C++
Master 1 IM

Roland RUELLE

Année 2016/2017

PRÉSENTATION DU MODULE

12 semaines : 12 cours / 12 TPs

Plan

Validation du module

Objectifs

Bibliographie

CONTACTS

- Au LJAD
  bureau 612 sur rendez-vous

- rruelle@unice.fr
  écrire avec le préfixe [M1 IM] dans l'objet du mail

- Supports de cours
  http://math.unice.fr/~rruelle

PLAN DES COURS

- 12 cours sur le semestre
- Plan des cours prévisionnel
  - Aspects impératifs du C++, éléments de syntaxe, structures de contrôles, fonctions, pointeurs, tableaux et références.
  - Structures de données et types utilisateurs
  - Objets & classes, constructeurs, destructeurs
  - Surdéfinition d'opérateurs
  - Héritage simple, Héritage multiple, polymorphisme
  - Template et métaprogrammation
  - Entrées/sorties
  - Standard Template Library (STL)
  - Outils : makefile / débuggeurs …

VALIDATION

Pour valider le module :

Contrôle des connaissances :

- Un TP noté : 1/3 de la note finale
- Un examen de fin de semestre : 2/3 de la note finale

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, Denis-M Ritchie
- Le site de référence : http://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.

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.

Impératifs : 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.

HELLOWORLD.CPP

Comme dans la plupart des cours de programmation, on commence par un HelloWorld : Programme le plus simple qui affiche un message à l'écran.

Dans un éditeur de texte quelconque. (notez ici la coloration syntaxique)

```
#include <iostream>

int main()
{
  std::cout << "Hello World" << std::endl;

}
```

COMPILATION ET EXÉCUTION DU HELLOWORLD

```
roland@DESKTOP-M1EA3EP ~
$ g++ HelloWorld.cpp -o HelloWorld

roland@DESKTOP-M1EA3EP ~
$ ./HelloWorld.exe
Hello World
```

LE HELLOWORLD LIGNE À LIGNE

#include <iostream>

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

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

LE HELLOWORLD LIGNE À LIGNE

int main()

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 apparaitre dans un programme. C'est une convention.

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 servir dans l'environnement appelant notre programme une valeur de bonne exécution ou un code erreur.

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

LE HELLOWORLD LIGNE À LIGNE

```
{
  std::cout << "Hello World" << std::endl;
}
```

Le corps de la fonction `main`.
Le corps d'une fonction se situe après l'entête de celle-ci et entre deux accolades. Elles définissent en fait le bloc d'instruction de la fonction.
Ici, il n'y a qu'une seule instruction, se terminant par un `;`
`std::cout` peut être vu comme 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 chaine de caractère, c'est-à-dire un emplacement mémoire contigüe contenant un caractère par octet de mémoire, et se terminant conventionnellement par le caractère nul. Nous verrons cela plus en détail lorsque nous reparlerons des types de données.
`std::endl` demande au flux de passer à la ligne suivante.

COMPILATION ET EXÉCUTION DU HELLOWORLD

```
roland@DESKTOP-M1EA3EP ~
$ g++ HelloWorld.cpp -o HelloWorld

roland@DESKTOP-M1EA3EP ~
$ ./HelloWorld.exe
Hello World
```

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 `a.out` (sous Linux, a.exe sous windows), sans cet argument.

`./HelloWorld.exe` dans un shell, permet d'exécuter notre 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 textes 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 parlé, 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 le déplace dans un fichier header suffixé en .h ou .hh qui sera inclus via une directive préprocesseur.

Enfin viennent les définitions des fonctions, le code du programme à proprement parlé, dans un fichier suffixé en .cpp pour le C++, .c pour le C.

ORGANISATION D'UN PROGRAMME EN C++

```cpp
#include <iostream>

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

//      point d'entrée de notre programme
int main()
{
  int a, b;

  std::cout << "Donnez deux entiers" << std::endl;
  std::cin >> a >> b;
  std::cout << a << " + " << b << "=" << somme(a, b) << std::endl;

  return 0;
}

// somme retourne la somme de a et b deux réels fournis en paramétres.
double somme(double a , double b)
{
  return a+b;
}
```

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 signifie 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.

ORGANISATION D'UN PROGRAMME EN C++

```
#ifndef __SOMME_HH__
#define __SOMME_HH__

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

#endif
```

Un tel fichier header ressemblerait à celui-ci .
On voit apparaitre 3 nouvelles directives préprocesseurs ainsi que la déclarations de la fonction somme.
On remarquera aussi que ce fichier NE CONTIENT PAS DE CODE, seulement des déclarations.

```
#ifndef __SOMME_HH__
#define __SOMME_HH__
```

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

DÉCLARATIONS

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.

Exemple de déclarations :

```
int i; // On déclare une variable de type entier.

float x; // On déclare une variable de type flottant (approximation
d'un nombre réel)

const int N = 5; // On déclare une constante N de type entier dont
                 //  la valeur est 5.
```

VARIABLES

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

```
#include <iostream>

using namespace std;

int main()
{
  int a;

  a = 0;
  cout << "a vaut : " << a << endl;
  a = 5;
  cout << "a vaut à present : " << a << endl;

}
```

Représentation en mémoire de a = 0

		a							
		00000000	00000000	00000000	00000000				
…	101	102	103	104	105	106	107	108	…

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

		a							
		00000000	00000000	00000000	00000101				
…	101	102	103	104	105	106	107	108	…

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 a un type donné tout au long de son existence.

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

TYPES DE DONNÉES NUMÉRIQUES

```cpp
#include <iostream>

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 correspondantes en mémoire est allouée

  a = 0;            //On attribue à a la valeur 0, jusqu'à maintenant, il n'y a pas de règle quant à sa 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 10.000
  reel2 = 0.0001;
  std::cout << "a : " << a << " " << std::endl
            << "Interessant : "
            << "b : " << b << std::endl                    // HA !
            << "c ; " << c << " " << std::endl;
  std::cout << reel1 << std::endl;
  std::cout << reel2 << std::endl;

}
```

TYPES DE DONNÉES ALPHABETIQUES

	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	

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 caractère particulier.

Il y en a beaucoup d'autre, Unicode par exemple.

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	

Ainsi, `c = 'a';`

Donne la valeur 97 à la variable c.

Lorsqu'on affiche c. Ce n'est pas 97 qui apparait, mais 'a' car le compilateur sait qu'on veut afficher un caractère et non sa valeur, grâce à son type.

```
c = 'a' + 1;
```

Mais il s'agit bien d'un nombre, comme l'indique cet exemple, valide en C++, qui affiche le caractère suivant 'a' dans la table.
Soit 'b' !

OPERATEURS

OPÉRATEURS

Il y a des nombreux opérateurs en C++

Classiques :

Arithmétiques, relationnels, logiques

Moins classiques :

Manipulation de bits

Et des opérateurs originaux, d'affectation, d'incrémentation

OPÉRATEURS ARITHMÉTIQUES

C++ dispose d'opérateurs classiques binaires (deux opérandes)

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 à priori définis que pour des opérandes de même type parmi

Int, long int, float, double, et 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>

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

  double c = a +b;
  std::cout << sizeof(a) << ": " << a << std::endl;
  std::cout << sizeof(b) << ": " << b << std::endl;
  std::cout << sizeof(c) << ": " << c << std::endl;

}
```

Aura comme sorti :

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

OPÉRATEURS ARITHMÉTIQUES
OPÉRATEUR %

Reste de la division entière : n'est défini que sur les entiers en C++ (ce n'est pas le cas en Java par exemple)

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.

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++.

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 ET DE COMPARAISONS

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 DES OPÉRATEURS D'ÉGALITÉ SUR LES DOUBLES.

```cpp
#include <iostream>

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

  if(a + b == c) {
    std::cout << "a+b=c" << std::endl;
  }
  else {
    std::cout << "a+b != c" << std::endl;
  }

  if(a == c-b) {
    std::cout << "a = c-b" << std::endl;
  }
  else {
    std::cout << "a != c-b" << std::endl;
  }

}
```

OPÉRATEURS LOGIQUES

En C++ il y a trois opérateurs logiques : **et** (noté &&) **ou** noté (||) et **non** (noté !)

Ces opérateurs travaillent sur des valeurs numériques de tout type avec la simple convention

Nulle → faux

Autre que nulle → vrai

COURT-CIRCUIT DANS L'ÉVALUATION DES OPÉRATEURS LOGIQUES

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

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

On peut - mais ce n'est qu'un exemple - protéger une portion de code :

```
if (ptr != 0 && *ptr == 8)
```

Dans cet exemple, on vérifie d'abord que ptr n'est pas nul avant de le déréférencer pour comparer sa valeur pointée en mémoire.

OPÉRATEURS D'AFFECTATION ÉLARGIE

C++ permet d'alléger la syntaxe de certaines expressions en donnant la possibilité d'utiliser 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

+= -= *= /= %=

|= ^= &= <<= >>=

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 connaitre la taille en mémoire de l'objet passé en paramétre.

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 CONTRÔLES

STRUCTURES DE CONTRÔLES

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 langage de programmation permettent d'effectuer des choix et des boucles.

On va parler de blocs d'instructions :

Il s'agit d'un ensemble d'instructions entouré d'accolades ouvrantes et fermantes.

```
{
        a = 5;
        …
}
```

STRUCTURES DE CONTRÔLES

L'INSTRUCTION IF – SYNTAXE

```
if (expression)
  instruction_1
else                              // l'instruction else est facultative
  instruction_2
```

expression est une expression quelconque avec la convention

Différente de 0 → vrai

Egale à 0 → faux

Instruction_1 et **instruction_2** sont des instructions quelconques i.e. :

- Simple (terminée par un point virgule)
- bloc
- Instruction structurée

STRUCTURES DE CONTRÔLES

L'INSTRUCTION SWITCH – SYNTAXE

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

permet dans certain cas d'éviter une abondance d'instruction if imbriquées.

expression est une expression quelconque comme dans le cas de if, dont la valeur va être testé contre les constantes.

constante : expression constante de type entier (char est accepté car converti en int)

Instruction : suite d'instruction quelconque

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.

STRUCTURES DE CONTRÔLES

L'INSTRUCTION DO ... WHILE

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

permet de répéter une ou plusieurs instructions tant que la condition expression est vrai.

A noter que :

- La série d'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 CONTRÔLES

L'INSTRUCTION WHILE

```
while (expression)
    instruction
```

permet de répéter une ou plusieurs instructions tant que la condition expression est vrai.

A noté 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. Celles-ci ne sont donc pas forcément exécutées.

STRUCTURES DE CONTRÔLES

L'INSTRUCTION FOR – « BOUCLE AVEC COMPTEUR »

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

permet de répéter une ou plusieurs instructions avec une syntaxe parfois plus pratique que les boucles while.

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.

STRUCTURES DE CONTRÔLES

L'INSTRUCTION FOR – « BOUCLE AVEC COMPTEUR » - UN EXEMPLE SIMPLE

```
#include <iostream>
using namespace std;
int main(){
   for (int i = 0; i < 10; i++)
    {
      cout << "i = " << i << endl;
    }
}
```

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

```
roland@DESKTOP-M1EA3EP ~
$ g++ ex.cpp

roland@DESKTOP-M1EA3EP ~
$ ./a.exe
i = 0
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i = 8
i = 9
```

STRUCTURES DE CONTRÔLES

BREAK, CONTINUE ET GOTO

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.
- **goto** est une instruction déconseillée, elle s'utilise conjointement à des étiquettes dans le code et permet d'y aller directement. Même si cela semble intéressant en première approche, son usage sera interdit lors de ce cours.

LES FONCTIONS

LES FONCTIONS

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

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

En effet, une fonction permet de renvoyer un résultat scalaire, 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

UN EXEMPLE DE FONCTION : LA FONCTION PUISSANCE

```
#include <iostream>
double my_pow(double a, unsigned int exp)
{
  double res = 1;

  for (int i = 0; i < exp; i++)
    res *= a;
  return res;
}
int main()
{
  std::cout << "2^5 = " << my_pow(2.0,5) << std::endl;
}
```

Voici un exemple de fonction.

La fonction *my_pow* prend en argument un flottant et un entier non signé et retourne une valeur de type flottant.
res est une variable locale à la fonction qui permet de stocker les valeurs intermédiaires du calcul qui est effectué dans la boucle.
Le résultat de la fonction, un double, est donné grâce au mot clé *return*.
A noté que *return* marque la fin de l'exécution de la fonction : les instructions qui se trouvent après ne sont jamais exécutées.
Une fonction peut contenir plusieurs *return* (dans des conditions par exemple) ou aucun, si la fonction ne renvoie rien.

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 *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>

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

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

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

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 évidement pas la modification de l'original.

LES FONCTIONS

PASSAGE PAR RÉFÉRENCE

```
#include <iostream>

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

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

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

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.

Une modification dans la fonction est donc répercutée sur les paramètres transmis.

LES FONCTIONS

PASSAGE PAR RÉFÉRENCE

```
#include <iostream>

/*
  Cette fonction doit echanger la valeur des
  deux entiers passés en paramétres */
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;
}
```

```
$ g++ echange.cpp
echange.cpp: Dans la fonction ‘int main()’:
echange.cpp:12:15: erreur : invalid initialization of non-
const reference of type ‘int&’ from an rvalue of type ‘int’
   my_swap(2, 3);
               ^
echange.cpp:8:6: note :   initializing argument 1 of ‘void
my_swap(int&, int&)’
 void my_swap(int &, int &);
```

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 !

LES FONCTIONS

PASSAGE PAR RÉFÉRENCE

```
#include <iostream>

/*
  Cette fonction doit echanger la valeur des
  deux entiers passés en paramétres */
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;
}

$ g++ echange.cpp
echange.cpp: Dans la fonction ‘int main()’:
echange.cpp:12:15: erreur : invalid initialization of non-
const reference of type ‘int&’ from an rvalue of type ‘int’
   my_swap(2, 3);
               ^
echange.cpp:8:6: note :   initializing argument 1 of ‘void
my_swap(int&, int&)’
 void my_swap(int &, int &);
```

La fonction my_swap modifie ses paramètres. On ne peut donc évidement pas l’appeler avec des arguments constants.

Pour lever cette ambiguïté, 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.

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 générale 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 elle 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

Ce 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 repose 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. est « empilée » en mémoire et « dépilée » 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 possède plusieurs définitions. On choisit l'une ou l'autre de ces définitions en fonction du contexte.

On a en fait déjà rencontré des operateurs qui étaient surchargés.

Par exemple + peut être une addition d'entier 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

SURCHAGE – UN EXEMPLE

```
#include <iostream>

void print_me(int a)
{
  std::cout << "Hello ! i m an integer ! : " << a << std::endl;
}

void print_me(double a)
{
  std::cout << "Hello ! i m a double ! : " << a << std::endl;
}

main()
{
  print_me(2);
  print_me(2.0);
}
```

La fonction *print_me* est définie deux fois. Le nom ne change pas, la valeur de retour ne change pas.

Le type du 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.

LES *POINTEURS*

TABLEAUX & POINTEURS

PREMIER EXEMPLE

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

}
```

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 chaque élément du tableau pour l'afficher.

On notera que la notation `[]` s'emploie aussi bien pour la déclaration que pour l'accès à un élément du tableau.

TABLEAUX & POINTEURS

QUELQUES RÉGLES

Il ne faut pas confondre les éléments d'un tableau avec le tableau lui-même.

Ainsi, `t[2] = 3, tab[i]++` sont des écritures valides.

Mais `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.

TABLEAUX & POINTEURS

QUELQUES RÉGLES

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%1]
```

Il n'existe pas de mécanisme de contrôles des indices ! Il revient au programmeur de ne pas écrire dans des zones mémoires qu'il n'a pas alloué.

Source de nombreux bugs ….

TABLEAUX & POINTEURS

QUELQUES RÉGLES

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 l'acceptent néanmoins en tant qu'extension du langage.

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

TABLEAUX & POINTEURS

QUELQUES RÉGLES – UNE PARENTHÈSE SUR LES OPTIONS DE COMPILATION

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

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

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

C’est l’occasion d’ouvrir une parenthèse sur d’autres options du compilateur g++. Si l’on compile le code ci-contre avec les options ci-dessus, ce code ne compile pas. En effet, par défaut, un compilateur fera son possible pour compiler un programme, quitte à ne pas respecter scrupuleusement la norme du langage.

On peut, en rajoutant ces options, forcer le compilateur à respecter la norme.

Ceci a comme but de garantir un maximum de compatibilité et de portabilité si on change de compilateur ou de version de compilateur.

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

TABLEAUX & POINTEURS

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'indice peut être quelconque en C++. On prendra néanmoins en compte les limitations de la machine elle-même comme la quantité de mémoire à disposition.

TABLEAUX & POINTEURS

INITIALISATION

```
#include <iostream>
using namespace std;
main()
{
  int t[10] = {0,2,4,6,8,10,12,15,16,18};
  for (int i = 0; i < 10; i++)
    cout << t[i] << ";";
  cout << endl;
}
```

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.

TABLEAUX & POINTEURS

POINTEURS

C++ permet, comme C, un contrôle fin de la mémoire. En effet, il permet, grâce aux pointeurs, d'accéder directement à des zones de la mémoire, en travaillant sur les adresses.

	mémoire				
adresses	0x0	…	0x42	0x43	…
valeurs			'a'	'b'	

Par exemple, ici, on voit que l'octet situé à l'adresse mémoire 42 (en hexadécimal, noté `0x`) contient la valeur `'a'` (un `char` suivant la table ascii) . Le suivant contient lui la valeur `'b'`.

TABLEAUX & POINTEURS

POINTEURS – LES OPERATEURS * ET &

```
#include <iostream>
using namespace std;
main()
{
  int *ptr;
  int i = 42;

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

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

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 ce que contient `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`.

TABLEAUX & POINTEURS

POINTEURS – LES OPERATEURS * 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.

TABLEAUX & POINTEURS

RELATION TABLEAUX ET POINTEURS

En C++, l'identificateur d'un tableau (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.

TABLEAUX & POINTEURS

POINTEURS – ARITHMÉTIQUE DES POINTEURS

Une adresse est une valeur entière. Il parait 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 à 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*.

TABLEAUX & 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 operateurs 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.

TABLEAUX & POINTEURS

POINTEURS PARTICULIERS

- **Le pointeur nul**, 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 soucis de la taille du type. C'était un type très utile en C, notamment pour écrire des fonctions génériques valables quelque soit le type des données.
- Par exemple : voici l'entête de la fonction `qsort` de la bibliothèque standard de C. Utilisable en C++, mais déconseillée, on a des outils bien plus puissants ! On notera par ailleurs l'emploi d'un pointeur de fonction, que nous verrons plus tard.

```
void qsort (void *tableau , size_t nb_elem , size_t taille_elem , int (*compare) (void const *a, void const *b));
```

TABLEAUX & POINTEURS

ALLOCATION STATIQUE ET DYNAMIQUE

<table>
<tr><td rowspan="7">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>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 liberer cet espace.</td></tr>
</table>

Ceci est une représentation schématisée de la mémoire occupée par un processus au cours de son exécution.
On connait déjà la **pile** (ou stack en anglais) qui contient les variables et les tableaux que l'on a déclaré 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 ! »

TABLEAUX & POINTEURS

LES OPERATEURS NEW ET DELETE

`new` est un opérateur unaire prenant comme argument un type.

`new type` où *type* représente un type quelconque.

Il renvoie un pointeur de type `type*` dont la valeur est l'adresse de la zone mémoire allouée pour notre donnée de type `type`.

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

On peut maintenant utiliser notre pointeur pour accéder à un entier que l'on a alloué en mémoire.

Une autre syntaxe permet d'allouer un espace mémoire contiguë pour plusieurs données à la fois. Le pointeur renvoyé, toujours de type `type*` pointe vers la première valeur allouée.

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

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

TABLEAUX & POINTEURS

LES OPERATEURS NEW ET DELETE

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

TABLEAUX & POINTEURS

LES OPERATEURS 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 évidement arrêter de l'utiliser.

TABLEAUX & POINTEURS

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;
}
```

Ce programme calcule la moyenne des valeurs que l'utilisateur entre au clavier durant l' exécution. Mais le nombre de ces valeurs varient aussi ! Si nous avions alloué un tableau statiquement, sur la pile, comme jusqu'à présent, nous aurions du 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.

TABLEAUX & 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); // et le pointeur s'appellera fct_ptr
```

On notera l'utilisation des parenthèses. En effet, écrire `int *fct(double, double)` ne signifie pas du tout la même chose.

TABLEAUX & POINTEURS

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;
}
main()
{
  double t[10] = {1,2,3,4,5,6,7,8,9,10};
  aff_tab(t, 10);
  apply(t, 10, fct1);
  aff_tab(t, 10);
}
```

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 double donné en paramètre.

