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

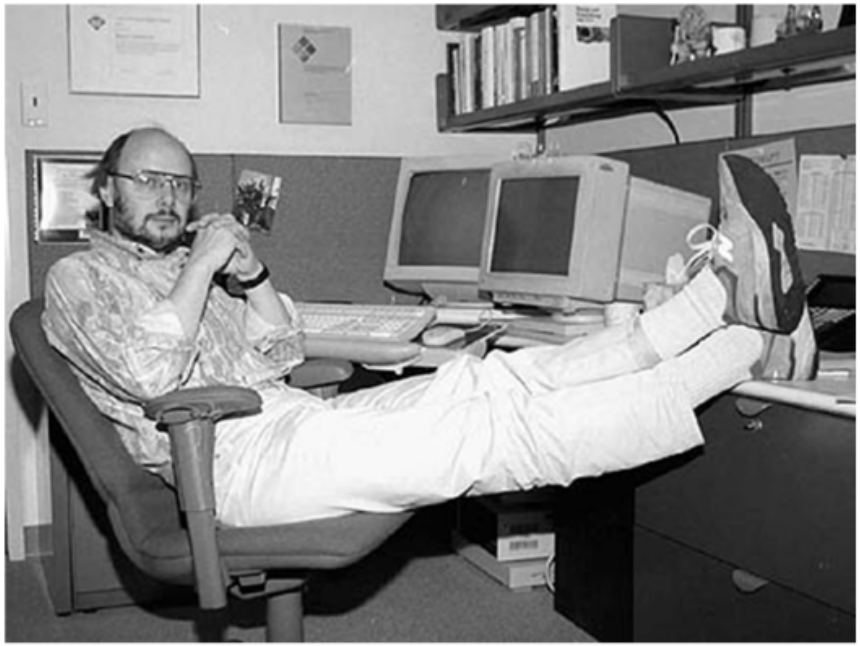
Petite histoire du C/C++

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.

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 11: 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 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 - 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 13: 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 - 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 14: 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 15: 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 16: 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 17: 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 17: 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 18: 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 19: 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 -
    utilisation déconseillée!
```

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, 3, 4, 7, 9, 13};

  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. C'est un nombre hexadécimal et il se définit à partir d'un type (*int*, *float*, ...).

Exemples de déclarations de pointeurs:

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

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 &

```cpp
#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 = 0x7ffde3edb19c
*ptr = 42
```

Pointeurs - Les opérateurs * et &

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

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

  ptr = &i;
  cout << "ptr =  " << ptr << endl;
  cout << "*ptr = " << *ptr << endl;
  i += 3;
  cout << "i = " << i << ", *ptr = " << *ptr
       << endl;
  *ptr = 122;
  cout << "*ptr = " << *ptr << ", i = " << i
       << endl;
  return 0;
}
```

Listing 22: code17V2.cpp

```
$ ./a.out
ptr = 0x7ffde3edb19c
*ptr = 42
i = 45, *ptr = 45
*ptr = 122, i = 122
```

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) ≠ *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];
```

$\Rightarrow$ On peut donc utiliser *new* pour manipuler des tableaux.

Les opérateurs new et delete

- Attention car 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 23: 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 24: 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++

```cpp
#include <iostream>

using namespace std;

int main()
{
  char *str = "Hello World"; // char*
  char str2[10] = {'C','o','u','c','o','u','\0'};
        // char[10]
  string str3 = "Bonjour"; // string

  cout << str << endl;
  cout << str2 << endl;
  cout << str3 << endl;

  return 0;
}
```

Listing 25: code20.cpp

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

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

Il existe trois manières de manipuler les chaînes de caractères en C++, comme on peut le voir dans l'exemple de gauche.

- En utilisant *char**, comme en C (attention, ne nécessite pas *new*!)
- Avec un tableau de char, d'une taille donnée
- En utilisant le type *string* (plus simple)

Chaînes de caractères en C++

```cpp
#include <iostream>

using namespace std;

int main()
{
  char *str = "Hello World"; // char*
  char str2[10] = {'C','o','u','c','o','u','\0'};
        // char[10]
  string str3 = "Bonjour"; // string

  cout << str << endl;
  cout << str2 << endl;
  cout << str3 << endl;

  return 0;
}
```

Listing 25: code20.cpp

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

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++

```cpp
#include <iostream>

using namespace std;

int main()
{
  char *str = "Hello World"; // char*
  char str2[10] = {'C','o','u','c','o','u','\0'};
        // char[10]
  string str3 = "Bonjour"; // string

  cout << str << endl;
  cout << str2 << endl;
  cout << str3 << endl;

  return 0;
}
```

Listing 25: code20.cpp

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

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*.

Les méthodes ne sont pas tout à fait équivalentes. En effet on peut modifier *str2[0]* mais pas *str[0]*. Et il est possible (mais compliqué) de modifier le premier caractère de *str3*.

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 26: 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 en C++

- 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 ce mécanisme, 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 27: code22.cpp

```cpp
#ifndef __STRUCT_HH__
#define __STRUCT_HH__

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

#endif
```

Listing 28: 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 (directives dans le header)

```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 27: code22.cpp

```cpp
#ifndef __STRUCT_HH__
#define __STRUCT_HH__

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

#endif
```

Listing 28: struct.hh

On aurait pu se passer des 3 lignes commençant par *#ifndef*, *#define* et *#endif* dans *struct.hh* car le code est très simple ici. Mais dans le cas général, il est fortement conseillé de les inclure!

Si on les inclut, les 3 lignes sont indispensables; si l'une manque, ça donne une erreur de compilation!

En général, le nom qui suit *#ifndef* est celui du fichier d'entête en majuscules, précédé et terminé par "__" et dans lequel on remplace "." par "_".

Les noms après *#ifndef* et *#define* doivent être identiques.

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 27: code22.cpp

```cpp
#ifndef __STRUCT_HH__
#define __STRUCT_HH__

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

#endif
```

Listing 28: 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 27: code22.cpp

```cpp
#ifndef __STRUCT_HH__
#define __STRUCT_HH__

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

#endif
```

Listing 28: 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

```
#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 29: 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 30: 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 31: 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 32: 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

```
#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.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 34: 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 35: 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 35: 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 36: 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 36: 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 37: 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 38: 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 39: 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
```

cmath

- La commande *#include<cmath>* permet l'utilisation de fonctions mathématiques dans votre code C++
- Exemples:
  - *sqrt*, *exp*, *log*, *sin*, *cos*, ...
  - *pow* permet l'élévation à la puissance

```
#include<cmath>
.....
double a = 2.1;
cout << pow(a, 6) << endl; // affiche la valeur de a^6
....
```

  - La constante π s'obtient par M_PI
- Enfin, par exemple, *sqrt(-1)* correspond à *NaN* (Not a Number) sans erreur de compilation, ni d'exécution! C'est au programmeur à faire attention.

Classes et Objets

Introduction

Nous entrons donc maintenant dans la partie **Programmation Orientée Objet** de ce cours.

Nous avons vu les structures: ce sont des types définis par l'utilisateur qui contiennent à la fois des données, mais aussi des fonctions membres.

En fait, en C++, ces structures sont un cas particulier d'un mécanisme plus général, les Classes.

Introduction

Pourquoi ne pas simplement travailler sur des structures ?

Les structures existent en C++ par souci de compatibilité avec C. Les Classes peuvent être considérées comme les structures mais leur comportement par défaut vis-à-vis de l'encapsulation est différent. De plus, le terme 'structure' ne renvoie pas à la terminologie Objet aussi bien que le terme 'classe', plus commun.

L'encapsulation fait partie intégrante de la POO. Elle permet de masquer certains membres et fonctions membres au monde extérieur.

Dans l'idéal, on ne devrait jamais accéder aux données d'une classe, mais agir sur ses méthodes (fonctions membres).

Exemple précédent: structure *Point*

```
#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 37: 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 38: Point.cpp

Déclaration

```
#ifndef __POINT2D_V1_HH__
#define __POINT2D_V1_HH__

class Point2D
{
private:
  float _x;
  float _y;

public:
  void initialise(float, float);
  void affiche();
};

#endif
```

Listing 40: Point2D_V1.hh

Une classe se déclare comme une structure.
Les étiquettes *private* et *public* sont utiles pour définir le niveau de visibilité des membres à l'extérieur de la classe.

Les membres déclarés *private* ne sont pas accessibles par des objets d'un autre type.
Les membres dits publics sont accessibles partout.
Ici, on ne pourra donc plus écrire, dans la fonction *main*:

```
Point2D p;
p._x = 5; //erreur-membre privé!
```

Par convention, les noms des membres privés commencent par '_' (tiret du bas).

private/public

```
#ifndef __POINT2D_V1_HH__
#define __POINT2D_V1_HH__

class Point2D
{
private:
  float _x;
  float _y;

public:
  void initialise(float, float);
  void affiche();
};

#endif
```

Listing 40: Point2D_V1.hh

Par défaut, si on ne spécifie rien, les membres d'une classe ont le statut privé. C'est le contraire pour une structure.

Dans une structure, on peut utiliser le mot-clé *private* pour rendre privés des attributs ou des fonctions.

Constructeurs

```
#ifndef __POINT2D_V2_HH__
#define __POINT2D_V2_HH__

class Point2D
{
private:
  float _x;
  float _y;

public:
  Point2D(float,float);
  void affiche();
};

#endif
```

Listing 41: Point2D_V2.hh

Au lieu d'utiliser une fonction *initialise*, il existe un autre mécanisme en C++: les **constructeurs**.
Les fonctions comme *initialise* ont, en effet, des inconvénients:

- On ne peut pas forcer l'utilisateur de la classe à l'appeler.
- Elle n'est pas appelée au moment de la création de l'objet ce qui peut être ennuyeux si des opérations doivent être effectuées dès le début de la vie de notre instance.

```
#include<iostream>
#include "Point2D_V2.hh"
using namespace::std;

Point2D::Point2D(float abs, float ord){
  _x = abs, _y = ord;
}
void Point2D::affiche(){
  cout << _x << " : " << _y << endl; }
```

Listing 42: Point2D_V2.cpp

Constructeurs

```
#ifndef __POINT2D_V2_HH__
#define __POINT2D_V2_HH__

class Point2D
{
private:
  float _x;
  float _y;

public:
  Point2D(float,float);
  void affiche();
};

#endif
```

Listing 41: Point2D_V2.hh

Un constructeur est une fonction membre ne renvoyant rien, qui porte le même nom que la classe.
Elle est appelée lors de la déclaration d'un objet.
Si nous voulons déclarer maintenant un objet de type *Point2D*, nous allons fournir les deux coordonnées.

Point2D p(3, 4); par exemple, déclare l'objet p de type *Point2D* et appelle dans la foulée le constructeur avec les paramètres 3 et 4.

```
#include<iostream>
#include "Point2D_V2.hh"
using namespace::std;

Point2D::Point2D(float abs, float ord){
  _x = abs, _y = ord;
}
void Point2D::affiche(){
  cout << _x << " : " << _y << endl; }
```

Listing 42: Point2D_V2.cpp

Constructeurs

```
#ifndef __POINT2D_V2_HH__
#define __POINT2D_V2_HH__

class Point2D
{
private:
  float _x;
  float _y;

public:
  Point2D(float,float);
  void affiche();
};

#endif
```

Listing 41: Point2D_V2.hh

Il peut y avoir autant de constructeurs que l'on veut, en fonction des besoins.

Il faudra les distinguer les uns des autres par des paramètres distincts.
On peut également, si cela est nécessaire, placer un constructeur dans la partie *private* de notre classe. Ainsi, son utilisation sera bloquée à l'extérieur de la classe.

```
#include<iostream>
#include "Point2D_V2.hh"
using namespace::std;

Point2D::Point2D(float abs, float ord){
  _x = abs, _y = ord;
}
void Point2D::affiche(){
  cout << _x << " : " << _y << endl; }
```

Listing 42: Point2D_V2.cpp

Constructeur - autre syntaxe

```
#include<iostream>
#include "Point2D_V2.hh"
using namespace::std;

Point2D::Point2D(float abs, float ord) : _x(abs), _y(ord) {}

void Point2D::affiche() {
  cout << _x << " : " << _y << endl;
}
```

Listing 43: Point2D_V3.cpp

On peut utiliser une syntaxe plus courte pour définir le constructeur (elle est équivalente à la syntaxe précédente). Le constructeur initialise les champs $\_x$ et $\_y$ de l'instance en lui passant les valeurs entre parenthèse.

Il est nécessaire d'ajouter les accolades, même si le corps est vide.

Destructeur

```
#ifndef __POINT2D_V3_HH__
#define __POINT2D_V3_HH__

class Point2D
{
private:
  float _x, _y;

public:
  Point2D(float,float);
  ~Point2D();
};

#endif
```

Listing 44: Point2D_V3.hh

La classe est déclarée dans un fichier header .hh

On voit les deux données membres déclarées *private*, ainsi qu'un constructeur de la classe ayant deux paramètres de type float.

Le destructeur de la classe porte le même nom que celle-ci précédé du caractère ~ (tilde).

Il est appelé **lors de la destruction de l'instance courante**. Son objectif est de permettre au programmeur de libérer de l'espace mémoire si nécessaire.
De même si des fichiers ont été ouverts ou des connexions (vers des bases de données par exemple) ont été initiées durant la vie de l'instance.

Accesseurs & mutateurs

```
#ifndef __MYCLASS_HH__
#define __MYCLASS_HH__

class MyClass{

private:
 double _v;

public:
 MyClass(double);
 ~MyClass();
 double getV() const;
 void setV(double d);
};

#endif
```

Listing 45: MyClass.hh

```
#include "MyClass.hh"

MyClass::MyClass(double v):
    _v(v) {}

MyClass::~MyClass() {}

double MyClass::getV() const {
  return _v;
}

void MyClass::setV(double v) {
  _v = v;
}
```

Listing 46: MyClass.cpp

On appelle accesseurs et mutateurs (ou modificateurs) des fonctions permettant l'accès à des attributs privés d'une classe.

On les appelle aussi getter et setter en Anglais.

Ces fonctions sont quasi systématiquement définies lors de la création d'une nouvelle classe.

Accesseurs & mutateurs

```
#ifndef __MYCLASS_HH__
#define __MYCLASS_HH__

class MyClass{

private:
 double _v;

public:
 MyClass(double);
 ~MyClass();
 double getV() const;
 void setV(double d);
};

#endif
```

Listing 45: MyClass.hh

```
#include "MyClass.hh"

MyClass::MyClass(double v):
    _v(v) {}

MyClass::~MyClass() {}

double MyClass::getV() const {
  return _v;
}

void MyClass::setV(double v) {
  _v = v;
}
```

Listing 46: MyClass.cpp

Leur nom correspond généralement au nom de l'attribut précédé de *get* pour les getters et *set* pour les setters.

En général, on déclare les fonctions getters comme étant *const*, comme dans l'exemple ci-contre.

Accesseurs & mutateurs

```
#ifndef __MYCLASS_HH__
#define __MYCLASS_HH__

class MyClass{

private:
 double _v;

public:
 MyClass(double);
 ~MyClass();
 double getV() const;
 void setV(double d);
};

#endif
```

Listing 45: MyClass.hh

```
#include "MyClass.hh"

MyClass::MyClass(double v):
    _v(v) {}

MyClass::~MyClass() {}

double MyClass::getV() const {
  return _v;
}

void MyClass::setV(double v) {
  _v = v;
}
```

Listing 46: MyClass.cpp

En effet, les fonctions membres déclarées comme *const* permettent à l'utilisateur de cette méthode de savoir que cette fonction ne modifiera pas les champs de l'objet.

Ce sont aussi les seules fonctions que l'on peut appeler sur des objets constants.

Accesseurs & mutateurs

```
#ifndef __MYCLASS_HH__
#define __MYCLASS_HH__

class MyClass{

private:
 double _v;

public:
 MyClass(double);
 ~MyClass();
 double getV() const;
 void setV(double d);
};

#endif
```

Listing 45: MyClass.hh

```
#include "MyClass.hh"

MyClass::MyClass(double v):
    _v(v) {}

MyClass::~MyClass() {}

double MyClass::getV() const {
  return _v;
}

void MyClass::setV(double v) {
  _v = v;
}
```

Listing 46: MyClass.cpp

On pourra noter également la syntaxe du constructeur qui initialise le champ $\_v$ de l'instance en lui passant la valeur entre parenthèse, comme dans le cas des objets imbriqués.

Il est néanmoins nécessaire d'ajouter les accolades, même si le corps est vide.

