Programmation Orientée Objet: C++

M1 IM/MF-MPA - Année 2023-2024

Présentation pratique du cours/modalités

Organisation et validation

- Charges horaires sur le semestre:
  - 7 cours de C++ d'1h30
  - 7 TPs de C++ de 2h15

Organisation et validation

- Charges horaires sur le semestre:
  - 7 cours de C++ d'1h30
  - 7 TPs de C++ de 2h15
- Pour valider le module (en contrôle continu)
  - Un TP noté (python - déjà fait) : 1*0.4 de la note finale
  - Deux TPs notés (C++): (0.5 + 0.5)*0.6 de la note finale.
    Un des deux TPs peut potentiellement être remplacé par une évaluation sur papier (questions de cours, QCM, pseudo-code, ...).

Contacts

- Gilles Scarella: gilles.scarella@univ-cotedazur.fr
- Simon Girel: simon.girel@univ-cotedazur.fr

  Ecrire avec le préfixe [M1 POO] dans l'objet du mail
- Supports de cours → Moodle

Plan du cours de C++ (prévisionnel)

- Aspects impératifs du C++, éléments de syntaxe, structures de contrôle, fonctions, pointeurs, tableaux et références
- Structures de données et types utilisateurs
- Objets & Classes, constructeurs, destructeurs
- Surdéfinition d'opérateurs
- Héritage simple, héritage multiple, polymorphisme
- Template
- Entrées/sorties
- Standard Template library (STL)

Auteur initial du cours (R. Ruelle, ingénieur au LJAD)

Objectifs

- Une introduction au langage C++ ainsi qu'au paradigme objet
- L'objectif est de faire découvrir le langage, d'être capable d'écrire et de concevoir un programme C++ simple de bout en bout.

Bibliographie

- *Apprendre le C++ de Claude DELANNOY* (sur lequel s'appuie en partie ce cours)
- Pour les puristes:
  Le langage C++ de Bjarne STROUSTRUP
- Pour les curieux:
  Le langage C. Norme ANSI de Brian W. KERNIGHAN, Dennis M. RITCHIE
- Le site de référence: https://www.cplusplus.com/

Introduction

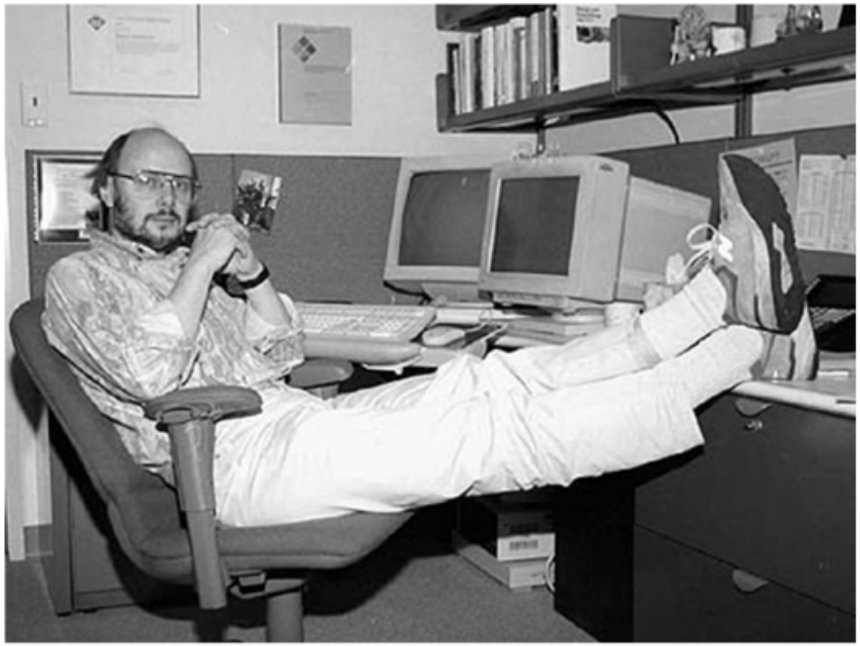
Petite histoire du C/C++

Ken Thompson (à gauche) et Dennis Ritchie (à droite).
(source Wikipédia)

Le C a été inventé au cours de l'année 1972 dans les laboratoires Bell par Dennis Ritchie et Ken Thompson.

En 1978, Brian Kernighan, qui aida à populariser le C, publia le livre "The C programming Language", le K&R, qui décrit le C "traditionnel" ou C ANSI.

Petite histoire du C/C++



Bjarne Stroustrup (source Wikipédia)

Dans les années 80, Bjarne Stroustrup développa le C++ afin d'améliorer le C, en lui ajoutant des "classes". Le premier nom de ce langage fut d'ailleurs "C with classes".

Ce fut en 1998 que le C++ fut normalisé pour la première fois. Une autre norme corrigée fut adoptée en 2003.

Une mise à jour importante fut C++11, suivie de C++14, ajoutant de nombreuses fonctionnalités au langage. Il existe aussi les normes C++17 et C++20.

Toutes ces normes permettent une écriture indépendante du compilateur. Le C++ est le même partout, pourvu qu'on respecte ces normes.