La nouveauté se situe dans la fonction `apply` qui va appliquer sur chaque éléments d'un tableau de n éléments la fonction passée en paramètre, à l'aide d'un pointeur.

On notera que pour appeler la fonction pointée, il faut la déréférencer, toujours à laide 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 math comme `cos` ou `sqrt`, sans réécrire pour chaque cas une boucle pour l'appliquer à chaque éléments de ce tableau.

TABLEAUX & POINTEURS

CHAINES DE CARACTÈRES EN C

```
#include <iostream>

using namespace std;

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

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

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

Bien que celui-ci offre d'autres mécanismes de plus haut niveau pour la manipulation de chaines, nous allons étudier celles-ci.

D'une part car c'est un bon exercice sur les pointeurs.

Pour comprendre ce qu'est une chaine de caractères.

Car elles sont encore utilisées en C++.

TABLEAUX & POINTEURS

CHAINES DE CARACTÈRES EN C

```
#include <iostream>

using namespace std;

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

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

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

il y a néanmoins une convention supplémentaire.

Ils s'agit d'un ensemble d'octets contigüe se terminant par le caractère nul noté `\0`. Ceci afin de donner une fin à la chaine.

La notation `"Hello World"` définie donc un pointeur sur caractère vers une zone de la mémoire où est définie la constante « `hello world` ». On récupère cette adresse dans le pointeur `str`.

TABLEAUX & POINTEURS

CHAINES DE CARACTÈRES EN C

```
#include <iostream>

using namespace std;

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

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

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

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

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

TABLEAUX & POINTEURS

LES ARGUMENTS D'UN PROGRAMME

```
#include <iostream>
using namespace std;
int main(int nb, char *args[])
{
  cout << "Mon programme possède " << nb
       << " arguments" << endl;
  for (int i = 0; i < nb; i++)
    cout << args[i] << endl;

}
```

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

On peut passer à un programme des valeurs lorsqu'on l'appelle sur la ligne de commande.
Le programme le reçoit comme un argument de la fonction `main`.(que nous avions ignoré jusqu'ici)
Le premier argument est conventionnellement un entier qui reçoit en valeur le 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 éléments de ce tableau est donc un pointeur vers une chaine de caractère 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 chaine de caractères.
Conventionnellement, le premier paramètre fourni est le nom du programme exécuté.

STRUCTURES

LES STRUCTURES

Jusqu'à présent, nous avons rencontré les tableaux qui étaient un regroupement de données de même type.

Il peut être utile de grouper des données de types différents et de les regrouper dans une même entité.

En C, il existait déjà un tel mécanisme connu sous le nom de structure.

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

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

STRUCTURES

DÉCLARATION

```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;
}
```

```cpp
#ifndef __STRUCT_HH__
#define __STRUCT_HH__

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

#endif
```

Dans cet exemple nous commençons par déclarer une 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.

STRUCTURES

DÉCLARATION

```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;
}
```

```cpp
#ifndef __STRUCT_HH__
#define __STRUCT_HH__

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

#endif
```

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

La structure Personne devient alors un type de donnée.

Ce type est un regroupement d'un entier et de deux *double*.

Dans la fonction *main*, après avoir inclus notre fichier header au début de notre code, nous pouvons déclarer une variable de type *Personne*.

Cette variable s'appellera p1.

STRUCTURES

DÉCLARATION

```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;
}
```

```cpp
#ifndef __STRUCT_HH__
#define __STRUCT_HH__

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

#endif
```

Les valeurs de différents types contenus dans la structure sont appelés 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.

STRUCTURES

DÉCLARATION

```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;
}
```

```cpp
#ifndef __STRUCT_HH__
#define __STRUCT_HH__

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

#endif
```

Une structure en soi n'a pas d'existence concrète en mémoire.

C'est une fois qu'elle a été **instanciée**, c'est-à-dire qu'une variable aura été créé à partir de sa déclaration, que la structure existe vraiment en mémoire pour cette variable.

Pour imager un peu, on ne peut pas habiter dans le plan d'une maison. Il n'y a qu'une fois que celle-ci a été créée à partir du plan qu'elle existe vraiment.

STRUCTURES

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;
}
```

Dans l'exemple précèdent, 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

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;
}
```

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

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

STRUCTURES

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;
}
```

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

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 << "à la date : " << v.date.jour << "/"
       << v.date.mois << "/" << v.date.annee << endl
       << "Valeur : " << v.x << endl;
}
```

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

STRUCTURES ET FONCTIONS

```
#include <iostream>
#include <cmath>
using namespace std;
struct Point
{
  float x;
  float y;
};
float my_norme(Point pt1, Point pt2)
{
  return sqrt(pow(pt1.x - pt2.x, 2) +
              pow(pt1.y - pt2.y, 2));
}
int  main()
{
  Point a = {2, 2};
  Point b = {3, 3};
  cout << "norme : " << my_norme(a, b) << endl;
}
```

On crée une fonction my_norme calculant la norme euclidienne de deux points 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.

my_norme reçoit donc une copie des points et non les points eux même.

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

STRUCTURES

STRUCTURES ET FONCTIONS

```
#include <iostream>
#include <cmath>
using namespace std;
struct Point
{
  float x;
  float y;
};
float my_norme(Point & pt1, Point & pt2)
{
  return sqrt(pow(pt1.x - pt2.x, 2) +
              pow(pt1.y - pt2.y, 2));
}
int  main()
{
  Point a = {2, 2};
  Point b = {3, 3};
  cout << "norme : " << my_norme(a, b) << endl;
}
```

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 (du !) les déclarer **const**).

STRUCTURES

STRUCTURES ET FONCTIONS

```
#include <iostream>
#include <cmath>
using namespace std;
struct Point
{
  float x;
  float y;
};
float my_norme(Point * pt1, Point * pt2)
{
  return sqrt(pow(pt1->x - pt2->x, 2) +
              pow(pt1->y - pt2->y, 2));
}
int  main()
{
  Point a = {2, 2};
  Point b = {3, 3};
  cout << "norme : " << my_norme(&a, &b) << endl;
}
```

On passe maintenant des pointeurs sur Point à notre fonction. On voit que l'appelle de la fonction s'en trouve modifié : on ne passe plus les Point eux même mais leurs adresses (obtenues 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

STRUCTURES ET FONCTIONS

```
#include <iostream>
#include <cmath>
using namespace std;
struct Point
{
  float x;
  float y;
};
float my_norme(Point * pt1, Point * pt2)
{
  return sqrt(pow(pt1->x - pt2->x, 2) +
              pow(pt1->y - pt2->y, 2));
}
int  main()
{
  Point a = {2, 2};
  Point b = {3, 3};
  cout << "norme : " << my_norme(&a, &b) << endl;
}
```

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,

$$(*pt1).x \leftrightarrow pt1->x$$

STRUCTURES

FONCTIONS MEMBRES

```
#include <iostream>
using namespace std;
struct Point2D{
  double x;  double y;
  void initialise(double, double);
  void deplace(double dx, double dy);
  void affiche();
};
void Point2D::initialise(double abs, double ord){  x = abs; y = ord;}
void Point2D::deplace(double dx, double dy){  x+= dx; y += dy;}
void Point2D::affiche(){  cout << "x : " << x << " y : " << y << endl;}
int main(){
  Point2D x;
  x.initialise(1,1);
  x.deplace(0.1, -0.1);
  x.affiche();
}
```

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 *Point2D*.

Une fonction *deplace*, qui prend deux paramètres et qui modifiera les coordonnées en fonction.

Une fonction *affiche* qui provoquera un affichage de notre point.

STRUCTURES

FONCTIONS MEMBRES

```
#include <iostream>
using namespace std;
struct Point2D{
  double x;  double y;
  void initialise(double, double);
  void deplace(double dx, double dy);
  void affiche();
};
void Point2D::initialise(double abs, double ord){  x = abs; y = ord;}
void Point2D::deplace(double dx, double dy){  x+= dx; y += dy;}
void Point2D::affiche(){  cout << "x : " << x << " y : " << y << endl;}
int main(){
  Point2D x;
  x.initialise(1,1);
  x.deplace(0.1, -0.1);
  x.affiche();
}
```

Les fonctions *initialise*, *deplace* et *affiche* sont des **fonctions membres** de la structure Point2D.

On les déclare dans la structure.

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

En effet, comment connaître *x* et *y* dans la fonction si le compilateur ne sait pas qu'elles appartiennent à Point2D ?

STRUCTURES

UN CODE ORDONNÉ

```
#ifndef __POINT2D_HH__
#define __POINT2D_HH__
#include <iostream>
using namespace std;
struct Point2D{
  double x;
  double y;
  void initialise(double, double);
  void deplace(double dx, double dy);
  void affiche();
};
#endif
```

Point2D.hh

```
#include "Point2D.hh"
void Point2D::initialise(double abs, double ord){
  x = abs;
  y = ord;
}
void Point2D::deplace(double dx, double dy){
  x += dx;
  y += dy;
}
void Point2D::affiche(){
  cout << "x : " << x << " y : " << y << endl;
}
```

Point2D.cpp

On sépare la déclaration de notre structure de sa définition. On créé un fichier NomDeLaStructure.hh qui sera destiné à la déclaration. Et un fichier NomDeLaStructure.cpp qui contiendra, du code, les définitions des fonctions membres.

STRUCTURES

UN CODE ORDONNÉ

```cpp
#include <iostream>
#include "Point2D.hh"

int main(){
  Point2D x;

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

ex_struct_fct_membres.cpp

```
roland@DESKTOP-M1EA3EP ~/structures
$ g++ ex_struct_fct_membres.cpp Point2D.cpp

roland@DESKTOP-M1EA3EP ~/structures
$ ./a.exe
x : 1.1 y : 0.9
```

Voilà à quoi va ressembler notre fichier contenant la fonction main désormais.

On inclus le fichier header Point2D.hh. Ainsi le compilateur connaîtra le type Point2D.

Pour la compilation, on compile en même temps les deux fichiers .cpp pour créer notre exécutable.

Il y a des méthodes plus propres, à l'aide de **make** par exemple.

CLASSES & OBJETS

CLASSES & OBJETS

INTRODUCTION

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.

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

CLASSES & OBJETS

INTRODUCTION

Pourquoi ne pas simplement travailler sur des structures ?

Les structures ne permettent pas d'encapsuler leurs membres, données ou fonctions.

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).

CLASSES & OBJETS

DÉCLARATION

```
#ifndef __POINT2D_HH__
#define __POINT2D_HH__
class Point2D
{
private:
  double x;
  double y;

public:
  void initialise(double, double);
  void affiche();

};
#endif
```

Une classe se déclare comme une structure.

On remarque les étiquettes *private* et *public* qui sont utiles pour déterminer 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

Point pt;

pt.x = 5;

dans la fonction *main*.

CLASSES & OBJETS

CONSTRUCTEURS

```
#ifndef __POINT2D_HH__
#define __POINT2D_HH__

class Point2D
{
private:
  double x;
  double y;

public:
  Point2D(double,double);
  void affiche();

};


#endif
```

```
#include <iostream>
#include "Point2D.hh"
using namespace std;
Point2D::Point2D(double abs, double ord){
  x = abs; y = ord;
}
void Point2D::affiche(){
  cout << x <<":"<< y << endl;
}
```

Au lieu d'utiliser une fonction *initialise*, nous allons voir un autre mécanisme de 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 dés le début de la vie de notre instance.

CLASSES & OBJETS

CONSTRUCTEURS

```
#ifndef __POINT2D_HH__
#define __POINT2D_HH__

class Point2D
{
private:
  double x;
  double y;

public:
  Point2D(double,double);
  void affiche();

};


#endif
```

```
#include <iostream>
#include "Point2D.hh"
using namespace std;
Point2D::Point2D(double abs, double ord){
  x = abs; y = ord;
}
void Point2D::affiche(){
  cout << x <<":"<< y << endl;
}
```

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 voulions déclarer maintenant un objet de type Point2D, nous serions obligés de fournir les deux coordonnées.

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

CLASSES & OBJETS

CONSTRUCTEURS

```
#ifndef __POINT2D_HH__
#define __POINT2D_HH__

class Point2D
{
private:
  double x;
  double y;

public:
  Point2D(double,double);
  void affiche();

};


#endif
```

```
#include <iostream>
#include "Point2D.hh"
using namespace std;
Point2D::Point2D(double abs, double ord){
  x = abs; y = ord;
}
void Point2D::affiche(){
  cout << x <<":"<< y << endl;
}
```

Il peut y avoir autant de constructeur 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.

CLASSES & OBJETS

DESTRUCTEUR

```
#ifndef __EX_DESTR_HH__
#define __EX_DESTR_HH__

class Point2D
{
private:
  double _x, _y;

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


#endif
```

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 double.

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.

CLASSES & OBJETS

OBJETS ET DYNAMIQUE

```
#ifndef __POINTND_HH__
#define __POINTND_HH__
class PointND {
private:
  int _n;
  double *_vals;
public:
  PointND(int);
  PointND(const PointND&);
  ~PointND();
  void print();
};
#endif
```

```
#include <iostream>
#include "PointND.hh"
using namespace std;
PointND::PointND(int n) {
  _n = n;
  _vals = new double[n];
  cout << "Constructeur : "  << "n = " << n << endl;
}
PointND::PointND(const PointND& pnd){
  _n = pnd._n;
  _vals = pnd._vals;
  cout << "Constructeur par copie" << endl;
}
PointND::~PointND(){
  cout << "Appel Destructeur" << endl;
}
void PointND::print(){
  for (int i = 0; i < _n; ++i)
    cout << _vals[i] << ":";
}
```

Notre objet contient un pointeur sur double qui contiendra toutes les coordonnées de notre n-point.
Le nombre de dimensions est un entier qui sera aussi un membre privé de notre classe.
Le 1er constructeur initialise les deux variables.
Pour le tableau de valeurs, pas d'autre choix que de passer par un « new » et donc une **allocation dynamique**. En effet, le nombre de valeurs étant variable, on ne peut pas utiliser un tableau dont la dimension serait fixée à l'avance.

Ce code contient plusieurs défauts.

CLASSES & OBJETS

OBJETS ET DYNAMIQUE

```
#ifndef __POINTND_HH__
#define __POINTND_HH__
class PointND {
private:
  int _n;
  double *_vals;
public:
  PointND(int);
  PointND(const PointND&);
  ~PointND();
  void print();
};
#endif
```

```
#include <iostream>
#include "PointND.hh"
using namespace std;
PointND::PointND(int n) {
  _n = n;
  _vals = new double[n];
  cout << "Constructeur : "  << "n = " << n << endl;
}
PointND::PointND(const PointND& pnd){
  _n = pnd._n;
  _vals = pnd._vals;
  cout << "Constructeur par copie" << endl;
}
PointND::~PointND(){
  cout << "Appel Destructeur" << endl;
}
void PointND::print(){
  for (int i = 0; i < _n; ++i)
    cout << _vals[i] << ":";
}
```

Premier défaut : le constructeur

Celui-ci initialise _n avec la valeur entière passée en paramètre. Il alloue aussi la place en mémoire dynamiquement pour les n valeurs de notre n-point.
Mais qu'en est il de ces n valeurs ? Elles ne sont pas initialisées.

On pourrait :

- Assumer leur caractère aléatoire .
- Initialiser le tableau à 0.
- Ajouter un second paramètre contenant des valeurs à recopier.
- Etc.

CLASSES & OBJETS

OBJETS ET DYNAMIQUE

```
#ifndef __POINTND_HH__
#define __POINTND_HH__
class PointND {
private:
  int _n;
  double *_vals;
public:
  PointND(int);
  PointND(const PointND&);
  ~PointND();
  void print();
};
#endif
```

```
#include <iostream>
#include "PointND.hh"
using namespace std;
PointND::PointND(int n) {
  _n = n;
  _vals = new double[n];
  cout << "Constructeur : "  << "n = " << n << endl;
}
PointND::PointND(const PointND& pnd){
  _n = pnd._n;
  _vals = pnd._vals;
  cout << "Constructeur par copie" << endl;
}
PointND::~PointND(){
  cout << "Appel Destructeur" << endl;
}
void PointND::print(){
  for (int i = 0; i < _n; ++i)
    cout << _vals[i] << ":";
}
```

Deuxième défaut : le constructeur par copie

Celui-ci recopie les valeurs de l'objet à copier, 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 ?
Ce code n'a probablement pas le comportement souhaité.
Il copie la valeur de _vals qui est un pointeur sur double. Sa valeur n'est donc que l'adresse du premier élément du tableau alloué par le new.
Les deux objets partageront alors le même tableau de valeur et modifier l'un modifiera l'autre également.
On pourrait aussi recopier les valeurs de pnd._vals dans le nouveau tableau à l'aide d'une boucle par exemple. Il faudra aussi penser à allouer un nouvel espace mémoire avec new[].

CLASSES & OBJETS

OBJETS ET DYNAMIQUE

```
#ifndef __POINTND_HH__
#define __POINTND_HH__
class PointND {
private:
  int _n;
  double *_vals;
public:
  PointND(int);
  PointND(const PointND&);
  ~PointND();
  void print();
};
#endif
```

```
#include <iostream>
#include "PointND.hh"
using namespace std;
PointND::PointND(int n) {
  _n = n;
  _vals = new double[n];
  cout << "Constructeur : "  << "n = " << n << endl;
}
PointND::PointND(const PointND& pnd){
  _n = pnd._n;
  _vals = pnd._vals;
  cout << "Constructeur par copie" << endl;
}
PointND::~PointND(){
  cout << "Appel Destructeur" << endl;
}
void PointND::print(){
  for (int i = 0; i < _n; ++i)
    cout << _vals[i] << ":";
}
```

Troisième défaut : le destructeur.

Celui-ci ne réalise pourtant qu'un affichage. 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 new[] dans le constructeur.

Quand est ce que celle-ci sera 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.

CLASSES & OBJETS

OBJETS MEMBRES

```cpp
#include <iostream>
#include "Cercle.hh"
using namespace std;
Point::Point(double x, double y)
{
  cout << "Constructeur de Point("
       << x <<";"<<y<<")"<< endl;
  _x = x, y = y;
}
Cercle::Cercle(Point pt, double r)
  :_centre(pt)
{
  cout << "Constructeur de Cercle"
       << endl;
  _rayon = r;
}
```

```cpp
#ifndef __CERCLE_HH__
#define __CERCLE_HH__

class Point{
  double _x, _y;
public:
  Point(double, double);
};
class Cercle{
  Point _centre;
  double _rayon;
public:
  Cercle(Point, double);
};

#endif
```

```cpp
#include "Cercle.hh"

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

```
$ ./a.exe
Constructeur de Point(2;3)
Constructeur de Cercle
```

CLASSES & OBJETS

OBJETS MEMBRES

```cpp
#include <iostream>
#include "Cercle.hh"
using namespace std;
Point::Point(double x, double y)
{
  cout << "Constructeur de Point("
       << x <<";"<<y<<")"<< endl;
  _x = x, y = y;
}
Cercle::Cercle(Point pt, double r)
  :_centre(pt)
{
  cout << "Constructeur de Cercle"
       << endl;
  _rayon = r;
}
```

```cpp
#ifndef __CERCLE_HH__
#define __CERCLE_HH__

class Point{
  double _x, _y;
public:
  Point(double, double);
};
class Cercle{
  Point _centre;
  double _rayon;
public:
  Cercle(Point, double);
};

#endif
```

On remarque que quand un objet est membre d'un autre objet, son constructeur est appelé en premier.
Si il n'existe pas de constructeur par défaut, cad sans argument, comment le construire ?
C++ a une syntaxe particulière. On fait suivre dans l'entête du constructeur de Cercle l'appel au constructeur de Point qui est du coup construit avant l'entrée dans le constructeur de Cercle.