Accesseurs & mutateurs

```
#ifndef __MYCLASS_HH__
#define __MYCLASS_HH__

class MyClass{

private:
 double _v;

public:
 MyClass(double);
 ~MyClass();
 double getV() const;
 void setV(double d);
};

#endif
```

Listing 45: MyClass.hh

```
#include "MyClass.hh"

MyClass::MyClass(double v):
      _v(v) {}

MyClass::~MyClass() {}

double MyClass::getV() const
      {
  return _v;
}

void MyClass::setV(double v)
      {
  _v = v;
}
```

Listing 46: MyClass.cpp

```
#include <iostream>
#include "MyClass.hh"
using namespace std;

int main()
{
  MyClass o(3);

  o.setV(4);
  cout << o.getV()
       << endl;
  return 0;
}
```

Listing 47: main_MyClass.cpp

```
g++ main_MyClass.cpp MyClass.cpp

./a.out
4
```

Le mot-clé this

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

class theClass{
public:
  theClass();
  ~theClass();
};

theClass::theClass() {
  cout << "C: " << this << endl;
}
theClass::~theClass() {
  cout << "D: " << this << endl;
}

int main() {
  theClass o, op;
}
```

Listing 48: main_this.cpp

Chaque fonction membre d'un objet peut recevoir une information supplémentaire.

Elle permet de faire le lien entre les corps des fonctions membres et l'instance courante de la classe.

Il s'agit de *this*.

C'est un pointeur transmis à toutes les fonctions membres et qui pointe vers l'instance courante.

```
 ./a.out
C: 0xffffcbff
C: 0xffffcbfe
D: 0xffffcbfe
D: 0xffffcbff
```

Membres statiques

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

class JeMeCompte
{
public:
  static int compteur; // Déclaration
  JeMeCompte();
  ~JeMeCompte();
};

int JeMeCompte::compteur = 0; // initialisation
                              // de la variable static
JeMeCompte::JeMeCompte(){
  cout << "C: " << compteur++ << endl; }

JeMeCompte::~JeMeCompte(){
  cout << "D: " << compteur++ << endl; }

int main(){
  JeMeCompte nbr;
  cout << "compteur: "<< JeMeCompte::compteur
       << endl; }
```

Listing 49: main_static.cpp

Jusqu'à présent, une donnée membre d'une classe était liée à chaque instance de la classe.

Il est également possible de lier une donnée à la classe elle-même, indépendamment d'éventuelles instances.

Et la valeur de cette donnée est partagée par toutes les instances et par la classe elle-même.

On utilise pour cela le mot clé *static* devant la ou les variable(s) désignée(s) pour ce rôle.

Membres statiques

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

class JeMeCompte
{
public:
  static int compteur; // Déclaration
  JeMeCompte();
  ~JeMeCompte();
};

int JeMeCompte::compteur = 0; // initialisation
                              // de la variable static
JeMeCompte::JeMeCompte(){
  cout << "C: " << compteur++ << endl; }

JeMeCompte::~JeMeCompte(){
  cout << "D: " << compteur++ << endl; }

int main(){
  JeMeCompte nbr;
  cout << "compteur: "<< JeMeCompte::compteur
       << endl; }
```

Listing 49: main_static.cpp

Il reste la question de l'initialisation de cette variable.

Celle-ci ne peut pas être initialisée par un constructeur: en effet celui-ci est intimement lié au cycle de vie d'un objet et donc à une instance elle-même.

On ne peut pas non plus l'initialiser dans la déclaration de la classe. En effet, cela risquerait, en cas de compilation séparée, de réserver plusieurs emplacements en mémoire pour cette donnée.

Membres statiques

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

class JeMeCompte
{
public:
  static int compteur; // Déclaration
  JeMeCompte();
  ~JeMeCompte();
};

int JeMeCompte::compteur = 0; // initialisation
                              // de la variable static
JeMeCompte::JeMeCompte(){
  cout << "C: " << compteur++ << endl; }

JeMeCompte::~JeMeCompte(){
  cout << "D: " << compteur++ << endl; }

int main(){
  JeMeCompte nbr;
  cout << "compteur: "<< JeMeCompte::compteur
       << endl; }
```

Listing 49: main_static.cpp

Il reste donc à l'initialiser hors de la déclaration, au côté de l'initialisation des fonctions membres.

La syntaxe est alors telle que dans l'exemple ci-contre.

On notera l'utilisation de l'opérateur de résolution de portée '::' pour signifier qu'il s'agit bien d'un membre de la classe.

Fonctions membres statiques

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

class JeMeCompte{
public:
  static int compteur;
  JeMeCompte();
  ~JeMeCompte();
  static void AfficheCompteur();
};

int JeMeCompte::compteur = 0;

JeMeCompte::JeMeCompte(){
  cout << "C: " << compteur++ << endl; }
JeMeCompte::~JeMeCompte(){
  cout << "D: " << compteur++ << endl; }
void JeMeCompte::AfficheCompteur(){
  cout << compteur << " objets" << endl;}

int main(){
  JeMeCompte nbr;
  JeMeCompte::AfficheCompteur();  }
```

Listing 50: main_static_V2.cpp

Il est également possible de déclarer des fonctions membres statiques.

Comme les données, elles ne dépendent plus d'une instance mais de la classe elle-même. On peut donc les appeler en dehors de tout objet, comme dans l'exemple ci-contre.

Pour signaler que l'on utilise la fonction de la classe *JeMeCompte*, on utilise l'opérateur ::

Fonction amie

```
#ifndef __MYCLASS_V2_HH__
#define __MYCLASS_V2_HH__

class MyClass{

private:
  double _v;

public:
  MyClass(double _v);
  ~MyClass();
  double getV() const;
  void setV(double d);
  friend bool isEqual(MyClass
        o, MyClass p);
};

#endif
```

Listing 51: MyClass_V2.hh

```
#include "MyClass_V2.hh"
#include<cmath>
MyClass::MyClass(double v): _v(v)
     {}

MyClass::~MyClass() {}

double MyClass::getV() const {
  return _v;
}
void MyClass::setV(double v) {
  _v = v;
}
bool isEqual(MyClass o, MyClass p)
      {
  return fabs(o._v - p._v) < 1e-9;
        }
```

Listing 52: MyClass_V2.cpp

```
#include <iostream>
#include "MyClass_V2.hh"
using namespace std;
int main()
{
  MyClass o(3);
  o.setV(4);
  cout<< o.getV() << endl;
  cout << boolalpha
       << isEqual(o,o) << endl;
}
```

Listing 53: main_MyClass_V2.cpp

Il existe un autre moyen pour une fonction d'accéder aux membres privés d'une classe. Il s'agit d'une fonction **amie**.

Celle-ci se déclare dans le corps de la classe, précédée du mot clé *friend*.

Elle ne fait pas partie de la classe et ne reçoit donc le pointeur *this* d'aucune instance.

Elle accède, par contre, aux données membres sans l'utilisation des getters ou des setters.

Fonction amie

```
#ifndef __MYCLASS_V2_HH__
#define __MYCLASS_V2_HH__

class MyClass{

private:
  double _v;

public:
  MyClass(double _v);
  ~MyClass();
  double getV() const;
  void setV(double d);
  friend bool isEqual(MyClass
        o, MyClass p);
};

#endif
```

Listing 51: MyClass_V2.hh

```
#include "MyClass_V2.hh"
#include<cmath>
MyClass::MyClass(double v): _v(v)
      {}

MyClass::~MyClass() {}

double MyClass::getV() const {
  return _v;
}
void MyClass::setV(double v) {
  _v = v;
}
bool isEqual(MyClass o, MyClass p)
       {
  return fabs(o._v - p._v) < 1e-9;
         }
```

Listing 52: MyClass_V2.cpp

```
#include <iostream>
#include "MyClass_V2.hh"
using namespace std;
int main()
{
  MyClass o(3);
  o.setV(4);
  cout<< o.getV() << endl;
  cout << boolalpha
       << isEqual(o,o) << endl;
}
```

Listing 53: main_MyClass_V2.cpp

Son utilisation se justifie quand on recherche un code optimisé. Car l'utilisation des getters et des setters génère du code moins optimisé.

On réservera donc son usage pour des cas particuliers de recherche de performance.

La plupart du temps et, pour un code plus lisible, on utilisera les modificateurs et accesseurs.

Fonction amie

```
#ifndef __MYCLASS_V2_HH__
#define __MYCLASS_V2_HH__

class MyClass{

private:
  double _v;

public:
  MyClass(double _v);
  ~MyClass();
  double getV() const;
  void setV(double d);
  friend bool isEqual(MyClass
        o, MyClass p);
};

#endif
```

Listing 51: MyClass_V2.hh

```
#include "MyClass_V2.hh"
#include<cmath>
MyClass::MyClass(double v): _v(v)
     {}

MyClass::~MyClass() {}

double MyClass::getV() const {
  return _v;
}
void MyClass::setV(double v) {
  _v = v;
}
bool isEqual(MyClass o, MyClass p)
      {
  return fabs(o._v - p._v) < 1e-9;
        }
```

Listing 52: MyClass_V2.cpp

```
#include <iostream>
#include "MyClass_V2.hh"
using namespace std;
int main()
{
  MyClass o(3);
  o.setV(4);
  cout<< o.getV() << endl;
  cout << boolalpha
       << isEqual(o,o) << endl;
}
```

Listing 53: main_MyClass_V2.cpp

On notera également l'utilisation de *boolalpha* qui est un modificateur de *cout*.

Il permet l'affichage de booléens de manière plus lisible.

Ici, on obtient true/false à la place de 1/0.

Objets et dynamique

```
#ifndef __POINTND_HH__
#define __POINTND_HH__

class PointND {
private:
  int _n;
  double *_vals;

public:
  PointND(int);
};

#endif
```

Listing 54: PointND.hh

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

using namespace std;

PointND::PointND(int n) {
  _n = n;
  _vals = new double[n];

  for(int i=0;i<_n;i++)
    _vals[i] = 0;

  cout <<"Constructeur: "
       << "n = " << n
       << endl;
}
```

Listing 55: PointND.cpp

Dans une classe, on peut avoir des pointeurs comme attributs. Dans cet exemple, l'objet contient un pointeur sur *double* qui contiendra toutes les coordonnées du n-point.

La dimension est un entier, membre privé de notre classe.

Le constructeur initialise les deux variables.

Pour le tableau de valeurs, pas d'autre choix que d'utiliser *new* et donc une allocation dynamique. En effet, le nombre de valeurs étant variable, on ne peut pas utiliser un tableau dont la taille serait fixée à l'avance.

Objets et dynamique: Le constructeur

```
#ifndef __POINTND_HH__
#define __POINTND_HH__

class PointND {
private:
  int _n;
  double *_vals;

public:
  PointND(int);
};

#endif
```

Listing 54: PointND.hh

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

using namespace std;

PointND::PointND(int n) {
  _n = n;
  _vals = new double[n];

  for(int i=0;i<_n;i++)
    _vals[i] = 0;

  cout <<"Constructeur: "
       << "n = " << n
       << endl;
}
```

Listing 55: PointND.cpp

Celui-ci initialise $\_n$ avec la valeur entière passée en paramètre. Il alloue aussi la place en mémoire dynamiquement pour les n valeurs du n-point.

Les $\_n$ valeurs sont initialisées à 0.

Objets et dynamique: Le destructeur

```
#ifndef __POINTND_V1_HH__
#define __POINTND_V1_HH__

class PointND {
private:
  int _n;
  double *_vals;

public:
  PointND(int);
  ~PointND();
};
#endif
```

Listing 56: PointND_V1.hh

```
#include <iostream>
#include "PointND_V1.hh"
using namespace std;

PointND::PointND(int n) {
  _n = n;
  _vals = new double[n];

  for(int i=0;i<_n;i++)
    _vals[i] = 0;

  cout << "Constructeur:"
       << " n = " << n
       << endl;
}

PointND::~PointND(){
  delete [] _vals;
  cout << "Appel
      Destructeur"
       << endl; }
```

Listing 57: PointND_V1.cpp

Celui-ci contient un affichage mais ce n'est pas le but premier d'un destructeur.

Celui-ci a pour but de détruire "proprement" l'objet. Il est chargé de libérer la mémoire, de clore des fichiers ou des connexions etc.

Ici, de la mémoire a été allouée dynamiquement par un appel à *new* dans le constructeur. Quand sera-t-elle libérée? C'est au destructeur de se charger de cette tâche.

On devra donc utiliser l'opérateur *delete* sur le tableau *_vals* afin de rendre au système cet espace qu'il pourra réutiliser pour d'autres allocations par exemple.

Objets et dynamique: Le constructeur par copie

```
#ifndef __POINTND_V2_HH__
#define __POINTND_V2_HH__

class PointND {
private:
  int _n;
  double *_vals;

public:
  PointND(int);
  PointND(const PointND&);
  ~PointND();
};
#endif
```

Listing 58: PointND_V2.hh

```
#include <iostream>
#include "PointND_V2.hh"

using namespace std;

PointND::PointND(int n) {
  _n = n;
  _vals = new double[n];
  for(int i=0;i<_n;i++)
    _vals[i] = 0;

  cout << "Constructeur: "
       << "n = " << n << endl;
}

PointND::PointND(const PointND&
    pnd){
  _n = pnd._n;
  _vals = new double[_n];
  for(int i=0;i<_n;i++)
    _vals[i] = pnd._vals[i];

  cout << "Constructeur par copie
       : n=" << _n << endl;
}

PointND::~PointND(){
  delete [] _vals;
  cout << "Appel Destructeur" <<
       endl; }
```

Listing 59: PointND_V2.cpp

Ce constructeur recopie les valeurs de l'objet passé en paramètre par référence. *const* permet d'assurer que celui-ci ne sera pas modifié, ce ne serait pas une copie sinon ···
Que se passe-t-il pour la copie des valeurs?

Ici on alloue la mémoire suffisante pour un tableau de *_n* doubles. On copie les valeurs des éléments de *pnd._vals* dans le tableau *_vals*.

Il ne faut pas faire une copie de pointeurs car le résultat ne serait pas celui espéré, notamment si un objet est modifié ou détruit avant l'autre.

Objets membres

```
#ifndef __CERCLE_HH__
#define __CERCLE_HH__

class Point2D{
  float _x, _y;
public:
  Point2D(float, float);
  Point2D();
};

class Cercle {
private:
  Point2D _centre;
  float _rayon;
};

#endif
```

Listing 60: Cercle.hh

```
#include<iostream>
#include "Cercle.hh"
using namespace::std;

Point2D::Point2D(
      float x, float y
      ){
  cout << "
       Constructeur
       Point2D("
       << x << ";"
       << y << ")"
       << endl;
  _x = x, _y = y;
}

Point2D::Point2D()
{
  cout << "
       Constructeur
       sans argument
       de Point2D"
       << endl;
}
```

Listing 61: Point2D_V4.cpp

```
#include "Cercle.hh"
int main()
{
  Cercle c;
  return 0;
}
```

Listing 62: code29.cpp

Soit le code ci-contre: On déclare deux classes, dont la classe *Point2D* ayant deux données membres et deux constructeurs.

La classe *Cercle* utilise le constructeur par défaut de la classe. Elle possède deux données membres dont une instance de la classe *Point2D*.

```
./a.out
Constructeur sans
argument de
Point2D
```

Objets membres

```cpp
#ifndef __CERCLE_HH__
#define __CERCLE_HH__

class Point2D{
  float _x, _y;
public:
  Point2D(float, float);
  Point2D();
};

class Cercle {
private:
  Point2D _centre;
  float _rayon;
};

#endif
```

Listing 60: Cercle.hh

```cpp
#include<iostream>
#include "Cercle.hh"
using namespace::std;

Point2D::Point2D(
      float x, float y
      ){
  cout << "
        Constructeur
        Point2D("
        << x << ";"
        << y << ")"
        << endl;
  _x = x, _y = y;
}

Point2D::Point2D()
{
  cout << "
        Constructeur
        sans argument
        de Point2D"
        << endl;
}
```

Listing 61: Point2D_V4.cpp

```cpp
#include "Cercle.hh"
int main()
{
  Cercle c;
  return 0;
}
```

Listing 62: code29.cpp

A l'exécution de ce code, on remarque que construire un objet de type *Cercle* entraîne la création d'un objet de type *Point2D*.

```
./a.out
Constructeur sans
argument de
Point2D
```

Objets membres

```
#ifndef __CERCLE_HH__
#define __CERCLE_HH__

class Point2D{
  float _x, _y;
public:
  Point2D(float, float);
  Point2D();
};

class Cercle {
private:
  Point2D _centre;
  float _rayon;
};

#endif
```

Listing 60: Cercle.hh

```
#include<iostream>
#include "Cercle.hh"
using namespace::std;

Point2D::Point2D(
      float x, float y
      ){
  cout << "
        Constructeur
        Point2D("
        << x << ";"
        << y << ")"
        << endl;
  _x = x, _y = y;
}

Point2D::Point2D()
{
  cout << "
        Constructeur
        sans argument
        de Point2D"
        << endl;
}
```

Listing 61: Point2D_V4.cpp

```
#include "Cercle.hh"
int main()
{
  Cercle c;
  return 0;
}
```

Listing 62: code29.cpp

En fait, le constructeur de *Point2D* est appelé avant celui de *Cercle*.

Ici, on remarque d'ailleurs que le constructeur par défaut de *Cercle* fait appel au constructeur sans argument de *Point2D*.

