Programmation Orientée Objet: C++

MAM4 - Année 2024-2025

Présentation pratique du cours/modalités

Organisation et validation

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

Organisation et validation

- Charges horaires sur le semestre:
  - 4 cours de C++ d'1h30
  - TPs de C++ de 2h15
- Pour valider le module (en contrôle continu)
  - 2 TPs notés (C++): (0.5 + 0.5)*0.6 de la note finale.
  - 1 Examen final sur machine.

Organisation: à noter

- Cours + TP du 2 au 5 Septembre 2024
- Un projet à rendre par groupe de 4
- Une évaluation, à définir

Contacts

- Stéphane Abide: stephane.abide@univ-cotedazur.fr
- Clément Rambaud : clement.rambaud@univ-cotedazur.fr

  Ecrire avec le préfixe [MAM4 CPP] 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
- Surcharge d'opérateurs
- Héritage simple, héritage multiple, polymorphisme
- Template
- Entrées/sorties
- Standard Template library (STL)

Auteurs du cours (R. Ruelle & G. Scarella ingénieurs 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

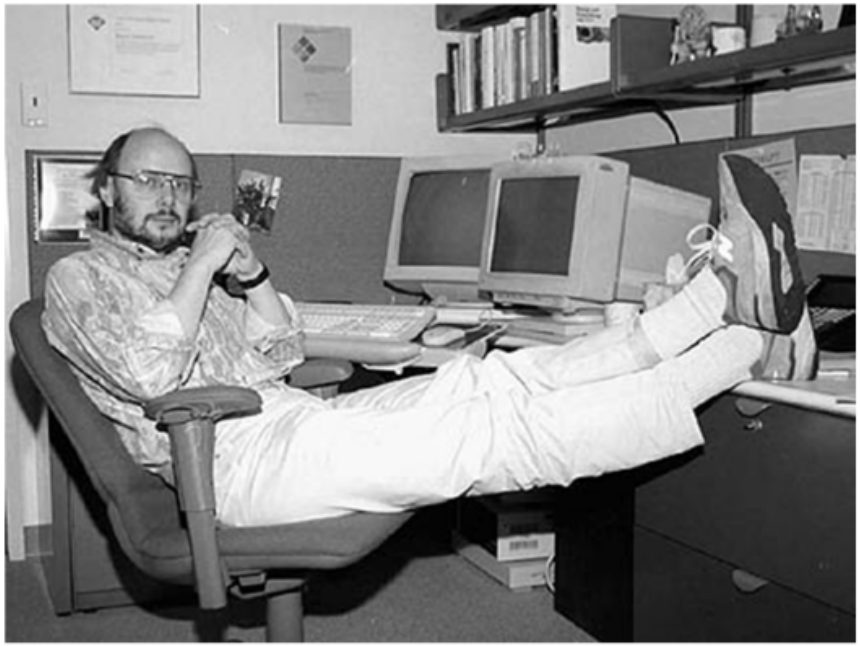
Petite histoire du C/C++

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 la norme C++17.

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>
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 2: 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 2: 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 3: 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 3: 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 4: 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 5: 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 6: 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 7: 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 8: 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 9: 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 9: exemple1f.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

```cpp
#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 9: 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.

L'instruction switch - Syntaxe

```
switch (variable)
{
case constante_1 : [
    instruction_1]
case constante_2 : [
    instruction_2]
...
case constante_n : [
    instruction_n]
[default :
    suite_instruction]
}
```

- L'instruction *switch* permet dans certains cas d'éviter une abondance d'instruction if imbriquées.
- **variable** est une variable quelconque de type entier, dont la valeur va être testée contre les constantes.
- **constante_1** : expression constante de type entier (*char* est accepté car converti en *int*) (idem pour constante_2, ..., constante_n)
- **instruction_1** : suite d'instructions quelconques (idem pour instruction_2, ..., instruction_n et suite_instruction)

Petite subtilité : Une fois un cas positif trouvé, les instructions suivantes sont exécutées, même si elles appartiennent à un autre cas. Ce peut être pratique, mais pas toujours. Pour éviter cela, on utilisera l'instruction *break* qui stoppe le flot d'exécution.

L'instruction switch - Syntaxe

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

int main()
{
  unsigned int a;
  cout << "Valeur de a ?" << endl;
  cin >> a;
  switch (a)
    {
    case 0 :
      cout << "a est nul" << endl;
      break;
    case 1 :
      cout << "a vaut 1" << endl;
      break;
    default:
      cout << "a est > 1" << endl;
      break;
    }
  return 0;
}
```

Listing 10: exempleSwitch.cpp

Voici un exemple d'utilisation de l'instruction *switch*.
Tout d'abord le programme demande à l'utilisateur d'entrer un nombre (entier non signé) au clavier.
Ensuite, en fonction de la valeur de *a* entrée, un affichage différent est obtenu.

Structures de contrôle - l'instruction do .. .while

```
do
instruction
while (expression);
```

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

- La série d'instructions dans **instruction** est exécutée au moins une fois.
- Il faut s'assurer que **expression** peut devenir fausse (sinon on ne sort jamais de la boucle !!)

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 while

```
#include <iostream>
using namespace std;
int main()
{
  int a = 0;

  while (a < 10)
    {
      cout << a << endl;
      a = a + 1;
    }

  do
    {
      cout << a << endl;
      a = a - 1;
    }
  while (a > 0);

  return 0;
}
```

Listing 11: exempleWhile.cpp

Voici un exemple de l'utilisation de *while* puis de *do...while*.

La première boucle affiche les nombres de 0 à 9 et se termine lorsque *a* vaut 10 (pas d'affichage).

La deuxième boucle affiche *a*, puis le décrémente tant que *a* est supérieur à 0.