```
$ ./a.exe
Constructeur de Point(2;3)
Constructeur de Cercle
```

CLASSES & OBJETS

MEMBRES STATIQUES

```
#include <iostream>
class JeMeCompte
{
  static int compteur;
public:
  JeMeCompte();
  ~JeMeCompte();
};
int JeMeCompte::compteur = 0;
JeMeCompte::JeMeCompte(){
  std::cout << compteur++ << std::endl;
}
JeMeCompte::~JeMeCompte(){
  std::cout << compteur++ << std::endl;
}
int main(){
  JeMeCompte nbr;
}
```

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 ainsi que par la classe elle-même.

On utilise pour cela le mot clé `static` devant la (ou les !) variable qui sera désignée pour ce rôle.

CLASSES & OBJETS

MEMBRES STATIQUES

```
#include <iostream>
class JeMeCompte
{
  static int compteur;
public:
  JeMeCompte();
  ~JeMeCompte();
};
int JeMeCompte::compteur = 0;
JeMeCompte::JeMeCompte(){
  std::cout << compteur++ << std::endl;
}
JeMeCompte::~JeMeCompte(){
  std::cout << compteur++ << std::endl;
}
int main(){
  JeMeCompte nbr;
}
```

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 classe elle-même dans la déclaration. En effet, cela risquerait en cas de compilation séparée de réserver plusieurs emplacement en mémoire pour cette donnée. Dans chaque fichier .o qui utiliserait cette classe.

CLASSES & OBJETS

MEMBRES STATIQUES

```
#include <iostream>
class JeMeCompte
{
  static int compteur;
public:
  JeMeCompte();
  ~JeMeCompte();
};
int JeMeCompte::compteur = 0;
JeMeCompte::JeMeCompte(){
  std::cout << compteur++ << std::endl;
}
JeMeCompte::~JeMeCompte(){
  std::cout << compteur++ << std::endl;
}
int main(){
  JeMeCompte nbr;
}
```

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'on s'adresse bien à un membre de la classe.

CLASSES & OBJETS

FONCTIONS MEMBRES STATIQUES

```
#include <iostream>
class JeMeCompte{
  static int compteur;
public:
  JeMeCompte();
  ~JeMeCompte();
  static void AfficheCompteur();
};
int JeMeCompte::compteur = 0;
JeMeCompte::JeMeCompte(){
  std::cout << "C : " << compteur++ << std::endl;}
JeMeCompte::~JeMeCompte(){
  std::cout << "D : " << compteur++ << std::endl;}
void JeMeCompte::AfficheCompteur(){
  std::cout << compteur <<" objet(s)«  << std::endl;}
int main(){
  JeMeCompte nbr;
  JeMeCompte::AfficheCompteur();
}
```

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 toute objet, comme dans l'exemple ci contre.

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

CLASSES & OBJETS

LE MOT CLÉ THIS

```
#include <iostream>

class Objet{
public:
  Objet();
  ~Objet();
};
Objet::Objet(){
  std::cout << "C : " << this << std::endl;
}
Objet::~Objet(){
  std::cout << "D : " << this << std::endl;
}
int main(){
  Objet o, op;
}
```

```
$ ./a.exe
C : 0xffffcbff
C : 0xffffcbfe
D : 0xffffcbfe
D : 0xffffcbff
```

Chaque fonction membre d'un objet reçoit 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 qui pointe vers l'instance courante.

CLASSES & OBJETS

ACCESSEURS & MUTATEURS

```hh
#ifndef __O_HH__
#define __O_HH__

class Objet{
private:
  double _v;
public:
  Objet(double _v);
  ~Objet();
  double getV() const;
  void setV(double d);
};

#endif
```

```cpp
#include "Objet.hh"

Objet::Objet(double v) : _v(v){}
Objet::~Objet() {}

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

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

```cpp
#include <iostream>
#include "Objet.hh"
using namespace std;
int main()
{
  Objet o(3);

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

```
roland@win ~/setter
$ g++ main.cpp Objet.cpp

roland@win ~/setter
$ ./a.exe
4
```

CLASSES & OBJETS

ACCESSEURS & MUTATEURS

```hh
#ifndef __O_HH__
#define __O_HH__

class Objet{
private:
  double _v;
public:
  Objet(double _v);
  ~Objet();
  double getV() const;
  void setV(double d);
};

#endif
```

```cpp
#include "Objet.hh"

Objet::Objet(double v) : _v(v){}
Objet::~Objet() {}

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

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

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

On les appelle aussi getter et setter en anglais.

Ce sont des fonctions qui doivent être presque automatiquement créées lors de la création d'une nouvelle classe.

CLASSES & OBJETS

ACCESSEURS & MUTATEURS

```hh
#ifndef __O_HH__
#define __O_HH__

class Objet{
private:
  double _v;
public:
  Objet(double _v);
  ~Objet();
  double getV() const;
  void setV(double d);
};

#endif
```

```cpp
#include "Objet.hh"

Objet::Objet(double v) : _v(v){}
Objet::~Objet() {}

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

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

Leurs noms rappellent les noms des attributs précédés 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.

CLASSES & OBJETS

ACCESSEURS & MUTATEURS

```hh
#ifndef __O_HH__
#define __O_HH__

class Objet{
private:
  double _v;
public:
  Objet(double _v);
  ~Objet();
  double getV() const;
  void setV(double d);
};

#endif
```

```cpp
#include "Objet.hh"

Objet::Objet(double v) : _v(v){}
Objet::~Objet() {}

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

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

En effet, les fonctions membres déclarés 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 seuls fonctions que l'on peut appeler sur des objets constants.

CLASSES & OBJETS

ACCESSEURS & MUTATEURS

```hh
#ifndef __O_HH__
#define __O_HH__

class Objet{
private:
  double _v;
public:
  Objet(double _v);
  ~Objet();
  double getV() const;
  void setV(double d);
};

#endif
```

```cpp
#include "Objet.hh"

Objet::Objet(double v) : _v(v){}
Objet::~Objet() {}

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

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

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.

CLASSES & OBJETS

FONCTION AMIE

```
#ifndef __O_HH__
#define __O_HH__
class Objet{
private:
  double _v;
public:
  Objet(double _v);
  ~Objet();
  double getV() const;
  void setV(double d);
  friend bool egale(Objet o1,
                    Objet o2);
};
#endif
```

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

```
#include "Objet.hh"
#include <cmath>
Objet::Objet(double v) : _v(v){}
Objet::~Objet() {}
double Objet::getV() const{
  return _v;
}
void Objet::setV(double v){
  _v = v;
}
bool egale(Objet o1, Objet o2)
{
  return std::abs(o1._v - o2._v) < 10e-10;
}
```

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.

CLASSES & OBJETS

FONCTION AMIE

```
#ifndef __O_HH__
#define __O_HH__
class Objet{
private:
  double _v;
public:
  Objet(double _v);
  ~Objet();
  double getV() const;
  void setV(double d);
  friend bool egale(Objet o1,
                    Objet o2);
};
#endif
```

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

```
#include "Objet.hh"
#include <cmath>
Objet::Objet(double v) : _v(v){}
Objet::~Objet() {}
double Objet::getV() const{
  return _v;
}
void Objet::setV(double v){
  _v = v;
}
bool egale(Objet o1, Objet o2)
{
  return std::abs(o1._v - o2._v) < 10e-10;
}
```

Son utilisation se justifie quand on recherche un code optimisé. Car l'utilisation des getters et des setters, comme les appels de fonctions en générale, 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.

CLASSES & OBJETS

FONCTION AMIE

```
#ifndef __O_HH__
#define __O_HH__
class Objet{
private:
  double _v;
public:
  Objet(double _v);
  ~Objet();
  double getV() const;
  void setV(double d);
  friend bool egale(Objet o1,
                    Objet o2);
};
#endif
```

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

```
#include "Objet.hh"
#include <cmath>
Objet::Objet(double v) : _v(v){}
Objet::~Objet() {}
double Objet::getV() const{
  return _v;
}
void Objet::setV(double v){
  _v = v;
}
bool egale(Objet o1, Objet o2)
{
  return std::abs(o1._v - o2._v) < 10e-10;
}
```

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

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

Ici, `true/false` à la place de `1/0` sinon.

SURCHARGE D'OPERATEURS

SURCHARGE D'OPERATEURS

INTRODUCTION

Imaginons que nous définissions une classe Complex chargée d'implémenter la gestion des 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) ;
```

SURCHARGE D'OPERATEURS

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 que nous puissions le définir dans ce contexte.

SURCHARGE D'OPERATEURS

INTRODUCTION

Le C++ permet de tels définitions supplémentaires. On appelle ce mécanisme la surcharge ou surdéfinition d'opérateurs. Ce mécanisme vient compléter la surcharge de fonction 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 permette. 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 bases.

Hors de question, donc, de redéfinir l'addition des entiers.

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

SURCHARGE D'OPERATEURS

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)

SURCHARGE D'OPERATEURS

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.

SURCHARGE D'OPERATEURS

UN EXEMPLE

```hh
#ifndef __COMPLEX_HH__
#define __COMPLEX_HH__
class Complex
{
private:
  double _real, _imag;

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

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

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
{
  std::cout << "(" << _real << ";" << _imag << ")"
            << std::endl;
}
```

```cpp
#include "Complex.hh"

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

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.

SURCHARGE D'OPERATEURS

UN EXEMPLE

```hh
#ifndef __COMPLEX_HH__
#define __COMPLEX_HH__
class Complex
{
private:
  double _real, _imag;

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

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

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
{
  std::cout << "(" << _real << ";" << _imag << ")"
            << std::endl;
}
```

```cpp
#include "Complex.hh"

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

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

SURCHARGE D'OPERATEURS

UN EXEMPLE

```
#ifndef __COMPLEX_HH__
#define __COMPLEX_HH__
class Complex
{
private:
  double _real, _imag;

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

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

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
{
  std::cout << "(" << _real << ";" << _imag << ")"
            << std::endl;
}
```

```
#include "Complex.hh"

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

Il reste une dernière fonction membre, appelée operator+. C'est cette fonction qui redéfinie l'opérateur + pour nos nombres complexes.
La syntaxe est toujours la même quelque soit l'opérateur.

SURCHARGE D'OPERATEURS

UN EXEMPLE

```
#ifndef __COMPLEX_HH__
#define __COMPLEX_HH__
class Complex
{
private:
  double _real, _imag;

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

On voit qu'il s'agit d'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.

SURCHARGE D'OPERATEURS

UN EXEMPLE

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

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
{
  std::cout << "(" << _real << ";" << _imag << ")"
            << std::endl;
}
```

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.

A noter :
Normalement un tel renvoi par valeur doit appeler le constructeur par copie de notre objet. Si celui-ci n'est pas explicitement défini, alors le constructeur par défaut est appelé.
Mais cet opération est couteuse en terme de performance et certains compilateurs choisissent d'optimiser le code, même si le constructeur par copie a des effets de bords, comme un affichage par exemple.
C'est le cas de g++. Pour désactiver cette optimisation, lors de la compilation, on pourra utiliser l'option : -fno-elide-constructors

SURCHARGE D'OPERATEURS

COMMUTATIVITÉ

Il n'y a pas d'hypothèse à priori sur la commutativité des opérateurs.

Ainsi, si nous voulions 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 quant à lui un `double` et le deuxième un `Complex`.

SURCHARGE D'OPERATEURS

COMMUTATIVITÉ

Ce constant nous amène à deux réflexions.

- Tout d'abord, même si on peut définir une fonction autant de fois qu'on le veut tant que les types des paramètres sont différents, il est quand même désagréable d'avoir à le faire dans ce cas ! On verra que ce n'est pas forcément nécessaire car un double n'est qu'un complexe particulier et on pourra convertir un double en complexe. Ce qui nous ramènera au problème précèdent.
- Une opération `3.5 + c` a comme premier paramètre un double. L'opérateur + que nous avions défini dans notre classe `Complex` recevait en effet l'instance en premier argument, car il s'agissait d'une fonction membre de notre classe `Complex`.

SURCHARGE D'OPERATEURS

OPÉRATEURS & FONCTIONS AMIES

Nous avions jusqu'à présent défini les opérateurs surchargés dans nos classes, en tant que méthode de classe.

Cela n'est pas nécessaire, on peut les définir aussi en tant que fonction amie de notre classe comme nous allons le voir dans l'exemple suivant.

SURCHARGE D'OPERATEURS

OPÉRATEURS & FONCTIONS AMIES

```
#ifndef __COMPLEX_HH__
#define __COMPLEX_HH__

class Complex
{
private:
  double _real, _imag;
public:
  Complex(double real, double imag);
  Complex(const Complex &);
  Complex operator+ (const Complex &) const;

  friend Complex operator-(const Complex&,
                           const Complex&);

  void affiche() const;
};

#endif
```

On définie maintenant dans notre classe Complex une fonction **amie** `operator-` qui comme on l'aura deviné sera chargée de définir l'opérateur soustraction '-' pour les nombres complexes.

Celle-ci n'est pas une fonction membre de notre classe, mais une fonction amie. Son implémentation pourrait être :

```
Complex operator-(const Complex& c1,
                  const Complex& c2)
{
  return Complex(c1._real - c2._real,
                 c1._imag - c2._imag);
}
```

SURCHARGE D'OPERATEURS

OPÉRATEURS & FONCTIONS AMIES

```
#ifndef __COMPLEX_HH__
#define __COMPLEX_HH__

class Complex
{
private:
  double _real, _imag;
public:
  Complex(double real, double imag);
  Complex(const Complex &);
  Complex operator+ (const Complex &) const;

  friend Complex operator-(const Complex&,
                           const Complex&);

  void affiche() const;
};

#endif
```

On n'est bien sûr pas obligé d'utiliser le mécanisme d'amitié. Ainsi, on pourrait très bien utiliser des fonctions de « publication » de la classe Complex, c'est-à-dire les mutateurs et accesseurs dont nous avons précédemment parlé.

```
Complex operator-(const Complex& c1,
                  const Complex& c2)
{
  return Complex(c1._real - c2._real,
                 c1._imag - c2._imag);
}
```

SURCHARGE D'OPERATEURS

OPÉRATEURS & FONCTIONS AMIES

```
#ifndef __COMPLEX_HH__
#define __COMPLEX_HH__

class Complex
{
private:
  double _real, _imag;
public:
  Complex(double real, double imag);
  Complex(const Complex &);
  Complex operator+ (const Complex &) const;

  friend Complex operator-(const Complex&,
                           const Complex&);

  void affiche() const;
};

#endif
```

Dans cet exemple, il n'est pas nécessaire de passer par une fonction externe à notre classe, car le premier paramètre est du type de notre Classe.

```
Complex operator-(const Complex& c1,
                  const Complex& c2)
{
  return Complex(c1._real - c2._real,
                 c1._imag - c2._imag);
}
```

SURCHARGE D'OPERATEURS

OPÉRATEURS D'INCRÉMENTATION

```
#ifndef __COMPTMODULO_HH__
#define __COMPTMODULO_HH__
class ComptModulo
{
private :
  int _val, _mod;
public:
  ComptModulo(int v, int mod);
  ~ComptModulo();
  ComptModulo operator++();
  ComptModulo operator++(int n);
  void affiche() const;
};
#endif
```

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

ComptModulo::ComptModulo(int n, int mod) :
  _val(n), _mod(mod) {}
ComptModulo::~ComptModulo() {}

ComptModulo ComptModulo::operator++(){
  _val = (_val + 1) % _mod;
  return *this;
}
ComptModulo ComptModulo::operator++(int n){
  ComptModulo c = *this;
  _val = (_val + 1) % _mod;
  return c;
}
void ComptModulo::affiche() const{
  std::cout << _val << " ";
}
```

```
#include <iostream>
#include "ComptModule.hh"
using namespace std;
int main()
{
  ComptModulo mod(0, 3);
  for (int i = 0; i < 10; i++)
    (mod++).affiche();
  cout << endl;
  ComptModulo c = ComptModulo(0,3);
  for (int i = 0; i < 10; i++)
    (++c).affiche();

}
```

```
$ ./a.exe
0 1 2 0 1 2 0 1 2 0
1 2 0 1 2 0 1 2 0 1
```

SURCHARGE D'OPERATEURS

OPÉRATEURS D'INCRÉMENTATION

```
#ifndef __COMPTMODULO_HH__
#define __COMPTMODULO_HH__
class ComptModulo
{
private :
  int _val, _mod;
public:
  ComptModulo(int v, int mod);
  ~ComptModulo();
  ComptModulo operator++();
  ComptModulo operator++(int n);
  void affiche() const;
};
#endif
```

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

ComptModulo::ComptModulo(int n, int mod) :
  _val(n), _mod(mod) {}
ComptModulo::~ComptModulo() {}

ComptModulo ComptModulo::operator++(){
  _val = (_val + 1) % _mod;
  return *this;
}
ComptModulo ComptModulo::operator++(int n){
  ComptModulo c = *this;
  _val = (_val + 1) % _mod;
  return c;
}
void ComptModulo::affiche() const{
  std::cout << _val << " ";
}
```

Les opérateurs d'incrémentation et de décrémentation se surchargent normalement à quelques nuances prés.

- Tout d'abord, il existe pour ces opérateurs une notation préfixée et suffixée, en fonction de si ils sont placés avant ou après la variable à in(dé)crémenter.
- Comment discriminer ces deux notations lors de la surdéfinition ?
- En fonction de cette position avant, ou après, la variable, le comportement n'est pas le même : En notation préfixée, la variable est modifié en premier, et sa valeur dans l'instruction courante est la nouvelle valeur.
- A l'inverse, en notation suffixée, la valeur de la variable est celle d'avant l'opération. Celle-ci n'est effective que lorsque l'instruction courante est terminée.

SURCHARGE D'OPERATEURS

OPÉRATEURS D'INCRÉMENTATION

```
#ifndef __COMPTMODULO_HH__
#define __COMPTMODULO_HH__
class ComptModulo
{
private :
  int _val, _mod;
public:
  ComptModulo(int v, int mod);
  ~ComptModulo();
  ComptModulo operator++();
  ComptModulo operator++(int n);
  void affiche() const;
};
#endif
```

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

ComptModulo::ComptModulo(int n, int mod) :
  _val(n), _mod(mod) {}
ComptModulo::~ComptModulo() {}

ComptModulo ComptModulo::operator++(){
  _val = (_val + 1) % _mod;
  return *this;
}
ComptModulo ComptModulo::operator++(int n){
  ComptModulo c = *this;
  _val = (_val + 1) % _mod;
  return c;
}
void ComptModulo::affiche() const{
  std::cout << _val << " ";
}
```

Pour les discriminer, C++ a adopté la convention suivante :
La notation postfixée contient un paramètre dans l'entête de la fonction (dans l'exemple ci-contre, il s'agit de int n.
Ce paramètre ne sert à rien et n'est là que pour faire la différence entre les deux notation !

SURCHARGE D'OPERATEURS

OPÉRATEURS D'INCRÉMENTATION

```
#ifndef __COMPTMODULO_HH__
#define __COMPTMODULO_HH__
class ComptModulo
{
private :
  int _val, _mod;
public:
  ComptModulo(int v, int mod);
  ~ComptModulo();
  ComptModulo operator++();
  ComptModulo operator++(int n);
  void affiche() const;
};
#endif
```

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

ComptModulo::ComptModulo(int n, int mod) :
  _val(n), _mod(mod) {}
ComptModulo::~ComptModulo() {}

ComptModulo ComptModulo::operator++(){
  _val = (_val + 1) % _mod;
  return *this;
}
ComptModulo ComptModulo::operator++(int n){
  ComptModulo c = *this;
  _val = (_val + 1) % _mod;
  return c;
}
void ComptModulo::affiche() const{
  std::cout << _val << " ";
}
```

Comment faire pour simuler le comportement de l'opérateur postfixé ?

Il faut, comme le montre l'exemple ci contre, d'abord réaliser une copie de l'objet.

Ensuite, on incrémente la donnée membre, ou tout autre opération qui modifie l'instance courante.

La valeur renvoyée sera la copie réalisée avant la modification.

Ainsi, la valeur de obj++ sera toujours obj avant la modification, même si celui est en fait déjà modifié à ce moment là.

SURCHARGE D'OPERATEURS

OPÉRATEURS D'INCRÉMENTATION

```
#ifndef __COMPTMODULO_HH__
#define __COMPTMODULO_HH__
class ComptModulo
{
private :
  int _val, _mod;
public:
  ComptModulo(int v, int mod);
  ~ComptModulo();
  ComptModulo operator++();
  ComptModulo operator++(int n);
  void affiche() const;
};
#endif
```

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

ComptModulo::ComptModulo(int n, int mod) :
  _val(n), _mod(mod) {}
ComptModulo::~ComptModulo() {}

ComptModulo ComptModulo::operator++(){
  _val = (_val + 1) % _mod;
  return *this;
}
ComptModulo ComptModulo::operator++(int n){
  ComptModulo c = *this;
  _val = (_val + 1) % _mod;
  return c;
}
void ComptModulo::affiche() const{
  std::cout << _val << " ";
}
```

On remarque par ailleurs que ces opérateurs renvoie l'objet par valeur.
`*this` est en effet un déréférencement de l'instance courante.
Cela ne devrait donc pas vous choquer que son type soit le bon.

