello.o
# L'édition des liens (étape 4):
g++ hello.o -o hello
```

Pour un programme de taille importante, on le fractionne en plusieurs sous-programmes répartis dans plusieurs fichiers (headers/sources), compilés séparément. On suit des règles d'écriture visant à rendre le programme lisible, portable, réutilisable et facile à maintenir.

- Factorisation de code : éviter de dupliquer du code
- La fragmentation de code : pour des raisons de lisibilité, il est nécessaire de découper un programme en plusieurs fichiers. De plus, cette règle permet de réutiliser facilement une partie du code pour d'autres applications.
- Le nommage des variables, fonctions, classes C++ : des noms bien choisis masquent les détails d'un algorithme qu'il n'est pas indispensable de connaître.

Un programme C++ est réparti dans un ou plusieurs fichiers.

Chacun peut contenir des définitions/ déclarations de fonctions, types, variables et classes. On distingue deux types de fichiers :

- les headers : contiennent les déclarations et peuvent aussi contenir les définitions. Ces fichiers sont généralement suffixés par .h ou .hpp
- les sources : contiennent les définitions. Ces fichiers sont généralement suffixés par .cpp ou .cxx

Le point d'entrée d'un programme est l'unique fonction main, sous deux formes possibles :

- `int main()`
- `int main(int argc, char ** argv)`

Cette fonction est définie dans un fichier source.

- Les directives préprocesseur, commençant par #, se placent généralement en début de fichier.

myfunc.hpp

```
#ifndef __MYFUNC_HPP__
#define __MYFUNC_HPP__

void mymsg();

#endif
```

myfunc.cpp

```
#include "myfunc.hpp"
#include <iostream>

void mymsg()
{
    std::cout << "Hello, world\n";
}
```

mymain.cpp

```
#include "myfunc.hpp"

int main()
{
    mymsg();
    return 0;
}
```

Compilation des fichiers objets associés à chaque fichier source

```
g++ -c myfunc.cpp -o myfunc.o
g++ -c mymain.cpp -o mymain.o
# L'édition des liens :
g++ myfunc.o mymain.o -o myprog
# L'exécution du programme :
./myprog
# Hello, world
```

```
#include <iostream>
class Point {
public:
    Point(double x, double y, double z)
        : M_x(x), M_y(y), M_z(z)
    {}
    double x() const { return M_x; }
    double y() const { return M_y; }
    double z() const { return M_z; }
    Point& operator+=(Point const& pt)
    {
        M_x += pt.x(); M_y += pt.y(); M_z += pt.z();
        return *this;
    }
private:
    double M_x, M_y, M_z;
};
```

```
int main()
{
    Point p(5.2, 10.1, 30);
    Point q(10.5, 20, 36.3);
    p += q;
    std::cout << "x=" << p.x() << " y=" << p.y() <<
    " z=" << p.z() << "\n";
    return 0;
}
```

```
#include <iostream>

template<typename T>
T max(T const& A, T const& B)
{
    return (A >= B) ? (A) : (B);
}

int main()
{
    double x = 3.12, y = 6.3;
    double dmax = max(x, y);
    std::cout << "dmax=" << dmax << "\n";

    int a = 2, b = 3;
    int imax = max(a, b);
    std::cout << "imax=" << imax << "\n";
    return 0;
}
```

```
./myprog
# dmax=6.3
# imax=3
```

```
#include <iostream>

int main()
{
    auto q = []() { std::cout << "Je suis une lambda
expression\n"; };
    q();

    int a = 2;
    auto p = [&a](double d) { return a * a + d; };
    double res1 = p(3.14);
    std::cout << "res1=" << res1 << "\n";
    double res2 = p(5.36);
    std::cout << "res2=" << res2 << "\n";
    return 0;
}
```

```
./myprog
# Je suis une lambda expression
# res1=7.14
# res2=9.36
```

4. ÉLÉMENTS DE BASE DU LANGAGE

- Les entiers :
 - `int` : précision 32 bits
 - `short int` : précision 16 bits
 - `long int` : précision 32 ou 64 bits
 - `long long int` : précision 64 bits
 - `unsigned int` : entiers strictement positifs ou nuls
- Les réels :
 - `float` : simple précision
 - `double` : double précision
 - `long double` : précision étendue
- Les caractères ('a', 'b', 'c', '#', '3', ...):
 - `char` : représente un caractère dans la table ASCII par un entier
 - `signed char` : entier compris entre -127 et 128
 - `unsigned char` : entier compris entre 0 et 255
- Le type logique ou booléen (`true`, `false`) :
 - `bool`

Une variable est un objet nommé contenant des données modifiables pendant l'exécution. En C++, les variables sont typées : leur type détermine l'espace mémoire occupé.