Aspect impératif du C++

- Le C++ est une surcouche de C, avec quelques incompatibilités de syntaxe.
- Un programme C est la plupart du temps un programme C++.
- Donc on peut faire du "C+", c'est-à-dire du C++ sans objet.
- *Impératif:* les instructions se suivent dans un ordre précis et transmis au processeur de la machine dans cet ordre.
- *Impératif* et *objet* ne se contredisent pas, C++ est un langage multi-paradigmes. Il respecte à la fois le paradigme objet et impératif.
- On va donc commencer par faire du C++ impératif.

Environnement de ce cours

- Lors des TPs, le système d'exploitation disponible sur les stations (et préféré dans ce cours) est Linux (Ubuntu).
- Mais vous pouvez aussi programmer en C++ sous Mac et Windows (cf la doc' sur Moodle pour configurer votre environnement)

Environnement de ce cours

- On pourra utiliser l'IDE *Visual Studio Code* pour écrire les codes en C++. Dans ce cours, un éditeur standard conviendra aussi.
- *VisualStudio* existe sous Windows. Sous Mac, il y a *Xcode*
- Les IDE ont certains avantages: choix des options de compilation, aide, *refactoring*, *profiling*. Mais leur utilisation peut être assez compliquée, a fortiori pour des non initiés!
- En résumé, sur les stations sous Linux, on pourra utiliser *Visual Studio Code* ou un éditeur standard, le compilateur GNU *g++* déjà installé et ça suffira!

Avant toute chose

Nous allons rappeler quelques notions élémentaires communes à tous les langages de programmation.

La syntaxe

La syntaxe d'un langage de programmation est l'ensemble de règles d'écriture d'un programme dans ce langage en particulier.

Par exemple, les langages Python, C++ ou même Scilab n'obéissent pas aux mêmes règles syntaxiques.

La syntaxe

Par exemple, pour calculer une puissance:

- En python, on pourra écrire : 5**3 pour 5 au cube, soit 125.
- En Scilab, 5**3 ou 5^3 donneront également 125.
- En C++ ... il n'y a pas d'opérateur de base permettant de calculer une puissance.
  Il faudra appeler une fonction!

Bref:

- En python, on écrit en utilisant la syntaxe python.
- En C++, on écrit en C++ avec la syntaxe du C++.
  Ne préjugez pas de ce que vous connaissez des autres langages pour tenter de deviner la syntaxe d'un autre langage!

La syntaxe

Par exemple, pour calculer une puissance:

- En python, on pourra écrire : 5**3 pour 5 au cube, soit 125.
- En Scilab, 5**3 ou 5^3 donneront également 125.
- En C++ ... il n'y a pas d'opérateur de base permettant de calculer une puissance.
  Il faudra appeler une fonction!

Bref:

- En python, on écrit en utilisant la syntaxe python.
- En C++, on écrit en C++ avec la syntaxe du C++.
  Ne préjugez pas de ce que vous connaissez des autres langages pour tenter de deviner la syntaxe d'un autre langage!

Les paradigmes

Derrière ce mot compliqué se cache le besoin de regrouper les langages de programmation selon leurs fonctionnalités et la façon dont ils seront utilisés.

Le paradigme est un ensemble de notions qui forment un ensemble cohérent et, si un langage respecte ces notions, il pourra alors être considéré comme respectant le paradigme correspondant.

Les paradigmes

Il y a de nombreux paradigmes différents. Et un langage de programmation peut se revendiquer de plusieurs paradigmes distincts.

Par exemple, le C++ est un langage respectant les paradigmes Objet, mais aussi impératif, générique, et même en partie le paradigme fonctionnel.

C'est un langage multi-paradigme.

HelloWorld.cpp

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

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

Listing 1: HelloWorld.cpp

Comme dans la plupart des cours de programmation, on commence par un *HelloWorld*: il s'agit d'un programme très simple qui affiche un message - en l'occurrence "hello world !" - à l'écran.

Dans un éditeur de texte quelconque (ou *Visual Studio Code*), on écrira donc le code ci-contre dans le fichier nommé *HelloWorld.cpp*.

Compilation et exécution

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

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

Listing 1: HelloWorld.cpp

```
g++ HelloWorld.cpp -o HelloWorld
```

On compile ensuite le fichier *HelloWorld.cpp* pour créer un exécutable nommé ici *HelloWorld*.

Le HelloWorld ligne à ligne

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

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

Listing 1: HelloWorld.cpp

```
g++ HelloWorld.cpp -o HelloWorld
```

En C++ comme en C, les lignes commençant par '#' sont des "directives préprocesseur". Elles s'adressent à un programme appelé préprocesseur cpp (pour "c preprocessor"), qui prépare le code source en traitant ces directives.

Ici, en utilisant *#include*, on dit au préprocesseur d'inclure le fichier *iostream* de la bibliothèque standard C++, qui contient les définitions pour afficher quelque chose à l'écran via des "flots".

Le HelloWorld ligne à ligne

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

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

Listing 1: HelloWorld.cpp

```
g++ HelloWorld.cpp -o HelloWorld
```

La deuxième ligne sera considérée comme "magique" dans ce cours. Elle devra figurer systématiquement dans vos codes afin de simplifier leur écriture.

On dit ici au compilateur que les objets de l'espace de nom "std" peuvent être utilisés sans que ce soit précisé lors de leur appel.