NB : `this` est un pointeur sur l'instance courante, ici il est donc de type `ComptModulo *`, pas de type `ComptModulo` …

SURCHARGE D'OPERATEURS

OPÉRATEURS D'INCRÉMENTATION

```
#ifndef __COMPTMODULO_HH__
#define __COMPTMODULO_HH__
class ComptModulo
{
private :
  int _val, _mod;
public:
  ComptModulo(int v, int mod);
  ~ComptModulo();
  ComptModulo operator++();
  ComptModulo operator++(int n);
  void affiche() const;
};
#endif
```

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

ComptModulo::ComptModulo(int n, int mod) :
  _val(n), _mod(mod) {}
ComptModulo::~ComptModulo() {}

ComptModulo ComptModulo::operator++(){
  _val = (_val + 1) % _mod;
  return *this;
}
ComptModulo ComptModulo::operator++(int n){
  ComptModulo c = *this;
  _val = (_val + 1) % _mod;
  return c;
}
void ComptModulo::affiche() const{
  std::cout << _val << " ";
}
```

Enfin, on remarquera que la notation préfixée réalise bien moins d'opération que la notation postfixée …

On fera donc attention à celle que l'on utilise, notamment dans une boucle for …

SURCHARGE D'OPERATEURS

OPÉRATEURS D'INCRÉMENTATION

```
#ifndef __COMPTMODULO_HH__
#define __COMPTMODULO_HH__
class ComptModulo
{
private :
  int _val, _mod;
public:
  ComptModulo(int v, int mod);
  ~ComptModulo();
  ComptModulo operator++();
  ComptModulo operator++(int n);
  void affiche() const;
};
#endif
```

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

ComptModulo::ComptModulo(int n, int mod) :
  _val(n), _mod(mod) {}
ComptModulo::~ComptModulo() {}

ComptModulo ComptModulo::operator++(){
  _val = (_val + 1) % _mod;
  return *this;
}
ComptModulo ComptModulo::operator++(int n){
  ComptModulo c = *this;
  _val = (_val + 1) % _mod;
  return c;
}
void ComptModulo::affiche() const{
  std::cout << _val << " ";
}
```

L'exemple ci contre ne redéfinie l'opération que pour l'incrémentation, mais la décrémentation est évidemment un copié-collé adapté de celui-ci …

SURCHARGE D'OPERATEURS

OPÉRATEUR []

```
#ifndef __VECTOR_HH__
#define __VECTOR_HH__

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

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

};

#endif
```

```
#include "Vector.hh"

Vector::Vector(int n) : _n(n){
  _val = new int[n];
}
Vector::~Vector(){
  delete[] _val;
}
int & Vector::operator[] (int i)
{
  return _val[i];
}
```

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

}
```

La notation [] que nous avons vu, par exemple lorsqu'on veut 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 surcouchent un tableau. C'est ici notre cas, avec une très simple (et très incomplète !) implémentation d'une classe Vector.

SURCHARGE D'OPERATEURS

OPÉRATEUR []

```
#ifndef __VECTOR_HH__
#define __VECTOR_HH__

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

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

};

#endif
```

```
#include "Vector.hh"

Vector::Vector(int n) : _n(n){
  _val = new int[n];
}
Vector::~Vector(){
  delete[] _val;
}
int & Vector::operator[] (int i)
{
  return _val[i];
}
```

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

}
```

On ne vas pas s'attarder sur les constructeurs et destructeurs que vous connaissez maintenant bien. Leur rôle est ici juste 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.

SURCHARGE D'OPERATEURS

OPÉRATEUR []

```
#ifndef __VECTOR_HH__
#define __VECTOR_HH__

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

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

};

#endif
```

```
#include "Vector.hh"

Vector::Vector(int n) : _n(n){
  _val = new int[n];
}
Vector::~Vector(){
  delete[] _val;
}
int & Vector::operator[] (int i)
{
  return _val[i];
}
```

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

}
```

Intéressons nous à la surcharge de [].
Que désire-t-on faire exactement lors de cette surcharge ?
On souhaite d'une part accéder à un élément donné de notre tableau, pour l'afficher par exemple.
Mais, on veut également pouvoir le modifier !
Ce sont les deux cas que nous voyons dans la fonction main. D'abord une boucle dans laquelle les valeurs accédées sont modifiées, et une autre dans laquelle elles sont juste accédées.

SURCHARGE D'OPERATEURS

OPÉRATEUR []

```
#ifndef __VECTOR_HH__
#define __VECTOR_HH__

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

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

};

#endif
```

```
#include "Vector.hh"

Vector::Vector(int n) : _n(n){
  _val = new int[n];
}
Vector::~Vector(){
  delete[] _val;
}
int & Vector::operator[] (int i)
{
  return _val[i];
}
```

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

}
```

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

SURCHARGE D'OPERATEURS

OPÉRATEUR <<

```
#ifndef __VECTOR_HH__
#define __VECTOR_HH__
#include <iostream>

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

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

#endif
```

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

using namespace std;

int main()
{
  Vector v(5);

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

}
```

```
#include "Vector.hh"

Vector::Vector(int n) : _n(n){
  _val = new int[n];
}
Vector::~Vector(){
  delete[] _val;
}
int & Vector::operator[] (int i)
{
  return _val[i];
}

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

SURCHARGE D'OPERATEURS

OPÉRATEUR <<

```hh
#ifndef __VECTOR_HH__
#define __VECTOR_HH__
#include <iostream>

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

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

#endif
```

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 connait 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 provoque les affichage défini dans le corps de la fonction.

```cpp
#include "Vector.hh"

Vector::Vector(int n) : _n(n){
  _val = new int[n];
}
Vector::~Vector(){
  delete[] _val;
}
int & Vector::operator[] (int i)
{
  return _val[i];
}

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

SURCHARGE D'OPERATEURS

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
```

```
#include "Affine.hh"


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


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

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

using namespace std;

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

}

int main()
{
  Affine a(2, 3);

  valeurEn0(a);
}
```

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 sous la forme d'un objet que l'on paramètre lors de sa construction, et l'utiliser comme tel, par exemple comme paramètre d'une autre fonction.
Ici, l'exemple est trivial, et on aurait pu aussi utiliser un pointeur sur fonction.
Mais dans l'exemple de matrice, par exemple, ou la surcharge de l'opérateur () a un sens certain, celui-ci peut s'avérer pratique à surcharger.

SURCHARGE D'OPERATEURS

L'OPÉRATEUR =

```
#ifndef __VECTOR_HH__
#define __VECTOR_HH__
#include <iostream>

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

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

#endif
```

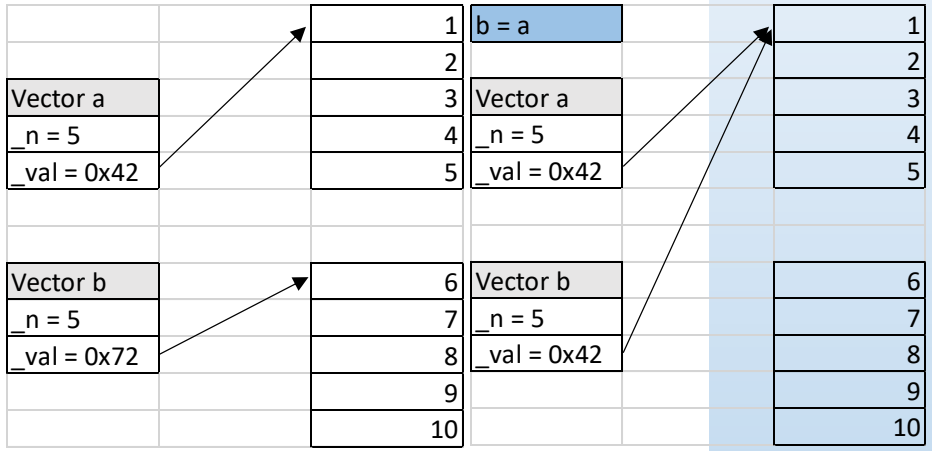
L'opérateur = va permettre de redéfinir les opérations d'affectations de variable objet.

On l'utilisera surtout, comme dans le cas de l'opérateur de recopie, lorsqu'une ou plusieurs données membres sont des pointeurs. En effet, l'opérateur par défaut recopiera les valeurs des données membres, c'est-à-dire des adresses, et non pas les valeurs pointées par celles-ci.

SURCHARGE D'OPERATEURS

L'OPÉRATEUR =

```
#ifndef __VECTOR_HH__
#define __VECTOR_HH__
#include <iostream>

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

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

#endif
```



On voit sur ce schémas les deux conséquences que cela peut engendrés.

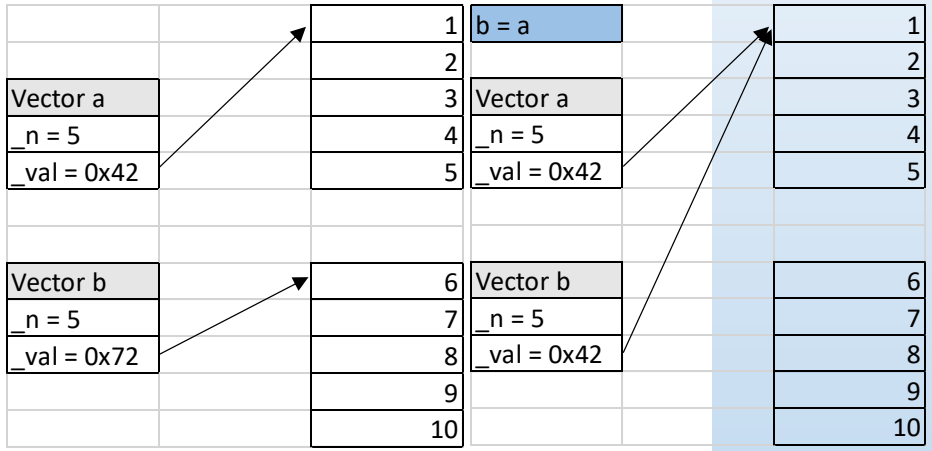
D'une part les deux vecteurs partagent maintenant les même données, ce que nous ne voulons probablement pas.

D'autre part, on ne libère pas l'espace alloué pour le tableau qui a pour adresse 0x72, ce que nous ne voulons pas non plus.

SURCHARGE D'OPERATEURS

L'OPÉRATEUR =

```
#ifndef __VECTOR_HH__
#define __VECTOR_HH__
#include <iostream>

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

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

#endif
```



La surcharge de l'opérateur = doit donc pouvoir gérer ce genre de cas.

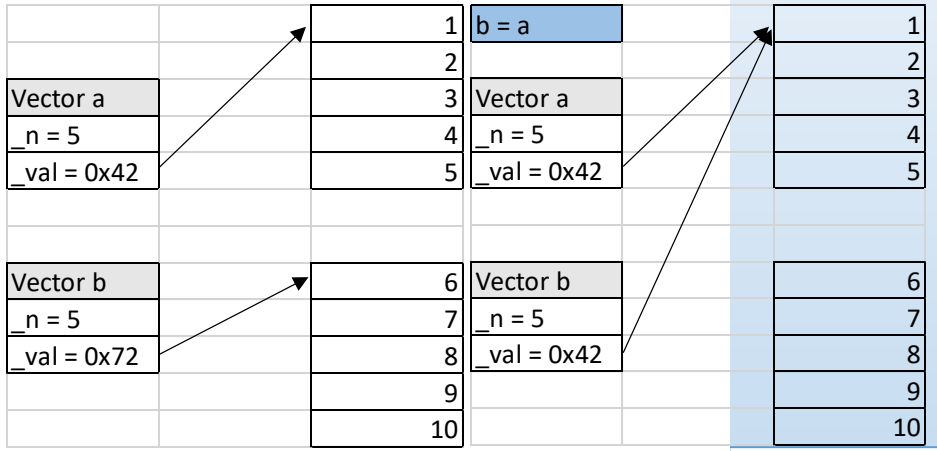
Il va donc libérer l'espace de l'ancien objet, et faire une « copie profonde » de l'objet affecté.

Il y a cependant un cas particulier à prendre en compte, l'affectation d'une variable à elle-même : a = a;

SURCHARGE D'OPERATEURS

L'OPÉRATEUR =

```
#ifndef __VECTOR_HH__
#define __VECTOR_HH__
#include <iostream>

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

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

#endif
```



Ci-dessous une implémentation possible de l'opérateur = dans le cas des Vector.
On règle le problème de l'auto affectation en le vérifiant par une condition.
Ensuite, on libère l'espace de l'ancienne valeur de notre objet, et on le remplace par le nouveau.

```
Vector Vector::operator= (const Vector&v)
{
  if (this != &v){
    delete _val;
    _val = new int[_n = v._n];
    for (int i = 0; i < _n; ++i) _val[i] = v._val[i];
  }
  return (*this);
}
```

SURCHARGE D'OPERATEURS

FORME CANONIQUE D'UNE CLASSE

- On appelle forme canonique d'une classe, une classe où on aura au moins défini un **constructeur**, un **constructeur par recopie**, un **destructeur**, et surchargé **l'opérateur d'affectation =**.
- On notera qu'il ne sera toutefois pas nécessaire que toutes ces fonctions soient déclarées publiques.
- On pourra par exemple interdire l'usage de =, qui peut être parfois indésirable, en le déclarant private.

LES PATRONS DE FONCTIONS

LES PATRONS DE FONCTIONS

Nous allons maintenant introduire une fonctionnalité très puissante 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.

LES PATRONS DE FONCTIONS

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 = 2.0, bd = 3.0;

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

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

En fait, le compilateur va générer de manière transparente pour l'utilisateur, autant de fonction qu'il est nécessaire en fonction des types de paramètres qu'on passera à la fonction.

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

LES PATRONS DE FONCTIONS

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 = 2.0, bd = 3.0;

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

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 un type générique T qu'on pourra utiliser au sein de notre fonction.

LES PATRONS DE FONCTIONS

PARAMÈTRES EXPRESSIONS

```
#include <iostream>

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[] = {2.0, 3.0, 4.0, 6.0, 2.0, 1.0};
  double max = maxtab(tab, 6);
  std::cout << "plus grand element : "
            << max << std::endl;
}
```

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.

LES PATRONS DE FONCTIONS

PARAMÈTRES EXPRESSIONS

```
#include <iostream>

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[] = {2.0, 3.0, 4.0, 6.0, 2.0, 1.0};
  double max = maxtab(tab, 6);
  std::cout << "plus grand element : "
            << max << std::endl;
}
```

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 de notre tableau.

On remarque par ailleurs que des types non « templaté » peuvent entrer comme paramètres d'une fonction template, il s'agit de paramètre expressions.

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

LES PATRONS DE FONCTIONS

SURDÉFINITIONS DE FONCTIONS

```
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;
}
```

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

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

LES PATRONS DE FONCTIONS

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;
}

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

On peut également définir un template pour un algorithme général qui est valable quelque 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 char *, l'opérateur < n'aurait pas le sens auquel on s'attendrait : il comparerait les valeurs des pointeurs et non des chaines de caractère. Pour cela, on spécialise la fonction et on utilise la fonction strcmp pour comparer les chaines une à une.

LES PATRONS DE FONCTIONS

POUR FINIR

Enfin, il n'est pas forcément évident d'écrire une et de spécialiser une fonction template. En effet, il faut faire attention aux cas ambigües – c'est-à-dire où le compilateur ne sait pas si 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 CLASSE

LES PATRONS DE CLASSE

Comme pour les fonctions template, il existe un mécanisme similaire pour les classes.

Bien que semblable aux patrons de fonctions sur de nombreux points, il existe des différences avec les tempate de classe.

On va voir que cela permet d'implémenter un code générique et réutilisable.

LES PATRONS DE CLASSE

LE RETOUR DE LA CLASSE VECTOR

```
#ifndef __VECTOR_HPP__
#define __VECTOR_HPP__

template <typename T>
class Vector
{
private:
    T* _data;
    unsigned int _n;
public:
    Vector(unsigned int n):_n(n){
        _data = new T[_n];
    }
    ~Vector(){
        if (_data) delete [] _data;
    }
    T& operator[] (unsigned int n){
        return _data[n];
    }
};
#endif
```

On se souvient de la classe Vector que nous avions définie dans ce cours.

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

Perte de temps, d'énergie …. Et d'argent !

LES PATRONS DE CLASSE

LE RETOUR DE LA CLASSE VECTOR

```
#ifndef __VECTOR_HPP__
#define __VECTOR_HPP__

template <typename T>
class Vector
{
private:
   T* _data;
   unsigned int _n;
public:
   Vector(unsigned int n):_n(n){
      _data = new T[_n];
   }
   ~Vector(){
      if (_data) delete [] _data;
   }
   T& operator[] (unsigned int n){
      return _data[n];
   }
};
#endif
```

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 :

A l'inverse des classe 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.

LES PATRONS DE CLASSE

LE RETOUR DE LA CLASSE VECTOR

```
#ifndef __VECTOR_HPP__
#define __VECTOR_HPP__

template <typename T>
class Vector
{
private:
    T* _data;
    unsigned int _n;
public:
    Vector(unsigned int n):_n(n){
        _data = new T[_n];
    }
    ~Vector(){
        if (_data) delete [] _data;
    }
    T& operator[] (unsigned int n){
        return _data[n];
    }
};
#endif
```

En résumé,

Déclaration d'une classe → fichier .hh

Définition d'une classe → fichier .cpp

Déclaration d'un template de classe → fichier .hpp

Il ne s'agit que d'une convention. Ce sera celle adoptée dans ce cours.

LES PATRONS DE CLASSE

LE RETOUR DE LA CLASSE VECTOR

```
#ifndef __VECTOR_HPP__
#define __VECTOR_HPP__

template <typename T>
class Vector
{
private:
    T* _data;
    unsigned int _n;
public:
    Vector(unsigned int n):_n(n){
        _data = new T[_n];
    }
    ~Vector(){
        if (_data) delete [] _data;
    }
    T& operator[] (unsigned int n){
        return _data[n];
    }
};
#endif
```

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

LES PATRONS DE CLASSE

LE RETOUR DE LA CLASSE VECTOR

```
#ifndef __VECTOR_HPP__
#define __VECTOR_HPP__

template <typename T>
class Vector
{
private:
    T* _data;
    unsigned int _n;
public:
    Vector(unsigned int n):_n(n){
        _data = new T[_n];
    }
    ~Vector(){
        if (_data) delete [] _data;
    }
    T& operator[] (unsigned int n){
        return _data[n];
    }
};
#endif
```

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

int main()
{
    Vector<double> vect(10);
    for (int i = 0; i < 10; i++)
        vect[i] = i;
    for (int i = 0; i < 10; i++)
        std::cout << vect[i] << std::endl;
}
```

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

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

LES PATRONS DE CLASSE

LE RETOUR DE LA CLASSE VECTOR

```
#ifndef __VECTOR_HPP__
#define __VECTOR_HPP__

template <typename T>
class Vector
{
private:
    T* _data;
    unsigned int _n;
public:
    Vector(unsigned int n):_n(n){
        _data = new T[_n];
    }
    ~Vector(){
        if (_data) delete [] _data;
    }
    T& operator[] (unsigned int n){
        return _data[n];
    }
};
#endif
```

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

int main()
{
    Vector<double> vect(10);
    for (int i = 0; i < 10; i++)
        vect[i] = i;
    for (int i = 0; i < 10; i++)
        std::cout << vect[i] << std::endl;
}
```

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.

LES PATRONS DE CLASSE

LE RETOUR DE LA CLASSE VECTOR

```
#ifndef __VECTOR_HPP__
#define __VECTOR_HPP__

template <typename T>
class Vector
{
private:
    T* _data;
    unsigned int _n;
public:
    Vector(unsigned int n):_n(n){
        _data = new T[_n];
    }
    ~Vector(){
        if (_data) delete [] _data;
    }
    T& operator[] (unsigned int n){
        return _data[n];
    }
};
#endif
```

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

int main()
{
    Vector<double> vect(10);
    for (int i = 0; i < 10; i++)
        vect[i] = i;
    for (int i = 0; i < 10; i++)
        std::cout << vect[i] << std::endl;
}
```

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.

LES PATRONS DE CLASSE

PLUS COMPLIQUÉ

```hpp
#ifndef __VECTOR_HPP__
#define __VECTOR_HPP__

template <typename T>
class Vector
{
private:
    T* _data;
    unsigned int _n;
public:
    Vector(unsigned int n):_n(n){
        _data = new T[_n];
    }
    ~Vector(){
        if (_data) delete [] _data;
    }
    T& operator[] (unsigned int n){
        return _data[n];
    }
};
#endif
```

```hpp
#ifndef __POINT_HPP__
#define __POINT_HPP__

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

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

#endif
```

```cpp
#include <iostream>
#include "Vector.hpp"
#include "Point.hpp"
int main()
{
    Vector<double> vect(10);
    for (int i = 0; i < 10; i++)
        vect[i] = i;
    for (int i = 0; i < 10; i++)
        std::cout << vect[i] << std::endl;

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

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

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

Mais on peut aussi définir un vecteur de point d'entier !