```
./a.out
Constructeur sans
argument de
Point2D
```

Objets membres

```
#ifndef __CERCLE_HH__
#define __CERCLE_HH__

class Point2D{
  float _x, _y;
public:
  Point2D(float, float);
  Point2D();
};

class Cercle {
private:
  Point2D _centre;
  float _rayon;
};

#endif
```

Listing 60: Cercle.hh

```
#include<iostream>
#include "Cercle.hh"
using namespace::std;

Point2D::Point2D(
      float x, float y
      ){
  cout << "
        Constructeur
        Point2D("
        << x << ";"
        << y << ")"
        << endl;
  _x = x, _y = y;
}

Point2D::Point2D()
{
  cout << "
        Constructeur
        sans argument
        de Point2D"
        << endl;
}
```

Listing 61: Point2D_V4.cpp

```
#include "Cercle.hh"
int main()
{
  Cercle c;
  return 0;
}
```

Listing 62: code29.cpp

Comment faire pour passer des paramètres au constructeur de *Point2D* lorsqu'on crée un objet de type *Cercle* ?

```
./a.out
Constructeur sans
argument de
Point2D
```

Objets membres

```
#ifndef __CERCLE_V2_HH__
#define __CERCLE_V2_HH__

class Point2D{
  float _x, _y;
public:
  Point2D(float, float);
  Point2D();
};

class Cercle {
private:
  Point2D _centre;
  float _rayon;
public:
  Cercle(float x, float y,
         float r);
};

#endif
```

Listing 63: Cercle_V2.hh

```
#include<iostream>
#include "Cercle_V2.hh"
using namespace::std;

Point2D::Point2D(float x, float y){
  cout << "Constructeur Point2D("
       << x << ";"
       << y << ")"
       << endl;
  _x = x, _y = y;
}

Point2D::Point2D()
{
  cout << "Constructeur sans
       argument de Point2D"
       << endl;
}

Cercle::Cercle(float x, float y,
     float r): _centre(x,y)
{
  _rayon = r;
  cout << "Constructeur de Cercle"
       << endl;
}
```

Listing 64: Point2D_V5.cpp

On modifie le code: on ajoute un constructeur qui possède 3 paramètres, à la classe *Cercle*. On va passer 2 de ces paramètres au constructeur de la classe *Point2D*.

```
#include "Cercle_V2.hh"

int main()
{
  Cercle c(2, 3, 4);
}
```

Listing 65: code30.cpp

Objets membres

```
#ifndef __CERCLE_V2_HH__
#define __CERCLE_V2_HH__

class Point2D{
  float _x, _y;
public:
  Point2D(float, float);
  Point2D();
};

class Cercle {
private:
  Point2D _centre;
  float _rayon;
public:
  Cercle(float x, float y,
         float r);
};

#endif
```

Listing 63: Cercle_V2.hh

```
#include<iostream>
#include "Cercle_V2.hh"
using namespace::std;

Point2D::Point2D(float x, float y){
  cout << "Constructeur Point2D("
       << x << ";"
       << y << ")"
       << endl;
  _x = x, _y = y;
}

Point2D::Point2D()
{
  cout << "Constructeur sans
       argument de Point2D"
       << endl;
}

Cercle::Cercle(float x, float y,
     float r): _centre(x,y)
{
  _rayon = r;
  cout << "Constructeur de Cercle"
       << endl;
}
```

Listing 64: Point2D_V5.cpp

On va utiliser une syntaxe particulière: entre l'entête de la fonction et son corps, on insère ':' et la définition de la donnée membre : *_centre(x, y)*

```
#include "Cercle_V2.hh"

int main()
{
  Cercle c(2, 3, 4);
}
```

Listing 65: code30.cpp

```
./a.out
Constructeur
Point2D(2;3)
Constructeur de Cercle
```

Objets membres

```
#ifndef __CERCLE_V3_HH__
#define __CERCLE_V3_HH__

class Point2D{
  float _x, _y;
public:
  Point2D(float, float);
};

class Cercle {
private:
  Point2D _centre;
  float _rayon;
public:
  Cercle(Point2D, float);
};

#endif
```

Listing 66: Cercle_V3.hh

```
#include<iostream>
#include "Cercle_V3.hh"
using namespace std;

Point2D::Point2D(float x, float y){
  cout << "Constructeur Point2D("
       << x << ";"
       << y << ")"
       << endl;
  _x = x, _y = y;
}

Cercle::Cercle(Point2D p, float r)
    : _centre(p)
{
  cout << "C. de Cercle avec
       Point2D" << endl;
  _rayon = r;
}
```

Listing 67: Point2D_V6.cpp

```
#include "Cercle_V3.hh"

int main()
{
  Point2D P(2, 3);
  Cercle c(P, 4);
}
```

Listing 68: code31.cpp

Voici un autre constructeur pour *Cercle* en utilisant le type *Point2D*.

La syntaxe est proche de l'exemple précédent, l'objet *Point2D* est construit en premier via l'instruction *_centre(p)*.

```
 ./a.out
Constructeur
Point2D(2;3)
C. de Cercle avec
Point2D
```

Surcharge d'opérateurs

Introduction

Imaginons que nous définissions une classe *Complex* pour gérer les nombres complexes dans un programme.

On va certainement être amené à définir les opérations courantes sur ces nombres. Par exemple, l'addition entre deux nombres complexes pourrait se définir comme une fonction ayant comme prototype:

```
Complex add(const Complex &) const;
```

Pour utiliser cette fonction dans un programme, on écrirait par exemple:

```
Complex c(1, 1), c2(2, 2);
Complex c3 = c.add(c2) ;
```

Introduction

C'est bien, mais on est habitué à une écriture qui nous semble plus naturelle:
On aurait envie d'écrire:

```
Complex c4 = c + c2;
```

Que nous faut-il pour cela ?
L'opérateur '+' existe bien en C++, mais il n'existe que pour des types prédéfinis, de base, tels que *int*, *float*, *double*, etc.
Pour que nous puissions l'utiliser dans le cadre des nombres complexes que nous avons définis, il faudrait le définir dans ce contexte.

Introduction

Le C++ permet de telles définitions supplémentaires. On appelle ce mécanisme la surcharge ou surdéfinition d'opérateurs. Ce mécanisme vient compléter la surcharge de fonctions que nous avons déjà vue.

Il ne faut pas croire qu'il s'agit là de quelque chose de naturel et que tous les langages le permettent. Le C, par exemple, ne permet pas cela.

De plus, il existe une contrainte importante en C++ à cette possibilité. Il n'est pas possible de redéfinir un opérateur qui s'appliquerait à des types de base.
Hors de question, donc, de redéfinir l'addition des entiers.

Mais qui aurait envie de faire cela, au risque de rendre son programme incompréhensible?

Introduction

Presque tous les opérateurs peuvent être redéfinis. Seuls quelques-uns échappent à cette règle.

Ainsi, parmi les opérateurs que nous connaissons déjà, ceux qui suivent ne peuvent pas être redéfinis:

- :: (opérateur de résolution de portée)
- . (opérateur point, pour accéder aux champs d'un objet)
- *sizeof*
- ?: (opérateur ternaire)

Introduction

Les autres peuvent donc prendre une définition différente en fonction du contexte dans lequel ils s'appliquent.

Néanmoins, ils gardent la même priorité et la même associativité que les opérateurs que nous connaissons déjà.

Ainsi, même si nous les redéfinissons, l'opérateur * de la multiplication sera "plus prioritaire" que l'addition +.

De même, les opérateurs conservent leur pluralité. Un opérateur unaire le reste, un binaire le restera également.

Un exemple

```
#ifndef __COMPLEX_V1_HH__
#define __COMPLEX_V1_HH__

class Complex
{
private:
 double _real, _imag;

public:
 Complex(double real,
      double imag);
 Complex operator+(const
      Complex &) const;
 void affiche() const;
};

#endif
```

Listing 69: Complex_V1.hh

```
#include <iostream>
#include "Complex_V1.hh"

using namespace std;

Complex::Complex(double real,
     double imag):
  _real(real), _imag(imag) {}

Complex Complex::operator+(const
     Complex &c) const
{
  return Complex(_real + c._real,
       _imag + c._imag);
}

void Complex::affiche() const
{
  cout << "(" << _real << ", "
       << _imag << ")"<< endl;
}
```

Listing 70: Complex_V1.cpp

```
#include <iostream>
#include "Complex_V1.hh"

int main()
{
  Complex c(1,1);
  Complex c2(2,2);
  Complex cres = c + c2;
  cres.affiche();
}
```

Listing 71: main_Complex.cpp

On définit une classe *Complex* ayant deux données privées, de type *double*, destinées à recevoir les parties réelles et imaginaires de nos nombres complexes.

On définit aussi un constructeur pour initialiser ces deux champs.

Un exemple

```
#ifndef __COMPLEX_V1_HH__
#define __COMPLEX_V1_HH__

class Complex
{
private:
 double _real, _imag;

public:
 Complex(double real,
      double imag);
 Complex operator+(const
      Complex &) const;
 void affiche() const;
};

#endif
```

Listing 69: Complex_V1.hh

```
#include <iostream>
#include "Complex_V1.hh"

using namespace std;

Complex::Complex(double real,
      double imag):
  _real(real), _imag(imag) {}

Complex Complex::operator+(const
      Complex &c) const
{
  return Complex(_real + c._real,
       _imag + c._imag);
}

void Complex::affiche() const
{
  cout << "(" << _real << ", "
       << _imag << ")"<< endl;
}
```

Listing 70: Complex_V1.cpp

```
#include <iostream>
#include "Complex_V1.hh"

int main()
{
  Complex c(1,1);
  Complex c2(2,2);
  Complex cres = c + c2;
  cres.affiche();
}
```

Listing 71: main_Complex.cpp

La fonction *affiche* est classique et son but est simplement de produire un affichage des données membres de notre instance.

Un exemple

```
#ifndef __COMPLEX_V1_HH__
#define __COMPLEX_V1_HH__

class Complex
{
private:
 double _real, _imag;

public:
 Complex(double real,
      double imag);
 Complex operator+(const
      Complex &) const;
 void affiche() const;
};

#endif
```

Listing 69: Complex_V1.hh

```
#include <iostream>
#include "Complex_V1.hh"

using namespace std;

Complex::Complex(double real,
     double imag):
  _real(real), _imag(imag) {}

Complex Complex::operator+(const
     Complex &c) const
{
  return Complex(_real + c._real,
       _imag + c._imag);
}

void Complex::affiche() const
{
  cout << "(" << _real << ", "
       << _imag << ")"<< endl;
}
```

Listing 70: Complex_V1.cpp

```
#include <iostream>
#include "Complex_V1.hh"

int main()
{
  Complex c(1,1);
  Complex c2(2,2);
  Complex cres = c + c2;
  cres.affiche();
}
```

Listing 71: main_Complex.cpp

Il reste une dernière fonction membre, appelée *operator+*. C'est cette fonction qui redéfinit l'opérateur + pour nos nombres complexes.

La syntaxe est toujours la même quel que soit l'opérateur.

Un exemple

```
#ifndef __COMPLEX_V1_HH__
#define __COMPLEX_V1_HH__

class Complex
{
private:
 double _real, _imag;

public:
 Complex(double real, double imag);
 Complex operator+(const Complex &) const;
 void affiche() const;
};

#endif
```

Listing 69: Complex_V1.hh

On voit qu'*operator+* est une fonction membre de la classe *Complex*.
Son argument est une référence sur une autre instance de type *Complex*. Celle-ci est déclarée *const*, car cet objet n'est pas appelé à changer lors de notre addition.
De même, la fonction elle-même est déclarée comme *const* car l'instance courante n'est pas amenée à être modifiée lors de l'opération.
Cela nous permet d'utiliser notre addition sur des objets constants, ce qui semble raisonnable.
Enfin, celle-ci renverra un objet de type *Complex*. Car, pour rester cohérent, l'addition de deux nombres complexes est aussi un nombre complexe.

Un exemple

```
#include <iostream>
#include "Complex_V1.hh"

using namespace std;

Complex::Complex(double real, double imag):
  _real(real), _imag(imag) {}

Complex Complex::operator+(const Complex &c) const
{
  return Complex(_real + c._real, _imag + c._imag);
}

void Complex::affiche() const
{
  cout << "(" << _real << ", "
       << _imag << ")"<< endl;
}
```

Listing 70: Complex_V1.cpp

Intéressons-nous à l'implémentation proprement dite:

On voit que notre fonction retourne simplement un nouvel objet de type *Complex* en fixant les deux valeurs passées à son constructeur comme la somme des parties réelles et imaginaires.

Le tout est ensuite renvoyé par valeur en sortie de la fonction.

Commutativité

- Si nous voulons définir une opérateur + avec un *Complex* c et un *double*, l'opérateur que nous surchargeons ne peut s'appliquer que dans l'ordre dans lequel on le définit.

  $c + 3.5$ n'appellera pas la même fonction que $3.5 + c$.

  Car la première porte sur un *Complex* en premier argument et un *double* en second.

  Le premier argument de $3.5 + c$ est un *double* et le deuxième un *Complex*.
- On va utiliser une fonction amie pour le deuxième opérateur d'addition.

Opérateurs & fonctions amies

```
#ifndef __COMPLEX_V2_HH__
#define __COMPLEX_V2_HH__

class Complex
{
private:
  double _real, _imag;

public:
  Complex(double real, double imag);
  Complex(const Complex &);
  Complex operator+(const Complex &) const;
  friend Complex operator+(double, const
      Complex &);
  void affiche() const;
};

#endif
```

Listing 71: Complex_V2.hh

On souhaite ici définir, dans notre classe *Complex*, une fonction **amie** *operator+*, chargée de définir l'opérateur d'addition entre un *double* et un *Complex*.

Ici on n'a pas d'autre choix que d'utiliser une fonction amie pour l'opération souhaitée, car l'ordre des arguments est important.

Si on avait voulu définir l'opérateur d'addition entre un *Complex* et un *double*, on aurait eu le choix entre utiliser une fonction amie ou une fonction membre.

```
Complex operator+(double d,
      const Complex & c1)
{
  return Complex(d + c1._real,
        c1._imag);
}
```

Opérateur []

```
#ifndef __VECTOR_V1_HH__
#define __VECTOR_V1_HH__

#include <iostream>
using namespace std;

class Vector
{
private :
 int *_val;
 int _n;

public:
 Vector(int n);
 ~Vector();
 int & operator[] (int i);
};

#endif
```

Listing 72: Vector_V1.hh

```
#include "Vector_V1.hh"

Vector::Vector(int n) :
      _n(n){
  _val = new int[n];
}

Vector::~Vector(){
  delete [] _val;
}

int & Vector::operator[]
      (int i)
{
  return _val[i];
}
```

Listing 73: Vector_V1.cpp

```
#include "Vector_V1.hh"

int main()
{
 Vector v(5);

 for (int i=0;i<5;i++)
   v[i] = i;

 for (int i=0;i<5;i++)
   cout << v[i] << " ";
 cout << endl;
 return 0;
}
```

Listing 74: main_Vector_V1.cpp

La notation [] vue, par exemple, pour accéder aux éléments d'un tableau, est un opérateur que l'on peut redéfinir lorsqu'il s'applique à un objet.

Evidemment, son utilisation s'applique particulièrement bien aux objets qui sont la surcouche d'un tableau, comme ici pour la classe *Vector*.

Opérateur []

```
#ifndef __VECTOR_V1_HH__
#define __VECTOR_V1_HH__

#include <iostream>
using namespace std;

class Vector
{
private :
 int *_val;
 int _n;

public:
 Vector(int n);
 ~Vector();
 int & operator[] (int i);
};

#endif
```

Listing 72: Vector_V1.hh

```
#include "Vector_V1.hh"

Vector::Vector(int n) :
      _n(n){
  _val = new int[n];
}

Vector::~Vector(){
  delete [] _val;
}

int & Vector::operator[]
      (int i)
{
  return _val[i];
}
```

Listing 73: Vector_V1.cpp

```
#include "Vector_V1.hh"

int main()
{
 Vector v(5);

 for (int i=0;i<5;i++)
   v[i] = i;

 for (int i=0;i<5;i++)
   cout << v[i] << " ";
 cout << endl;
 return 0;
}
```

Listing 74: main_Vector_V1.cpp

On a déjà parlé des constructeur et destructeur. Leur rôle est simplement ici de gérer le tableau de données - un pointeur sur entier - qui est un membre privé de notre classe.

Celui-ci, bien qu'alloué, n'est pas initialisé lors du constructeur et ses valeurs sont donc considérées comme aléatoires.

Opérateur []