Le HelloWorld ligne à ligne

```cpp
#include <iostream>

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

Listing 2: HelloWorld2.cpp

```
g++ HelloWorld2.cpp -o HelloWorld2
```

La version sans la ligne *'using namespace std;'* contient une syntaxe plus compliquée.

Dans ce cours, par souci de simplification, la ligne *'using namespace std;'* devra toujours figurer dans vos codes, en début de fichier.

Le HelloWorld ligne à ligne

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

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

Listing 1: HelloWorld.cpp

```
g++ HelloWorld.cpp -o HelloWorld
```

Il s'agit de l'entête de la fonction *main*. En C++, une fonction se divise en deux parties principales: l'entête et le corps de la fonction.

On peut voir ici trois éléments fondamentaux dans l'écriture d'une fonction.

main est le nom de la fonction. **En C++, c'est aussi le point d'entrée du programme.** Nous verrons plus tard ce que cela signifie. Il faut juste retenir que *main* est la seule fonction qui doit absolument apparaître dans un programme. C'est une convention.

Le HelloWorld ligne à ligne

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

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

Listing 1: HelloWorld.cpp

```
g++ HelloWorld.cpp -o HelloWorld
```

int est le type de retour de la fonction main. *int* pour integer, c'est-à-dire que la fonction *main*, une fois terminée, doit retourner une valeur entière.

Pour information, cette valeur peut être récupérée dans l'environnement appelant notre programme pour indiquer une bonne exécution ou au besoin un code erreur.

Pour la fonction *main*, on n'est pas obligé de renvoyer un entier.

Le HelloWorld ligne à ligne

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

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

Listing 1: HelloWorld.cpp

```
g++ HelloWorld.cpp -o HelloWorld
```

() ici vide, il s'agit de la liste des arguments fournis lors de l'appel de notre fonction.

Le HelloWorld ligne à ligne (corps de la fonction main)

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

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

Listing 1: HelloWorld.cpp

```
g++ HelloWorld.cpp -o HelloWorld
```

- Le corps d'une fonction se situe après l'entête de celle-ci et entre deux accolades. Elles définissent en fait le bloc d'instructions de la fonction. Ici il y a une instruction principale, se terminant par un ";"

Le HelloWorld ligne à ligne (corps de la fonction main)

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

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

Listing 1: HelloWorld.cpp

```
g++ HelloWorld.cpp -o HelloWorld
```

- Le corps d'une fonction se situe après l'entête de celle-ci et entre deux accolades. Elles définissent en fait le bloc d'instructions de la fonction. Ici il y a une instruction principale, se terminant par un ";"
- *cout* peut être vu comme l'affichage à l'écran, il s'agit du flot de sortie standard.
- << est un opérateur opérant sur un flot de sortie à sa gauche et une donnée à lui transmettre, à sa droite.
- "hello world !" est une chaîne de caractères, c'est-à-dire un emplacement mémoire contigü contenant un caractère par octet de mémoire et se terminant conventionnellement par le caractère nul. Nous verrons cela plus en détail quand on reparlera des types de données.
- *endl* demande au flux de passer à la ligne suivante.
- La ligne *return 0;* est facultative

Compilation et exécution du HelloWorld

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

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

Listing 1: HelloWorld.cpp

```
g++ HelloWorld.cpp -o HelloWorld
```

Comme il s'agit d'un programme simplissime, la ligne de compilation est elle-même très simple. En la décomposant élément par élément:

g++ est le nom du compilateur C++ de GNU, celui utilisé pour ce cours.

HelloWorld.cpp est le nom du fichier dans lequel on vient d'écrire notre code.

-o HelloWorld est une **option transmise au compilateur** lui demandant de créer un fichier exécutable portant ce nom-là. Il s'agit d'un argument optionnel. Notre programme se nommerait sous Linux *a.out*, sous Windows *a.exe* sans cet argument.

Compilation et exécution du HelloWorld

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

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

Listing 1: HelloWorld.cpp

```
g++ HelloWorld.cpp -o HelloWorld
```

```
./HelloWorld
hello world !
```

L'instruction *./HelloWorld* dans un terminal (sous Linux, Mac ou Windows), permet d'exécuter le programme, qui, comme prévu, affiche "hello world !" et se termine.

Organisation d'un programme en C++

- Le code source d'un programme est un ensemble de fichiers texte qui contiennent les déclarations et les définitions des différents éléments qui seront ensuite transmises au compilateur.
- Un fichier se décompose généralement en 2 ou 3 parties.

Organisation d'un programme en C++

- Le code source d'un programme est un ensemble de fichiers texte qui contiennent les déclarations et les définitions des différents éléments qui seront ensuite transmises au compilateur.
- Un fichier se décompose généralement en 2 ou 3 parties.
- Les directives préprocesseur (souvenez-vous, elles commencent par #) se situent généralement en début de fichier.
- Viennent ensuite les définitions et déclarations de variables ou de fonctions ou de type de données. Il ne s'agit pas d'instructions à proprement parler, mais plutôt d'informations qui permettront au compilateur de vérifier la cohérence du code écrit ensuite. Cela peut être assez long, et, souvent le programmeur les déplace dans un fichier header suffixé en .h, .hh ou .hpp qui sera inclus via une directive préprocesseur.
- Enfin viennent les définitions des fonctions et le code du programme à proprement parler, dans un fichier suffixé en .cpp pour le C++ (pour rappel, on utilise .c pour le C).

Organisation d'un programme en C++

```
#include <iostream>
using namespace std;

//  Déclaration de la fonction somme
double somme(double a, double b);