LES PATRONS DE CLASSE

PLUS COMPLIQUÉ

```
#ifndef __VECTOR_HPP__
#define __VECTOR_HPP__

template <typename T>
class Vector
{
private:
    T* _data;
    unsigned int _n;
public:
    Vector(unsigned int n):_n(n){
        _data = new T[_n];
    }
    ~Vector(){
        if (_data) delete [] _data;
    }
    T& operator[] (unsigned int n){
        return _data[n];
    }
};
#endif
```

```
#ifndef __POINT_HPP__
#define __POINT_HPP__

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

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

#endif
```

```
#include <iostream>
#include "Vector.hpp"
#include "Point.hpp"
int main()
{
    Vector<double> vect(10);
    for (int i = 0; i < 10; i++)
        vect[i] = i;
    for (int i = 0; i < 10; i++)
        std::cout << vect[i] << std::endl;

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

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 Point.

Comme Point est un type template, il est moral qu'une fonction l'utilisant soit aussi template… sinon quel type de Point le compilateur instancierait ?

LES PATRONS DE CLASSE

TYPES EXPRESSION

```
#ifndef __MATRICE_HPP__
#define __MATRICE_HPP__
#include <iostream>
using namespace std;
template <typename T, int N, int M>
class Matrice
{
  private:
    T _datas[N][M];  int _n; int _m;
  public:
    Matrice():_n(N), _m(M){}
    T* operator[] (int i){ return _datas[i]; }
    int getN() const { return _n; }
    int getM() const { return _m; }
    friend ostream& operator <<(ostream& o, Matrice& m) {
      for (int i = 0; i < m._n; ++i){
        for (int j = 0; j < m._m; ++j)
          o << m._datas[i][j] <<" ";
        o << endl;
      }
      return o;
    }
};
#endif
```

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

int main()
{
  Matrice<double, 2, 2> mat;

  mat[0][0] = 0;
  mat[0][1] = 1;
  mat[1][0] = 1;
  mat[1][1] = 0;

  cout << mat << endl;
}
```

On peut aussi définir des types expressions. Il s'agit de donner des valeurs à travers la déclaration du template à des constantes présentes dans le code déclaré.
Ici, on définit une Matrice 2x2 dans le main, au moment de la déclaration de la variable mat.
Le compilateur va alors générer à partir du patron que nous avons déclaré une matrice dont les données seront un tableau de 2x2 double.
L'avantage est qu'on ne passe pas par un mécanisme de type allocation dynamique, toutes les données étant stockées dans l'instance courante.
Mais la dimension de celle-ci est figé et ne changera pas de manière dynamique à l'exécution.

LES PATRONS DE CLASSE

TYPES EXPRESSION

```
#ifndef __MATRICE_HPP__
#define __MATRICE_HPP__
#include <iostream>
using namespace std;
template <typename T, int N, int M>
class Matrice
{
  private:
    T _datas[N][M];  int _n; int _m;
  public:
    Matrice():_n(N), _m(M){}
    T* operator[] (int i){ return _datas[i]; }
    int getN() const { return _n; }
    int getM() const { return _m; }
    friend ostream& operator <<(ostream& o, Matrice& m) {
      for (int i = 0; i < m._n; ++i){
        for (int j = 0; j < m._m; ++j)
          o << m._datas[i][j] <<" ";
        o << endl;
      }
      return o;
    }
};
#endif
```

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


int main()
{
  Matrice<double, 2, 2> mat;


  mat[0][0] = 0;
  mat[0][1] = 1;
  mat[1][0] = 1;
  mat[1][1] = 0;


  cout << mat << endl;
}
```

Ici, on surcharge l'opérateur [] pour pouvoir accéder aux éléments de notre matrice.
On remarquera que cela doit se faire en deux temps.
En fait on surcharge [] pour accéder aux tableaux représentant les lignes. Ensuite on accédera aux colonnes en utilisant l'opérateur [] habituel sur un tableau.

LES PATRONS DE CLASSE

TYPES EXPRESSION

```
#ifndef __MATRICE_HPP__
#define __MATRICE_HPP__
#include <iostream>
using namespace std;
template <typename T, int N, int M>
class Matrice
{
  private:
    T _datas[N][M];  int _n; int _m;
  public:
    Matrice():_n(N), _m(M){}
    T* operator[] (int i){ return _datas[i]; }
    int getN() const { return _n; }
    int getM() const { return _m; }
    friend ostream& operator <<(ostream& o, Matrice& m) {
      for (int i = 0; i < m._n; ++i){
        for (int j = 0; j < m._m; ++j)
          o << m._datas[i][j] <<" ";
        o << endl;
      }
      return o;
    }
};
#endif
```

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


int main()
{
  Matrice<double, 2, 2> mat;


  mat[0][0] = 0;
  mat[0][1] = 1;
  mat[1][0] = 1;
  mat[1][1] = 0;


  cout << mat << endl;
}
```

La surcharge de l'opérateur << pour l'affichage se fait quant à lui à l'aide d'une fonction amie.
On remarque que ici, dans le cadre des patrons de classe, la fonction amie est directement inclue dans le code de la classe.

LES PATRONS DE CLASSE

POUR FINIR

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 définie 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++ (http://eigen.tuxfamily.org/) est une librairie pour l'algèbre linéaire et des algorithmes très optimisés qui lui sont associés.

LES PATRONS DE CLASSE

POUR FINIR

La notion de template est plus vaste que celle abordée dans ce cours.

Ainsi, on n'a pas abordé 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++.

HÉRITAGE

L'HÉRITAGE

La notion d'héritage en programmation orienté 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.

L'HÉRITAGE

EXEMPLE

```
#ifndef __FORME_HH__
#define __FORME_HH__

#include <string>
class Forme
{
private:
    std::string _nom;

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

#endif
```

Forme.hh

```
#include "Forme.hh"

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

Forme.cpp

On commence par écrire une classe très simple, Forme, qui ne contient qu'une chaine qui contiendra le nom de notre forme.

Une fonction sera chargée d'initialiser notre chaine.

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

L'HÉRITAGE

EXEMPLE

```
#ifndef __FORME_HH__
#define __FORME_HH__

#include <string>
class Forme
{
private:
   std::string _nom;

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

#endif
```

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 initialiser la chaine 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 .o).

L'HÉRITAGE

EXEMPLE

```
#ifndef __FORME_HH__
#define __FORME_HH__

#include <string>
class Forme
{
private:
    std::string _nom;

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

#endif
```

Forme.hh

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

class Rond: public Forme
{
private:
    double _diametre;

public:
    void setDiametre(double);
};
#endif
```

Rond.hh

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

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

Rond.cpp

On va donc maintenant créer une première classe fille de la classe Forme.

A savoir la classe Rond.

On remarque la ligne :

class Rond: public Forme

Dans la déclaration de la classe rond. Les « : » suivi du public Forme signifie que Rond hérite de la classe Forme.

Le mot clé « public » ici sera expliqué dans la suite de ce cours.

L'HÉRITAGE

EXEMPLE

Forme.hh

```
#ifndef __FORME_HH__
#define __FORME_HH__

#include <string>
class Forme
{
private:
    std::string _nom;

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

#endif
```

Rond.hh

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

class Rond: public Forme
{
private:
    double _diametre;

public:
    void setDiametre(double);
};
#endif
```

Rond.cpp

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

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

Carre.hh

```
#ifndef __CARRE_HH__
#define __CARRE_HH__
#include "Forme.hh"
class Carre: public Forme
{
private:
    double _longueur;
public:
    void setLongueur(double);
};
#endif
```

Carre.cpp

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

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

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 diamètres, le carré une longueur.

L'HÉRITAGE

EXEMPLE

```
#ifndef __FORME_HH__
#define __FORME_HH__

#include <string>
class Forme
{
private:
    std::string _nom;

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

#endif
```

Forme.hh

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

class Rond: public Forme
{
private:
    double _diametre;

public:
    void setDiametre(double);
};
#endif
```

Rond.hh

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

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

Rond.cpp

```
#ifndef __CARRE_HH__
#define __CARRE_HH__
#include "Forme.hh"
class Carre: public Forme
{
private:
    double _longueur;
public:
    void setLongueur(double);
};
#endif
```

Carre.hh

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

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

Carre.cpp

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

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

main.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 !

L'HÉRITAGE

EXEMPLE

Forme.hh

```
#ifndef __FORME_HH__
#define __FORME_HH__

#include <string>
class Forme
{
private:
    std::string _nom;

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

};

#endif
```

Rond.hh

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

class Rond: public Forme
{
private:
    double _diametre;

public:
    void setDiametre(double);
};
#endif
```

Carre.hh

```
#ifndef __CARRE_HH__
#define __CARRE_HH__
#include "Forme.hh"
class Carre: public Forme
{
private:
    double _longueur;
public:
    void setLongueur(double);
};
#endif
```

Rond.cpp

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

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

Carre.cpp

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

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

Forme.cpp

```
void Forme::affiche()
{
    std::cout << "Je suis de type " <<  _nom << std::endl;
}
```

On définie 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 également ce que contient leur donnée `_nom` héritée de classe Forme.
On a donc modifié les fonctionnalité de 3 classes en en modifiant qu'une seule. Pas mal !
Mais, l'affichage ne prend pas en compte la spécificité de chaque classe…

L'HÉRITAGE

EXEMPLE

Forme.hh

```
#ifndef __FORME_HH__
#define __FORME_HH__

#include <string>
class Forme
{
private:
    std::string _nom;

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

};

#endif
```

Rond.hh

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

class Rond: public Forme
{
private:
    double _diametre;

public:
    void setDiametre(double);
};
#endif
```

Carre.hh

```
#ifndef __CARRE_HH__
#define __CARRE_HH__
#include "Forme.hh"
class Carre: public Forme
{
private:
    double _longueur;
public:
    void setLongueur(double);
};
#endif
```

Rond.cpp

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

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

Carre.cpp

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

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

Forme.cpp

```
void Forme::affiche()
{
    std::cout << "Je suis de type " <<  _nom << std::endl;
}
```

main.cpp

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

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

```
$ ./a.exe
Je suis de type forme générale.
Je suis de type carre
Je suis de type rond
```

L'HÉRITAGE

REDÉFINITION D'UNE FONCTION HÉRITÉE

Forme.hh

```
#ifndef __FORME_HH__
#define __FORME_HH__

#include <string>
class Forme
{
private:
    std::string _nom;

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

};

#endif
```

Rond.hh

```
#ifndef __ROND_HH__
#define __ROND_HH__
#include "Forme.hh"
class Rond: public Forme
{
private:
    double _diametre;
public:
    void setDiametre(double);
    void affiche();
};
#endif
```

Rond.cpp

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

void Rond::setDiametre(double d)
{
_diametre = d;
}
void Rond::affiche()
{
    std::cout << "Mon diamètre est de
"
    << _diametre << std::endl;
}
```

Carre.hh

```
#ifndef __CARRE_HH__
#define __CARRE_HH__
#include "Forme.hh"
class Carre: public Forme
{
private:
    double _longueur;
public:
    void setLongueur(double);
    void affiche();
};
#endif
```

Carre.cpp

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

void Carre::setLongueur(double l)
{
    _longueur = l;
}
void Carre::affiche()
{
    std::cout << "Ma longueur est de "
    << _longueur << std::endl;
}
```

Pour afficher les spécificité de chaque enfant, on va déclarer puis définir une fonction affiche dans chaque classe fille…

```
Je suis de type forme générale.
Ma longueur est de 5
Mon diamètre est de 3
```

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 classe fille, il faudrait aussi l'écrire ??

C'est pas très orienté objet ca …

L'HÉRITAGE

REDÉFINITION D'UNE FONCTION HÉRITÉE

Forme.hh

```
#ifndef __FORME_HH__
#define __FORME_HH__

#include <string>
class Forme
{
private:
    std::string _nom;

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

};

#endif
```

Rond.hh

```
#ifndef __ROND_HH__
#define __ROND_HH__
#include "Forme.hh"
class Rond: public Forme
{
private:
    double _diametre;
public:
    void setDiametre(double);
    void affiche();
};
#endif
```

Rond.cpp

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

void Rond::setDiametre(double d)
{
_diametre = d;
}
void Rond::affiche()
{
Forme::affiche();
std::cout << "Mon diamètre est de "
    << _diametre << std::endl;
}
```

Carre.hh

```
#ifndef __CARRE_HH__
#define __CARRE_HH__
#include "Forme.hh"
class Carre: public Forme
{
private:
    double _longueur;
public:
    void setLongueur(double);
    void affiche();
};
#endif
```

Carre.cpp

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

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

On va plutôt essayer d'appeler dans la fonction affiche fille, la fonction affiche parente.

Si on écrit juste 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.

L'HÉRITAGE

REDÉFINITION D'UNE FONCTION HÉRITÉE

Forme.hh

```
#ifndef __FORME_HH__
#define __FORME_HH__

#include <string>
class Forme
{
private:
    std::string _nom;

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

};

#endif
```

Rond.hh

```
#ifndef __ROND_HH__
#define __ROND_HH__
#include "Forme.hh"
class Rond: public Forme
{
private:
    double _diametre;
public:
    void setDiametre(double);
    void affiche();
};
#endif
```

Carre.hh

```
#ifndef __CARRE_HH__
#define __CARRE_HH__
#include "Forme.hh"
class Carre: public Forme
{
private:
    double _longueur;
public:
    void setLongueur(double);
    void affiche();
};
#endif
```

Rond.cpp

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

void Rond::setDiametre(double d)
{
_diametre = d;
}
void Rond::affiche()
{
Forme::affiche();
std::cout << "Mon diamètre est de "
    << _diametre << std::endl;
}
```

Carre.cpp

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

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

```
$ ./a.exe
Je suis de type forme générale.

Je suis de type carre
Ma longueur est de 5

Je suis de type rond
Mon diamètre est de 3
```

Nous avons maintenant un affichage adapté en fonction de, 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 !

L'HÉRITAGE

REDÉFINITION D'UNE FONCTION HÉRITÉE

Forme.hh

```
#ifndef __FORME_HH__
#define __FORME_HH__

#include <string>
class Forme
{
private:
    std::string _nom;

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

};

#endif
```

Rond.hh

```
#ifndef __ROND_HH__
#define __ROND_HH__
#include "Forme.hh"
class Rond: public Forme
{
private:
    double _diametre;
public:
    void setDiametre(double);
    void affiche();
};
#endif
```

Carre.hh

```
#ifndef __CARRE_HH__
#define __CARRE_HH__
#include "Forme.hh"
class Carre: public Forme
{
private:
    double _longueur;
public:
    void setLongueur(double);
    void affiche();
};
#endif
```

Rond.cpp

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

void Rond::setDiametre(double d)
{
_diametre = d;
}
void Rond::affiche()
{
Forme::affiche();
std::cout << "Mon diamètre est de "
    << _diametre << std::endl;
}
```

Carre.cpp

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

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

```
$ ./a.exe
Je suis de type forme générale.

Je suis de type carre
Ma longueur est de 5

Je suis de type rond
Mon diamètre est de 3
```

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

Il faut un peu modifier la syntaxe de l'appel de la fonction :
r.Forme::affiche();

L'HÉRITAGE

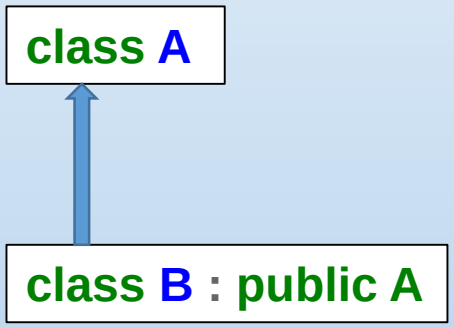
SURCHARGE ET REDÉFINITION

On fera attention sur un point :

Une fonction membre redéfinie dans une classe masque automatiquement les fonctions héritée, 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.

L'HÉRITAGE

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 paramètres 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.

L'HÉRITAGE

CONSTRUCTEURS & DESTRUCTEURS

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

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

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

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

On va rencontrer une difficulté si on doit fournir des paramètres au constructeur de A. En effet, à priori, les paramètres fournis au constructeur de B sont sensés être utilisés par lui. Or le constructeur de A doit être appelé avant. On va avoir recours à la même syntaxe que pour les objets membres lors de la définition du constructeur de B.

L'HÉRITAGE

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 une troisième liée 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.

L'HÉRITAGE

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 pouvoirs 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.

L'HÉRITAGE

CONTRÔLE DES ACCÈS

```
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();
};
```

```
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;
}
```

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.

L'HÉRITAGE

DÉRIVATION PUBLIQUE & DÉRIVATION PRIVÉE

Depuis le début de ce cours sur l'héritage, nous avons déclaré qu'une classe dérivée 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 publiques de la classe parente, comme si elles faisaient parties de la classe fille.

L'HÉRITAGE

DÉRIVATION PUBLIQUE & DÉRIVATION PRIVÉE

```
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);
};
```

```
#include "A.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) {}
```

```
#include "A.hh"

int main()
{
B b(5);
C c(5);

b.setA(6);
c.setA(6);
}
```

```
 g++ A.cpp  main.cpp
In file included from main.cpp:1:0:
A.hh: Dans la fonction ‘int main()’:
A.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 publiques de la classe A de l'extérieur.

L'HÉRITAGE

DÉRIVATION PUBLIQUE & DÉRIVATION PRIVÉE

```
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);
};
```

```
#include "A.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) {}
```

```
#include "A.hh"

int main()
{
B b(5);
C c(5);

b.setA(6);
c.setA(6);
}
```

```
 g++ A.cpp  main.cpp
In file included from main.cpp:1:0:
A.hh: Dans la fonction ‘int main()’:
A.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’
```

L'intérêt de la dérivation privée, par exemple, existe quand une classe redéfinie une fonction d'une classe parente. Il n'y a probablement plus de raison d' accéder à cette fonction dans la classe parente. Et on peut ainsi en interdire l'utilisation.

L'HÉRITAGE

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.

L'HÉRITAGE

CLASSE DE BASE & CLASSE DÉRIVÉE

En Programmation orienté objet, on considère qu'un objet d'une classe dérivée peut « remplacer » un objet d'une classe de base. Un objet dérivée d'une classe A peut intervenir quand une classe A est attendu.

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 prés. Elle ne s'applique que dans le cas d'un héritage publique.

Il existe donc une conversion implicite d'une classe fille vers un type de la classe parente.

L'HÉRITAGE

CONVERSION

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

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

.hh

```cpp
A::A(int a)
{
    std::cout << "Cons A"
    << std::endl;
    _a = a;
}
void A::affiche()
{
    std::cout << "A : ";
    std::cout << _a
    << std::endl;
}

B::B(int a, int b) : A(a)
{
    std::cout << "CONS B"
    << std::endl;
    _b = b;
}
void B::affiche()
{
    std::cout << "B : ";
    A::affiche();
    std::cout << _b
    << std::endl;
}
```

.cpp

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

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

main.cpp

Dans cet exemple – cas d'école – on déclare deux classes, A, et 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.

L'HÉRITAGE

CONVERSION

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

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

.hh

```cpp
A::A(int a)
{
    std::cout << "Cons A"
    << std::endl;
    _a = a;
}
void A::affiche()
{
    std::cout << "A : ";
    std::cout << _a
    << std::endl;
}

B::B(int a, int b) : A(a)
{
    std::cout << "CONS B"
    << std::endl;
    _b = b;
}
void B::affiche()
{
    std::cout << "B : ";
    A::affiche();
    std::cout << _b
    << std::endl;
}
```

.cpp

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

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

main.cpp

```
$ ./a.exe
On construit a, b
Cons A
Cons A
CONS 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 'b' est converti, il 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 'a', sa donnée privée a bien pris la valeurs de la partie A de b.

L'HÉRITAGE

CONVERSION, POINTEURS & RÉFÉRENCES

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

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

.hh

```
A::A(int a)
{
    std::cout << "Cons A"
    << std::endl;
    _a = a;
}
void A::affiche()
{
    std::cout << "A : ";
    std::cout << _a
    << std::endl;
}

B::B(int a, int b) : A(a)
{
    std::cout << "CONS B"
    << std::endl;
    _b = b;
}
void B::affiche()
{
    std::cout << "B : ";
    A::affiche();
    std::cout << _b
    << std::endl;
}
```

.cpp

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

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

main.cpp

```
$ ./a.exe
On construit b
Cons A
CONS 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.