```
#ifndef __VECTOR_V1_HH__
#define __VECTOR_V1_HH__

#include <iostream>
using namespace std;

class Vector
{
private :
 int *_val;
 int _n;

public:
 Vector(int n);
 ~Vector();
 int & operator[] (int i);
};

#endif
```

Listing 72: Vector_V1.hh

```
#include "Vector_V1.hh"

Vector::Vector(int n) :
      _n(n){
  _val = new int[n];
}

Vector::~Vector(){
  delete [] _val;
}

int & Vector::operator[]
      (int i)
{
  return _val[i];
}
```

Listing 73: Vector_V1.cpp

```
#include "Vector_V1.hh"

int main()
{
 Vector v(5);

 for (int i=0;i<5;i++)
   v[i] = i;

 for (int i=0;i<5;i++)
   cout << v[i] << " ";
 cout << endl;
 return 0;
}
```

Listing 74: main_Vector_V1.cpp

Intéressons nous à la surcharge de [].

Que veut-on faire exactement avec cette surcharge?

On souhaite, d'une part, accéder à un élément donné du tableau, pour l'afficher par exemple.

Mais on veut également pouvoir le modifier!

Ces deux cas sont visibles dans la fonction *main*. D'abord une boucle dans laquelle les valeurs accédées sont modifiées et une autre dans laquelle elles ne sont qu'accédées.

Opérateur []

```
#ifndef __VECTOR_V1_HH__
#define __VECTOR_V1_HH__

#include <iostream>
using namespace std;

class Vector
{
private :
 int *_val;
 int _n;

public:
 Vector(int n);
 ~Vector();
 int & operator[] (int i);
};

#endif
```

Listing 72: Vector_V1.hh

```
#include "Vector_V1.hh"

Vector::Vector(int n) :
      _n(n){
  _val = new int[n];
}

Vector::~Vector(){
  delete [] _val;
}

int & Vector::operator[]
      (int i)
{
  return _val[i];
}
```

Listing 73: Vector_V1.cpp

```
#include "Vector_V1.hh"

int main()
{
 Vector v(5);

 for (int i=0;i<5;i++)
   v[i] = i;

 for (int i=0;i<5;i++)
   cout << v[i] << " ";
 cout << endl;
 return 0;
}
```

Listing 74: main_Vector_V1.cpp

En fait, tout réside dans le type de la valeur de retour de notre fonction.

On voit ici qu'elle retourne une référence sur l'élément auquel on veut accéder.

Cela permet, une fois la fonction terminée, de pouvoir modifier cette valeur.

Si nous avions travaillé avec un retour par valeur, nous n'aurions eu qu'une copie de notre valeur et celle-ci n'aurait pas pu être modifiée.

Opérateur <<

```
#ifndef __VECTOR_V2_HH__
#define __VECTOR_V2_HH__

#include <iostream>
using namespace std;

class Vector
{
private :
 int *_val;
 int _n;

public:
 Vector(int n);
 ~Vector();
 int & operator[] (int i);
 friend ostream& operator<<(
    ostream &c, Vector& v);
};

#endif
```

Listing 75: Vector_V2.hh

```
#include "Vector_V2.hh"

Vector::Vector(int n) : _n(n){
  _val = new int[n];
}

Vector::~Vector(){
  delete [] _val;
}

int & Vector::operator[](int i)
{
  return _val[i];
}

ostream& operator<<(ostream &c,
      Vector& v)
{
  for(int i=0; i<v._n; i++)
    c << v[i] << " ";
  c << endl;
  return c;
}
```

Listing 76: Vector_V2.cpp

Un opérateur que l'on peut aussi surcharger avec intérêt est l'opérateur <<, pour l'objet *ostream*.

Il ne peut pas se surcharger comme fonction membre de la classe car le premier opérande est un objet de type *ostream*. Il s'agit d'un objet de flux de sortie, comme *cout* que l'on connaît déjà.

Il peut donc être défini comme fonction amie de la classe.

Sa valeur de retour est aussi un objet de type *ostream*, renvoyé en référence. On fait cela afin de pouvoir utiliser l'opérateur en série.
Ainsi lorsqu'on utilise cet opérateur avec *cout* et un objet de type *Vector*, il est appelé et produit les affichages définis dans le corps de la fonction.

Opérateur <<

```
#ifndef __VECTOR_V2_HH__
#define __VECTOR_V2_HH__

#include <iostream>
using namespace std;

class Vector
{
private :
 int *_val;
 int _n;

public:
 Vector(int n);
 ~Vector();
 int & operator[] (int i);
 friend ostream& operator<<(
    ostream &c, Vector& v);
};

#endif
```

Listing 74: Vector_V2.hh

```
#include "Vector_V2.hh"

Vector::Vector(int n) : _n(n){
  _val = new int[n];
}

Vector::~Vector(){
  delete [] _val;
}

int & Vector::operator[](int i)
{
  return _val[i];
}

ostream& operator<<(ostream &c,
      Vector& v)
{
  for(int i=0; i<v._n; i++)
    c << v[i] << " ";
  c << endl;
  return c;
}
```

Listing 75: Vector_V2.cpp

```
#include <iostream>
#include "Vector_V2.hh"
using namespace std;

int main()
{
  Vector v(5);

  for (int i=0; i<5; i++)
    v[i] = i;

  cout << v;
  return 0;
}
```

Listing 76: main_Vector_V2.cpp

On voit ici l'intérêt de la définition de l'opérateur: la syntaxe dans le *main* pour afficher un objet de la classe devient très simple!

Les foncteurs - Opérateur ()

```
#ifndef __AFFINE_HH__
#define __AFFINE_HH__

class Affine
{
private:
 double _a, _b;
public :
 Affine(double, double);
 double operator()(double x) const;
};

#endif
```

Listing 77: Affine.hh

```
#include "Affine.hh"

Affine::Affine(double a,
        double b): _a(a), _b(b)
{}

double Affine::operator()(double
                x) const
{
  return (_a*x + _b);
}
```

Listing 78: Affine.cpp

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

using namespace std;

void valeurEn0(const
       Affine & a)
{
 cout << a(0)
      << endl;
}

int main()
{
  Affine a(2, 3);
  valeurEn0(a);
  return 0;
}
```

Listing 79: main_Affine.cpp

Un opérateur qu'il peut être pratique de surcharger est l'opérateur ().

On peut ainsi transformer un objet en fonction et l'utiliser comme tel. Par exemple, ici, on construit une fonction affine *a* sous la forme d'un objet que l'on paramétrise lors de sa construction et on peut l'utiliser, par exemple comme paramètre d'une autre fonction (*valeurEn0*).

Ici, l'exemple est trivial et on aurait pu aussi utiliser un pointeur sur fonction.

Il existe de nombreux autres opérateurs qu'il est possible de surcharger et ce cours ne permet malheureusement pas de les étudier tous.

L'opérateur '=', ainsi que les opérateurs d'incrémentation ($++$ et $--$) peuvent néanmoins faire l'objet d'une étude plus approfondie en raison de leur subtilité.

Les patrons de fonctions

- Nous allons maintenant introduire une fonctionnalité très intéressante de C++: les patrons, ou *template* en Anglais (ce cours utilisera indistinctement les deux appellations).
- Pour comprendre tout l'intérêt de ce concept, il faut se souvenir de la surcharge des fonctions.

  Si l'on voulait introduire une fonction *min* sur les entiers qui renverrait le plus petit de deux entiers passés en paramètres, on créerait cette fonction, mais on ne pourrait pas l'utiliser pour des *float* etc. Il faudrait créer une fonction par type de données que l'on veut comparer.

Un exemple

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

template <typename T>
T minimum(T a, T b)
{
  return (a < b) ? a : b;
}

int main()
{
  int a = 2, b = 3;
  double ad = 5.1, bd = 6.2;

  cout << a << ", " << b << " -> "
       << minimum(a,b) << endl;
  cout << ad << ", " << bd << " -> "
       << minimum(ad,bd) << endl;
  return 0;
}
```

Listing 80: code40.cpp

Dans cet exemple, on définit la fonction *minimum* une fois pour toute et on peut l'utiliser quel que soit le type passé en paramètre.

Concernant la syntaxe, on commence donc par le mot clé *template*. Ensuite, le contenu des chevrons définira le caractère générique de notre fonction.

Ici *typename T* définira donc le type générique T qu'on pourra utiliser au sein de notre fonction.

Un exemple

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

template <typename T>
T minimum(T a, T b)
{
  return (a < b) ? a : b;
}

int main()
{
  int a = 2, b = 3;
  double ad = 5.1, bd = 6.2;

  cout << a << ", " << b << " -> "
       << minimum(a,b) << endl;
  cout << ad << ", " << bd << " -> "
       << minimum(ad,bd) << endl;
  return 0;
}
```

Listing 78: code40.cpp

En fait, le compilateur va générer, de manière transparente pour l'utilisateur, autant de fonctions que nécessaire selon les types de paramètres qu'on passera à la fonction.

Il y a un seul bémol à cela en comparaison de la surcharge de fonction: l'algorithme ne varie pas en fonction du type des paramètres.

Paramètres expression

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

template <typename T>
T maxtab(T* arr, unsigned int n)
{
  T tmp = arr[0];
  for (unsigned int i=0; i<n; ++i)
    if (tmp < arr[i])
      tmp = arr[i];
  return tmp;
}

int main()
{
  double tab[6] = {2.0, 3.0, 4.0,
      8.0, 2.0, 1.0};
  double tmax = maxtab(tab, 6);
  cout << "Plus grand element: "
       << tmax << endl;
  return 0;
}
```

Listing 79: code41.cpp

Voici un exemple plus complexe.

On veut une fonction qui retourne le plus grand élément d'un tableau qu'on lui fournit en paramètre.

Il n'y a pas de raison de se limiter aux tableaux d'un type particulier.

En fait, tant qu'une relation de comparaison peut être définie entre deux éléments du tableau, notre algorithme peut fonctionner.

Paramètres expression

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

template <typename T>
T maxtab(T* arr, unsigned int n)
{
  T tmp = arr[0];
  for (unsigned int i=0; i<n; ++i)
    if (tmp < arr[i])
      tmp = arr[i];
  return tmp;
}

int main()
{
  double tab[6] = {2.0, 3.0, 4.0,
      8.0, 2.0, 1.0};
  double tmax = maxtab(tab, 6);
  cout << "Plus grand element: "
       << tmax << endl;
  return 0;
}
```

Listing 78: code41.cpp

On définit donc notre fonction comme une fonction *template*, prenant un pointeur sur type en paramètre ainsi qu'un entier non signé qui contiendra le nombre d'éléments du tableau.

On remarque par ailleurs que des types non "templatés" peuvent entrer comme paramètres d'une fonction *template*, il s'agit de paramètres expression.

L'algorithme ensuite est classique, en prenant soin d'utiliser le type *template* quand c'est nécessaire.

Surdéfinition de fonctions template

```
template <typename T>
T maxtab(T * arr, unsigned int n)
{
  T tmp= arr[0];
  for (unsigned int i=0; i<n; i++)
    if (tmp < arr[i])
      tmp = arr[i];
  return tmp;
}

template <typename T>
T maxtab(T* arr, T* arr2,
         unsigned int n, unsigned
             int n2)
{
  T tmp  = maxtab(arr, n);
  T tmp2 = maxtab(arr2, n2);
  return (tmp < tmp2) ? tmp2 : tmp;
}
```

Listing 79: code42_1.cpp

On peut surcharger une fonction *template* en faisant varier son nombre d'éléments ou le type de ceux-ci.

Ici nous avons surchargé la fonction *maxtab* en donnant la possibilité de renvoyer le plus grand élément de deux tableaux.

Spécialisation

```
template <typename T>
T maxtab(T * arr, unsigned int n)
{
  T tmp= arr[0];
  for (unsigned int i=0; i<n; i++)
    if (tmp < arr[i])
      tmp = arr[i];
  return tmp;
}

string maxtab(string* arr, unsigned int n)
{
  string tmp = to_lower(arr[0]);
  for (unsigned int i=0; i<n; i++)
    {
      if (to_lower(tmp)< to_lower(arr[i]))
        tmp = arr[i];
    }
  return tmp;
}
```

Listing 80: code42_2.cpp

On peut également définir un *template* pour un algorithme général qui est valable quel que soit le type, mais aussi spécialiser une fonction, c'est-à-dire définir un algorithme pour un type particulier.

Ici, dans le cas d'un tableau de *string*, l'opérateur $<$, pour comparer deux *strings*, existe mais est sensible à la casse: on ne veut pas cela ici. On spécialise ici la fonction *maxtab* et on utilise une fonction non standard *to_lower* convertissant une chaîne de caractères en minuscules.

Pour finir

Enfin, il n'est pas forcément évident d'écrire et de spécifier une fonction template. En effet, il faut faire attention aux cas ambigüs - c'est-à-dire où le compilateur ne sait pas s'il doit utiliser une fonction plutôt qu'une autre car les deux conviennent.

Par ailleurs, la règle pour les patrons de fonctions est que le type doit convenir "parfaitement", c'est-à-dire qu'un *const int* n'est pas un *int* etc.

Les patrons de classes

- Il existe un mécanisme similaire aux fonctions template pour les classes.
- Bien que semblable aux patrons de fonctions sur de nombreux points, il existe des différences avec les templates de classe.
- On va voir que cela permet d'implémenter un code générique et réutilisable.

Classe template Vector

```
#ifndef __VECTOR_HPP__
#define __VECTOR_HPP__
#include <iostream>
template <typename T>
class Vector
{
private:
 T *_val;
 int _n;
public:
 Vector(int n):_n(n){
   _val = new T[_n];
 }
 ~Vector(){
   if (_val) {
     delete [] _val;
   }
 }
 T& operator[] (int i){
   return _val[i];
 }
};
#endif
```

Listing 81: Vector.hpp

On se souvient de la classe *Vector* que nous avions définie dans ce cours, dans la partie *Surcharge d'opérateurs*.

Celle-ci, bien que déclarant une surcouche "objet" à des tableaux d'entiers, n'était pas utilisable si nous voulions stocker d'autres types. Il aurait alors fallu tout refaire.

Perte de temps, d'énergie ...

Classe template Vector

```
#ifndef __VECTOR_HPP__
#define __VECTOR_HPP__
#include <iostream>
template <typename T>
class Vector
{
private:
 T *_val;
 int _n;
public:
 Vector(int n):_n(n){
   _val = new T[_n];
 }
 ~Vector(){
   if (_val) {
     delete [] _val;
   }
 }
 T& operator[] (int i){
   return _val[i];
 }
};
#endif
```

Listing 75: Vector.hpp

En pratique, comme on le voit ci-contre, on définit les types dans l'entête de la déclaration de la classe, de la même façon que pour les fonctions template.

On remarquera aussi un point important: à l'inverse des classes que nous déclarons d'habitude, ici tout est dans le même fichier!

En effet, le code, ainsi que la déclaration de la classe ne sont, en somme, qu'une déclaration. Le code n'est vraiment généré qu'à la compilation, à partir de ce template, en fonction des besoins.

Classe template Vector

```
#ifndef __VECTOR_HPP__
#define __VECTOR_HPP__
#include <iostream>
template <typename T>
class Vector
{
private:
 T *_val;
 int _n;
public:
 Vector(int n):_n(n){
   _val = new T[_n];
 }
 ~Vector(){
   if (_val) {
     delete [] _val;
   }
 }
 T& operator[] (int i){
   return _val[i];
 }
};
#endif
```

Listing 75: Vector.hpp

En résumé, on adoptera les conventions suivantes dans ce cours:

- Déclaration d'une classe → fichier .hh
- Définition d'une classe → fichier .cpp
- Déclaration d'un template de classe → fichier .hpp

Classe template Vector

```
#ifndef __VECTOR_HPP__
#define __VECTOR_HPP__
#include <iostream>
template <typename T>
class Vector
{
private:
 T *_val;
 int _n;
public:
 Vector(int n):_n(n){
   _val = new T[_n];
 }
 ~Vector(){
   if (_val) {
     delete [] _val;
   }
 }
 T& operator[] (int i){
   return _val[i];
 }
};
#endif
```

Listing 75: Vector.hpp

Comme on n'a, au final, déclaré qu'un template de classe, celui-ci ne se compile pas "séparément". Il ne sera compilé que s'il est inclus dans un fichier de code et que si ce code l'utilise!