// Point d'entrée de notre programme
int main()
{
  double a, b;
  cout << "Donnez deux reels" << endl;
  cin >> a >> b;
  cout << a << " + " << b << " = " << somme(
       a, b) << endl;
  return 0;
}

// définition de la fonction somme
double somme(double a, double b)
{
  return a+b;
}
```

Listing 3: somme0.cpp

Voici l'exemple d'un programme un peu plus complexe.

On commence par déclarer une fonction *somme*, sans la définir, c'est-à-dire sans écrire son code. On indique seulement qu'il existe une telle fonction, prenant deux réels en argument et renvoyant un réel.

On peut à présent l'utiliser dans le code qui suit la déclaration.

On pourrait aussi, et on doit même le faire pour plus de propreté, écrire cette partie dans un fichier header (ou aussi d'entête).

Organisation d'un programme en C++

```
#include <iostream>
using namespace std;

//  Déclaration de la fonction somme
double somme(double a, double b);

// Point d'entrée de notre programme
int main()
{
  double a, b;
  cout << "Donnez deux reels" << endl;
  cin >> a >> b;
  cout << a << " + " << b << " = " << somme(
       a, b) << endl;
  return 0;
}

// définition de la fonction somme
double somme(double a, double b)
{
  return a+b;
}
```

Listing 3: somme0.cpp

La fonction *somme* doit toutefois être définie quelque part; ici, on la définit en fin de programme.

On aurait pu également la définir dans un autre fichier .cpp que l'on compilerait séparément ou non.

Nous verrons tout cela dans la suite de ce cours.

Organisation d'un programme en C++

```
#ifndef __SOMME_HH__
#define __SOMME_HH__

/* Déclaration de la fonction
    somme */

double somme (double a, double b);

#endif
```

Listing 4: somme.hh

On utilise ici un fichier d'entête (appelé aussi fichier header).

C'est un exemple typique de fichier header.

On voit apparaître 3 nouvelles directives préprocesseur ainsi que la déclaration de la fonction *somme*.

On remarquera aussi que ce fichier **ne contient pas de code** mais seulement des déclarations.

Organisation d'un programme en C++

```
#ifndef __SOMME_HH__
#define __SOMME_HH__

/* Déclaration de la fonction
    somme */

double somme (double a, double b);

#endif
```

Listing 4: somme.hh

Il s'agit simplement d'une protection, évitant lors de programmes plus complexes, d'inclure deux fois un fichier header. Le compilateur terminerait alors en erreur car il ne veut pas de multiples déclarations (surtout lorsqu'on définira des classes).
En gros, si la constante *__SOMME_HH__* n'est pas définie, alors il faut inclure le code ci-après.
Dans le code en question, on commence par définir une telle constante et on déclare notre fonction.
Enfin, on ferme la condition *#ifndef* par *#endif* .

Déclarations de variables

```
int i;   // On déclare une variable de type entier appelée i.
float x; // On déclare une variable de type flottant x (
    approximation d'un nombre réel)
const int N = 5;  // On déclare une constante N de type entier
    égale à 5.
```

En C++, avant d'utiliser une variable, une constante ou une fonction, on doit la déclarer, ainsi que son type. Ainsi, le compilateur pourra faire les vérifications nécessaires lorsque celle-ci sera utilisée dans les instructions.

Ci-dessus on peut voir l'exemple de trois déclarations.

Variables

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

int main()
{
  int a;

  a = 0;
  cout << "a vaut " << a << endl;
  a = 5;
  cout << "a vaut à présent: " << a << endl;
}
```

Listing 5: variablea.cpp

Une variable, comme son nom l'indique, est un espace mémoire dont le contenu peut varier au cours de l'exécution.

La variable *a* vaut d'abord 0.

Puis on lui donne la valeur 5; son emplacement en mémoire n'a pas changé, mais le contenu si.

Types de données

Types de données

Le C++ est un langage "fortement typé".

La compilation permet de détecter des erreurs de typage.

Chaque variable d'un programme possède un type donné tout au long de son existence.

Un type peut représenter une valeur numérique sur 1, 2, 4 ou 8 octets, signée ou non. La valeur en question peut être un nombre à virgule flottante dont l'encodage en mémoire est assez complexe.

Types de données numériques

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

int main()
{
  int a;                // On déclare un entier a; on réserve donc 4 octets en mémoire que
        l'on nomme 'a'
  unsigned int b;       // On déclare un entier non signé b, 4 octets sont aussi alloués
  char c;               // On déclare un caractère 'c', un octet est réservé
  double reel1, reel2; // deux réels sont déclarés et la place correspondante en mémoire
       est allouée