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, même si on l'initialise avec un autre type, les valeurs supplémentaires sont perdues lors de la conversion.

L'HÉRITAGE

LIMITATIONS

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

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

```
A::A(int a)
{
    std::cout << "Cons A"
    << std::endl;
    _a = a;
}
void A::affiche()
{
    std::cout << "A : ";
    std::cout << _a
    << std::endl;
}

B::B(int a, int b) : A(a)
{
    std::cout << "CONS B"
    << std::endl;
    _b = b;
}
void B::affiche()
{
    std::cout << "B : ";
    A::affiche();
    std::cout << _b
    << std::endl;
}
```

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

   a->affiche();
}
```

```
$ ./a.exe
Cons A
CONS B
A : 5
```

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

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 … 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.

L'HÉRITAGE

LIMITATIONS

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

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

```
A::A(int a)
{
    std::cout << "Cons A"
    << std::endl;
    _a = a;
}
void A::affiche()
{
    std::cout << "A : ";
    std::cout << _a
    << std::endl;
}

B::B(int a, int b) : A(a)
{
    std::cout << "CONS B"
    << std::endl;
    _b = b;
}
void B::affiche()
{
    std::cout << "B : ";
    A::affiche();
    std::cout << _b
    << std::endl;
}
```

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

   a->affiche();
}
```

```
$ ./a.exe
Cons A
CONS 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 c'est 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 un peu plus tard un mécanisme qui permet de passer outre cette difficulté.

L'HÉRITAGE

CONSTRUCTEUR DE RECOPIE & HÉRITAGE

Nous avons déjà parlé du constructeur de recopie. Celui-ci est appelé dans le cas d'une initialisation d'un objet par un objet de même type, ou d'une transmission par valeur ou par retour d'une fonction.

Il y a plusieurs cas possible, suivant si le constructeur de recopie est déclaré ou non dans la classe dérivée. Et des nuances un peu délicates comme on va le voir.

L'HÉRITAGE

CONSTRUCTEUR DE RECOPIE & HÉRITAGE

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

class B : public A
{
private:
    int _b;
public:
    B(int, int);
    B(const B&);
};
```

```
A::A(int a)
{
    std::cout << "Cons A"
    << std::endl;
    _a = a;
}

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

B::B(const B& b) : A(b)
{
    std::cout << "Recopie B"
    << std::endl;
    _b = b._b;
}
```

```
void affiche(B b){
}

int main()
{
    B b(5,6);
    affiche(b);
}
```

```
$ ./a.exe
Cons A
CONS B
Cons A
Recopie B
```

Si l'objet b de type B définit un constructeur de recopie, celui-ci n' appellera pas automatiquement celui de A, comme c'est déjà le cas pour les constructeurs. Il faut donc appeler le constructeur de recopie de A dans le constructeur de recopie de B, en lui passant … 'b' en paramètre. Celui-ci sera converti au passage et la partie A de B sera copiée.

Pour le reste, on copie les données membres de 'B privée de A' dans le constructeur de recopie de B.

L'HÉRITAGE

CONSTRUCTEUR DE RECOPIE & HÉRITAGE

```
class A
{
private:
   int _a;
   A(const A&);

public:
   A(int);
};

class B : public A
{
private:
   int _b;
public:
   B(int, int);
   B(const B&);
};
```

```
A::A(int a)
{
   std::cout << "Cons A"
   << std::endl;
   _a = a;
}

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

B::B(const B& b) : A(b)
{
   std::cout << "Recopie B"
   << std::endl;
   _b = b._b;
}
```

```
void affiche(B b){
}

int main()
{
   B b(5,6);
   affiche(b);
}
```

Que se passe t il si on déclare un constructeur de recopie de A en privé ?

On interdit alors aux objets héritant de A de se copier !

```
$ g++ A.cpp main.cpp
In file included from A.cpp:2:0:
A.hh: Dans le constructeur de copie ‘B::B(const B&)’:
A.hh:8:5: erreur : ‘A::A(const A&)’ is private
     A(const A&);
     ^
A.cpp:18:23: erreur : à l'intérieur du contexte
 B::B(const B& b) : A(b)
                       ^
```

L'HÉRITAGE

OPÉRATEUR D'AFFECTATION & HÉRITAGE

Si la classe dérivée ne surdéfinit pas l'opérateur d'affectation '=', l'affectation se déroule en utilisant l'opérateur par défaut.

La partie héritée de A est alors traitée par l'affectation prévue par A si elle existe et qu'elle est public, ou par l'affectation par défaut de A si elle n'est pas redéfinie.

Si la classe A surdéfinit son opérateur d'affectation en privé, alors l'affectation est interdite pour elle, ainsi que pour ses classes dérivées, comme dans le cas des constructeurs de recopie.

L'HÉRITAGE

OPÉRATEUR D'AFFECTATION & HÉRITAGE

```
class A
{
private:
    int _a;
public:
    A(int);
    A& operator=(const A&);
};

class B : public A
{
private:
    int _b;
public:
    B(int, int);
    //B& operator=(const B&);
};
```

```
A::A(int a)
{
    std::cout << "Cons A"
    << std::endl;
    _a = a;
}
A& A::operator=(const A&a)
{
    std::cout << "= A"
        << std::endl;
    _a = a._a;
    return *this;
}

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

/*B& B::operator=(const B&b)
{
    std::cout << "= B"
        << std::endl;
    _b = b._b;
    return *this;
}*/
```

```
int main()
{
    B b1(5,6);
    B b2(6,7);

    b2 = b1;
}
```

```
$ ./a.exe
Cons A
CONS B
Cons A
CONS B
= A
```

Ici, B ne surdéfinit pas d'opérateur d'affectation (celui-ci est en commentaire…).

L'opérateur '=' de A est par contre défini, et celui-ci est appelé lorsqu'on écrit b2 = b1.

L'HÉRITAGE

OPÉRATEUR D'AFFECTATION & HÉRITAGE

```
class A
{
private:
    int _a;
public:
    A(int);
    A& operator=(const A&);
};

class B : public A
{
private:
    int _b;
public:
    B(int, int);
    B& operator=(const B&);

};
```

```
A::A(int a)
{
    std::cout << "Cons A"
    << std::endl;
    _a = a;
}
A& A::operator=(const A&a)
{
    std::cout << "= A"
        << std::endl;
    _a = a._a;
    return *this;
}

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

B& B::operator=(const B&b)
{
    std::cout << "= B"
        << std::endl;
    _b = b._b;
    return *this;
}
```

```
int main()
{
    B b1(5,6);
    B b2(6,7);

    b2 = b1;
}
```

```
$ ./a.exe
Cons A
CONS B
Cons A
CONS B
= B
```

Ici, l'opérateur '=' a été défini aussi dans B.

On voit alors que celui-ci n'appelle pas l'opérateur '=' de A !

On se retrouve un peu comme dans le cas du constructeur de recopie, à ceci prêt qu'on ne peut pas passer les paramètres au constructeur de A comme plus haut…

On doit donc prévoir la recopie aussi des membres de A mais …

L'HÉRITAGE

OPÉRATEUR D'AFFECTATION & HÉRITAGE

```
class A
{
private:
    int _a;
public:
    A(int);
    A& operator=(const A&);
};

class B : public A
{
private:
    int _b;
public:
    B(int, int);
    B& operator=(const B&);

};
```

```
A::A(int a)
{
    std::cout << "Cons A"
    << std::endl;
    _a = a;
}
A& A::operator=(const A&a)
{
    std::cout << "= A"
        << std::endl;
    _a = a._a;
    return *this;
}

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

B& B::operator=(const B&b)
{
    std::cout << "= B"
        << std::endl;
    _a = b._a;
    _b = b._b;
    return *this;
}
```

```
int main()
{
    B b1(5,6);
    B b2(6,7);

    b2 = b1;
}
```

… mais on ne peut pas faire ca dans notre exemple !

En effet , _a est un membre privé de A et on ne peut pas l'utiliser comme cela dans B !

```
$ g++ A.cpp main.cpp
In file included from A.cpp:2:0:
A.hh: Dans la fonction membre ‘B& B::operator=(const B&)’:
A.hh:7:9: erreur : ‘int A::_a’ is private
     int _a;
         ^
A.cpp:29:5: erreur : à l'intérieur du contexte
     _a = b._a;
     ^
In file included from A.cpp:2:0:
A.hh:7:9: erreur : ‘int A::_a’ is private
     int _a;
         ^
A.cpp:29:12: erreur : à l'intérieur du contexte
     _a = b._a;
            ^
```

L'HÉRITAGE

OPÉRATEUR D'AFFECTATION & HÉRITAGE

```
class A
{
private:
    int _a;
public:
    A(int);
    A& operator=(const A&);
};

class B : public A
{
private:
    int _b;
public:
    B(int, int);
    B& operator=(const B&);

};
```

```
A::A(int a)
{
    std::cout << "Cons A"
    << std::endl;
    _a = a;
}
A& A::operator=(const A&a)
{
    std::cout << "= A"
        << std::endl;
    _a = a._a;
    return *this;
}
B::B(int a, int b) : A(a)
{
    std::cout << "CONS B"
    << std::endl;
    _b = b;
}
B& B::operator=(const B&b)
{
    std::cout << "= B"
        << std::endl;
    A* pa = this;
    const A* pb = &b;
    *pa = *pb;
    _b = b._b;
    return *this;
}
```

```
int main()
{
    B b1(5,6);
    B b2(6,7);

    b2 = b1;
}
```

```
$ ./a.exe
Cons A
CONS B
Cons A
CONS B
= B
= A
```

On va donc s'arranger pour pouvoir quand même utiliser l'opérateur d'affectation de A.

On va créer un pointeur de type A* qui pointera vers this. Le pointeur this(de type B*) est alors converti implicitement vers un pointeur de type A*.

On fait de même pour b que l'on met dans un pointeur de type const A* également. (Il est passé const en paramètre, donc un pointeur const également). Ensuite on affecte la valeur pointée par le nouveau, sur l'ancien (this).

Et le tour est joué !

L'HÉRITAGE

HÉRITAGE & TEMPLATE

```
template <typename T>
class A
{
private:
   T _a;
public:
   A(T a) {_a = a;}
};
```

```
class B : public A<int>
{
private:
    int _b;
public:
    B(int, int);
};
```

L'héritage et les template se mélangent plutôt bien.

Il y a plusieurs cas possibles.

Ici, seule la classe parente est template. La classe fille qui ne l'est pas, doit donc définir lors de sa déclaration un type pour A.

Ici, A est une classe template et B ne l'est pas.

Donc B ne peut hériter que d'un type particulier de A.

L'HÉRITAGE

HÉRITAGE & TEMPLATE

```
template <typename T>
class A
{
private:
   T _a;
public:
   A(T a) {_a = a;}
};
```

```
class B : public A<int>
{
private:
    int _b;
public:
    B(int, int);
};
```

L'héritage et les template se mélangent plutôt bien.

Il y a plusieurs cas possibles.

Ici, seule la classe parente est template. La classe fille qui ne l'est pas, doit donc définir lors de sa déclaration un type pour A.

Ici, A est une classe template et B ne l'est pas.

Donc B ne peut hériter que d'un type particulier de A.

L'HÉRITAGE

HÉRITAGE & TEMPLATE

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

template <typename T>
class B : public A
{
private:
    T _b;
public:
    B(int a, T b) : A(a)
    {
    _b = b;
    }

};
```

Dans le cas où c'est la classe dérivée qui est templaté, tout se déroule naturellement…

L'HÉRITAGE

HÉRITAGE & TEMPLATE

```
template <typename T>
class A
{
private:
    T _a;
public:
    A(T a){ _a = a;}
};
template <typename T>
class B : public A<T>
{
private:
    T _b;
public:
    B(T a, T b) : A<T>(a)
    { _b = b; }
};
```

Enfin, dans le cas où la classe dérivée et parente sont templatées, on utilisera la syntaxe suivante.

Ici, on passe le type template de B à A. Elles seront donc templatées par le même type.

On fera également attention à l'appel du constructeur de A dans le constructeur de B.

L'HÉRITAGE

HÉRITAGE & TEMPLATE

```
template <typename T>
class A
{
private:
   T _a;
public:
   A(T a){ _a = a;}
};

template <typename T, typename U>
class B : public A<U>
{
private:
   T _b;
public:
   B(U a, T b) : A<U>(a)
   { _b = b; }
};
```

Il n'est pas obligatoire de templater la classe dérivée et héritée par le même type !

Ici, le type T est utilisé dans B, alors que U servira à définir le type de A.

HÉRITAGE MULTIPLE

L'HÉRITAGE MULTIPLE

EXEMPLE

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

class B
{
private:
    int _b;
public:
    B(int b);
    void affiche();
};

class C : public A, public B
{
private:
    int _c;
public:
    C(int, int, int);
    void affiche();
};
```

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

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

B::B(int b) {
     _b = b;
}

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

C::C(int a, int b, int c) : A(a), B(b)
{
     _c = c;
}

void C::affiche()
{
   A::affiche();
   B::affiche();
   cout << _c << endl;
}
```

Nous allons dire quelques mots au sujet de l'héritage multiple. C'est une possibilité offerte par C++, au final assez rarement utilisée.

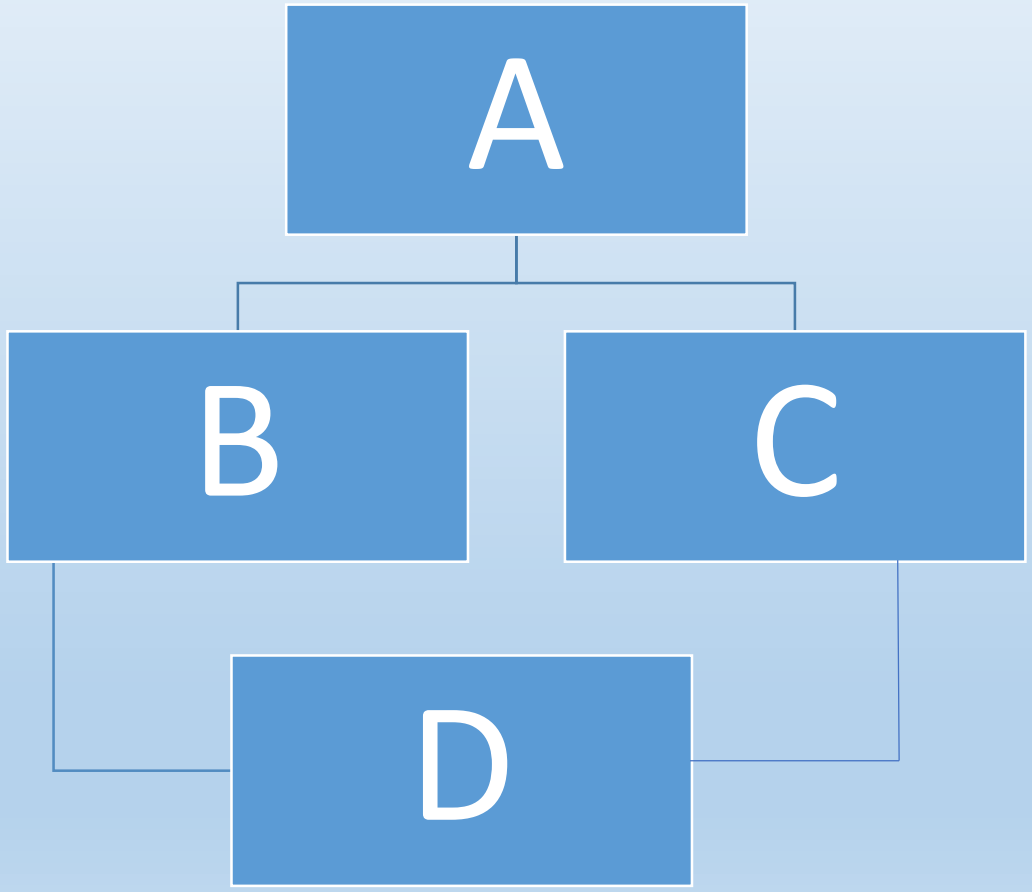
Comme on le voit dans l'exemple ci contre, on peut, comme on déclare qu'une classe hérite d'une autre classe, déclarer n classes parentes.

Le mécanisme est le même en ce qui concerne le passage d'arguments dans les constructeurs.

L'appel de fonction parente peut être forcé comme dans le cas de l'héritage simple.

L'HÉRITAGE MULTIPLE

HÉRITAGE EN DIAMANT



La difficulté de ce type d'héritage commence quand deux classes parentes héritent d'une même classe.

On se retrouve alors dans la situation suivante :

C hérite de A

B hérite de A

D hérite de B et de C.

B et C vont donc construire chacun une classe A ! Il y a donc un risque de conflit quand, dans D, on veut accéder à une donnée de A …

L'HÉRITAGE MULTIPLE

CLASSES VIRTUELLES

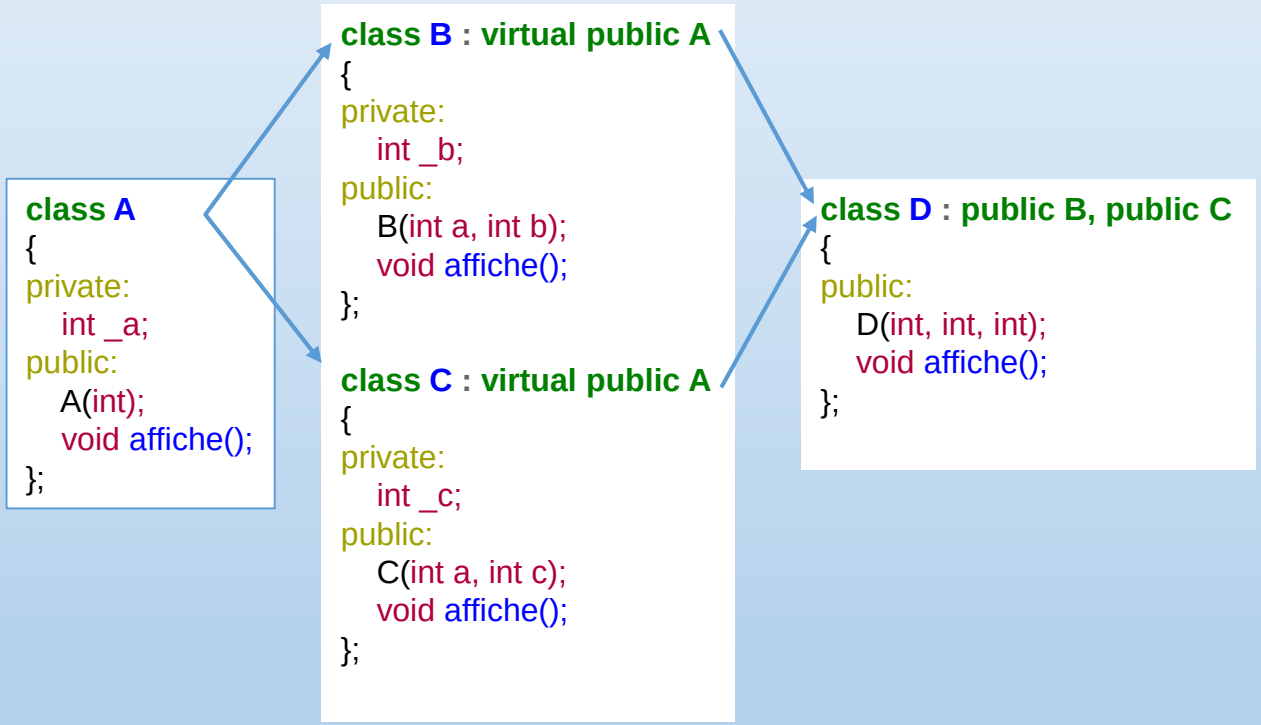
Pour pallier à ce type de problématiques. C++ met à notre disposition les « classes virtuelles ».

Ainsi, B et C vont hériter « virtuellement » de A. Le compilateur saura alors qu'il faudra construire non pas deux classes A, mais une seule. Ainsi quand dans D on accède à une donnée membre de A, il n'y aura pas de risque de conflit.

Cela implique par contre des choses au niveau des constructeurs ! En effet, si les deux classes filles B et C ont prévu des paramètres à passer au constructeur de A, lesquelles seront choisies ?

L'HÉRITAGE MULTIPLE

CLASSES VIRTUELLES

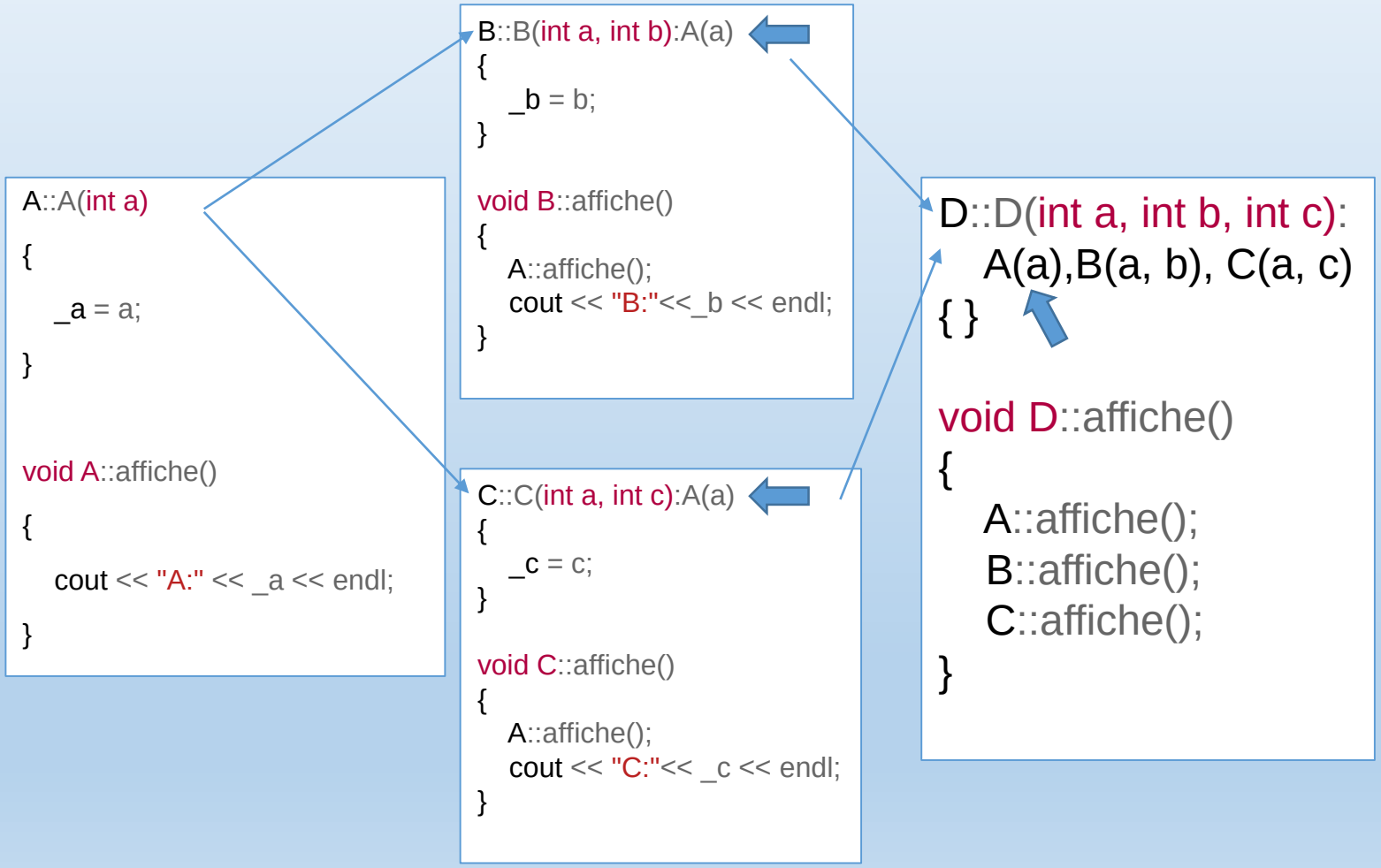


Dans cet exemple, on voit que l'on a ajouter le mot clé virtual aux deux classes B et C dérivant A.

La classe D hérite quant à elle de B et de C.

L'HÉRITAGE MULTIPLE

CLASSES VIRTUELLES



Que se passe t il au niveau de l'appel des constructeurs ?

B et C prévoient un appel du constructeur de A avec des paramètres. Il n'y a aucune raison pour que ces valeurs soient les mêmes ! En fait dans le cas d'un héritage virtuel, cet appel intermédiaire au constructeur parent est ignoré au seul profit de l'appel dans la classe fille D.

POLYMORPHISME

POLYMORPHISME

DÉFINITION

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

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

Cela provient du fait qu'à la compilation, le compilateur ne connait 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 là, qui correspond au type de la classe parente. Il s'agit d'un typage statique.

POLYMORPHISME

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, quelle sera la fonction 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.

POLYMORPHISME

FONCTIONS VIRTUELLES

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

};

class B : public A
{
private:
    int _b;

public:
    B(int, int);
    void affiche();
};
```