Que vaut *a* lorsque la deuxième boucle se termine ?

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 12: 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
```

Structures de contrôle - break et continue

Instructions de branchement inconditionnel :

- *break* et *continue* s'utilisent principalement dans des boucles afin de contrôler plus finement le flux d'exécution.
- *break* permet de sortir de la boucle à n'importe quel moment (souvent une condition validée dans la boucle par un *if*)
- *continue* va stopper prématurément le tour de boucle actuel et passer directement au suivant.

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 13: 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
```

Les fonctions - Déclaration de fonctions

Avant de pouvoir utiliser une fonction, c'est-à-dire de l'appeler, il est nécessaire que le compilateur "connaisse" la fonction. Il pourra ainsi réaliser les contrôles nécessaires qui pourront donner lieu à des erreurs de compilation le cas échéant.

Ainsi, on prendra soin d'écrire le "prototype " de la fonction. Pour la fonction *my_pow*,

```
double my_pow(double, unsigned int);
```

- Il n'est pas nécessaire de préciser le nom des paramètres dans ce cas.
- La déclaration se termine par un point virgule

Les fonctions - Passage par valeur

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

/*
  Cette fonction doit échanger la valeur des
  deux entiers passés en paramètres */
void my_swap(int, int);

int main()
{
  int a = 2, b = 3;
  cout <<  "a : " << a << " b : "
       << b << endl;
  my_swap(a, b);
  cout <<  "a : " << a << " b : "
       << b << endl;
  return 0;
}

void my_swap(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}
```

Listing 14: mySwapV0.cpp

Quand on exécute ce programme, on remarque qu'il ne fait pas ce qu'on veut.

Les valeurs de a et de b sont les mêmes avant et après l'appel à la fonction *my_swap*.
Pourquoi?

Par défaut en C++, le passage des arguments à une fonction se fait "par valeur" : c'est-à-dire que la valeur du paramètre est copiée en mémoire, et une modification sur la copie n'entraîne évidemment pas la modification de l'original.

Les fonctions - Passage par référence

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

/*
  Cette fonction doit échanger la valeur des
  deux entiers passés en paramètres */
void my_swap(int &, int &);

int main()
{
  int a = 2, b= 3;
  cout <<  "a : " << a << " b : "
       << b << endl;
  my_swap(a, b);
  cout <<  "a : " << a << " b : "
       << b << endl;
}

void my_swap(int &a, int &b) {
  int tmp = a;
  a = b;
  b = tmp;
}
```

Listing 15: mySwapV1.cpp

Modifions la fonction *my_swap*.

Cette fois-ci, le programme a bien l'effet désiré!

Pourquoi?

La notation 'int &' signifie qu'on ne passe plus un entier par valeur mais par référence. Il n'y a donc plus copie de la valeur. On passe directement la valeur elle-même.

Si les arguments sont modifiés dans la fonction, ils sont donc modifiés après l'appel à la fonction.

Les fonctions - Passage par référence

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

/*
  Cette fonction doit échanger la valeur des
  deux entiers passés en paramètres */
void my_swap(int &, int &);

int main()
{
  my_swap(2, 3);
}

void my_swap(int &a, int &b) {
  int tmp = a;
  a = b;
  b = tmp;
}
```

Listing 16: mySwapV2.cpp

Quand on tente de compiler ce programme, le compilateur termine en erreur.
Pourquoi?
A la lecture du message, on comprend qu'on ne fournit pas à la fonction un paramètre du bon type.
En effet, on ne peut pas modifier la constante 2 ou 3! Heureusement!

```
$ g++ mySwapV2.cpp
mySwapV2.cpp: Dans la fonction 'int main()':
mySwapV2.cpp:12:15: erreur : invalid initialization of
my_swap(2, 3);
^
mySwapV2.cpp:8:6: note : initializing argument 1 of 'v
void my_swap(int &, int &);
```

Les fonctions - Passage par référence

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

/*
  Cette fonction doit échanger la valeur des
  deux entiers passés en paramètres */
void my_swap(int &, int &);

int main()
{
  my_swap(2, 3);
}

void my_swap(int &a, int &b) {
  int tmp = a;
  a = b;
  b = tmp;
}
```

Listing 16: mySwapV2.cpp

La fonction *my_swap* modifie ses paramètres. On ne peut donc évidemment pas l'appeler avec des arguments constants.

Pour lever cette ambiguité, on considère qu'une fonction qui ne modifie pas ses arguments doit le spécifier dans sa déclaration en ajoutant le mot-clé **const** au type de ses arguments. Sinon, on considère qu'ils sont modifiables.

```
$ g++ mySwapV2.cpp
mySwapV2.cpp: Dans la fonction 'int main()':
mySwapV2.cpp:12:15: erreur : invalid initialization of
my_swap(2, 3);
^
mySwapV2.cpp:8:6: note : initializing argument 1 of 'v
void my_swap(int &, int &);
```

Les fonctions - Mot-clé const

On a déjà vu ce mot-clé.

- Il permet de définir des variables par une seule instruction et qui ne peuvent pas être modifiées dans la suite du programme, elles restent donc constantes tout au long de leur existence.
- L'attribut *const* permet de protéger les variables.
- Une variable *const* est déclarée et définie en même temps.

```
...
const int N=20; // entier constant
N += 2; // erreur de compilation!
....
```

Les fonctions - Variables globales

La portée d'une variable peut varier.

On dit qu'une variable est **globale** lorsque la portée de celle-ci s'étend sur une portion de code ou de programme groupant plusieurs fonctions. On les utilise en général pour définir des constantes qui seront utilisées dans l'ensemble du programme, par exemple si nous devions définir dans une bibliothèque de maths, la valeur PI. Elles sont définies hors de toute fonction, ou dans un fichier header, et sont connues par le compilateur dans le code qui suit cette déclaration.