Classe template Vector

```
#ifndef __VECTOR_HPP__
#define __VECTOR_HPP__
#include <iostream>
template <typename T>
class Vector
{
private:
 T *_val;
 int _n;
public:
 Vector(int n):_n(n){
   _val = new T[_n];
 }
 ~Vector(){
   if (_val) {
     delete [] _val;
   }
 }
 T& operator[] (int i){
   return _val[i];
 }
};
#endif
```

Listing 75: Vector.hpp

```
#include <iostream>
#include "Vector.hpp"
using namespace std;

int main()
{
  Vector<double> vect(10);

  for (int i=0;i<10;i++)
    vect[i] = i*0.5;

  for (int i=0;i<10;i++)
    cout << vect[i]
         << endl;

  return 0;
}
```

Listing 76: main_Vector.cpp

Pour utiliser le patron de classe, on va devoir instancier le type, lors de la déclaration de notre variable.

Ainsi, on crée un objet *vect*, de type *Vector<double>*, soit un *Vector* dont le type sera des *double*.

Classe template Vector

```
#ifndef __VECTOR_HPP__
#define __VECTOR_HPP__
#include <iostream>
template <typename T>
class Vector
{
private:
 T *_val;
 int _n;
public:
 Vector(int n):_n(n){
   _val = new T[_n];
 }
 ~Vector(){
   if (_val) {
     delete [] _val;
   }
 }
 T& operator[] (int i){
   return _val[i];
 }
};
#endif
```

Listing 75: Vector.hpp

```
#include <iostream>
#include "Vector.hpp"
using namespace std;

int main()
{
  Vector<double> vect(10);

  for (int i=0;i<10;i++)
    vect[i] = i*0.5;

  for (int i=0;i<10;i++)
    cout << vect[i]
         << endl;

  return 0;
}
```

Listing 69: main_Vector.cpp

Et pour le compiler?

On ne compile que le fichier contenant le *main*, car c'est, au final, le seul fichier cpp de notre programme ici.

Plus compliqué

```
#ifndef __VECTOR_HPP__
#define __VECTOR_HPP__
#include <iostream>
template <typename T>
class Vector
{
private:
 T *_val;
 int _n;
public:
 Vector(int n):_n(n){
   _val = new T[_n];
 }
 ~Vector(){
   if (_val) {
     delete [] _val;
   }
 }
 T& operator[] (int i){
   return _val[i];
 }
};
#endif
```

Listing 75: Vector.hpp

```
#ifndef __POINT_HPP__
#define __POINT_HPP__
#include<iostream>
using namespace std;

template <typename T>
struct Point
{
  T _x;
  T _y;
};

template <typename T>
ostream & operator<< (ostream& c,
      Point<T>& x) {
  c << x._x << ";" << x._y;
  return c;
}

#endif
```

Listing 70: Point.hpp

```
#include <iostream>
#include "Vector.hpp"
#include "Point.hpp"

int main()
{
 Vector<double> vect(10);

 for (int i=0;i<10;i++)  {
  vect[i] = i*0.5;
  cout << vect[i] << endl; }

 Vector< Point<int> > vect2(10);

 for (int i=0;i<10;i++) {
  vect2[i]._x = i;
  vect2[i]._y = 10-i;
  cout << vect2[i] << endl; }
}
```

Listing 71: main_Vector2.cpp

Dans cet exemple, on a ajouté la structure template *Point*, qui s'utilise comme une classe template, avec les particularités liées aux structures.

On peut ainsi définir un type *Point* dont le type générique est un entier.

On peut alors aussi définir un vecteur de *Point* d'entiers!

Plus compliqué

```
#ifndef __VECTOR_HPP__
#define __VECTOR_HPP__
#include <iostream>
template <typename T>
class Vector
{
private:
 T *_val;
 int _n;
public:
 Vector(int n):_n(n){
   _val = new T[_n];
 }
 ~Vector(){
   if (_val) {
     delete [] _val;
   }
 }
 T& operator[] (int i){
   return _val[i];
 }
};
#endif
```

Listing 75: Vector.hpp

```
#ifndef __POINT_HPP__
#define __POINT_HPP__
#include<iostream>
using namespace std;

template <typename T>
struct Point
{
  T _x;
  T _y;
};

template <typename T>
ostream & operator<< (ostream& c,
      Point<T>& x) {
  c << x._x << ";" << x._y;
  return c;
}

#endif
```

Listing 62: Point.hpp

```
#include <iostream>
#include "Vector.hpp"
#include "Point.hpp"

int main()
{
 Vector<double> vect(10);

 for (int i=0;i<10;i++)  {
  vect[i] = i*0.5;
  cout << vect[i] << endl; }

 Vector< Point<int> > vect2(10);

 for (int i=0;i<10;i++) {
  vect2[i]._x = i;
  vect2[i]._y = 10-i;
  cout << vect2[i] << endl; }
}
```

Listing 63: main_Vector2.cpp

La structure *Point* est très simple, elle ne contient que deux variables de type T générique.

On notera aussi la surcharge de l'opérateur << pour pouvoir afficher un objet *Point*.

Comme *Point* est un type template, il est "logique" qu'une fonction l'utilisant soit aussi template ··· sinon quel type de *Point* le compilateur instancierait-il?

Pour finir sur les patrons de classe

- Les patrons de classe sont un outil très puissant et largement utilisés par les développeurs pour créer de nouvelles librairies.
- On verra, dans le cadre de ce cours, la librairie STL pour Standard Template Library, qui définit ainsi de nombreux outils rapides et puissants.
- La librairie Boost, célèbre également, est une librairie basée sur les template.
- En maths, par exemple, la librairie Eigen++ (https://eigen.tuxfamily.org/) est une librairie pour l'algèbre linéaire et contient des algorithmes très optimisés.

Pour finir sur les patrons de classe

- La notion de template est plus vaste que celle abordée dans ce cours.
- Ainsi, on n'a pas abordé les notions de type expression, ni la spécialisation de classe template, ou la spécialisation partielle. Nous ne parlerons pas non plus de meta-programmation ou de template récursifs.
- Bien qu'utiles et intéressantes, ces notions sortent de l'objectif de ce cours qui est une introduction au C++ et à la POO.

Héritage

- La notion d'héritage en programmation orientée objet est une notion fondamentale.
- Il s'agit de créer de nouvelles classes, de nouveaux types, en se basant sur des classes déjà existantes.
- On pourra alors non seulement hériter, utiliser leurs capacités, leurs données et leurs fonctions, mais aussi étendre ces capacités. Il s'agit encore ici de ne pas écrire du code qui existe déjà mais de l'utiliser et de l'étendre sans avoir à modifier quelque chose qui existe et qui fonctionne déjà.

  On pourra ainsi écrire une classe dérivant d'une autre classe, mais aussi plusieurs classes héritant d'une autre classe.

  De même une classe peut hériter d'une classe qui peut elle-même hériter d'une classe etc...

Exemple

```
#ifndef __FORME_HH__
#define __FORME_HH__

#include <string>
using namespace std;

class Forme
{
private:
 string _nom;

public:
 void setNom(const string&);
};
#endif
```

Listing 64: Forme.hh

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

void Forme::setNom(const
    string& nom) {
  _nom = nom;
}
```

Listing 65: Forme.cpp

On commence par écrire une classe très simple, *Forme*, constituée seulement d'une chaîne qui contiendra le nom de notre forme.

Une fonction sera chargée d'initialiser notre chaîne.

Pour l'instant, on ne définit pas de constructeur ni de destructeur.

Exemple

```
#ifndef __FORME_HH__
#define __FORME_HH__

#include <string>
using namespace std;

class Forme
{
private:
 string _nom;

public:
 void setNom(const string&);
};
#endif
```

Listing 61: Forme.hh

On va oublier maintenant le code de la fonction *setNom*. On retiendra juste que celui-ci, comme son nom l'indique, sert à changer la valeur de la chaîne *_nom*, qui est une donnée membre.

En fait, pour hériter d'une classe, nous n'avons besoin que de sa déclaration, c'est-à-dire du fichier header, et du code de la classe compilé (en gros un fichier objet d'extension .o, obtenu après compilation).

Exemple

```
#ifndef __FORME_HH__
#define __FORME_HH__

#include <string>
using namespace std;

class Forme
{
private:
 string _nom;

public:
 void setNom(const string&);
};
#endif
```

Listing 61: Forme.hh

```
#ifndef __ROND_HH__
#define __ROND_HH__
#include "Forme.hh"

class Rond: public Forme
{
private:
  double _diametre;
public:
  void setDiametre(double);
};
#endif
```

Listing 63: Rond.hh

```
#include "Forme.hh"
#include "Rond.hh"
#include <iostream>
void Rond::setDiametre(
    double d) {
_diametre = d;
}
```

Listing 64: Rond.cpp

On crée maintenant une première classe fille de la classe *Forme*: à savoir la classe *Rond*. On remarque la ligne:

class Rond: public Forme

Dans la déclaration de la classe *Rond*, ":" suivi de *public Forme* signifie que *Rond* hérite de la classe *Forme*.

Le mot clé *public* sera expliqué dans la suite de ce cours.

Exemple

```
#ifndef __FORME_HH__
#define __FORME_HH__

#include <string>
using namespace std;

class Forme
{
private:
 string _nom;

public:
 void setNom(const string&);
};
#endif
```

Listing 61: Forme.hh

```
#ifndef __ROND_HH__
#define __ROND_HH__
#include "Forme.hh"

class Rond: public Forme
{
private:
  double _diametre;
public:
  void setDiametre(double);
};
#endif
```

Listing 56: Rond.hh

```
#include "Forme.hh"
#include "Rond.hh"
#include <iostream>
void Rond::setDiametre(double d) {
_diametre = d;
}
```

Listing 57: Rond.cpp

```
#ifndef __CARRE_HH__
#define __CARRE_HH__
#include "Forme.hh"

class Carre: public Forme
{
private:
 double _longueur;
public:
 void setLongueur(double);
};
#endif
```

Listing 58: Carre.hh

```
#include "Forme.hh"
#include "Carre.hh"
#include <iostream>

void Carre::setLongueur(
      double l)
{
 _longueur = l;
}
```

Listing 59: Carre.cpp

On déclare de même une classe *Carre*, héritant également de la classe *Forme*. Ces deux classes ont chacune leurs spécificités, comme on peut le voir, le rond ayant un diamètre et le carré une longueur.

Exemple

```
#ifndef __FORME_HH__
#define __FORME_HH__

#include <string>
using namespace std;

class Forme
{
private:
 string _nom;

public:
 void setNom(const string&);
};
#endif
```

Listing 61: Forme.hh

```
#include "Forme.hh"
#include "Carre.hh"
#include "Rond.hh"

int main(void)
{
 Carre c; Rond r; Forme f;
 c.setNom("carre");
 c.setLongueur(5.0);
 r.setNom("rond");
 r.setDiametre(3);
 f.setNom("forme generale");
}
```

Listing 59: main_Forme.cpp

```
#ifndef __ROND_HH__
#define __ROND_HH__
#include "Forme.hh"

class Rond: public Forme
{
private:
  double _diametre;
public:
  void setDiametre(double);
};
#endif
```

Listing 56: Rond.hh

```
#include "Forme.hh"
#include "Rond.hh"
#include <iostream>
void Rond::setDiametre(double d) {
_diametre = d;
}
```

Listing 57: Rond.cpp

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

void Forme::setNom(const string&
     nom) {
  _nom = nom;
}
```

Listing 62: Forme.cpp

```
#ifndef __CARRE_HH__
#define __CARRE_HH__
#include "Forme.hh"

class Carre: public Forme
{
private:
 double _longueur;
public:
 void setLongueur(double);
};
#endif
```

Listing 57: Carre.hh

```
#include "Forme.hh"
#include "Carre.hh"
#include <iostream>

void Carre::setLongueur(
     double l)
{
 _longueur = l;
}
```

Listing 58: Carre.cpp

Dans la fonction *main*, on déclare une variable de chaque type et on peut, sur les classes filles, appeler des fonctions de la classe *Forme*. L'inverse, évidemment, n'est pas possible!

Exemple

```
#ifndef __FORMEV2_HH__
#define __FORMEV2_HH__

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

class Forme
{
private:
 string _nom;

public:
 void setNom(const
      string&);
 void affiche();
};

#endif
```

Listing 65: Forme_V2.hh

```
#ifndef __RONDV1_HH__
#define __RONDV1_HH__
#include "Forme_V2.hh"

class Rond: public Forme
{
private:
  double _diametre;
public:
  void setDiametre(double);
};
#endif
```

Listing 66: Rond_V1.hh

```
#include "Rond_V1.hh"
void Rond::setDiametre(double d)
      {
 _diametre = d;
}
```

Listing 67: Rond_V1.cpp

```
#include "Forme_V2.hh"
void Forme::setNom(const string&
      nom) {
  _nom = nom;
}
void Forme::affiche() {
  cout << "Je suis de type " <<
       _nom << endl; }
```

Listing 68: Forme_V2.cpp

```
#ifndef __CARREV1_HH__
#define __CARREV1_HH__
#include "Forme_V2.hh"

class Carre: public
     Forme
{
private:
 double _longueur;
public:
 void setLongueur(
      double);
};
#endif
```

Listing 69: Carre_V1.hh

```
#include "Carre_V1.hh"

void Carre::setLongueur
     (double l)
{
 _longueur = l;
}
```

Listing 70: Carre_V1.cpp

On définit une fonction *affiche* dans la classe *Forme*. Celle-ci se contente d'afficher le nom de la forme.

Cette fonction est alors utilisable par toutes les classes filles, qui afficheront ce que contient la donnée _nom héritée de la classe *Forme*.

On a donc modifié les fonctionnalités de 3 classes après modification d'une seule. Pas mal! Mais l'affichage ne prend pas en compte la spécificité de chaque classe ...

Exemple

```
#include "Carre_V1.hh"
#include "Rond_V1.hh"

int main(void)
{
  Carre c;
  Rond r;
  Forme f;
  c.setNom("carre");
  c.setLongueur(5.0);

  r.setNom("rond");
  r.setDiametre(3);

  f.setNom("forme
       générale");
  f.affiche();
  c.affiche();
  r.affiche();
}
```

Listing 71: main_FormeV2.cpp

```
#ifndef __RONDV1_HH__
#define __RONDV1_HH__
#include "Forme_V2.hh"

class Rond: public Forme
{
private:
  double _diametre;
public:
  void setDiametre(double);
};
#endif
```

Listing 62: Rond_V1.hh

```
#include "Rond_V1.hh"
void Rond::setDiametre(double d)
      {
 _diametre = d;
}
```

Listing 63: Rond_V1.cpp

```
#include "Forme_V2.hh"
void Forme::setNom(const string&
      nom) {
  _nom = nom;
}
void Forme::affiche() {
  cout << "Je suis de type " <<
       _nom << endl; }
```

Listing 64: Forme_V2.cpp

```
#ifndef __CARREV1_HH__
#define __CARREV1_HH__
#include "Forme_V2.hh"

class Carre: public
      Forme
{
private:
 double _longueur;
public:
 void setLongueur(
       double);
};
#endif
```

Listing 65: Carre_V1.hh

```
#include "Carre_V1.hh"

void Carre::setLongueur
     (double l)
{
 _longueur = l;
}
```

Listing 66: Carre_V1.cpp

```
./a.out

Je suis de
  type forme
  générale
Je suis de
  type carre
Je suis de
  type rond
```

Redéfinition d'une fonction héritée

```
#ifndef __FORMEV2_HH__
#define __FORMEV2_HH__

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

class Forme
{
private:
 string _nom;

public:
 void setNom(const
      string&);
 void affiche();
};

#endif
```

Listing 61: Forme_V2.hh

```
#ifndef __RONDV2_HH__
#define __RONDV2_HH__
#include "Forme_V2.hh"

class Rond: public Forme
{
private:
  double _diametre;
public:
  void setDiametre(double);
  void affiche();
};
#endif
```

Listing 68: Rond_V2.hh

```
#include "Rond_V2.hh"

void Rond::setDiametre(double d)
      {
  _diametre = d;
}

void Rond::affiche() {
  cout << "Mon diametre est de "
       << _diametre << endl;
}
```

Listing 69: Rond_V2.cpp

```
#ifndef __CARREV2_HH__
#define __CARREV2_HH__
#include "Forme_V2.hh"

class Carre: public
     Forme
{
private:
  double _longueur;
public:
  void setLongueur(
       double);
  void affiche();
};
#endif
```

Listing 70: Carre_V2.hh

```
#include "Carre_V2.hh"
void Carre::setLongueur
     (double l) {
  _longueur = l;   }

void Carre::affiche() {
  cout << "Ma longueur
       est de "
       << _longueur
       << endl;    }
```

Listing 71: Carre_V2.cpp

Pour afficher les spécificités de chaque enfant, on va déclarer puis définir une fonction *affiche* dans chaque classe fille ...

En créant une fonction *affiche* dans la classe fille, on masque la fonction *affiche* de la classe mère.

On a donc bien une fonction *affiche* par classe, mais si on voulait aussi le nom de la forme dans les classes filles, il faudrait l'écrire en dur??

Ce n'est pas très orienté objet ça ...