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

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

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

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

```
int main()
{
    A *pa;
    B  b(2, 3);

    pa = &b;

    pa->affiche();
}
```

```
$ ./a.exe
A: 2
```

Dans cet exemple, nous rappelons le problème.

Dans le main, on créé 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.

POLYMORPHISME

FONCTIONS VIRTUELLES

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

};

class B : public A
{
private:
    int _b;

public:
    B(int, int);
    void affiche();
};
```

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

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

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

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

```
int main()
{
    A *pa;
    B  b(2, 3);

    pa = &b;

    pa->affiche();
}
```

```
$ ./a.exe
A: 2
B: 3
```

On se contente d'ajouter le mot clé « virtual » à la fonction affiche.

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

POLYMORPHISME

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…
- En revanche, un destructeur peut être virtuel.

POLYMORPHISME

FONCTIONS VIRTUELLES - DESTRUCTEURS

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


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

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

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

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

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

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

    delete pa;

}
```

```
$ ./a.exe
Des 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 stocké 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 …

POLYMORPHISME

FONCTIONS VIRTUELLES - DESTRUCTEURS

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

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

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

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

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

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

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

    delete pa;

}
```

```
$ ./a.exe
Des B
Des 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.

POLYMORPHISME

CLASSES ABSTRAITES – FONCTIONS VIRTUELLES PURES

```
#ifndef _A_HH_
#define _A_HH_

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

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

#endif
```

Fonction virtuelle pure

On la redéfinit ici

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 la dérivant 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 définitions à cette fonction, et c'est la classe l'héritant qui devra définir cette fonction.

POLYMORPHISME

CLASSES ABSTRAITES – FONCTIONS VIRTUELLES PURES

```
#ifndef _A_HH_
#define _A_HH_

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

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

#endif
```

Fonction virtuelle pure

On la redéfinit ici

```
#include "A.hh"

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

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

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.

```
$ g++ A.cpp main.cpp
main.cpp: Dans la fonction ‘int main()’:
main.cpp:5:15: erreur : invalid new-expression of abstract class type ‘A’
  A* a = new A();
               ^
In file included from main.cpp:1:0:
A.hh:4:7: note :   because the following virtual functions are pure within ‘A’:
 class A
       ^
A.hh:9:15: note :       virtual void A::affiche()
  virtual void affiche() = 0;
               ^
```

POLYMORPHISME

CLASSES ABSTRAITES – FONCTIONS VIRTUELLES PURES

```
#ifndef _A_HH_
#define _A_HH_

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

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

#endif
```

Fonction virtuelle pure

On la redéfinit ici

```
#include "A.hh"

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

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

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.

```
$ g++ A.cpp main.cpp
main.cpp: Dans la fonction ‘int main()’:
main.cpp:5:15: erreur : invalid new-expression of abstract class type ‘A’
  A* a = new A();
               ^
In file included from main.cpp:1:0:
A.hh:4:7: note :   because the following virtual functions are pure within ‘A’:
 class A
       ^
A.hh:9:15: note :       virtual void A::affiche()
  virtual void affiche() = 0;
               ^
```

POLYMORPHISME

INTERFACES

En Java, ou dans d'autres langages, il existe la notion d'interface.

Une interface peut être vue comme une certaine forme de classe, ininstantiable, comme les classes abstraites.

Elles permettent de définir les actions (fonctions membres) nécessaires lors de l'écriture d'une classe « implémentant » cette interface.

PAR DEFINITION :

Une **interface** sera une classe abstraite dont toutes les fonctions sont virtuelles pures, sauf le déstructeur, et qui ne contient pas de données membres. (sauf des données static const).

POLYMORPHISME

INTERFACES

```
#ifndef __IFORME_HH__
#define __IFORME_HH__
class IForme
{
public:
    virtual void affiche() = 0;
    virtual void deplace() = 0;
};
class Rond : public IForme
{
private:
    double _x, _y, _r;
public:
    Rond(double, double, double);
    void affiche();
    void deplace();
};
class Carre : public IForme
{
private:
    double _x, _y, _long;
public:
    Carre(double, double, double);
    void affiche();
    void deplace();
};
#endif
```

```
#include "IForme.hh"

Rond::Rond(double x, double y, double r){
    _x = x; _y = y; _r = r;
}
void Rond::affiche(){
    /* AFFICHE UN ROND*/
}
void Rond::deplace(){
    /* Deplace un rond */
}

Carre::Carre(double x, double y, double l){
    _x = x; _y = y; _long = l;
}
void Carre::affiche(){
    /* AFFICHE UN ROND*/
}
void Carre::deplace(){
    /* Deplace un rond */
}
```

```
#include "IForme.hh"

int main()
{
    IForme *r = new Rond(1, 1, 1);
    IForme *c = new Carre(1, 1, 1);

    r->affiche();
    c->affiche();
}
```

On déclare une interface `IForme`.

Le « `I` » en tête du nom de la classe est là pour rappeler qu'il s'agit d'une interface.

C'est une convention communément admise.

On déclare deux méthodes que devront au moins définir des classes dérivant de cette interface : affiche et déplace.

POLYMORPHISME

INTERFACES

```
#ifndef __IFORME_HH__
#define __IFORME_HH__
class IForme
{
public:
    virtual void affiche() = 0;
    virtual void deplace() = 0;
};
class Rond : public IForme
{
private:
    double _x, _y, _r;
public:
    Rond(double, double, double);
    void affiche();
    void deplace();
};
class Carre : public IForme
{
private:
    double _x, _y, _long;
public:
    Carre(double, double, double);
    void affiche();
    void deplace();
};
#endif
```

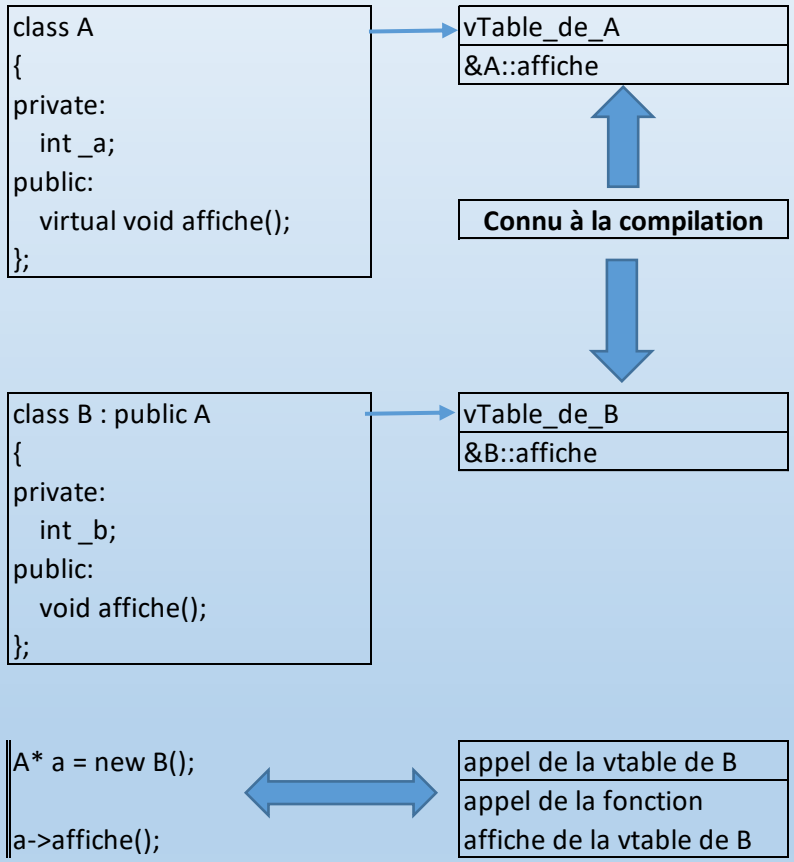
```
#include "IForme.hh"

Rond::Rond(double x, double y, double r){
    _x = x; _y = y; _r = r;
}
void Rond::affiche(){
    /* AFFICHE UN ROND*/
}
void Rond::deplace(){
    /* Deplace un rond */
}

Carre::Carre(double x, double y, double l){
    _x = x; _y = y; _long = l;
}
void Carre::affiche(){
    /* AFFICHE UN ROND*/
}
void Carre::deplace(){
    /* Deplace un rond */
}
```

```
#include "IForme.hh"

int main()
{
    IForme *r = new Rond(1, 1, 1);
    IForme *c = new Carre(1, 1, 1);

    r->affiche();
    c->affiche();
}
```

On peut alors déclarer des pointeurs sur cette interface dont le type réellement instancié est celui des classes filles.

POLYMORPHISME

COMMENT CA MARCHE ? – LA VTABLE



Il n'y a pas de façon prévue dans la norme pour l'implémentation du mécanisme de fonctions virtuelles.

Une façon commune de la représenter est la suivante :

- Pour chaque classe contenant une fonction virtuelle, et dans les classes dérivées de cette classe, on créé un tableau de pointeurs sur fonctions qui contiendront les pointeurs vers les fonctions virtuelles de cette classe.
- Une adresse vers cette vTable est ajoutée dans chaque instance de la classe.
- Lors de l'appel d'une fonction d'un objet instancié, au lieu d'appeler directement la fonction de la classe pointée, on appelle la fonction contenue dans la vTable pointée dans l'objet.

POLYMORPHISME

IDENTIFICATION DU TYPE À L'EXECUTION

```
#include <iostream>
#include <typeinfo>
#include "IForme.hh"

void affiche_type(IForme *f){
    std::cout <<
    typeid(f).name()
    << std::endl <<
    typeid(*f).name()
    << std::endl;
}

int main()
{
    IForme* a = new Carre(1,1,1);
    IForme* b = new Rond(1,1,1);
    affiche_type(a);
    affiche_type(b);
}
```

```
$ ./a.exe
P6IForme
5Carre
P6IForme
4Rond
```

Il existe un opérateur unaire, typeid, qui permet de connaître le type réel d'un objet pointé.

Cette opérateur renvoie un objet de type type_info.

Cette classe contient la fonction name qui permet d'obtenir, sous forme d'un `char *` le type d'un objet.

Par contre, il n'y a aucune obligation que le nom du type soit identique à celui qui est définie dans le code du programme. La contrainte est que pour deux types différents, le nom soit différent.

POLYMORPHISME

IDENTIFICATION DU TYPE À L'EXÉCUTION

```
#include <iostream>
#include <typeinfo>
#include "IForme.hh"

void affiche_type(IForme *f){
    std::cout <<
    typeid(f).name()
    << std::endl <<
    typeid(*f).name()
    << std::endl;
}

int main()
{
    IForme* a = new Carre(1,1,1);
    IForme* b = new Rond(1,1,1);
    affiche_type(a);
    affiche_type(b);
}
```

```
$ ./a.exe
P6IForme
5Carre
P6IForme
4Rond
```

Par exemple, ici nous affichons le type du pointeur, dans la fonction `affiche_type` qui prend un pointeur sur `IForme` en paramètre.

Ensuite on affiche le type de l'objet pointé.

Le type du pointeur est toujours le même, c'est un pointeur sur `IForme`.

Le type de l'objet pointé, par contre, varie en fonction du type réel de l'objet pointé (celui qui a servi a son instanciation.)

POLYMORPHISME

IDENTIFICATION DU TYPE À L'EXÉCUTION

```
#include <iostream>
#include <typeinfo>
#include "IForme.hh"

void affiche_type(IForme *f){
    std::cout <<
    typeid(f).name()
    << std::endl <<
    typeid(*f).name()
    << std::endl;
}

int main()
{
    IForme* a = new Carre(1,1,1);
    IForme* b = new Rond(1,1,1);
    affiche_type(a);
    affiche_type(b);
}
```

```
$ ./a.exe
P6IForme
5Carre
P6IForme
4Rond
```

Il existe aussi une surcharge des opérateurs == et != pour les objets de type type_info qui permettent de comparer le type de deux objets.

Le fait de pouvoir connaître dynamiquement le type d'objet lors de l'exécution s'appelle

R.T.T.I. pour RunTime Type Information .

POLYMORPHISME

LES CAST DYNAMIQUES

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

int main()
{
    IForme* a = new Carre(1,1,1);

    Rond *r = dynamic_cast<Rond*>(a);
    if (r)
        r->pour_rond();

    Carre *c = dynamic_cast<Carre*>(a);
    if (c)
        c->pour_carre();
}
```

```
void Rond::pour_rond(){
    cout << "ROND" << endl;
}
```

```
void Carre::pour_carre(){
    cout <<"CARRE"<< endl;
}
```

```
$ ./a.exe
CARRE
```

Nous avons vu comment créer un pointeur différent du type réel de l'objet pointé.

Ensuite, nous avons vu comment travailler sur ce type réel à travers le type générique.

Nous aimerions maintenant pouvoir récupérer un pointeur vers le type réel, pour, par exemple, appeler des fonctions spécifiques à cet objet.

POLYMORPHISME

LES CAST DYNAMIQUES

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

int main()
{
    IForme* a = new Carre(1,1,1);

    Rond *r = dynamic_cast<Rond*>(a);
    if (r)
        r->pour_rond();

    Carre *c = dynamic_cast<Carre*>(a);
    if (c)
        c->pour_carre();
}
```

```
void Rond::pour_rond(){
    cout << "ROND" << endl;
}
```

```
void Carre::pour_carre(){
    cout <<"CARRE"<< endl;
}
```

```
$ ./a.exe
CARRE
```

Pour cela, nous allons utiliser un opérateur de cast appelé dynamic_cast.

Il s'utilise de la façon suivante :

```
ptr_res = dynamic_cast<vers_type>(ptr)
```

Où

- `ptr_res` est un pointeur vers le type réel de l'objet.
- `vers_type` est le type du pointeur réel.
- `ptr` est un pointeur vers le type général.

POLYMORPHISME

LES CAST DYNAMIQUES

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


int main()
{
    IForme* a = new Carre(1,1,1);


    Rond *r = dynamic_cast<Rond*>(a);
    if (r)
        r->pour_rond();


    Carre *c = dynamic_cast<Carre*>(a);
    if (c)
        c->pour_carre();
}
```

```
void Rond::pour_rond(){
    cout << "ROND" << endl;
}
```

```
void Carre::pour_carre(){
    cout <<"CARRE"<< endl;
}
```

```
$ ./a.exe
CARRE
```

Et si le pointeur n'est pas du bon type ?

Le pointeur renvoyé par dynamic_cast est alors le pointeur nul.

C'est pour cela que l'on compare la valeur du pointeur dans le code ci-contre avant d'utiliser la fonction spécifique du type.

POLYMORPHISME

LES CAST DYNAMIQUES

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


int main()
{
    IForme* a = new Carre(1,1,1);


    Rond *r = dynamic_cast<Rond*>(a);
    if (r)
        r->pour_rond();


    Carre *c = dynamic_cast<Carre*>(a);
    if (c)
        c->pour_carre();
}
```

```
void Rond::pour_rond(){
    cout << "ROND" << endl;
}
```

```
void Carre::pour_carre(){
    cout <<"CARRE"<< endl;
}
```

```
$ ./a.exe
CARRE
```

Pour information, il existe trois autres types de cast.

`static_cast` : Pour convertir des types de base ou défini par l'utilisateur.

`reinterpret_cast` : pour convertir des pointeurs vers des types.

`const_cast` : pour convertir des variables const en non const.

On ne s'attardera pas plus sur les opérateurs de cast dans ce cours.

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 connait 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'iterateurs, et d'algorithmes sur ces conteneurs. Cette librairie est connue sous le nom de Standard Template Library ou STL. C'est une librairie puissante conçue à base de classe template, 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 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 chainée.

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 contiguë : 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-éme 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 fonctions 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 possède une valeur qui désigne un élément donné du conteneur. On dira qu'il « pointe » vers cet élément.
- 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 iterateur se déclare de la façon suivante :

std::vector<double>::iterator iv;

La première partie est un vector de double.

Que signifie la deuxième partie ?

Les objets iterateurs 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'iterateur pointant sur le début du vecteur.

Comme on peut comparer l'égalité ou l'inégalité de deux iterateurs, 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() et end() sont égaux.

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++.

Pour comprendre pourquoi, on se souviendra de la surchage de l'opérateur d'incrémentation et que la notation postfixée entraine 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++;
   }
   iv = v.begin();
   for (int i = 0; i < v.size(); i++)
   {
      std::cout << *(iv+i) << ";";
   }
}
```

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 entraine 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

ITERATEUR SUR MAP

```
#include <map>
#include <string>
#include <iostream>

int main()
{
    std::map<std::string, double> m;
    m["Charles"] = 5;
    m["Zoé"] = 6;
    m["Toto"] = 7;
    for (std::map<std::string, double>::iterator im = m.begin();
        im != m.end(); ++im)
    {
        std::cout << (*im).first <<  " " << (*im).second << std::endl;
    }
}
```

On définit ici une map templatée par une chaine de caractéres et un double.