  a = 0;        // On attribue à 'a' la valeur 0 -> jusqu'à maintenant, elle n'avait pas
       de valeur
  b = -1;       // On essaye de donner une valeur négative à b !
  c = 'a';      // 'a' est la notation pour le caractère a.
  reel1 = 1e4; //  reel1 prend la valeur 10000
  reel2 = 0.0001;
  cout << "a : " << a << " " << endl
            << "Interessant : "
            << "b : " << b << endl    // HA ! - ça n'est pas -1!
            << "c ; " << c << " " << endl;
  cout << reel1 << endl;
  cout << reel2 << endl;
}
```

Listing 6: typeA.cpp

Types de données numériques

En C++, comme dans de nombreux autres langages, une distinction forte est faite entre la représentation des entiers d'une part, et des nombres à virgule d'autre part.

Il y a au moins 8 types permettant de stocker des entiers, 3 types permettant de stocker des nombres à virgule flottante et 1 type particulier pour représenter les booléens.

A noter que les nombres complexes (par exemple) ne sont pas des types natifs du langage. Si on veut les utiliser, il faudra les coder soi-même ou utiliser des codes déjà faits (librairies etc.)

Types de données numériques

Les entiers peuvent être déclarés en utilisant les types suivants:

Type de donnée	Signification	Taille (en octets)	Plage de valeurs acceptée
char	Caractère	1	-128 à 127
unsigned char	Caractère non signé	1	0 à 255
short int	Entier court	2	-32 768 à 32 767
unsigned short int	Entier court non signé	2	0 à 65 535
int	Entier	2 (sur processeur 16 bits) 4 (sur processeur 32 bits)	-32 768 à 32 767 -2 147 483 648 à 2 147 483 647
unsigned int	Entier non signé	2 (sur processeur 16 bits) 4 (sur processeur 32 bits)	0 à 65 535 0 à 4 294 967 295
long int	Entier long	4 (processeur 32 bits) 8 (processeur 64 bits)	-2 147 483 648 à 2 147 483 647 -9 223 372 036 854 775 808 à 9 223 372 036 854 775 807
unsigned long int	Entier long non signé	4 (processeur 32 bits) 8 (processeur 64 bits)	0 à 4 294 967 295 0 à 18 446 744 073 509 551 615

Types de données numériques

On notera qu'une distinction est faite pour certains types en fonction de la catégorie de processeur qui est utilisée. Sans entrer dans les détails, il est bon de le savoir, même si les processeurs actuels sont très majoritairement en 64 bits.

Pourquoi ne pas utiliser tout le temps le type le plus large?

Pour des raisons évidentes de coût ! Utiliser des types de données sur 8 octets pour stocker des données qui ne dépasseront jamais une valeur assez basse est un gaspillage mémoire.

Et la mémoire coûte cher.

Types de données numériques

Le langage C++, nativement, ne sait travailler qu'avec des représentations numériques.

Ainsi, il n'est pas possible dans une machine dont la mémoire est finie de représenter l'ensemble des réels qui est infini.

On va donc travailler dans un sous-ensemble de l'ensemble Réel avec des approximations plus ou moins précises en fonction de la quantité de mémoire que l'on va utiliser pour la représentation.

C'est ainsi que plusieurs types de données sont là aussi possibles.

Types de données numériques

Type de Base	Taille du type en octets	Taille de l'exposant	Taille de la mantisse	Nombres de valeurs possibles
float	4	8 bits	23 bits	4 294 967 296
double	8	11 bits	52 bits	18 446 744 073 509 551 616
long double	12	15 bits	64 bits	79 008 162 513 705 374 134 343 950 336

Types de données numériques

Le type *bool* permet de représenter les valeurs *true* et *false*: c'est-à-dire 0 ou 1 (booléens)

Il ne s'agit d'un type pratique permettant de distinguer lors de la lecture du code une valeur qui est destinée à une utilisation logique (vraie ou fausse).

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

Types de données alphabétiques

	30	40	50	60	70	80	90	100	110	120
0:		(	2	<	F	P	Z	d	n	x
1:		)	3	=	G	Q	[	e	o	y
2:		*	4	>	H	R	\	f	p	z
3:	!	+	5	?	I	S	]	g	q	{
4:	"	,	6	@	J	T	^	h	r	|
5:	#	-	7	A	K	U	_	i	s	}
6:	$	.	8	B	L	V	`	j	t	~
7:	%	/	9	C	M	W	a	k	u	DEL
8:	&	0	:	D	N	X	b	l	v	
9:	'	1	;	E	O	Y	c	m	w	

```
...
char mychar = 'a';
...
```

Un caractère (de type char) est un élément de la table ASCII codé sur un octet. Il s'agit en fait d'un nombre entre 0 et 127.
Le caractère 'a' est donc la valeur 97.
'A' se code 65.
Il s'agit d'un jeu de caractères particulier.
Il y en a beaucoup d'autres, Unicode par exemple.

Types de données alphabétiques

```
#include<iostream>
using namespace::std;

int main(void)
{
  char w = 'a';       // w <-> 97
      et 'a'
  cout << w << endl; // ça
      affiche 'a'!
  cout << w+2 << " "
       << 'a' + 1 << endl; //
           affiche 99 et 98!
  char z = 'a' + 1;
  cout << z << endl; // ça
      affiche 'b'!
  return 0;
}
```

Listing 7: essai_char.cpp

- Lorsqu'on affiche w, ce n'est pas 97 qui apparaît, mais 'a' car le compilateur sait qu'on veut afficher un caractère et non sa valeur, grâce à son type.
- *z = 'a' + 1;*
  z est de type *char*, soit le caractère suivant 'a' dans la table.
  Soit 'b' !
- Pour gérer les chaînes de caractères, on utilisera en général le type *string* dans la suite du cours

Opérateurs

Opérateurs

Il y a de nombreux opérateurs en C++.

Tout d'abord, sont définis des opérateurs classiques:

- opérateurs arithmétiques
- opérateurs relationnels (ou de comparaison)
- opérateurs logiques

Il y a des opérateurs moins classiques comme les opérateurs de manipulation de bits.
Et des opérateurs "originaux" d'affectation et d'incrémentation.

Opérateurs arithmétiques

C++ dispose des opérateurs binaires (à deux opérandes) arithmétiques que nous connaissons tous:

Addition '+'; soustraction '−'; multiplication '∗' et division '/'

Il y a aussi un opérateur unaire : '−' pour les nombres négatifs ($-x + y$).

Opérateurs arithmétiques

Ces opérateurs binaires ne sont a priori définis que pour des opérandes de même type parmi:
int, *long int*, *float*, *double*, *long double*, ...

Alors comment faire pour ajouter 1 (entier *int*) et 2.5 (flottant simple précision) ? Ce qui semble assez naturel?

Par le jeu des conversions implicites de type. Le compilateur se chargera de convertir un opérande ou les deux dans un type, rendant l'opération possible.

Opérateurs arithmétiques