Redéfinition d'une fonction héritée

```
#ifndef __FORMEV2_HH__
#define __FORMEV2_HH__

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

class Forme
{
private:
 string _nom;

public:
 void setNom(const
      string&);
 void affiche();
};

#endif
```

Listing 61: Forme_V2.hh

```
#ifndef __RONDV2_HH__
#define __RONDV2_HH__
#include "Forme_V2.hh"

class Rond: public Forme
{
private:
  double _diametre;
public:
  void setDiametre(double);
  void affiche();
};
#endif
```

Listing 64: Rond_V2.hh

```
#include "Rond_V2.hh"

void Rond::setDiametre(double d)
      {
  _diametre = d;
}

void Rond::affiche() {
  Forme::affiche();
  cout << "Mon diametre est de "
       << _diametre << endl;
}
```

Listing 68: Rond_V22.cpp

```
#ifndef __CARREV2_HH__
#define __CARREV2_HH__
#include "Forme_V2.hh"

class Carre: public
     Forme
{
private:
  double _longueur;
public:
  void setLongueur(
       double);
  void affiche();
};
#endif
```

Listing 66: Carre_V2.hh

```
#include "Carre_V2.hh"
void Carre::setLongueur
     (double l) {
  _longueur = l;  }

void Carre::affiche() {
  Forme::affiche();
  cout << "Ma longueur
       est de "
       << _longueur
       << endl;    }
```

Listing 65: Carre_V22.cpp

On va plutôt essayer d'appeler dans la fonction *affiche* fille, la fonction *affiche* parente.
Si on écrit simplement *affiche()* dans la fonction *affiche* de la classe fille, le compilateur va croire à un appel récursif de la fonction.
Il faut donc distinguer la fonction de la classe *Forme* avec la fonction de la classe fille, *Rond* ou *Carre*.
Pour cela, on va utiliser l'opérateur :: qui va nous permettre de se placer dans le contexte "parent" quand on le désirera.

Redéfinition d'une fonction héritée

```
#ifndef __FORMEV2_HH__
#define __FORMEV2_HH__

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

class Forme
{
private:
 string _nom;

public:
 void setNom(const
      string&);
 void affiche();
};

#endif
```

Listing 61: Forme_V2.hh

```
#ifndef __RONDV2_HH__
#define __RONDV2_HH__
#include "Forme_V2.hh"

class Rond: public Forme
{
private:
  double _diametre;
public:
  void setDiametre(double);
  void affiche();
};
#endif
```

Listing 64: Rond_V2.hh

```
#include "Rond_V2.hh"

void Rond::setDiametre(double d)
      {
  _diametre = d;
}

void Rond::affiche() {
  Forme::affiche();
  cout << "Mon diametre est de "
       << _diametre << endl;
}
```

Listing 64: Rond_V22.cpp

```
#ifndef __CARREV2_HH__
#define __CARREV2_HH__
#include "Forme_V2.hh"

class Carre: public
     Forme
{
private:
  double _longueur;
public:
  void setLongueur(
       double);
  void affiche();
};
#endif
```

Listing 66: Carre_V2.hh

```
#include "Carre_V2.hh"
void Carre::setLongueur
     (double l) {
  _longueur = l;  }

void Carre::affiche() {
  Forme::affiche();
  cout << "Ma longueur
       est de "
       << _longueur
       << endl;    }
```

Listing 61: Carre_V22.cpp

Nous avons maintenant un affichage adapté selon si on appelle la fonction *affiche* d'une classe *Forme*, *Rond* ou *Carre*, et ce, sans réécrire la fonction *affiche* de la classe *Forme*.

Une fois c'est suffisant!

```
./a.out

Je suis de
  type forme
  générale
Je suis de
  type carre
Ma longueur
  est de 5
Je suis de
  type rond
Mon diametre
  est de 3
```

Redéfinition d'une fonction héritée

```
#ifndef __FORMEV2_HH__
#define __FORMEV2_HH__

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

class Forme
{
private:
 string _nom;

public:
 void setNom(const
      string&);
 void affiche();
};

#endif
```

Listing 61: Forme_V2.hh

```
#ifndef __RONDV2_HH__
#define __RONDV2_HH__
#include "Forme_V2.hh"

class Rond: public Forme
{
private:
  double _diametre;
public:
  void setDiametre(double);
  void affiche();
};
#endif
```

Listing 64: Rond_V2.hh

```
#include "Rond_V2.hh"

void Rond::setDiametre(double d)
      {
  _diametre = d;
}

void Rond::affiche() {
  Forme::affiche();
  cout << "Mon diametre est de "
       << _diametre << endl;
}
```

Listing 64: Rond_V22.cpp

```
#ifndef __CARREV2_HH__
#define __CARREV2_HH__
#include "Forme_V2.hh"

class Carre: public
     Forme
{
private:
  double _longueur;
public:
  void setLongueur(
       double);
  void affiche();
};
#endif
```

Listing 66: Carre_V2.hh

```
#include "Carre_V2.hh"
void Carre::setLongueur
     (double l) {
  _longueur = l;  }

void Carre::affiche() {
  Forme::affiche();
  cout << "Ma longueur
       est de "
       << _longueur
       << endl;    }
```

Listing 61: Carre_V22.cpp

Et si on avait voulu appeler la fonction *affiche* de *Rond* héritée de *Forme*, et non la fonction *affiche* redéfinie?

Dans le fichier cpp contenant le *main*, il faut alors un peu modifier la syntaxe de l'appel de la fonction *r.Forme::affiche()*,

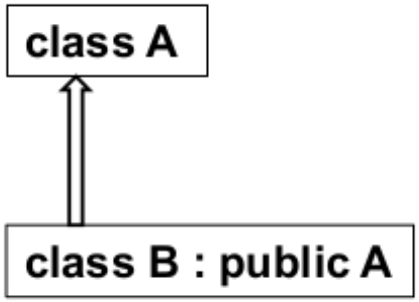
```
./a.out

Je suis de
  type forme
  générale
Je suis de
  type carre
Ma longueur
  est de 5
Je suis de
  type rond
```

Surcharge et définition

On fera attention sur un point:

Une fonction membre redéfinie dans une classe masque automatiquement les fonctions héritées, même si elles ont des paramètres différents. La recherche d'un symbole dans une classe se fait uniquement au niveau de la classe; si elle échoue, la compilation s'arrête en erreur, même si une fonction convenait dans une classe parente.

Constructeurs & destructeurs



Soit une classe A et une classe B héritant de A.

On va maintenant s'intéresser à la construction et à la destruction de la classe B et à son lien avec la classe A.

Pour construire la classe B, le compilateur va devoir d'abord créer la classe A. Il va donc appeler un constructeur de la classe A. Puis il va appeler celui de B.

Dans le cas où il n'y a pas de paramètre au constructeur de A, ou dans le cas d'un constructeur par défaut, il n'y a rien à faire, celui-ci est appelé automatiquement.

Les destructeurs, eux, sont appelés dans le sens inverse des appels des destructeurs, c'est-à-dire que B sera détruite avant A.

Constructeurs & destructeurs

```
#ifndef __AB1__HH__
#define __AB1__HH__
class A
{
private:
  int _a;
public:
  A(int);
};

class B : public A
{
private:
  int _b;
public:
  B(int, int);
};
#endif
```

Listing 62: AB_1.hh

```
#include "AB_1.hh"
A::A(int a)
{
  _a = a;
}

B::B(int a, int b) : A(a)
{
  _b = b;
}
```

Listing 63: AB_1.cpp

Pour construire un objet B, on va rencontrer une difficulté si on doit fournir des paramètres au constructeur de A. En effet, a priori, les paramètres fournis au constructeur de B sont sensés être utilisés par le constructeur de A.

Or ce dernier doit être appelé avant.

$\rightarrow$ Lors de la définition du constructeur de B, on utilise la même syntaxe que pour les objets membres.

Contrôle des accès

Nous avons déjà vu *private* et *public* comme statuts possibles pour une donnée ou une fonction membre. Il en existe en fait un troisième, lié à la notion d'héritage: *protected*.

On rappelle d'abord que:

- *private*: le membre n'est accessible qu'aux fonctions membres et aux fonctions amies d'une classe.
- *public*: le membre est accessible aux fonctions membres et fonctions amies, mais également à l'utilisateur de la classe.

Contrôle des accès

- Le mot-clé *protected* va quant à lui permettre de protéger une donnée ou une fonction membre d'un usage utilisateur, mais le membre reste accessible à partir de fonctions membres de classes dérivées.
- Il constitue donc un intermédiaire entre le concepteur d'une classe, qui a tout pouvoir sur elle, et un "simple" utilisateur de celle-ci.
- Il va permettre à un développeur qui souhaite étendre les fonctionnalités d'une classe d'avoir plus de pouvoir qu'un utilisateur extérieur de la classe.
- Il aurait été obligé sinon de passer par "l'interface de la classe", c'est-à-dire les fonctions membres "accesseur" et "modificateur", au risque d'une baisse de performance et de praticité.

Contrôle des accès

```
#ifndef __AB2__HH__
#define __AB2__HH__
class A
{
private:
  int _a;
protected:
  double _p;
public:
  A(int);
};

class B : public A
{
private:
  int _b;
public:
  B(int, int);
  void modifyA();
  void modifyP();
};
#endif
```

Listing 64: AB_2.hh

```
#include "AB_2.hh"
A::A(int a)
{
  _a = a;
}

B::B(int a, int b) : A(a)
{
  _b = b;
}

void B::modifyA()
{
  //_a = 1;
  //NE COMPILE PAS
}
void B::modifyP()
{
  _p = 1;
}
```

Listing 65: AB_2.cpp

On voit donc ici que la variable $\_p$, déclarée en *protected* dans la classe A, est accessible depuis la classe B.

La variable $\_a$, privée, reste inaccessible et un accès depuis B ne compilera pas.

Dérivation publique & dérivation privée

Depuis le début de cette partie sur l'héritage, nous avons déclaré qu'une classe dérivait d'une autre classe de la façon suivante:

```
class B: public A
```

En écrivant le mot-clé *public* ici, nous avons en fait déclaré une dérivation publique.

Une dérivation publique permet aux utilisateurs d'une classe dérivée d'accéder aux membres publics de la classe parente, comme s'ils faisaient partie de la classe fille.

Dérivation publique & dérivation privée

```
#ifndef __AB3__HH__
#define __AB3__HH__
class A
{
private:
  int _a;
protected:
  double _p;
public:
  A(int);
  void setA(int);
};

class B : private A
{
public:
  B(int);
};

class C : public A
{
public:
  C(int);
};
#endif
```

Listing 66: AB_3.hh

```
#include "AB_3.hh"

A::A(int a)
{
_a = a;
}

void A::setA(int a)
{
  _a=a;
}

B::B(int a):A(a) {}
C::C(int a):A(a) {}
```

Listing 67: AB_3.cpp

```
#include "AB_3.hh"

int main(void){
  A a(7);
  B b(5);
  C c(5);

  a.setA(3);
  //  b.setA(6);
  c.setA(6);
}
```

Listing 68: main_AB3.cpp

(Compilation avec la ligne *b.setA(6);*)

```
g++ AB_3.cpp main_AB3.cpp
In file included from main.cpp:1:0:
AB.hh: Dans la fonction 'int main()':
AB.hh:10:10: erreur :
'void A::setA(int)'
is inaccessible
void setA(int);
^
main.cpp:8:10: erreur : à l'intérieur
du contexte
b.setA(6);
^
main.cpp:8:10: erreur : 'A' is not
an accessible base of 'B'
```

La classe *B* hérite en privé de la classe *A*.

Elle masque alors son héritage à l'extérieur de la classe. Un utilisateur ne peut pas accéder à partir de *B* à des membres même publics de la classe *A*.

La classe *C*, par contre, hérite publiquement de la classe *A* et on peut utiliser les membres publics de la classe *A* de l'extérieur.

Dérivation protégée

Comme il existe le mot-clé *protected* pour les membres d'une classe, il existe aussi une notion de dérivation protégée.

Les membres de la classe parente seront ainsi déclarés comme protégés dans la classe fille et lors des dérivations successives.

On n'utilisera pas cette forme d'héritage dans le cours.

Classe de base & classe dérivée

- En Programmation Orientée Objet, on considère qu'un objet d'une classe dérivée peut "remplacer" un objet d'une classe de base. Un objet dérivé d'une classe A peut être utilisé quand celui d'une classe A est attendue.
- En effet, tout se qui se trouve dans une classe A se trouve également dans ses classes dérivées.
- En C++, on retrouve également cette notion, à une nuance près. Elle ne s'applique que dans le cas d'un héritage public.
- Il existe donc une conversion implicite d'une classe fille vers le type de la classe parente.

Conversion

```
#ifndef __AB4__HH__
#define __AB4__HH__

#include<iostream>
using namespace std;

class A
{
private:
  int _a;
public:
  A(int);
  void affiche();
};

class B : public A
{
private:
  int _b;
public:
  B(int, int);
  void affiche();
};
#endif
```

Listing 69: AB_4.hh

```
#include "AB_4.hh"

A::A(int a)
{
  cout << "Constr A"
       << endl;
  _a = a;
}

void A::affiche()
{
  cout << "A : ";
  cout << _a << endl;
}

B::B(int a, int b) : A(a)
{
  cout << "Constr B"
       << endl;
  _b = b;
}

void B::affiche()
{
  cout << "B : ";
  A::affiche();
  cout << _b << endl;
}
```

Listing 70: AB_4.cpp

```
#include "AB_4.hh"

int main()
{
  cout << "On construit
       a, b" << endl;
  A a(5);
  B b(6,7);

  cout << "On affiche a,
       b" << endl;
  a.affiche();
  b.affiche();
  cout << "on dit a = b"
       << endl;
  a = b;
  cout << "On affiche a,
       b" << endl;
  a.affiche();
  b.affiche();
}
```

Listing 71: main_AB4.cpp

Dans cet exemple - cas d'école - on déclare deux classes: la classe A et la classe B dérivant publiquement de A.

Dans le *main*, on construit deux objets: 'a' de type A, et 'b' de type B. Ensuite on dit $a = b$.

On a le droit de le faire car 'b' est de type B, dérivant de A, donc peut faire l'affaire dans le cas où un type A est attendu.

Ainsi, $a = b$ est accepté par le compilateur.
Dans ce cas, 'b' est converti en un objet de type A.

Conversion

```
#ifndef __AB4__HH__
#define __AB4__HH__

#include<iostream>
using namespace std;

class A
{
private:
  int _a;
public:
  A(int);
  void affiche();
};

class B : public A
{
private:
  int _b;
public:
  B(int, int);
  void affiche();
};
#endif
```

Listing 65: AB_4.hh

```
#include "AB_4.hh"

A::A(int a)
{
  cout << "Constr A"
       << endl;
  _a = a;
}

void A::affiche()
{
  cout << "A : ";
  cout << _a << endl;
}

B::B(int a, int b) : A(a)
{
  cout << "Constr B"
       << endl;
  _b = b;
}

void B::affiche()
{
  cout << "B : ";
  A::affiche();
  cout << _b << endl;
}
```

Listing 66: AB_4.cpp

```
#include "AB_4.hh"

int main()
{
  cout << "On construit
       a, b" << endl;
  A a(5);
  B b(6,7);

  cout << "On affiche a,
       b" << endl;
  a.affiche();
  b.affiche();
  cout << "on dit a = b"
       << endl;
  a = b;
  cout << "On affiche a,
       b" << endl;
  a.affiche();
  b.affiche();
}
```

Listing 67: main_AB4.cpp

```
 ./a.out
On construit a, b
Constr A
Constr A
Constr B
On affiche a, b
A : 5
B : A : 6
7
on dit a = b
On affiche a, b
A : 6
B : A : 6
7
```

Néanmoins, comme le montre l'affichage de notre programme, lorsque la variable *b* est convertie, elle perd une partie de ses données membres - celles de *B* - pour ne garder que les données membres héritées: celles de *A*. Ce qui est normal car *a* est de type *A*.

Comme on le voit, quand on réaffiche la variable *a*, sa donnée privée a bien pris la valeur de la partie *A* de *b*.

Conversion, pointeurs & références

```
#ifndef __AB4__HH__
#define __AB4__HH__

#include<iostream>
using namespace std;

class A
{
private:
  int _a;
public:
  A(int);
  void affiche();
};

class B : public A
{
private:
  int _b;
public:
  B(int, int);
  void affiche();
};
#endif
```

Listing 65: AB_4.hh

```
#include "AB_4.hh"

A::A(int a)
{
  cout << "Constr A"
       << endl;
  _a = a;
}

void A::affiche()
{
  cout << "A : ";
  cout << _a << endl;
}

B::B(int a, int b) : A(a)
{
  cout << "Constr B"
       << endl;
  _b = b;
}

void B::affiche()
{
  cout << "B : ";
  A::affiche();
  cout << _b << endl;
}
```

Listing 66: AB_4.cpp

```
#include "AB_4.hh"

int main()
{
  cout << "On construit b" <<
       endl;
  A* pa;
  B b(6,7);

  pa = &b;
  cout << "On affiche pa, b"
       << endl;
  pa->affiche();
  b.affiche();
  cout << "ra: reference sur
       b" << endl;
  A &ra = b;
  cout << "On affiche ra, b"
       << endl;
  ra.affiche();
  b.affiche();
}
```

Listing 68: main_AB42.cpp

```
./a.out

On construit b
Constr A
Constr B
On affiche pa, b
A : 6
B : A : 6
7
ra: reference
    sur b
On affiche ra, b
A : 6
B : A : 6
7
```

Le mécanisme avec les pointeurs et les références reste similaire.