Leur utilisation est cependant **déconseillée** tant elles peuvent rendre un code compliqué à comprendre et à maintenir.

Nous ne nous attarderons pas sur elles pour l'instant, il faut juste savoir que cela existe.

Les fonctions - Variables locales

Les variables locales sont les variables les plus couramment utilisées dans un programme informatique impératif (de loin !)
Elles sont déclarées dans une fonction et n'existent que dans celle-ci.
Elles disparaissent (=leur espace mémoire est libéré) une fois que la fonction se termine.

L'appel des fonctions et la création des variables locales reposent sur un système LIFO (Last In - First Out) ou de pile.

Lors de l'appel d'une fonction, les valeurs des variables, des paramètres, etc. sont "empilées" en mémoire et "dépilées" lors de la sortie de la fonction.

Le système considère donc que cet espace mémoire est réutilisable pour d'autres usages !!

Les fonctions - Surcharge

- Aussi appelé overloading ou surdéfinition.
- Un même symbole peut posséder plusieurs définitions. On choisit l'une ou l'autre de ces définitions en fonction du contexte.
- On a en fait déjà rencontré des opérateurs qui étaient surchargés. Par exemple, + peut être une addition d'entiers ou de flottants en fonction du type de ses opérandes.
- Pour choisir quelle fonction utiliser, le C++ se base sur le type des arguments.

Les fonctions - Surcharge - Un exemple

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

void printMe(int a)
{
  cout << "Hello ! I'm an integer ! : "
       << a << endl;
}

void printMe(double a)
{
  cout << "Hello ! I'm a double ! : "
       << a << endl;
}

int main()
{
  printMe(2);
  printMe(2.0);
  return 0;
}
```

Listing 17: printMe.cpp

La fonction *printMe* est définie deux fois. Son nom et sa valeur de retour ne changent pas.

Le type de son paramètre change.

Lorsque l'on appelle la fonction, le compilateur se base sur le type de l'argument pour choisir quelle fonction il va appeler.

Dans certains cas, le compilateur n'arrive pas à faire un choix. Il se terminera alors en erreur.

```
Hello ! I'm an integer ! : 2
Hello ! I'm a double ! : 2
```

Tableaux & pointeurs

Exemple

```
#include <iostream>

using namespace std;

int main()
{
  int t[10];

  for(int i=0; i<10; i++)
    t[i] = i;

  for(int i=0; i<10; i++)
    cout << "t[" << i << "]: " << t[i]
         << endl;

  return 0;
}
```

Listing 18: code14.cpp

- La déclaration *int t[10];* réserve en mémoire l'emplacement pour 10 éléments de type entier.
- Dans la première boucle, on initialise chaque élément du tableau, le premier étant conventionnellement numéroté 0.
- Dans la deuxième boucle, on parcourt le tableau pour afficher chaque élément.
- On notera qu'on utilise la notation [] pour l'accès à un élément du tableau.

Quelques règles

Il ne faut pas confondre les éléments d'un tableau avec le tableau lui-même.
Ainsi, par exemple, *t[2] = 3; t[0]++;* sont des écritures valides.

Mais écrire t1 = t2, si t1 et t2 sont des tableaux, n'est pas possible!

Il n'existe pas, en C++, de mécanisme d'affectation globale pour les tableaux.

Quelques règles

Les indices peuvent prendre la forme de n'importe quelle expression arithmétique d'un type entier.

Par exemple, si *n*, *p*, *k* et *j* sont de type *int*, il est valide d'écrire:

```
t[n-3], t[3*p-2*k+j%4]
```

Il n'existe pas de mécanisme de contrôle des indices! Il revient au programmeur de ne pas écrire dans des zones mémoire qu'il n'a pas allouées.
Source de nombreux bugs ...

Quelques règles

En C ANSI et en iso C++, la dimension d'un tableau (=son nombre d'éléments) ne peut être qu'une constante, ou une expression constante. Certains compilateurs acceptent néanmoins le contraire en tant qu'extension du langage.

```
const int N = 50;
int t[N]; // Valide quels que soient la norme et le
    compilateur
int n = 50;
int t[n]; // n'est pas valide systématiquement et doit
    être utilisé avec précaution.
```

Quelques règles - Une parenthèse sur les options de compilation

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

int main()
{
   int n = 50;
   int t[n];
   cout << sizeof(t) << endl;
   return 0;
}
```

Listing 19: test_tab.cpp

C'est l'occasion d'ouvrir une parenthèse sur d'autres options du compilateur g++. Si l'on compile le code ci-contre avec les options ci-dessous, ce code ne compile pas. En effet, par défaut, un compilateur fera son possible pour compiler un programme, quitte à ne pas respecter scrupuleusement la norme du langage.
On peut, en rajoutant ces options, forcer le compilateur à respecter la norme.
Ceci a comme but de garantir un maximum de compatibilité et de portabilité si on change de compilateur ou de version de compilateur.

Pour avoir plus d'information sur ces options, on pourra consulter le manuel de g++. (man g++)

```
$ g++ -std=c++98 -Wall -pedantic -Werror test_tab.cpp
test_tab.cpp: Dans la fonction 'int main()':
test_tab.cpp:8:10: erreur : ISO C++ forbids variable length array 't' [-Werror=vla]
int t[n];
^
cc1plus : les avertissements sont traités comme des erreurs
```

Tableaux à plusieurs indices

On peut écrire:

```
int t[5][3];
```

pour réserver un tableau de 15 éléments (5*3) de type entier.
On accède alors à un élément en jouant sur les deux indices du tableau.

Le nombre d'indices peut être quelconque en C++. On prendra néanmoins en compte les limitations de la machine, comme la quantité de mémoire à disposition.