```cpp
#include <iostream>

using namespace std;

int main()
{
 int a = 1;
 double b = 3.14;

 cout << sizeof(a) << ": " << a << endl;
 cout << sizeof(b) << ": " << b << endl;
 cout << sizeof(a+b) << ": " << a+b << endl;
 return 0;
}
```

Listing 8: test_aritm.cpp

Ainsi, le programme suivant

- Déclare et initialise un entier a dont la valeur sera 1
- Déclare un "réel" b dont la valeur est 3.14

Mais, a et b ne sont pas du même type! Quel va être alors le type de a+b ? Comme b est un réel, pour ne pas perdre d'information, il faut que a+b soit aussi "grand" que b.

C'est le cas et nous voyons cela grâce à l'opérateur **sizeof** qui donne la taille en mémoire de la valeur transmise en argument.

```
 ./a.exe
4: 1
8: 3.14
8: 4.14
```

Opérateurs arithmétiques - Opérateurs % et /

- L'opérateur '%' fournit le reste de la division entière (=le modulo): il n'est défini que sur les entiers en C++ (ce n'est pas le cas en Java par exemple).

```
cout << 134%5 << endl; // affiche 4
```

  Pour les entiers négatifs : le résultat dépend de l'implémentation ! ne pas utiliser !

- Par ailleurs, il est à noter que la division '/' est différente suivant le type des opérandes:
  s'ils sont entiers alors la division est entière, sinon s'il s'agit de flottants, la division sera réelle.
  →DANGER

Opérateurs arithmétiques

Il n'y a pas d'opérateur pour la puissance : il faudra alors faire appel aux fonctions de la bibliothèques standard du C++ (par exemple: pow(a,b) calcule a^b)

En termes de priorité, elles sont les mêmes que dans l'algèbre traditionnel. Et en cas de priorité égale, les calculs s'effectuent de gauche à droite.

On peut également se servir de parenthèses pour lever les ambiguïtés et rendre son code plus lisible !!

Opérateurs relationnels (ou de comparaison)

Il s'agit des opérateurs classiques, vous les connaissez déjà.
Ils ont deux opérandes et renvoient une valeur booléenne:
$<$, $>$, $<=$, $>=$, $==$, $!=$
Les deux derniers sont l'égalité et la différence.
En effet $=$ est déjà utilisé pour l'affectation!

Petite facétie sur les opérateurs d'égalité

```
#include <iostream>

using namespace std;

int main()
{
    double a = 3.6;
    double b = 4.5;
    double c = 8.1;

    if (a+b == c){
        cout << "a+b=c" << endl;
    }
    else {
        cout << "a+b != c" << endl;
    }
    if (a == c - b){
        cout << "a = c - b" << endl;
    }
    else {
        cout << "a != c - b" << endl;
        cout << "a – (c – b) =" << a-(c-b) <<endl;
    }
}
```

Ce programme va démontrer la difficulté qui existe dans la comparaison des nombres à virgules flottantes.

Explicitons un peu les différentes parties qui deviendrons plus claires lors de la suite de ce cours.

On déclare ici 3 variables de type double a, b et c et on leur assigne immédiatement 3 valeurs. Mathématiquement, on voit que a +b = c.

Ici, on se sert de l'instruction 'if' qui vérifie si la condition 'a + b = c' est vraie ou fausse.

Si la condition est vraie, alors l'affichage de 'a + b = c' sera déclenché.

Sinon, le programme affichera 'a +b != c'.

Ici, l'instruction 'if' va examiner la condition 'a = c – b '. Si la condition est vraie, alors le programme affichera 'a = c – b'. Sinon sera affiché 'a != c – b' ainsi que la différence.

Petite facétie sur les opérateurs d'égalité

```
#include <iostream>

using namespace std;

int main()
{
   double a = 3.6;
   double b = 4.5;
   double c = 8.1;

   if (a+b == c){
      cout << "a+b=c" << endl;
   }
   else {
      cout << "a+b != c" << endl;
   }
   if (a == c - b){
      cout << "a = c - b" << endl;
   }
   else {
      cout << "a != c - b" << endl;
      cout << "a – (c – b) =" << a-(c-b) <<endl;
   }
}
```

On exécute le programme créé par la compilation.

Le résultat n'est pas tel qu'on s'y attend !

En effet 'a +b = c' est bien affiché, mais pour le programme, a est différent de c – b.

Ce qui est mathématiquement faux !

Un nombre à virgule flottante en C++ n'est pas un réel mais une approximation.

Lors d'opérations, une infime erreur d'arrondi peut apparaître et rendre inopérante la comparaison.