Si on définit un pointeur sur A, on peut l'initialiser avec une adresse de type pointeur sur B.

De même, une référence sur A peut prendre l'adresse d'un objet de type B.

Conversion, pointeurs & références

```
#ifndef __AB4__HH__
#define __AB4__HH__

#include<iostream>
using namespace std;

class A
{
private:
  int _a;
public:
  A(int);
  void affiche();
};

class B : public A
{
private:
  int _b;
public:
  B(int, int);
  void affiche();
};
#endif
```

Listing 65: AB_4.hh

```
#include "AB_4.hh"

A::A(int a)
{
  cout << "Constr A"
       << endl;
  _a = a;
}

void A::affiche()
{
  cout << "A : ";
  cout << _a << endl;
}

B::B(int a, int b) : A(a)
{
  cout << "Constr B"
       << endl;
  _b = b;
}

void B::affiche()
{
  cout << "B : ";
  A::affiche();
  cout << _b << endl;
}
```

Listing 66: AB_4.cpp

```
#include "AB_4.hh"

int main()
{
  cout << "On construit b" <<
       endl;
  A* pa;
  B b(6,7);

  pa = &b;
  cout << "On affiche pa, b"
       << endl;
  pa->affiche();
  b.affiche();
  cout << "ra: reference sur
       b" << endl;
  A &ra = b;
  cout << "On affiche ra, b"
       << endl;
  ra.affiche();
  b.affiche();
}
```

Listing 57: main_AB42.cpp

```
./a.out

On construit b
Constr A
Constr B
On affiche pa, b
A : 6
B : A : 6
7
ra: reference
    sur b
On affiche ra, b
A : 6
B : A : 6
7
```

On commence à entrevoir une chose intéressante: un pointeur ou une référence peuvent pointer vers des objets qui ne sont pas de leur type, mais d'un type dérivé.

C'est plus intéressant que pour les objets, car le type d'un objet ne varie pas, mais si on l'initialise avec un autre type, les valeurs supplémentaires sont perdues lors de la conversion.

Conversion, pointeurs & références

```
#ifndef __AB4__HH__
#define __AB4__HH__

#include<iostream>
using namespace std;

class A
{
private:
  int _a;
public:
  A(int);
  void affiche();
};

class B : public A
{
private:
  int _b;
public:
  B(int, int);
  void affiche();
};
#endif
```

Listing 65: AB_4.hh

```
#include "AB_4.hh"

A::A(int a)
{
  cout << "Constr A"
       << endl;
  _a = a;
}

void A::affiche()
{
  cout << "A : ";
  cout << _a << endl;
}

B::B(int a, int b) : A(a)
{
  cout << "Constr B"
       << endl;
  _b = b;
}

void B::affiche()
{
  cout << "B : ";
  A::affiche();
  cout << _b << endl;
}
```

Listing 66: AB_4.cpp

```
#include "AB_4.hh"

int main()
{
  cout << "On construit b" <<
       endl;
  A* pa;
  B b(6,7);

  pa = &b;
  cout << "On affiche pa, b"
       << endl;
  pa->affiche();
  b.affiche();
  cout << "ra: reference sur
       b" << endl;
  A &ra = b;
  cout << "On affiche ra, b"
       << endl;
  ra.affiche();
  b.affiche();
}
```

Listing 57: main_AB42.cpp

```
./a.out

On construit b
Constr A
Constr B
On affiche pa, b
A : 6
B : A : 6
7
ra: reference
    sur b
On affiche ra, b
A : 6
B : A : 6
7
```

Alors qu'un pointeur (ou une référence) n'est qu'une adresse qui pointe vers un type en vérité plus "grand" que le type réel du pointeur.

Limitations

```
#ifndef __AB4__HH__
#define __AB4__HH__

#include<iostream>
using namespace std;

class A
{
private:
  int _a;
public:
  A(int);
  void affiche();
};

class B : public A
{
private:
  int _b;
public:
  B(int, int);
  void affiche();
};
#endif
```

Listing 65: AB_4.hh

```
#include "AB_4.hh"

A::A(int a)
{
  cout << "Constr A"
       << endl;
  _a = a;
}

void A::affiche()
{
  cout << "A : ";
  cout << _a << endl;
}

B::B(int a, int b) : A(a)
{
  cout << "Constr B"
       << endl;
  _b = b;
}

void B::affiche()
{
  cout << "B : ";
  A::affiche();
  cout << _b << endl;
}
```

Listing 66: AB_4.cpp

```
#include "AB_4.hh"

int main()
{
  A* a = new B(5,6);

  a->affiche();

  return 0;
}
```

Listing 58: main_AB43.cpp

```
  ./a.out
Constr A
Constr B
A : 5
```

On reprend les classes A et B de l'exemple précédent.

Dans le *main*, on déclare un pointeur sur A, qu'on initialise avec un objet de type B. C'est donc le constructeur de B qui est appelé. C'est valide comme on l'a vu précédemment.

On appelle alors la fonction *affiche* de notre pointeur et $\cdots$ déception, c'est la fonction *affiche* d'un objet de type A qui est appelée, et non pas celle d'un objet de type B, même si c'est bien un objet de ce type qui a été créé.

Limitations

```
#ifndef __AB4__HH__
#define __AB4__HH__

#include<iostream>
using namespace std;

class A
{
private:
  int _a;
public:
  A(int);
  void affiche();
};

class B : public A
{
private:
  int _b;
public:
  B(int, int);
  void affiche();
};
#endif
```

Listing 65: AB_4.hh

```
#include "AB_4.hh"

A::A(int a)
{
  cout << "Constr A"
       << endl;
  _a = a;
}

void A::affiche()
{
  cout << "A : ";
  cout << _a << endl;
}

B::B(int a, int b) : A(a)
{
  cout << "Constr B"
       << endl;
  _b = b;
}

void B::affiche()
{
  cout << "B : ";
  A::affiche();
  cout << _b << endl;
}
```

Listing 66: AB_4.cpp

```
#include "AB_4.hh"

int main()
{
  A* a = new B(5,6);

  a->affiche();

  return 0;
}
```

Listing 56: main_AB43.cpp

```
./a.out
Constr A
Constr B
A : 5
```

Cela vient du fait que les fonctions appelées sont "figées" lors de la compilation. Et pour le compilateur, un pointeur sur un objet correspond au type de cet objet, et ce sont donc les fonctions de la classe de cet objet qui seront appelées. Même si lors de l'exécution, l'objet pointé est en réalité "plus grand".

On verra dans la suite un mécanisme qui permet de passer outre cette difficulté.

Polymorphisme

Définition

Nous avons vu qu'un pointeur sur une classe parente pouvait recevoir l'adresse de n'importe quel objet dérivant la classe parente.

Il y avait néanmoins une contrainte: lorsque l'on appelait une fonction de l'objet pointé, c'était la fonction de la classe parente qui était appelée et pas la fonction de l'objet réellement pointé.

Cela provient du fait qu'à la compilation, le compilateur ne connaît pas le type de l'objet réellement pointé, et se base uniquement sur le type du pointeur. Il inclura, dans le code compilé, les appels aux fonctions de ce type, qui correspond au type de la classe parente. Il s'agit d'un typage statique.

Définition

C++ sait faire mieux que cela et permet un typage dynamique de ces objets.

Lors de la compilation, il sera alors mis en place un mécanisme permettant de choisir au moment de l'exécution la fonction qui sera appelée.

Il s'agit du Polymorphisme.

Des objets de types différents peuvent être pointés par le même pointeur et l'exécution du code se fait de manière cohérente avec les types réellement pointés.

Pour cela, nous allons voir un nouveau type de fonctions membres: les fonctions virtuelles.

Fonctions virtuelles

```
#ifndef __AB4__HH__
#define __AB4__HH__

#include<iostream>
using namespace std;

class A
{
private:
  int _a;
public:
  A(int);
  void affiche();
};

class B : public A
{
private:
  int _b;
public:
  B(int, int);
  void affiche();
};
#endif
```

Listing 65: AB_4.hh

```
#include "AB_4.hh"

A::A(int a)
{
  cout << "Constr A"
       << endl;
  _a = a;
}

void A::affiche()
{
  cout << "A : ";
  cout << _a << endl;
}

B::B(int a, int b) : A(a)
{
  cout << "Constr B"
       << endl;
  _b = b;
}

void B::affiche()
{
  cout << "B : ";
  A::affiche();
  cout << _b << endl;
}
```

Listing 66: AB_4.cpp

```
#include "AB_4.hh"

int main()
{
  A* pa;
  B b(6,7);

  pa = &b;
  pa->affiche();
}
```

Listing 57: main_AB44.cpp

```
./a.out

Constr A
Constr B
A : 6
```

Dans cet exemple, nous rappelons le problème.

Dans le *main*, on crée un pointeur sur un objet de type A. Mais celui-ci pointe en réalité sur un objet de type *B* par le jeu des conversions implicites.

Lorsqu'on appelle la fonction *affiche*, c'est celle de *A* qui est appelée et non celle de B.

Fonctions virtuelles

```
#ifndef __AB7__HH__
#define __AB7__HH__

#include<iostream>
using namespace std;

class A
{
private:
  int _a;
public:
  A(int);
  virtual void affiche();
        // fonction
  // virtuelle
};

class B : public A
{
private:
  int _b;
public:
  B(int, int);
  void affiche(); //
       définie pour B
};
#endif
```

Listing 58 : AB_7.hh

```
#include "AB_7.hh"

A::A(int a)
{
  _a = a;
}

void A::affiche()
{
  cout << "A : ";
  cout << _a << endl;
}

B::B(int a, int b) : A(a)
{
  _b = b;
}

void B::affiche()
{
  A::affiche();
  cout << "B : ";
  cout << _b << endl;
}
```

Listing 59 : AB_7.cpp

```
#include "AB_7.hh"

int main()
{
  A* pa;
  B b(6,7);

  pa = &b;
  pa->affiche();
}
```

Listing 60 : main_AB7.cpp

```
./a.out

A : 6
B : 7
```

On se contente ensuite d'ajouter le mot clé *"virtual"* à la déclaration de la fonction *affiche*.

Lors de l'appel dans *main*, c'est maintenant la fonction de *B* qui est appelée!

Que s'est-il passé?

Fonctions virtuelles

Le mot clé "virtual" placé dans la déclaration d'une fonction permet de rendre une fonction "virtuelle".

Même si visuellement, il semblerait que peu de chose ait changé dans le code, en vérité un système relativement complexe a été mis en place par le compilateur pour obtenir un comportement cohérent dans le cas du polymorphisme : c'est en effet la fonction membre du type réel de l'objet qui est appelée et plus celle du type du pointeur.

Fonctions virtuelles - Limitations

Les fonctions virtuelles ont néanmoins quelques règles à respecter:

- Seule une fonction membre peut être virtuelle. Les fonctions "ordinaires" ou amies sont exclues de ce mécanisme.
- Un constructeur ne peut pas être virtuel. En effet, un constructeur est appelé pour construire une classe. Cela n'aurait pas trop de sens qu'en réalité, il construise une autre classe $\cdots$
- En revanche, un destructeur peut être virtuel.

Fonctions virtuelles - Destructeur

```
#ifndef __AB8__HH__
#define __AB8__HH__

#include<iostream>
using namespace std;

class A
{
private:
  int _a;
public:
  A(int);
  ~A();
};

class B: public A
{
private:
  int _b;
public:
  B(int, int);
  ~B();
};
#endif
```

Listing 61: AB_8.hh

```
#include "AB_8.hh"

A::A(int a)
{
  _a = a;
}

A::~A()
{
  cout << "Destr A" << endl;
}

B::B(int a, int b) : A(a)
{
  _b = b;
}

B::~B()
{
  cout << "Destr B" << endl;
}
```

Listing 62: AB_8.cpp

```
#include "AB_8.hh"

int main()
{
  A* pa = new B(2,3);

  delete pa;
}
```

Listing 63: main_AB8.cpp

```
./a.out

Destr A
```

Que se passe-t-il lorsqu'un destructeur n'est pas virtuel dans cet exemple?

On construit un objet de type *B* avec *new*. Il faudra donc le détruire.

Mais son adresse est stockée dans un pointeur de type A*.

C'est donc le destructeur de *A* qui est appelé! Et l'objet *B* n'est pas entièrement détruit ...

Fonctions virtuelles - Destructeur

```hh
#ifndef __AB9__HH__
#define __AB9__HH__

#include<iostream>
using namespace std;

class A
{
private:
  int _a;
public:
  A(int);
  virtual ~A();
};

class B: public A
{
private:
  int _b;
public:
  B(int, int);
  ~B();
};
#endif
```

Listing 64: AB_9.hh

```cpp
#include "AB_9.hh"

A::A(int a)
{
  _a = a;
}

A::~A()
{
  cout << "Destr A" << endl;
}

B::B(int a, int b) : A(a)
{
  _b = b;
}

B::~B()
{
  cout << "Destr B" << endl;
}
```

Listing 65: AB_9.cpp

```cpp
#include "AB_9.hh"

int main()
{
  A* pa = new B(2,3);

  delete pa;
}
```

Listing 66: main_AB9.cpp

```
./a.out

Destr B
Destr A
```

On déclare maintenant le destructeur de A comme étant virtuel.

Lors de l'exécution, c'est donc bien le destructeur de B qui est appelé.

Intérêt

```
#ifndef __AB7__HH__
#define __AB7__HH__

#include<iostream>
using namespace std;

class A
{
private:
  int _a;
public:
  A(int);
  virtual void affiche(); //
      fonction
  // virtuelle
};

class B : public A
{
private:
  int _b;
public:
  B(int, int);
  void affiche(); // définie
      pour B
};
#endif
```

Listing 52 : AB_7.hh

```
#include "AB_7.hh"

int main()
{
  A* tab[2]; // tableau de 2
      pointeurs sur A
  tab[0] = new A(5);
  tab[0]->affiche();

  tab[1] = new B(7,9);
  tab[1]->affiche();

  delete tab[0];
  delete tab[1];
}
```

Listing 61 : main_AB72.cpp

Un des intérêts du polymorphisme est de pouvoir créer des tableaux de pointeurs sur une classe.

Ici, à l'intérieur du tableau *tab*, on peut avoir une adresse d'un objet *A* ou d'un objet B, ce qui permet de manipuler différents types de données.

On n'oublie pas le *delete* ici (sans [])!

```
#include "AB_7.hh"

A::A(int a)
{
  _a = a;
}

void A::affiche()
{
  cout << "A : ";
  cout << _a << endl;
}

B::B(int a, int b) : A(a)
{
  _b = b;
}

void B::affiche()
{
  A::affiche();
  cout << "B : ";
  cout << _b << endl;
}
```

Listing 53 : AB_7.cpp

```
./a.out

A : 5
A : 7
B : 9
```

Classes abstraites - Fonctions virtuelles pures

```
#ifndef __AB12__HH__
#define __AB12__HH__

#include<iostream>
using namespace std;

class A
{
private:
  int _a;
public:
  virtual void affiche() = 0; //
       fonction virtuelle pure
};

class B : public A
{
private:
  int _b;
public:
  void affiche(); // définie pour B
};
#endif
```

Listing 58: AB_11.hh

Les classes abstraites en POO sont des classes qui n'ont pas pour but d'être instanciées directement.

Il s'agira alors pour l'utilisateur de la classe de créer une classe et d'hériter de celle-ci en créant le code supplémentaire si besoin.

Pour cela, le C++ introduit des fonctions virtuelles dites "pures", c'est-à-dire qu'on ne donne pas de définition à cette fonction et c'est la classe fille qui devra définir cette fonction.