Initialisation d'un tableau

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

int main()
{
  int t[6] = {0, 2, 4, 6, 8, 10};

  for (int i=0; i<6; i++)
    cout << t[i] << ";" ;
  cout << endl;

  return 0;
}
```

Listing 20: code16.cpp

Nous avons déjà initialisé des tableaux grâce à des boucles.

On peut en fait les initialiser "en dur" lors de leur déclaration.

On utilisera alors la notation {} comme dans l'exemple ci-contre.

Pointeurs

Un pointeur est une adresse mémoire. Il se définit à partir d'un type et c'est un nombre hexadécimal.

Exemples de déclarations de pointeurs:

```
double* p; // pointeur sur double
float* q; // pointeur sur float
int *v; // pointeur sur int
```

Pour écrire la déclaration d'un seul pointeur, on peut placer '*' comme on veut: le coller au type, au nom de la variable ou laisser un espace entre les deux.
Mais pour déclarer simultanément deux pointeurs sur double, p1 et p2, on écrira:

```
double *p1, *p2; // p1 et p2 pointeurs sur double
double* q1, q2; //q1 pointeur sur double, q2 double!
```

Pointeurs - Les opérateurs * et &

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

int main()
{
  int *ptr;
  int i = 42;

  ptr = &i;
  cout << "ptr: " << ptr << endl;
  cout << "*ptr: " << *ptr
       << endl;
  return 0;
}
```

Listing 21: code17.cpp

On commence par déclarer une variable *ptr* de type *int **: un pointeur sur entier. Puis une variable *i* de type entier.

On assigne à *ptr* **l'adresse** en mémoire de la variable *i*, grâce à l'opérateur &.

On affiche ensuite *ptr*: une **adresse**, une valeur qui sera affichée en hexadécimal.

Puis on affiche la valeur pointée par *ptr* (la même que la valeur de *i*). On dit que l'on a **déréférencé** le pointeur *ptr*.

```
$ ./a.out
ptr : 0xffffcbf4
*ptr : 42
```

Pointeurs - Les opérateurs * et &

	adresses	valeurs	variables
mémoire	0x0		
	⋮		
	0x42	0xffffcbf4	ptr
	0x43		
	⋮		
	0xffffcbf4	42	i
	0xffffcbf5		
	0xffffcbf6		
	0xffffcbf7		
	⋮	⋮	⋮

Voici une représentation schématisée de l'exemple précédent.
On voit bien que la valeur de la variable *ptr* est l'adresse en mémoire de la variable *i*.
Le type du pointeur est important: il permet de connaître la taille en mémoire de la valeur pointée!

Pour un type entier, il s'agira des 4 octets suivant l'adresse *0xffffcbf4*.
La taille du pointeur lui-même varie en fonction du nombre de bits du système : 16, 32, ou pour les machines actuelles: 64 bits.

Relation tableaux et pointeurs

En C++, l'identificateur d'un tableau (càd son nom, sans indice à sa suite) est considéré comme un pointeur.

Par exemple, lorsqu'on déclare le tableau de 10 entiers

```
int t[10];
```

La notation t est équivalente à &t[0], c'est-à-dire à l'adresse de son premier élément.
La notation &t est possible aussi (c'est équivalent à t).

Pointeurs - Arithmétique des pointeurs

Une adresse est une valeur entière. Il paraît donc raisonnable de pouvoir lui additionner ou lui soustraire un entier, en suivant toutefois des règles particulières.

Que signifie ajouter 1 à un pointeur sur entier ? Est-ce la même chose que pour un pointeur sur char par exemple?

Non

Ajouter 1 à un pointeur a pour effet de le décaler en mémoire du nombre d'octets correspondant à la taille du type pointé.
En ajoutant (soustrayant) 1 à un pointeur sur *int* (*float*, *double*, *char* ...), on le décale en mémoire de la taille d'un *int* (resp. *float*, *double*, *char* ...).

On appelle ce mécanisme l'arithmétique des pointeurs.

Relation tableaux et pointeurs

On sait maintenant qu'un tableau peut être considéré comme un pointeur.
Plus précisément, il s'agit d'un pointeur constant.
Pour accéder aux éléments d'un tableau, on a donc deux possibilités :

- La notation indicielle : t[5]
- La notation pointeur : *(t+5)

Attention:

- La priorité des opérateurs est importante : *(t+5) $\neq$ *t + 5
- Un nom de tableau est un pointeur constant! On ne peut pas écrire *tab*+=1 ou *tab*=*tab*+1 ou encore *tab*++ pour parcourir les éléments d'un tableau.

Pointeurs particuliers

- **Le pointeur nul** (noté NULL), dont la valeur vaut 0.
  Il est utile car il permet de désigner un pointeur ne pointant sur rien.
  Evidemment déréférencer le pointeur nul conduit irrémédiablement à une erreur de segmentation.
- **Le pointeur générique void ***.
  Un pointeur est caractérisé par deux informations: la valeur de l'adresse pointée et la taille du type pointé.
  *void ** ne contient que l'adresse. Il permet donc de manipuler n'importe quelle valeur sans souci de la taille du type. C'était un type très utile en C, notamment pour écrire des fonctions génériques valables quel que soit le type des données.