```
$ g++ diff_double.cpp -o diff_double
```

```
$ ./diff_double
a+b=c
a != c - b
a-(c-b) = 4.440889e-16
```

Opérateurs logiques

En C++, il y a trois opérateurs logiques:

- **et** (noté &&)
- **ou** (noté ||)
- **non** (noté !)

Ces opérateurs travaillent sur des valeurs numériques de tout type avec la simple convention:
Nul $\Leftrightarrow$ faux
Autre que nul $\Leftrightarrow$ vrai

Court-circuit dans l'évaluation des opérateurs logiques

La seconde opérande d'un opérateur n'est évaluée que lorsque sa connaissance est indispensable.

Typiquement, si on sait déjà, par son premier opérande, qu'un 'ou' ou un 'et' sera vrai ou faux, on n'évalue pas la deuxième partie.

Par exemple:

....
int $a = 4$;
if $(a > 5 \,\&\&\, a <= 7)$
...
Dans cet exemple, on vérifie d'abord si a>5. Comme ce n'est pas le cas, le test a<=7 n'est pas effectué ensuite.

Opérateurs d'affectation élargie

C++ permet d'alléger la syntaxe de certaines expressions en donnant la possibilité de condenser des opérations classiques du type:
variable = variable opérateur expression

Ainsi, au lieu d'écrire 'a = a*b;' on pourra écrire 'a *= b;'

Liste des opérateurs d'affectation élargie:
+ =, − =, * =, / =, % =
| =, ^ =, & =, <<=, >>=

Opérateurs d'incrémentation

```cpp
#include <iostream>
using namespace::std;
// = using namespace std;

int main(void) {  // = int main()
  int a=12, b=5;
  cout << "a = " << a
       << ", b = " << b << endl;
  cout << "a++ = " << a++
       << ", a = " << a << endl;
  cout << "++b = " << ++b
       << ", b = " << b << endl;
  return 0;
}
```

Listing 9: operateurs++.cpp

```
./operateurs++.exe
a = 12, b = 5
a++ = 12, a = 13
++b = 6, b = 6
```

Dans la syntaxe des boucles *for* qu'on verra dans la suite, on utilise l'opérateur d'incrémentation ++.

Pour une variable i, on peut utiliser $++i$ et $i++$ pour un résultat identique (= incrémentation de 1). Mais il y a une différence mineure, visible dans l'exemple ci-contre.

- $++i$ modifie immédiatement la valeur de i
- $i++$ modifie "en différé" la valeur de i

Opérateur conditionnel

- Il s'agit d'un opérateur ternaire.
- Il permet des affectations du type:
  Si condition est vraie alors variable vaut valeur, sinon variable vaut autre valeur.
- On l'écrit de la manière suivante:
  $x = (cond)\,?\,a : b;$
- Par exemple:
  $int\, x = (y > 0)\,?\,2 : 3;$

Autres opérateurs

- *sizeof* : Son usage ressemble à celui d'une fonction, il permet de connaître la taille en mémoire (càd le nombre d'octets) de l'objet passé en paramètre.
- Opérateurs de manipulation de bit:
  - & → ET bit à bit
  - | → OU bit à bit
  - ^ → OU Exclusif bit à bit
  - << → Décalage à gauche
  - >> → Décalage à droite
  - ~ → Complément à un (bit à bit)

Structures de contrôle

Structures de contrôle

Un programme est un **flux d'instructions** qui est exécuté dans l'ordre.
Pour casser cette linéarité et donner au programme une relative intelligence, les langages de programmation permettent d'effectuer des choix et des boucles.
On utilise très souvent un bloc d'instructions: il s'agit d'un ensemble d'instructions entouré de deux accolades, l'une ouvrante et l'autre fermante.

```
{
...
int a = 5;
/* des commentaires */
double c = a + 5;
...
}
```

Les instructions placées entre les accolades ouvrante '{' et fermante '}' font partie du même bloc d'instructions.

Les conditions - l'instruction if

```
#include <iostream>
using namespace std;

int main()
{
  int a;

  a = 0;
  if (a == 0)
    {
      cout << "a est nul" << endl;
    }
  else
    {
      cout << "a est non nul" << endl;
    }
  return 0;
}
```

Listing 10: exempleIf.cpp

L'instruction *if* permet de choisir si une partie du code sera exécutée ou pas.

Son utilisation est fondamentale lors de l'écriture d'un programme.

Sa syntaxe est :
if (expression)
instruction_1
else
instruction_2

Les conditions - l'instruction if

```
#include <iostream>
using namespace std;

int main()
{
  int a;

  a = 0;
  if (a == 0)
    {
      cout << "a est nul" << endl;
    }
  else
    {
      cout << "a est non nul" << endl;
    }
  return 0;
}
```

Listing 10: exempleIf.cpp

expression est une expression quelconque avec la convention:
Différent de 0 → vrai
Egal à 0 → faux

instruction_1 et **instruction_2** sont des instructions quelconques, à savoir:

- une instruction simple (terminée par un point virgule)
- un bloc d'instructions
- une instruction structurée

Les conditions - l'instruction if

```
#include <iostream>
using namespace std;

int main()
{
  int a;

  a = 0;
  if (a == 0)
    {
      cout << "a est nul" << endl;
    }
  else
    {
      cout << "a est non nul" << endl;
    }
  return 0;
}
```

Listing 10: exempleIf.cpp

else est un mot-clé du langage permettant d'exécuter le code dans le cas où la condition n'est pas vérifiée.

Son utilisation est facultative.

Structures de contrôle - l'instruction while

```
while (expression)
  instruction
```

L'instruction *while* permet de répéter une ou plusieurs instructions tant que la condition **expression** est vraie.

A noter que:

- Il faut s'assurer que **expression** peut devenir fausse (sinon on ne sort jamais de la boucle !!)
- L'expression est évaluée avant l'exécution des instructions de **instruction**. Celles-ci ne sont donc pas forcément exécutées.

Structures de contrôle - l'instruction for - boucle avec compteur

L'instruction *for* permet de répéter une ou plusieurs instructions avec une syntaxe parfois plus pratique que les boucles *while*.