```
int a;  
char b;  
a = 3;  
b = 'c';  
double pi = 3.1415926;
```

Une constante est une variable dont la valeur ne peut être modifiée, définie avec le mot-clé `const`.

```
const int A = 26;  
const double pi = 3.1415926;
```

```
const int varGlobalConst = 1;
int varGlobal;

namespace SomeVariables
{
    const int varGlobalConstNS = 2;
    int varGlobalNS;
}

int main(int argc, char* argv[])
{
    int j = 2;
    if (1 < argc)
    {
        int i = 1;
        j = i;
    }

    SomeVariables::varGlobalNS = varGlobalConst;
    varGlobal = SomeVariables::varGlobalConstNS;
    return 0;
}
```

- Opérateurs arithmétiques
 - Addition : +
 - Soustraction : -
 - Multiplication : *
 - Division : / (division entière dans le cas de int)
 - Modulo : % (reste de la division entière)
- Opérateurs logiques
 - Négation : !
 - ET logique : &&
 - OU logique : ||
- Opérateurs de relation
 - Syntaxes : <, <=, ==, !=
 - Retourne 1 si la condition est vraie et 0 sinon

- Opérateurs d'affectation : =, +=, -=, *=, /=, %=

```
int a = 3;  
a += 5; // a == 8  
a *= 3; // a == 24
```

- Pré/post incrémentation : ++ et --

```
int a = 3;  
int b = a++; // b == 3 et a == 4 (post  
incrémentation)  
int c = ++a; // c == 5 et a == 5 (pré  
incrémentation)  
++c; // c == 6;
```

- Opérateurs bit-à-bit
- ET bit-à-bit : &
- OU bit-à-bit : |
- OU exclusif bit-à-bit : ^

L'instruction d'affectation copie une valeur dans une variable :

```
i = 2;
```

[
lvalue = rvalue
i (gauche) → emplacement mémoire
2 (droite) → valeur]



- **lvalue** (left-value) : emplacement de stockage en mémoire, modifiable. Exemples : une variable, une case de tableau.
- **rvalue** (right-value) : une valeur d'un type précis, sans forcément de zone de stockage. Exemples : une valeur littérale, une constante, une variable, une expression.

Les énumérations définissent un ensemble de constantes nommées (les énumérateurs). Elles donnent du sens à de simples numéros et limitent une variable à un ensemble fini de valeurs symboliques.

Par défaut, la numérotation est automatiquement choisie en partant de 0 et avec une progression de 1.

```
enum couleurs { noir , bleu , vert , rouge ,  
    blanc , jaune };  
// noir==0, bleu==1 , vert==2 , rouge==3 ,  
    blanc==4 , jaune==5
```

Mais on peut les définir partiellement ou complètement.

```
enum couleurs { noir = -2 ,bleu ,vert ,rouge = 5,  
    blanc ,jaune};
```

```
// noir==-2, bleu==-1 , vert==0 , rouge==5 ,  
    blanc==6 , jaune==7
```

Exemple d'utilisation

```
couleurs a = couleurs::vert;  
std::cout << "couleur vert :" << a << "\n";
```

Les pointeurs permettent d'accéder aux données en mémoire par leur adresse. Un pointeur sur un objet de type T se déclare par T*. Il contient l'adresse pointée (un entier non signé) et la taille du type pointé.

```
char x = 'A'; // création d'une variable de type char
char* p = &x; // pointeur sur la variable
```

Le déréférencement d'un pointeur permet de récupérer la variable pointée par le pointeur.