Allocation statique et dynamique

<table>
<tr><td rowspan="6">MÉMOIRE</td><td rowspan="3">PILE
(stack)</td><td>main</td><td>variables de la fonction main</td></tr>
<tr><td>fct_1</td><td>variables et arguments de la fonction fct_1
appelée dans main</td></tr>
<tr><td>fct_2</td><td>variables et arguments de la fonction fct_2
appelée dans fct_1</td></tr>
<tr><td colspan="3">La pile peut grandir en occupant la mémoire libre</td></tr>
<tr><td colspan="3">mémoire libre</td></tr>
<tr><td colspan="3">Le tas peut grandir en occupant la mémoire libre</td></tr>
<tr><td></td><td>TAS
(heap)</td><td colspan="2">Le tas offre un espace de mémoire dite d'allocation dynamique. C'est un espace mémoire qui est géré par le programmeur, en faisant appel aux opérateurs d'allocation new pour allouer un espace et delete pour libérer cet espace.</td></tr>
</table>

Ceci est une représentation schématisée de la mémoire occupée par un processus au cours de son exécution.

On connaît déjà la **pile** (ou stack en anglais) qui contient les variables et les tableaux que l'on a déclarés jusqu'à présent.

Le **tas** (ou heap) est une zone de la mémoire qui peut grandir au fil de l'exécution et dont le contenu est géré par le programmeur. Mais,
"Un grand pouvoir implique de grandes responsabilités!"

Les opérateurs new et delete

- *new* est un opérateur unaire prenant comme argument un type.
- La syntaxe est: *new montype* où *montype* représente un type quelconque.
- Il renvoie un pointeur de type *montype** dont la valeur est l'adresse de la zone mémoire allouée pour notre donnée de type *montype*. Par exemple:

```
int *ptr = new int;
```

- On peut maintenant utiliser notre pointeur pour accéder à un entier dont on a alloué l'espace mémoire.
- **(Important)** Une autre syntaxe permet d'allouer un espace mémoire contigü pour plusieurs données à la fois. Le pointeur renvoyé, toujours de type *montype**, pointe vers la première valeur allouée.

```
int* ptr2 = new int[10];
```

- **L'allocation peut-elle échouer? Si oui que se passe-t-il?**

Les opérateurs new et delete

- On ne peut évidemment pas allouer indéfiniment de la mémoire, celle-ci étant finie. Un programme trop gourmand risque de mettre le système entier en danger et bien souvent celui-ci préfèrera le terminer de manière brutale.
- *delete* est l'opérateur permettant de faire le ménage dans la mémoire en libérant l'espace qui ne sert plus.
- Lorsqu'un pointeur *ptr* a été alloué par *new*, on écrira alors *delete ptr* pour le libérer.
- S'il s'agit d'un tableau, on écrira *delete [] ptr;*

Les opérateurs new et delete

Remarques:

- Des précautions doivent être prises lors de l'utilisation de *delete*.
- *delete* ne doit pas être utilisé pour des pointeurs déjà détruits.
- *delete* ne doit pas être utilisé pour des pointeurs obtenus autrement que par l'utilisation de *new*.
- Une fois un pointeur détruit, on doit évidemment arrêter de l'utiliser.

Exemple

```
#include <iostream>
using namespace std;
double sum(double *val, int n)
{
  double res = 0;
  for (int i=0; i<n; i++)
    res += val[i];
  return res;
}

int main()
{
  int n;
  double *val;
  cout << "nombres de valeurs: ";
  cin >> n;
  val = new double[n];
  for (int i = 0; i<n; i++) {
    cout << i << ": " ;
    cin >> val[i];
  }
  cout << "Moyenne de ces valeurs: " <<
          sum(val,n)/n << endl;
  delete [] val;
  return 0;
}
```

Listing 22: code18.cpp

Ce programme calcule la moyenne des valeurs que l'utilisateur entre au clavier durant l'exécution. Mais le nombre de ces valeurs varie aussi! Si nous avions alloué un tableau statiquement, sur la pile, comme jusqu'à présent, nous aurions dû entrer une valeur constante pour sa taille qui aurait pu être soit trop grande, soit trop petite.
On notera

- l'utilisation de *delete* qui permet de libérer notre pointeur proprement à la fin de notre programme.
- L'utilisation du pointeur *val* avec la notation indicielle [], au lieu d'utiliser l'arithmétique des pointeurs.

Pointeurs sur fonctions

Lorsqu'un exécutable est chargé en mémoire, ses fonctions le sont évidemment aussi. Par voie de conséquence, elles ont donc une adresse, que C++ permet de pointer.

Si nous avons une fonction dont le prototype est le suivant :

```
int fct(double, double);
```

Un pointeur sur cette fonction sera déclaré de la façon suivante :

```
int (* fct_ptr)(double, double); // le pointeur s'appellera fct_ptr
```

On notera l'utilisation des parenthèses.

En effet, écrire

```
int * fct(double, double) //même chose que int* fct(double, double)
```

ne signifie pas du tout la même chose.

Pointeurs sur fonctions

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

double fct1(double x){
  return x*x;
}
double fct2(double x){
  return 2*x;
}
void apply(double *val, int n, double (*fct
     )(double)){
  for (int i=0; i<n; i++)
    val[i]= (*fct)(val[i]);
}

void aff_tab(double *val, int n){
  for (int i=0; i<n; i++)
    cout << i << ": " << val[i] << endl;
}
int main()
{
  double t[10] = {1,2,3,4,5,6,7,8,9,10};
  aff_tab(t, 10);
  apply(t, 10, fct1);
  cout << "Apres apply (elevation au carre)
       : " << endl;
  aff_tab(t, 10);
}
```

Listing 23: code19.cpp

On définit deux fonctions ayant même valeur de retour et même type d'argument, *fct1* et *fct2*.
La fonction *aff_tab* n'est là que pour aider et affiche un tableau de doubles donné en paramètre.
La nouveauté se situe dans la fonction *apply* qui va appliquer la fonction passée en paramètre sur chaque élément d'un tableau de n éléments, à l'aide d'un pointeur.
On notera que, pour appeler la fonction pointée, il faut la déréférencer, toujours à l'aide de l'opérateur *.
Cela permet d'écrire des fonctions génériques puissantes et se passer de l'écriture de code redondants!

En effet, on aurait pu appliquer des fonctions de la librairie *cmath* comme *cos* ou *sqrt*, sans réécrire pour chaque cas une boucle pour l'appliquer à chaque élément de ce tableau.

Chaînes de caractères en C