Classes abstraites - Fonctions virtuelles pures

```
#ifndef __AB12__HH__
#define __AB12__HH__

#include<iostream>
using namespace std;

class A
{
private:
  int _a;
public:
  virtual void affiche() = 0; //
       fonction virtuelle pure
};

class B : public A
{
private:
  int _b;
public:
  void affiche(); // définie pour
       B
};
#endif
```

Listing 59: AB_12.hh

```
#include "AB_12.hh"

int main()
{
  A* a = new A(); // ko!
  a->affiche();

  A* b = new B();
  b->affiche();
}
```

Listing 60: main_AB12.cpp

```
  g++ AB_7.cpp main_AB12.cpp -o main_AB12
main_AB12.cpp: In function 'int main()':
main_AB12.cpp:5:16: error: invalid
new-expression of abstract class type 'A'
    5 |   A* a = new A();
      |                ^
In file included from main_AB12.cpp:1:
AB_12.hh:7:7: note:
because the following virtual functions
are pure within 'A':
    7 | class A
      |       ^
      AB_12.hh:12:16: note:
      'virtual void A::affiche()'
      12 |   virtual void affiche() = 0;
      // fonction virtuelle pure
```

Dans cet exemple, dans la fonction *main*, on essaie d'instancier un objet de type A.

Le compilateur refuse car on essaie d'instancier une classe abstraite.

Classes abstraites - Fonctions virtuelles pures

```
#ifndef __AB12__HH__
#define __AB12__HH__

#include<iostream>
using namespace std;

class A
{
private:
  int _a;
public:
  virtual void affiche() = 0; //
       fonction virtuelle pure
};

class B : public A
{
private:
  int _b;
public:
  void affiche(); // définie pour
       B
};
#endif
```

Listing 57 : AB_12.hh

```
#include "AB_12.hh"

int main()
{
  A* a = new A(); // ko!
  a->affiche();

  A* b = new B();
  b->affiche();
}
```

Listing 58 : main_AB12.cpp

```
  g++ AB_7.cpp main_AB12.cpp -o main_AB12
main_AB12.cpp: In function 'int main()':
main_AB12.cpp:5:16: error: invalid
new-expression of abstract class type 'A'
    5 |   A* a = new A();
      |                ^
In file included from main_AB12.cpp:1:
AB_12.hh:7:7: note:
because the following virtual functions
are pure within 'A':
    7 | class A
      |       ^
      AB_12.hh:12:16: note:
      'virtual void A::affiche()'
      12 |   virtual void affiche() = 0;
      // fonction virtuelle pure
```

Une classe abstraite est une classe contenant au moins une fonction virtuelle pure.

On ne peut pas instancier cette classe directement.

La classe *B* hérite publiquement de *A* et redéfinit la fonction *affiche*. On pourra alors instancier un objet de type *B*.

Classes abstraites - Fonctions virtuelles pures

```
#ifndef __AB12__HH__
#define __AB12__HH__

#include<iostream>
using namespace std;

class A
{
private:
  int _a;
public:
  virtual void affiche() = 0; // fonction
        virtuelle pure
};

class B : public A
{
private:
  int _b;
public:
  void affiche(); // définie pour B
};
#endif
```

Listing 57: AB_12.hh

```
#include "AB_12.hh"

int main()
{
  A* b = new B();
  b->affiche();
}
```

Listing 59: main_AB12ok.cpp

```
  ./a.out

B : 0
```

STANDARD TEMPLATE LIBRARY

LA STL

Le C++, tout comme le C et de nombreux langages, possède une librairie disponible dés l'installation, connue sous le nom de librairie standard.

On connaît déjà quelques éléments de cette librairie notamment par les *#include <cmath> <iostream> <string>* etc.

Une partie de cette librairie concerne des versions stables, optimisées et testées de conteneurs, d'itérateurs et d'algorithmes sur ces conteneurs. Cette sous-librairie est connue sous le nom de *Standard Template Library* ou *STL*.

C'est une librairie puissante conçue à base de classes templates, dont nous allons maintenant étudier quelques éléments.

LA STL

LA CLASSE VECTOR

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

int main()
{
  double tab[5] = {1, 2, 3, 4, 5};
  std::vector<double> v(tab, tab + 5);
  v.push_back(10);
  std::cout << v[0] << std::endl;
  std::cout << v.back() << std::endl;
  v.pop_back();
  std::cout << v.back() << std::endl;
}
```

Nous avons déjà plus ou moins construit une classe *vector* telle qu'implémentée dans la STL.

On en trouve la documentation complète sur *cplusplus.com*

http://www.cplusplus.com/reference/vector/vector/

Ci-contre, un exemple d'utilisation.

LA STL

LA CLASSE LIST

```
#include <iostream>
#include <list>

int main()
{
  double tab[5] = {1, 2, 3, 4, 5};
  std::list<double> v(tab, tab + 5);
  v.push_back(10);
  /* v[0] = 5; N'EXISTE PAS POUR UNE LISTE */
  std::cout << v.front() << std::endl;
  std::cout << v.back() << std::endl;
  v.pop_back();
  std::cout << v.back() << std::endl;
}
```

La classe *std::list* s'apparente en terme d'utilisation à la classe *std::vector*.

Il y a néanmoins des différences d'implémentation entre un *vector* et une *list* qui font qu'on préfèrera l'une ou l'autre classe en fonction de l'utilisation de celle-ci.

LA STL

LA CLASSE LIST

```
#include <iostream>
#include <list>

int main()
{
  double tab[5] = {1, 2, 3, 4, 5};
  std::list<double> v(tab, tab + 5);
  v.push_back(10);
  /* v[0] = 5; N'EXISTE PAS POUR UNE LISTE */
  std::cout << v.front() << std::endl;
  std::cout << v.back() << std::endl;
  v.pop_back();
  std::cout << v.back() << std::endl;
}
```

La classe *list* telle qu'implémentée dans la STL est ce qu'on appelle une liste **doublement chaînée**.

On remarque qu'il n'y a pas de surcharge de l'opérateur [].

En effet, une liste a des avantages par rapports aux vecteurs/tableaux/espaces mémoire contigus : si cet ensemble d' éléments est appelé à grandir/diminuer ou si on veut insérer un élément dans le tableau.

LA STL

LA CLASSE LIST

```
#include <iostream>
#include <list>

int main()
{
  double tab[5] = {1, 2, 3, 4, 5};
  std::list<double> v(tab, tab + 5);
  v.push_back(10);
  /* v[0] = 5; N'EXISTE PAS POUR UNE LISTE */
  std::cout << v.front() << std::endl;
  std::cout << v.back() << std::endl;
  v.pop_back();
  std::cout << v.back() << std::endl;
}
```

Par contre, pour accéder au n-ième élément d'une liste, il faut la parcourir du début, élément par élément.

Le temps d'accès est donc linéaire par rapport à la taille de la liste.

En notation de Landau : O(n)

Dans un tableau, le temps d'accès est constant. O(1)

LA STL

LA CLASSE LIST

```
#include <iostream>
#include <list>

int main()
{
  double tab[5] = {1, 2, 3, 4, 5};
  std::list<double> v(tab, tab + 5);
  v.push_back(10);
  /* v[0] = 5; N'EXISTE PAS POUR UNE LISTE */
  std::cout << v.front() << std::endl;
  std::cout << v.back() << std::endl;
  v.pop_back();
  std::cout << v.back() << std::endl;
}
```

Pour agrandir une liste, par contre, le temps est constant, alors qu'il est difficile d'augmenter le nombre d' éléments d'un tableau…

On doit copier les éléments de l'ancien tableau dans le nouveau avant de libérer l'ancien espace mémoire etc.

LA STL

LA CLASSE MAP

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

int main()
{
  std::map<std::string, double> m;
  m["Charles"] = 5.0;
  m["Toto"] = 10;
  m["Sally"] = 15;

  std::cout << m.size() << std::endl;
  std::cout << m["Sally"] << std::endl;

}
```

La classe *std::map* est un autre conteneur qui permet d'associer une clé à une valeur.

On parle de tableaux associatifs ou de dictionnaires.

Comme c'est une classe template, le type des clés/valeurs peut être attribué lors de l'écriture du code.

L'opérateur [] est cette fois-ci surchargé pour accéder à la valeur d'une clé.

LA STL

Il existe d'autres conteneurs dans la STL et nous ne les verrons pas tous.

On pourra néanmoins se reporter à la référence sur *cplusplus.com* pour connaître les autres types en fonction des besoins.

LA STL

LES ITERATEURS

Pour homogénéiser les actions possibles sur les différents conteneurs, il est apparu la notion d'itérateurs. Un itérateur, qui peut être vu comme une généralisation de la notion de pointeur, permet de parcourir un conteneur.

Il possède ces propriétés :

- À chaque instant, un itérateur pointe vers une valeur qui désigne un élément donné du conteneur.
- Un itérateur peut être incrémenté par l'opérateur ++.
- Il peut être déréférencé, c'est-à-dire que l'utilisation de l'opérateur * permet (comme sur un pointeur) d'accéder à la valeur courante de l'itérateur.
- Deux itérateurs sur un même conteneur peuvent être comparés.

LA STL

ITERATEUR SUR VECTEUR

```
#include <vector>
#include <iostream>
int main()
{
  std::vector<double> v(5);
  std::vector<double>::iterator iv;
  double i = 0;
  for (iv = v.begin();
        iv != v.end();
        ++iv)
  {
    *iv = i++;
  }
  for (iv = v.begin();
        iv != v.end();
        ++iv)
  {
    std::cout << *iv << ";";
  }
}
```

Un itérateur (sur vecteur) se déclare de la façon suivante :

std::vector<double>::iterator iv;

La première partie est un vector de doubles.

Que signifie la deuxième partie ?

Les objets itérateurs sont définis à l'intérieur du conteneur qu'ils itèrent.

On dit qu'il s'agit de classes imbriquées, ou « nested class ». En effet, on peut déclarer une classe à l'intérieur d'une autre classe. Pour s'adresser à cette classe imbriquée, on va faire appel à l'opérateur « :: ».

LA STL

ITERATEUR SUR VECTEUR

```
#include <vector>
#include <iostream>
int main()
{
  std::vector<double> v(5);
  std::vector<double>::iterator iv;
  double i = 0;
  for (iv = v.begin();
       iv != v.end();
       ++iv)
  {
    *iv = i++;
  }
  for (iv = v.begin();
       iv != v.end();
       ++iv)
  {
    std::cout << *iv << ";";
  }
}
```

Tous les conteneurs fournissent deux valeurs particulières, sous forme d'itérateur, permettant de fournir un début et une fin.

Ainsi, on initialise l'itérateur avec comme valeur l'itérateur pointant sur le début du vecteur.

Comme on peut comparer l'égalité ou l'inégalité de deux itérateurs, on arrête la boucle *for* quand l'itérateur est égal à l'itérateur pointant sur la fin du vecteur.

Attention : *end()* est un itérateur particulier qui ne pointe pas sur le dernier élément, mais un cran plus loin. De sorte que pour un conteneur vide, *begin()* sont égaux.

et end()

LA STL

ITERATEUR SUR VECTEUR

```
#include <vector>
#include <iostream>
int main()
{
  std::vector<double> v(5);
  std::vector<double>::iterator iv;
  double i = 0;
  for (iv = v.begin();
       iv != v.end();
       ++iv)
  {
    *iv = i++;
  }
  for (iv = v.begin();
       iv != v.end();
       ++iv)
  {
    std::cout << *iv << ";";
  }
}
```

Pour passer à l'élément suivant, l'itérateur peut être incrémenté grâce à l'opérateur « ++ ».

~~**Attention :** cela ne veut pas dire qu'il existe un opérateur « -- ».~~

On remarquera aussi que l'on écrit ++iv et non iv++.

En effet, la notation postfixée entraîne plus d'opérations que la notation préfixée.

LA STL

ITERATEUR SUR VECTEUR

```
#include <vector>
#include <iostream>
int main()
{
  std::vector<double> v(5);
  std::vector<double>::iterator iv;
  double i = 0;
  for (iv = v.begin();
       iv != v.end();
       ++iv)
  {
    *iv = i++;
  }
  for (iv = v.begin();
       iv != v.end();
       ++iv)
  {
    std::cout << *iv << ";";
  }
}
```

Pour accéder à la valeur pointée par l'itérateur, on le déréférence grâce à l'opérateur « * ».

On peut ainsi modifier ou accéder à cette valeur.

Pour les vecteurs, il existe aussi une surcharge de l'opérateur « [] » permettant d'accéder immédiatement à un élément donné.

Cela entraîne la possibilité d'une arithmétique des itérateurs sur vecteur.

Ainsi, dans l'exemple ci contre,

*(iv+i) est équivalent à v[i]

LA STL

ITERATEUR SUR LIST

```
#include <list>
#include <iostream>

int main()
{
  std::list<int> lis;
  lis.push_back(2);
  lis.push_back(3);
  std::list<int>::iterator il;
  for (il = lis.begin(); il != lis.end(); ++il)
     std::cout << *il << std::endl;
  // NE COMPILE PAS
  //std::cout << *(lis.begin() + 1) << std::endl;
}
```

Les itérateurs sur list se déclarent de la même façon que les itérateurs sur vector.

De même, une liste possède deux fonctions particulières begin() et end() qui permettent d'initialiser et de comparer un itérateur sur list.

De même, l'opérateur de déréférencement permet d'accéder à la valeur pointée par l'itérateur.

LA STL

ITERATEUR SUR LIST

```
#include <list>
#include <iostream>

int main()
{
  std::list<int> lis;
  lis.push_back(2);
  lis.push_back(3);
  std::list<int>::iterator il;
  for (il = lis.begin(); il != lis.end(); ++il)
     std::cout << *il << std::endl;
  // NE COMPILE PAS
  //std::cout << *(lis.begin() + 1) << std::endl;
}
```

Par contre, il n'y a pas d'opérateur [] sur une liste.

Une liste ne s'utilise pas comme un vector.

Accéder directement à un élément n'est pas souhaitable car c'est une procédure lente.

Il n'y a donc pas d'arithmétique sur les itérateurs de list : on garde une cohérence entre le conteneur et son itérateur.

LA STL

ALGORITHMES – FOR_EACH

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

void affiche(int i){
    std::cout << i << ";";
}

int main()
{
    int myints[] = {32,71,12,45,26,80,53,33};
    std::vector<int> v(myints, myints+8);
    std::for_each(v.begin(), v.end() - 2, affiche);
}
```

La STL contient également des algorithmes. Ces fonctions template travaillent sur des conteneurs et des itérateurs.

On présente ici la fonction *for_each* qui exécute une fonction (ou un foncteur) passée en paramètre, entre deux bornes d'un conteneur – c'est-à-dire entre deux valeurs d'itérateurs.

```
$ ./a.out
32;71;12;45;26;80;
```

LA STL

ALGORITHMES – COUNT/COUNT_IF

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

bool est_pair(int i){
    return i % 2 == 0;
}

int main()
{
    int myints[] = {32,71,12,45,26,80,53,33,32};
    vector<int> v(myints, myints+9);
    int c = count_if(v.begin(), v.end(),  est_pair);
    cout << "Nbr pair :" << c << endl;
    cout << "32 :"<< count(v.begin(), v.end(), 32)
            << endl;
}
```

La fonction ***count*** permet de compter le nombres d'occurrences d'un élément entre deux itérateurs d'un conteneur.

La fonction ***count_if*** permet de compter le nombre d'éléments satisfaisant une condition.

Cette condition prend la forme d'une fonction (ou d'un foncteur) - renvoyant un bool et prenant en paramètre un élément du conteneur - passée en paramètre.

```
$ ./a.out
Nbr pair :5
32 :2
```

LA STL

ALGORITHMES – SORT

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


int main()
{
    int myints[] = {32,71,12,45,26,80,53,33,32};
    vector<int> v(myints, myints+9);
    sort(v.begin(), v.end());
    for (int i = 0; i < v.size(); ++i)
        cout << v[i] << "-";
    cout << endl;
}
```

La fonction *sort* permet d'appliquer une fonction de tri sur un conteneur entre deux valeurs d'itérateurs.

Le troisième argument - optionnel - est une fonction renvoyant vrai si deux éléments du tableau sont dans la bonne position.

```
$ ./a.out
12-26-32-32-33-45-53-71-80-
```

LA STL

STL ET POLYMORPHISME

```
#ifndef _AB_HH_
#define _AB_HH_
#include<iostream>
class A
{
private:
   int _a;
public:
   virtual void affiche() ;
};

class B : public A
{
private:
   int _b;
public:
   void affiche();
};

#endif
```

Fonction virtuelle

AB.hh

```
#include "AB.hh"
#include <vector>
using namespace std;

int main()
{
   vector<A*> tab;  // tableau de  A*
  tab.push_back(new A() );
  tab[0]->affiche();

  tab.push_back(new B( ));
  tab[1]->affiche();

delete tab[0];
delete tab[1];
}
```

main.cpp

```
#include "AB.hh"
using namespace std;

void A::affiche() {
cout << "classe A: " << _a << endl;  }

void B::affiche() {
cout << "classe B: "  << _b << endl; }
```

AB.cpp

Ici on utilise un *vector* pour stocker des pointeurs sur A.

Comme vu précédemment, à l'intérieur du tableau *tab*, on peut avoir une adresse d'un objet A ou d'un objet B, ce qui permet de manipuler différents types de données.