```
char y = *p;
```

adresse	valeur	nom variable
0x7fff55bf19a0	'A'	x
0x7fff55bf1998	0x7fff55bf19a0	p
0x7fff5249798f	'A'	y

- Le pointeur nul permet de désigner un pointeur qui ne pointe sur rien.

```
int *a = 0;  
int *a = NULL;  
int *a = nullptr; // recommended from C++11
```

- Le type `void*` représente un pointeur générique. Il stocke l'adresse de n'importe quelle variable mais pas sa taille. On ne peut pas déréférencer un pointeur générique.

```
void *ptr = nullptr;  
int a = 26;  
ptr = &a;  
char b = 'b';  
ptr = &b;
```

En C++, on peut déclarer une référence `j` sur une variable `i` : c'est un alias de `i`. On peut ainsi modifier le contenu de `i` via `j`.

La déclaration se fait en précisant le type de l'objet, le symbole `&`, et le nom de la référence.

```
int i = 26;  
int &j = i;  
j = 33; // i == j == 33  
int * pi = &i;  
int * pj = &j; // adresses identiques pi == pj
```

- Une référence ne peut être initialisée qu'une seule fois : à la déclaration.
- Une référence ne peut donc référencer qu'une seule variable tout au long de sa durée de vie.

Un tableau est une variable composée de données de même type, stockées de manière contiguë en mémoire. La taille de chaque case dépend du type contenu.

L'instruction suivante permet de réserver en mémoire l'emplacement de 10 entiers.

```
int tab[10];
```

Remarque : la dimension d'un tableau doit être une expression constante.

L'accès au tableau s'effectue à l'aide de l'opérateur [].

```
tab[0]=1;tab[1]=3; // écriture  
int val = tab[3]; // lecture
```

On peut également initialiser les tableaux lors de leur définition :

```
int tab[10] = { 1,2,3,4,5,6,7,8,9,10 };
```

L'identificateur d'un tableau est un pointeur. La notation `tab` est équivalente à `&tab[0]`, l'adresse du premier élément. Par arithmétique des pointeurs, il est équivalent d'écrire :

```
int val1 = tab[5];  
int val2 = *(tab+5)
```

On peut aussi définir des tableaux à plusieurs indices.

```
int multitab[10][3][5]; // definition  
multitab[0][1][3] = 4; // initialisation
```

La mémoire d'un ordinateur est une succession d'octets (8 bits), directement accessibles par leur adresse.

En C++, la mémoire utilisée pour les variables se répartit en deux catégories :

- La pile (stack) stocke les variables désallouées automatiquement. Sa taille est limitée mais réglable (appel POSIX `setrlimit`). Elle suit le modèle LIFO (Last In First Out).
- Le tas (heap) est utilisé pour l'allocation dynamique durant l'exécution. Cette zone est virtuellement illimitée : tant qu'il y a de la mémoire libre, on peut y créer des objets.

Le C++ offre quatre opérateurs d'allocation dynamique : `new`, `new[]`, `delete` et `delete[]`. L'opérateur `new` prend un type en argument et renvoie un pointeur sur l'objet alloué.

Les variables

```
int * var = new int(10); // construction de
l'objet dans le tas
*var = 13; // on modifie la valeur
delete var; // suppression de l'objet dans le tas
```

L'opérateur `new[]` alloue un espace mémoire contigu pour plusieurs données. Le pointeur renvoyé pointe vers la première donnée.

Les tableaux

```
int * tab = new int[10];
tab[3] = 13; // on modifie la case 3 du tableau
delete [] tab;
```

Une chaîne de caractères est décrite en C/C++ par un tableau de caractères se terminant par le caractère de fin de chaîne (valeur `'\0'`).

```
char hola[5] = { 'h', 'o', 'l', 'a', '\\0' };  
char toto[] = "toto";
```

La classe `std::string`

C'est un type de la STL permettant de stocker et manipuler facilement des chaînes de caractères.