```
#include <iostream>

using namespace std;

int main()
{
  char *str = "Hello World";
  char str2[10] = {'C','o','u','c','o','u','\0'};
  int i = 0;

  cout << str << endl;
  while(str2[i++]) cout << str2[i-1];
  return 0;
}
```

Listing 24: code20.cpp

```
 ./code20
Hello World
Coucou
```

Les chaînes de caractères telles qu'elles sont représentées en C sont toujours valables en C++.

Bien que C++ offre d'autres mécanismes de plus haut niveau pour la manipulation de chaînes, nous allons en parler.

- D'une part car c'est un bon exercice sur les pointeurs.
- Pour comprendre ce qu'est une chaîne de caractères.
- Car elles sont encore utilisées en C++.

Chaînes de caractères en C

```
#include <iostream>

using namespace std;

int main()
{
  char *str = "Hello World";
  char str2[10] = {'C','o','u','c','o','u','\0'};
  int i = 0;

  cout << str << endl;
  while(str2[i++]) cout << str2[i-1];
  return 0;
}
```

Listing 24: code20.cpp

```
 ./code20
Hello World
Coucou
```

En C, les chaînes de caractères peuvent êtres vues comme des tableaux de *char*.

Il y a néanmoins une convention supplémentaire.

Une chaîne de caractères est donc un ensemble d'octets contigus se terminant par le caractère nul noté \0, ceci afin de donner une fin à la chaîne.

La notation *"Hello World"* définit donc un pointeur sur caractères vers une zone de la mémoire où est définie la constante *"Hello World"*. On récupère cette adresse dans le pointeur *str*.

Chaînes de caractères en C

```
#include <iostream>

using namespace std;

int main()
{
  char *str = "Hello World";
  char str2[10] = {'C','o','u','c','o','u','\0'};
  int i = 0;

  cout << str << endl;
  while(str2[i++]) cout << str2[i-1];
  return 0;
}
```

Listing 24: code20.cpp

```
 ./code20
Hello World
Coucou
```

On peut tout aussi bien définir une chaîne en déclarant un tableau et en l'initialisant avec la notation $\{\}$ comme dans l'exemple.

Pour les afficher, on utilise *cout*, ou une boucle qui parcourt le tableau tant que le caractère nul n'est pas rencontré.

Ces deux méthodes ne sont pas tout à fait équivalentes. En effet on peut modifier *str2[0]* mais pas *str[0]*.

Chaînes de caractères en C

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

int main(int nb, char *args[])
{
  cout << "Mon programme possede "
       << nb << " arguments" << endl;
  for (int i=0; i<nb; i++)
    cout << args[i] << endl;
  return 0;
}
```

Listing 25: code21.cpp

```
$ ./a.out salut tout le monde
Mon programme possède 5 arguments
./a.out
salut
tout
le
monde
```

On peut passer à un programme des valeurs, lorsqu'on l'appelle sur la ligne de commande, dans le terminal.
Le programme les reçoit comme des arguments de la fonction *main* (dont nous n'avons pas parlé jusqu'ici).

- Le premier argument de la fonction est conventionnellement un entier qui correspond au nombre d'arguments fournis au programme.
- Le deuxième paramètre est un peu plus complexe. Il s'agit d'un tableau de pointeurs sur *char* de taille non définie.

Chaque élément de ce tableau est donc un pointeur vers une chaîne de caractères qui existe quelque part en mémoire. On peut donc le balayer, comme dans la boucle de l'exemple. Chaque élément *args[i]* est donc de type *char** et peut être considéré comme une chaîne de caractères.

Conventionnellement, le premier élément *args[0]* est le nom du programme exécuté.

Structures

Les structures

- Jusqu'à présent, nous avons rencontré les tableaux qui étaient un regroupement de données de même type.
- Il peut être utile de grouper des données de types différents et de les regrouper dans une même entité.
- En C, il existait déjà un tel mécanisme connu sous le nom de structure.

  C++ conserve cette possibilité, tout en lui ajoutant de nombreuses possibilités.

  Ainsi, nous allons créer de nouveaux types de données, plus complexes, à partir des types que nous connaissons déjà.

Déclaration d'une structure

```cpp
#include <iostream>
#include "struct.hh"

using namespace std;

int main()
{
  Personne p1;

  p1.age = 35;
  p1.poids = 55.7;
  p1.taille = 160.5;
  cout << p1.age << " "
       << p1.poids << " "
       << p1.taille
       << endl;
  return 0;
}
```

Listing 26: code22.cpp

```cpp
#ifndef __STRUCT_HH__
#define __STRUCT_HH__

struct Personne
{
  int age;
  double poids;
  double taille;
};

#endif
```

Listing 27: struct.hh

Dans cet exemple nous commençons par déclarer la structure *Personne* dans un fichier *struct.hh*.

Ce n'est pas obligatoire, nous aurions pu déclarer cette structure dans le fichier contenant la fonction *main*, avant de l'utiliser, mais c'est une bonne habitude qui deviendra de plus en plus importante au fur et à mesure que nous avancerons dans ce cours.

Déclaration d'une structure

```cpp
#include <iostream>
#include "struct.hh"

using namespace std;

int main()
{
  Personne p1;

  p1.age = 35;
  p1.poids = 55.7;
  p1.taille = 160.5;
  cout << p1.age << " "
       << p1.poids << " "
       << p1.taille
       << endl;
  return 0;
}
```

Listing 26: code22.cpp

```cpp
#ifndef __STRUCT_HH__
#define __STRUCT_HH__

struct Personne
{
  int age;
  double poids;
  double taille;
};

#endif
```

Listing 27: struct.hh

Une structure se déclare grâce au mot-clé *struct* suivi du nom de la structure.