```
for (expression_declaration; expression_2; expression_3)
  instruction
```

- **expression_declaration** → va permettre d'initialiser le compteur de boucle.
- **expression_2** → une condition sur le compteur pour arrêter la boucle.
- **expression_3** → l'incrémentation du compteur.
- **instruction** → il s'agit d'une instruction simple, d'un bloc, d'une structure de contrôle ...

Structures de contrôle - l'instruction for - boucle avec compteur

```cpp
#include <iostream>

using namespace std;

int main()
{
  for (int i=0; i<10; i++)
    {
      cout << "i = "
           << i << endl;
    }
  return 0;
}
```

Listing 11: exempleFor.cpp

Ce programme, une fois compilé et exécuté, affichera simplement à l'écran les nombres de 0 à 9.
On aurait pu évidemment obtenir ce résultat avec une boucle *while*.

```
g++ exempleFor.cpp

./a.out
i = 0
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i = 8
i = 9
```

Les fonctions

Les fonctions

Pour structurer un programme, un des moyens les plus courants est de diviser le code en briques appelées **fonctions**.

Le terme n'est pas strictement équivalent au terme mathématique.

En effet, une fonction permet de renvoyer un résultat, mais pas seulement: elle peut modifier les valeurs de ses arguments, ne rien renvoyer, agir autrement qu'en renvoyant une valeur: affichage, ouverture et écriture dans un fichier, etc.

Les fonctions - syntaxe

```
type_de_retour nom_de_la_fonction(paramètres)
{
   instructions ...
   return ...
}
```

Il est tout d'abord nécessaire de faire la différence entre la déclaration d'une fonction et sa définition.

Dans un premier temps, une fonction peut être déclarée - c'est-à-dire qu'on signifie au compilateur que cette fonction existe et on lui indique les types des arguments de la fonction et son type de retour - et être définie, dans un second temps : c'est-à-dire qu'on définit l'ensemble des instructions qui vont donner un comportement particulier à la fonction.

La syntaxe ci-dessus présente la définition d'une fonction.

Les fonctions - syntaxe

```
type_de_retour nom_de_la_fonction(paramètres)
{
  instructions ...
  return ...
}
```

type_de_retour → Une fonction peut renvoyer une valeur. Le compilateur doit connaître le type de la valeur afin de pouvoir vérifier la cohérence de l'utilisation de cette valeur de retour dans le reste du programme.

Les fonctions - syntaxe

```
type_de_retour nom_de_la_fonction(paramètres)
{
   instructions ...
   return ...
}
```

nom_de_la_fonction → il s'agit du nom de notre fonction. Il doit bien sûr être cohérent avec ce que fait la fonction en question ...

Les fonctions - syntaxe

```
type_de_retour nom_de_la_fonction(paramètres)
{
   instructions ...
   return ...
}
```

paramètres → Il peut s'agir d'un ou de plusieurs paramètres, séparés par des virgules et qui doivent être précédés par leur type respectif.

Les fonctions - syntaxe

```
type_de_retour nom_de_la_fonction(paramètres)
{
   instructions ...
   return ...
}
```

Vient ensuite le corps de la fonction: il s'agit d'un bloc d'instructions.

Il doit donc débuter avec une accolade ouvrante et se terminer avec une accolade fermante.

La fonction se termine lorsqu'une instruction *return* est exécutée.

Il peut y avoir plusieurs instructions *return* en différents points de la fonction (par exemple dans le cas de l'utilisation d'une condition *if* ...)

Les fonctions - syntaxe

```
type_de_retour nom_de_la_fonction(paramètres)
{
  instructions ...
  return ...
}
```

Il existe un type de retour un peu particulier: *void*

Ce type signifie que la fonction ne renvoie rien. C'est par exemple le cas si elle doit uniquement provoquer un affichage.
Dans ce cas, l'instruction *return* seule (sans argument) est autorisée (par exemple pour sortir de la fonction avant d'atteindre la dernière instruction) car elle ne doit pas retourner de valeur.

Les fonctions - Un exemple de fonction

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

unsigned int mySum(unsigned int N)
{
  unsigned int resu = 0;

  for(unsigned int i=0; i<N+1; i++)
    resu += i;
  return resu;
}

int main()
{
  cout << "Somme jusqu'à 5 inclus = "
       << mySum(5) << endl;
  return 0;
}
```

Listing 12: mySum.cpp

Voici un exemple de fonction.

La fonction *mySum* prend en argument un entier *N* non signé et retourne une valeur de type entier non signé calculant la somme des entiers jusqu'à *N* inclus.

On notera que la fonction *main* est bien sûr également une fonction.

```
./mySum.exe
 Somme jusqu'à 5 inclus
 = 15
```