```
std::string s1; // s1 contient 0 caractère  
std::string s2 = "hello, world"; // s2 contient 8  
caractères  
std::string s3(60, "*"); // s3 contient 60 étoiles  
(*)  
std::string s4 = s2+s3; // concaténation de s2 et  
s3  
std::string s5(s2, 3, 2); // s5 contient "lo"  
bool isEqual = (s2 == s3); // comparaison
```

```
const char* s6 = s2.c_str(); // récupère le  
tableau de caractères sous-jacent.
```

Il existe de nombreuses fonctionnalités détaillées ici.

Syntaxe

```
if ( expression )  
instruction1  
[else  
instruction2]
```

```
int a = 8, b = 0;  
if (a > 0)  
{  
    int c = 8;  
    b = 2 * c;  
}  
else  
{  
    b = 9;  
}  
int b = (a > 0) ? 3 : 6;  
int c = (a > 0) ? myfunc1(3) : myfunc2(6);
```

Syntaxe

```
switch ( expression )  
case constant-expression : statement  
[default : statement]
```

```
int a;  
std::cout << "Tapez la valeur de a : "; std::cin  
>> a;  
switch (a)  
{  
    case 1:  
        std::cout << "a vaut 1" << std::endl;  
        break;  
    case 2:  
        std::cout << "a vaut 2" << std::endl;  
        break;  
    case 3:  
        std::cout << "a vaut 3" << std::endl;  
        break;  
    default:  
        std::cout << "a ne vaut ni 1, ni 2, ni 3" <<
```

```
std::endl;  
    break;  
}
```

Syntaxe

```
for ( init-expression ; cond-expression ; loop-  
expression )  
instructions
```

```
// The counter variable can be declared in the  
init-expression  
for (int i = 0; i < 2; i++) {  
    std::cout << i;  
}  
// The counter variable can be declared outside  
the for loop.  
int i;  
for (i = 0; i < 2; i++) {  
    std::cout << i;  
}  
// The counter variable can be initialized outside  
the for loop.  
int j = 0; int k = 10;  
for (; j < k; j++) {
```

```
std::cout << i;  
}
```

Syntaxe

```
for( variable : collection )  
instructions
```

```
// un tableau C++  
int tableau[] = { 2, 3, 5, 8, 13 };  
for (int i : tableau)  
    std::cout << i;  
// parcours partiel du conteneur  
for (int m : tableau)  
{  
    if (m > 5) break;  
    std::cout << m;  
}
```

Syntaxe

```
while ( expression )  
instructions
```

```
int i = 0;  
  
while (i < 10)  
{  
    std::cout << "La valeur de i est : " << i << std::endl;  
    i++;  
}  
  
std::cout << "La valeur finale de i est : " << i << std::endl;
```

Syntaxe

```
do  
instructions  
while ( expression ) ;
```

```
int i = 0;  
  
do  
{  
    std::cout << "La valeur de i vaut : " << i << std::endl;  
    i++;  
}  
while (i < 10);  
  
std::cout << "La valeur finale de i est " << i << std::endl;
```

- L'instruction break sert à “casser” ou interrompre une boucle (for, while et do-while), ou un switch.

```
for (int i = 0; i < 10; i++)  
{  
    std::cout << "La variable i vaut: " << i <<  
std::endl;  
    if (i == 5)  
        break;  
}
```

- L'instruction continue sert à “continuer” une boucle (for, while et do-while) avec la prochaine itération.

```
for (int i = 0; i < 10; i++)  
{  
    if (i == 5) continue;  
    std::cout << "La variable i vaut: " << i <<  
std::endl;  
}
```

Une des manières d'interagir avec la console et les fichiers est d'utiliser la bibliothèque standard du C++. Ces flux d'entrées/sorties sont accessibles via l'en-tête `<iostream>`.

- `std::cin` : gère les entrées en provenance de l'entrée standard
- `std::cout` : gère les sorties vers la sortie standard, l'écran
- `std::cerr` : gère les sorties vers l'unité d'erreur standard, l'écran.

```
#include <iostream>

int main()
{
    int n;
    std::cout << "Entrez un nombre positif : ";
    std::cin >> n;
    if (n < 0)
        std::cerr << "Erreur: Le nombre " << n
                    << " n'est pas positif " << "\n";
    else
        std::cout << "Vous avez entré " << n << "\n";
    return 0;
}
```