La structure Personne devient alors un type de données.
Ce type est le regroupement d'un entier et de deux doubles.
Dans la fonction *main*, après avoir inclus le fichier header au début du code, nous pouvons déclarer une variable de type *Personne*.

Cette variable s'appellera *p1*.

Déclaration d'une structure

```cpp
#include <iostream>
#include "struct.hh"

using namespace std;

int main()
{
  Personne p1;

  p1.age = 35;
  p1.poids = 55.7;
  p1.taille = 160.5;
  cout << p1.age << " "
       << p1.poids << " "
       << p1.taille
       << endl;
  return 0;
}
```

Listing 26: code22.cpp

```cpp
#ifndef __STRUCT_HH__
#define __STRUCT_HH__

struct Personne
{
  int age;
  double poids;
  double taille;
};

#endif
```

Listing 27: struct.hh

Les valeurs des différents types contenus dans la structure sont appelées des champs.

Pour accéder aux champs d'une variable dont le type est une structure, on utilisera l'opérateur point '.' suivi du nom du champ.
Ainsi *p1.age* est de type entier et on peut lui associer une valeur pour l'afficher ensuite.

Initialisation

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

struct Personne
{
  int age;
  double poids;
  double taille;
};

int main()
{
  Personne toto = {35, 78, 168.5};

  cout << toto.age << " "
       << toto.poids << " "
       << toto.taille << endl;
  return 0;
}
```

Listing 28: code23.cpp

Dans l'exemple précédent, nous initialisions la structure en attribuant une valeur à chacun de ses champs.

Dans certains cas, cela peut s'avérer long et peu pratique.

Une autre façon est d'initialiser les champs de la structure au moment de son instanciation à la manière d'un tableau, grâce à l'opérateur $\{\}$.

Structures contenant des tableaux

```
#include <iostream>

using namespace std;

struct NamedPoint
{
  double x;
  double y;
  char nom[10];
};

int main()
{
  NamedPoint pt = {0,0, "Origine"};

  cout << " nom " << pt.nom
       << " x : " << pt.x
       << " y : " << pt.y << endl;
}
```

Listing 29: code24.cpp

Une structure peut contenir un tableau. La taille de celui-ci sera réservée en mémoire quand une variable issue de cette structure sera créée.

On notera l'initialisation de la structure dans l'exemple ci-contre.

Tableaux de structures

```
#include <iostream>
using namespace std;
struct NamedPoint
{
  double x;
  double y;
  char nom[10];
};
int main()
{
  NamedPoint pt[3] = {{0,0, "Origine"},
                      {1,0, "X"},
                      {0,1, "Y"}};
  for (int i=0; i<3; i++)
    cout << " nom : " << pt[i].nom
         << ", x : " << pt[i].x
         << ", y : " << pt[i].y
         << endl;
}
```

Listing 30: code25.cpp

On peut également créer des tableaux contenant des instances d'une même structure.
Dans l'exemple, on déclare un tableau de 3 points, que l'on initialise. Chaque élément du tableau est initialisé avec la notation {} et lui-même est initialisé comme cela.
Puis on parcourt le tableau avec une boucle for pour en afficher chaque champ.
On fera attention au type de chaque élément :

- *pt* est un tableau de 3 *NamedPoint*,
- *pt[0]* est de type *NamedPoint*,
- *pt[0].nom* est un tableau de 10 *char*.

Structures imbriquées

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

struct Date
{
  int jour;
  int mois;
  int annee;
};
struct Valeur
{
  double x;
  Date date;
};
int main()
{
  Valeur v = {5.5, {2,4,2017}};
  cout << "A la date: " << v.date.jour
       << "/" << v.date.mois << "/"
       << v.date.annee << endl
       << "Valeur: " << v.x << endl;
}
```

Listing 31: code26.cpp

Créer une structure revient à créer un nouveau type. On peut donc utiliser ce nouveau type comme champ d'une autre structure comme dans l'exemple ci-contre.

Ici encore, on notera le type des objets sur lesquels on travaille:

- *v* est de type *Valeur*
- *v.x* est un *double*
- *v.date* est de type *Date*
- *v.date.jour* est un entier

Structures et fonctions

```cpp
#include <iostream>
#include <cmath> //pour utiliser pow
using namespace std;

struct Point
{
  double x;
  double y;
};
double myNorme(Point p, Point q)
{
  return sqrt(pow(p.x - q.x, 2)
         + pow(p.y - q.y, 2));
}
int main()
{
  Point a = {2, 2};
  Point b = {3, 3};
  cout << "norme: " << myNorme(a,b)
       << endl;
}
```

Listing 32: code27.cpp

On crée une fonction *myNorme* calculant la norme euclidienne de deux points de $\mathbb{R}^2$ définissant un vecteur.

On remarque au passage l'emploi de fonctions de la librairie *cmath*.

Les deux *Point* a et b sont passés à la fonction par copie.
myNorme reçoit donc une copie des points et non les points eux-mêmes.

C'est le passage par valeur. Si on modifie les valeurs des champs dans myNorme, ceux-ci ne sont pas modifiés à l'extérieur de la fonction.

Structures et fonctions

```
#include <iostream>
#include <cmath> //pour utiliser pow
using namespace std;

struct Point
{
  double x;
  double y;
};
double myNorme(Point & p, Point & q)
{
  return sqrt(pow(p.x - q.x, 2) +
              pow(p.y - q.y, 2));
}
int main()
{
  Point a = {2, 2};
  Point b = {3, 3};
  cout << "norme: " << myNorme(a,b)
       << endl;
}
```

Listing 33: code27_V2.cpp

Cette fois-ci le passage se fait par référence. Rien ne change à part l'entête de la fonction.

Ici, les valeurs des champs des paramètres ne changent pas (on aurait pu (dû!) les déclarer *const*).

Structures et fonctions

```
#include <iostream>
#include <cmath> //pour utiliser pow
using namespace std;
struct Point
{
  double x;
  double y;
};
double myNorme(Point* p, Point* q)
{
  return sqrt(pow(p->x - q->x, 2) +
              pow(p->y - q->y, 2));
}

int main()
{
  Point a = {2, 2};
  Point b = {3, 3};
  cout << "norme: "
       << myNorme(&a, &b) << endl;
}
```

Listing 34: code27_V3.cpp

On passe maintenant des pointeurs sur *Point* à notre fonction. On voit que l'appel de la fonction s'en trouve modifié: on ne passe plus les *Point* eux-mêmes mais leur adresse (obtenue grâce à l'opérateur &).

Le corps de la fonction aussi a changé.

Utiliser l'opérateur point '.' n'a plus de sens si on travaille sur un pointeur. Un pointeur n'est pas du même type qu'une structure! C'est une adresse!

Structures et fonctions

```
#include <iostream>
#include <cmath> //pour utiliser pow
using namespace std;
struct Point
{
  double x;
  double y;
};
double myNorme(Point* p, Point* q)
{
  return sqrt(pow(p->x - q->x, 2) +
              pow(p->y - q->y, 2));
}

int main()
{
  Point a = {2, 2};
  Point b = {3, 3};
  cout << "norme: "
       << myNorme(&a, &b) << endl;
}
```

Listing 34: code27_V3.cpp

Si nous avions voulu utiliser l'opérateur point '.' quand même, nous aurions pu, au prix d'une écriture un peu lourde.

En effet, si on déréférence le pointeur sur *Point*, on obtient un objet de type *Point*, et ainsi le traiter comme tel ...

C++ introduit une facilité syntaxique pour éviter cela: l'opérateur flèche '->'

Ainsi,
$(*p).x$ est équivalent à $p\text{->}x$

Fonctions membres

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

struct Point{
  double x; double y;
  void initialise(double, double);
  void deplace(double dx, double dy);
  void affiche();
};
void Point::initialise(double abs,
    double ord){ x = abs; y = ord;}
void Point::deplace(double dx, double
    dy){ x+= dx; y += dy;}
void Point::affiche(){
  cout << "x : " << x << " y : "
       << y << endl;}

int main(){
  Point p;
  p.initialise(1,4); p.deplace(0.1, 2);
  p.affiche(); }
```

Listing 35: code28.cpp

On ne se contente plus de données dans la structure. On ajoute aussi des fonctions:

- Une fonction *initialise* qui prend deux paramètres qui seront destinés à initialiser les coordonnées de notre *Point*.
- Une fonction *deplace*, qui prend deux paramètres et qui modifiera les coordonnées en fonction.
- Une fonction *affiche* qui provoquera un affichage de notre point.

Fonctions membres

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

struct Point{
  double x; double y;
  void initialise(double, double);
  void deplace(double dx, double dy);
  void affiche();
};
void Point::initialise(double abs,
    double ord){ x = abs; y = ord;}
void Point::deplace(double dx, double
    dy){ x+= dx; y += dy;}
void Point::affiche(){
  cout << "x : " << x << " y : "
       << y << endl;}

int main(){
  Point p;
  p.initialise(1,4); p.deplace(0.1, 2);
  p.affiche(); }
```

Listing 34: code28.cpp

Les fonctions *initialise*, *deplace* et *affiche* sont des fonctions membres de la structure *Point*.

On les déclare dans la structure.

On les définit à l'extérieur de celle-ci MAIS le nom de la structure doit apparaître, suivi de :: appelé **opérateur de résolution de portée**.

En effet, comment connaître x et y dans la fonction si le compilateur ne sait pas qu'elles appartiennent à *Point*?

Un code ordonné

```
#ifndef __POINT_HH__
#define __POINT_HH__

#include <iostream>
using namespace std;

struct Point{
  double x;
  double y;
  void initialise(double,
       double);
  void deplace(double dx,
       double dy);
  void affiche();
};

#endif
```

Listing 35: Point.hh

```
#include "Point.hh"
void Point::initialise(double abs,
     double ord){
  x = abs;
  y = ord;
}
void Point::deplace(double dx,
     double dy){
  x += dx;
  y += dy;
}
void Point::affiche(){
  cout << "x : " << x
       << " y : " << y << endl;
}
```

Listing 36: Point.cpp

On sépare la déclaration de la structure, de sa définition. On crée un fichier *NomStructure.hh* (par ex.) destiné à la déclaration. Et le fichier *NomStructure.cpp* contiendra le code définissant les fonctions membres.

Un code ordonné

```
#include <iostream>
#include "Point.hh"

int main(){
  Point x;

  x.initialise(1,1);
  x.deplace(0.1, -0.1);
  x.affiche();
}
```

Listing 37: ex_struct_fct_membres.cpp

Voilà à quoi va ressembler notre fichier contenant la fonction *main* désormais.
On inclut le fichier header *Point.hh*.
Ainsi le compilateur connaîtra le type *Point*.
Pour la compilation, on compile en même temps les deux fichiers .cpp pour créer notre exécutable.
Il y a d'autres méthodes de compilation, à l'aide de **make** par exemple.

```
g++ ex_struct_fct_membres.cpp Point.cpp
  ./a.out
  x : 1.1 y : 0.9
```