Dans la boucle for, on déclare et on initialise l'itérateur directement dans la boucle, comme on pourrait le faire avec un entier par exemple.

```
$ ./a.exe
Charles 5
Toto 7
Zoé 6
```

LA STL

ITERATEUR SUR MAP

```
#include <map>
#include <string>
#include <iostream>

int main()
{
    std::map<std::string, double> m;
    m["Charles"] = 5;
    m["Zoé"] = 6;
    m["Toto"] = 7;
    for (std::map<std::string, double>::iterator im = m.begin();
        im != m.end(); ++im)
    {
        std::cout << (*im).first <<  " " << (*im).second << std::endl;
    }
}
```

Ensuite, l'itérateur s'utilise de la même façon que ceux sur les vector ou les list.

On peut également déréférencer l'itérateur sur map.

Par contre, la valeur accédée est une paire !

```
$ ./a.exe
Charles 5
Toto 7
Zoé 6
```

LA STL

ITERATEUR SUR MAP

```
#include <map>
#include <string>
#include <iostream>

int main()
{
    std::map<std::string, double> m;
    m["Charles"] = 5;
    m["Zoé"] = 6;
    m["Toto"] = 7;
    for (std::map<std::string, double>::iterator im = m.begin();
        im != m.end(); ++im)
    {
        std::cout << (*im).first <<  " " << (*im).second << std::endl;
    }
}
```

Une paire est une classe qui existe également dans la STL.

Il s'agit d'associer une clé à une valeur, et ce en utilisant les template. Ainsi la clé et la valeur peuvent être de type quelconque.

Ainsi, le champ .first d'une paire permet d'accéder à sa clé.

Et le champ .second permet d'accéder à la valeur.

```
$ ./a.exe
Charles 5
Toto 7
Zoé 6
```

LA STL

ITERATEUR SUR MAP

```
#include <map>
#include <string>
#include <iostream>

int main()
{
    std::map<std::string, double> m;
    m["Charles"] = 5;
    m["Zoé"] = 6;
    m["Toto"] = 7;
    for (std::map<std::string, double>::iterator im = m.begin();
        im != m.end(); ++im)
    {
        std::cout << (*im).first <<  " " << (*im).second << std::endl;
    }
}
```

On remarque que l'affichage ne respecte pas l'ordre dans lequel on a rempli la map.

Cela vient du fait qu'une map, comme tous les conteneurs, est ordonnée.

Ceci afin d'accélérer les algorithmes de recherche etc.

```
$ ./a.exe
Charles 5
Toto 7
Zoé 6
```

LA STL

ITERATEUR SUR MAP

```
#include <map>
#include <string>
#include <iostream>

int main()
{
    std::map<std::string, double> m;
    m["Charles"] = 5;
    m["Zoé"] = 6;
    m["Toto"] = 7;
    for (std::map<std::string, double>::iterator im = m.begin();
        im != m.end(); ++im)
    {
        std::cout << (*im).first <<  " " << (*im).second << std::endl;
    }
}
```

Sur les chaines de caractères, la relation d'ordre est triviale, il s'agit de l'ordre alphabétique.

Mais que se passe-t-il pour des objets où la relation d'ordre est moins évidente ?

```
$ ./a.exe
Charles 5
Toto 7
Zoé 6
```

LA STL

ITERATEUR SUR MAP

```
#include <map>
#include <string>
#include <iostream>
class cmp {
public:
    bool operator()(const std::string&, const std::string&);
};
bool cmp::operator()(const std::string& s1,  const std::string& s2){
    return s2 < s1;
}
int main() {
    std::map<std::string, double, cmp> m;
    m["Charles"] = 5;
    m["Zoé"] = 6;
    m["Toto"] = 7;
    for (std::map<std::string, double>::iterator im = m.begin();
        im != m.end(); ++im)
        std::cout << (*im).first <<  " " << (*im).second << std::endl;
}
```

Dans le cas où la relation d'ordre « par défaut » ne convient pas, ou que celle-ci n'est pas évidente – c'est le cas si il n'y a pas d'opérateur de comparaison '<' défini pour le type de la clé – on peut définir lors de la construction de la map, une relation d'ordre.

```
$ ./a.exe
Zoé 6
Toto 7
Charles 5
```

LA STL

ITERATEUR SUR MAP

```
#include <map>
#include <string>
#include <iostream>
class cmp {
public:
    bool operator()(const std::string&, const std::string&);
};
bool cmp::operator()(const std::string& s1,  const std::string& s2){
    return s2 < s1;
}
int main() {
    std::map<std::string, double, cmp> m;
    m["Charles"] = 5;
    m["Zoé"] = 6;
    m["Toto"] = 7;
    for (std::map<std::string, double>::iterator im = m.begin();
          im != m.end(); ++im)
        std::cout << (*im).first <<  " " << (*im).second << std::endl;
}
```

Le troisième élément du template de map doit être un type. L'objet de ce type servira à comparer les clés de la map.

Ce type doit surcharger l'opérateur '()' et donc est un foncteur.

L'opérateur '()' doit renvoyer un booléen et prendre en paramètre deux éléments qui seront deux clés de la map.

```
$ ./a.exe
Zoé 6
Toto 7
Charles 5
```

LA STL

ITERATEUR SUR MAP

```
#include <map>
#include <string>
#include <iostream>
class cmp {
public:
    bool operator()(const std::string&, const std::string&);
};
bool cmp::operator()(const std::string& s1,  const std::string& s2){
    return s2 < s1;
}
int main() {
    std::map<std::string, double, cmp> m;
    m["Charles"] = 5;
    m["Zoé"] = 6;
    m["Toto"] = 7;
    for (std::map<std::string, double>::iterator im = m.begin();
          im != m.end(); ++im)
        std::cout << (*im).first <<  " " << (*im).second << std::endl;
}
```

Ici, on voit que l'on a changé la relation d'ordre de la map en lui passant l'ordre inverse de l'ordre alphabétique.

Ainsi le premier élément sera 'zoé', plus grande clé au sens alphabétique.

```
$ ./a.exe
Zoé 6
Toto 7
Charles 5
```

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 conteneurs – c'est-à-dire entre deux valeurs d'itérateurs.

```
$ ./a.exe
32;71;12;45;26;80;
```

LA STL

ALGORITHMES – COUNT/COUNT_IF

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

bool 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(),  pair);
    cout << "Nbr pair :" << c << endl;
    cout << "32 :"<< count(v.begin(), v.end(), 32)
            << endl;
}
```

La fonction count permet de compter le nombres d'occurrence 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.exe
Nbr pair :5
32 :2
```

LA STL

ALGORITHMES – SORT

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

bool petit(int i, int j){
    return i < j;
}

int main()
{
    int myints[] = {32,71,12,45,26,80,53,33,32};
    vector<int> v(myints, myints+9);
    sort(v.begin(), v.end(), petit);
    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 est une fonction renvoyant vrai si deux éléments du tableau sont dans la bonne position.

```
$ ./a.exe
12-26-32-32-33-45-53-71-80-
```

LES FLOTS

LES FLOTS

Depuis le début de ce cours, nous avons été amenés à utiliser les objets cout et cin ainsi que les opérateurs ‘<< ‘ et ‘>>’.

Nous allons étudier plus précisément ces objets et les possibilités qu’ils offrent.

Nous allons voir qu’ils font parti d’un ensemble plus vaste, les flots, qui vont nous servir à lire et écrire des fichiers.

LES FLOTS

FLOTS PRÉDÉFINIS POUR L'ÉCRITURE

```
#include <iostream>

using namespace std;

int main()
{
    // Ecriture sur la sortie standard
    cout << "Ok : " << 10 << endl;

    //  Ecriture non bufferisé sur la
    // sortie erreur
    cerr << "Erreur ! " << endl;

    //  Ecriture bufferisé
    //sur la sortir d'erreur
    clog << "Erreur 2 " << endl;
}
```

Il existe 3 flots prédéfinis pour la sortie.

- **cout** est le plus commun. Il permet d'écrire sur la sortie standard. Celle-ci désigne le plus souvent l'écran de la machine sur laquelle le code est exécuté.
- **cerr** désigne la sortie d'erreur. Celle-ci par défaut désigne également l'écran de la machine. Il s'agit de ce qu'on appelle une écriture non-bufferisé.
- **clog** désigne également la sortie d'erreur mais en bufferisant les sorties.

LES FLOTS

FLOTS PRÉDÉFINIS POUR L'ÉCRITURE

```
#include <iostream>

using namespace std;

int main()
{
    // Ecriture sur la sortie standard
    cout << "Ok : " << 10 << endl;

    //  Ecriture non bufferisé sur la
    // sortie erreur
    cerr << "Erreur ! " << endl;

    //  Ecriture bufferisé
    //sur la sortir d'erreur
    clog << "Erreur 2 " << endl;
}
```

Une écriture « bufferisé » est une écriture qui utilise un tampon pour les sorties à l'écran.

Pourquoi un tel mécanisme ?

Une écriture sur l'écran est lente : il s'agit d'executer des instructions sur la carte video qui les imprimera sur l'écran et ce pour chaque écriture.

Si on écrit caractère par caractère, ces jeux d'instructions sont exécutés un grand nombre de fois.

Un tampon permet alors de regrouper les sorties dans un espace mémoire, et, lorsque cet espace est plein, de l'imprimer sur l'écran en une seule fois.

LES FLOTS

FLOTS PRÉDÉFINIS POUR L'ÉCRITURE

```
#include <iostream>

using namespace std;

int main()
{
    // Ecriture sur la sortie standard
    cout << "Ok : " << 10 << endl;

    //  Ecriture non bufferisé sur la
    // sortie erreur
    cerr << "Erreur ! " << endl;

    //  Ecriture bufferisé
    //sur la sortir d'erreur
    clog << "Erreur 2 " << endl;
}
```

Cela peut avoir des conséquences imprévues :

Par exemple, si un programme plante, il se peut que le tampon ne soit pas vidé (flushé) sur l'écran et que le programme n'ait pas tout écrit avant de se terminer.

Pour info : C'est un peu le même système qui est mis en place lorsqu'une clé USB n'est pas retirée proprement d'une machine : des informations gardées en mémoire en attente d'écriture sur la clé n'ont pas été correctement vidées.

LES FLOTS

FLOTS PRÉDÉFINIS POUR L'ÉCRITURE

```
#include <iostream>

using namespace std;

int main()
{
    // Ecriture sur la sortie standard
    cout << "Ok : " << 10 << endl;

    //  Ecriture non bufferisé sur la
    // sortie erreur
    cerr << "Erreur ! " << endl;

    //  Ecriture bufferisé
    //sur la sortir d'erreur
    clog << "Erreur 2 " << endl;
}
```

Les flux d'erreur, même si ils écrivent par défaut sur l'écran, ne désigne pas la même sortie.

Ainsi, il existe des moyens de séparer les sorties standard des sorties d'erreur, par exemple en redirigeant celles–ci dans un fichier.

Par défaut, tout s'imprime sur l'écran

```
$ ./a.exe
Ok : 10
Erreur !
Erreur 2
```

Mais, on peut rediriger la sortie d'erreur dans un fichier par exemple pour l'exploiter ultérieurement en cas d'erreur

```
$ ./a.exe 2> LOG.txt
Ok : 10
```

```
$ cat LOG.txt
Erreur !
Erreur 2
```

LES FLOTS

FLOTS PRÉDÉFINIS POUR L'ÉCRITURE

```
#include <iostream>

using namespace std;

int main()
{
    char str[] = "Ok tout va bien";
    cout.write(str, 10);
}
```

La fonction write dont voici le prototype dans la documentation :

ostream& write (**const char* s, streamsize n);**

Permet d'écrire un certain nombre d'octets à l'écran sans aucune vérification quant au contenu de ces données.

Il n'y a pas non plus de bufferisation.

Cela peut servir, par exemple, lorsque l'on a besoin d'écrire des données brutes à l 'écran, typiquement telles qu'elles sont stockées en mémoire, et non réinterprétées pour le rendre lisible par un humain. On parlera d'affichage binaire, mais ce sera surtout utilisé pour les fichiers.

LES FLOTS

FORMATAGES

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

using namespace std;

int main()
{
    int i = 42;
    cout << "Entier" << endl;
    cout << "Déci : " << i << endl;
    cout << "hexa : "<< hex << i << endl;
    cout << "Oct : "<< oct << i << endl;
    cout << "Bool" << endl;
    cout << true << endl;
    cout << boolalpha << true << endl;
}
```

Nous avons déjà apercu les manipulateurs de flots lors de ce cours. Nous allons étudier un peu plus en profondeur les possibilités qu'ils offrent.

Ceux-ci font partis du header <iomanip>.

Pour les entiers tout d'abord, on peut convertir l'affichage en différentes bases.

Pour les booléens, on peut afficher ou non la valeur entière du booléen ou sa valeur 'true/false'.

```
$ ./a.exe
Entier
Déci : 42
hexa : 2a
Oct : 52
Bool
1
true
```

LES FLOTS

FORMATAGES

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

using namespace std;

int main()
{
    for (int i = 0; i <= 10; i++)
        cout << setw(2) << i << ":"
             << setw(i) << 4242 << endl;
}
```

```
$ ./a.exe
 0:4242
 1:4242
 2:4242
 3:4242
 4:4242
 5: 4242
 6:  4242
 7:   4242
 8:    4242
 9:     4242
10:      4242
```

Une autre possibilité offerte et celle de la taille des gabarits.

On va utiliser setw(int) avec un entier en paramètre qui va définir la taille du gabarit d'affichage.

Pour avoir des affichages propres par colonne par exemple.

Il s'agit d'un manipulateur paramétré comme on peut le voir dans l'exemple.

LES FLOTS

FORMATAGES

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

using namespace std;

int main()
{
    double d = 42.1234567;
    cout << setprecision(4) << d << endl;
    cout << scientific << d << endl;
    cout << setprecision(8) << fixed << d << endl;
}
```

```
$ ./a.exe
42.12
4.2123e+01
42.12345670
```

Pour les nombres à virgules flottantes, on peut jouer sur la précision et l'écriture scientifique ou non.

LES FLOTS

FLOT PRÉDÉFINI POUR LA LECTURE

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

using namespace std;

int main()
{
    string str;
    char txt[10+1];

    cin >> str;
    cin >> setw(10) >> txt;

    cout  << str << endl;
    cout << txt << endl;
}
```

Il existe un flot que nous avons déjà rencontré lisant les données sur l'entrée standard.

L'entrée standard par défaut est le clavier. (Ce n'est pas nécessairement le cas, car on peut par exemple rediriger le contenu d'un fichier sur l'entrée standard d'un exécutable).

Nous avons déjà vu l'objet cin, qui est le pendant de cout pour les lectures.

cin est un objet de type istream pour lequel est surchargé l'opérateur >> pour un certain nombre de types prédéfinis.

LES FLOTS

FLOT PRÉDÉFINI POUR LA LECTURE

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

using namespace std;

int main()
{
    string str;
    char txt[10+1];

    cin >> str;
    cin >> setw(10) >> txt;

    cout  << str << endl;
    cout << txt << endl;
}
```

Ainsi , on peut passer un objet de type string en tant qu'opérande de cin.

On peut également passer un char *, dont l'espace mémoire aura été préalablement alloué.

On fera attention à ce dernier point :

Il faut prévoir un octet pour le caractère nul.

On pensera aussi à limiter le nombre de caractères lus sur le flux pour ne pas déborder hors de la mémoire du tableau de char.

cin utilisé avec l'opérateur '>>' a aussi une particularité, il utilise lors de la lecture les espaces blancs comme séparateur de lecture.

Ainsi, pour lire la phrase : « Il fait beau », il faudra 3 appels de l'opérateur '>>' de cin.

LES FLOTS

LES FONCTIONS GETLINE & GCOUNT

```
#include <iostream>
using namespace std;
int main()
{
    char str[10+1];
    cin.getline(str, 10);

    cout << str << " " << cin.gcount();

    cin.getline(str, 10, ';');
    cout << str << " " << cin.gcount();
}
```

```
$ ./a.exe
test
test 5
tom;tom
tom 4
```

Une fonction non formatée est une fonction qui ne modifie pas le contenu qui a été lu. Il existe par exemple la fonction getline qui récupère une ligne entrée au clavier (c'est à dire jusqu'à ce qu'on appuie sur la touche 'entrée').

Elle prend en paramètre un char * qui aura été préalablement alloué ainsi que le nombre de caractères maximum à lire (on prendra garde de ne pas laisser la possibilité de dépasser la taille du tableau).

LES FLOTS

LES FONCTIONS GETLINE & GCOUNT

```
#include <iostream>
using namespace std;
int main()
{
    char str[10+1];
    cin.getline(str, 10);

    cout << str << " " << cin.gcount();

    cin.getline(str, 10, ';');
    cout << str << " " << cin.gcount();
}
```

```
$ ./a.exe
test
test 5
tom;tom
tom 4
```

La fonction gcount vient en complément de la fonction getline et permet après une utilisation de getline de connaitre le nombre de caractères qui ont été effectivement lus sur le flot.

LES FLOTS

LES FONCTIONS GETLINE & GCOUNT

```
#include <iostream>
using namespace std;
int main()
{
    char str[10+1];
    cin.getline(str, 10);

    cout << str << " " << cin.gcount();

    cin.getline(str, 10, ';');
    cout << str << " " << cin.gcount();
}
```

```
$ ./a.exe
test
test 5
tom;tom
tom 4
```

On peut aussi passer en troisième paramètre de la fonction getline un caractère dit séparateur. Celui-ci, par défaut, est le caractère de retour à la ligne ‘\n’.

LES FLOTS

LA FONCTION READ

```
#include <iostream>

using namespace std;

int main()
{
    char txt[10+1];

    cin.read(txt, 5);
    cout << txt << endl;
}
```

La fonction read est le pendant de la fonction write pour les flots entrant.

Comme la fonction write, elle est indispensable pour lire par exemple des entrées binaires.

LES FLOTS

ERREURS SUR LES FLOTS

Les flots, qu'ils soient en entrée ou en sortie, dérivent d'une classe appelée ios (Input/Output Stream).

Cette classe définit, entre autre, trois valeurs qui correspondent à trois statuts d'erreur sur les flots, il s'agit de bits, donc des valeurs binaires valant 0 ou 1.

- eofbit : ce bit est activé si la fin du fichier est atteinte. Autrement dit, si il n'y a plus de caractère à lire sur ce flot. Pratique si on veut savoir quand on a atteint la fin d'un fichier …
- failbit : activé si la prochaine instruction d'Entrée/Sortie ne peut aboutir.
- badbit : activé si le flot est dans un état irrécupérable.

LES FLOTS

ERREURS SUR LES FLOTS

Pour accéder à ces valeurs, il existe trois fonctions membres de ios (héritée donc par istream et ostream).

- eof() fournit vrai (1) si le bit de fin de flot est activé.
- bad() fournit vrai si le flot est dans un état irrécupérable.
- fail() : si le failbit vaut 1.

Il existe une 4éme fonction :

- good() si le statut du flot est bon : aucune des trois fonctions précédentes ne renvoie la valeur ‘vrai’.

LES FLOTS

ERREURS SUR LES FLOTS

On peut donner une valeur soi même au statut d'un flot grâce à la fonction membre clear qui prend un entier en paramètre.

Cela peut servir si on surcharge les opérateurs << ou >> pour un type utilisateur.

On peut alors dire si une opération s'est mal passée.

Par exemple, si 's' désigne un flot, l'instruction suivante :

```
s.clear(ios::badbit)
```

Mettra 1 au badbit de s et mettre les autres bits à 0.

LES FLOTS

ERREURS SUR LES FLOTS

Les opérateurs ‘()’ et ‘!’ ont également été surchargés pour connaitre le statut d’un flot.

Ainsi, si ‘s’ désigne un flot : s() prendra la valeur de s.good().

De même, ‘!s’ prend la valeur ‘vrai’ si une erreur est apparue sur le flot.

```
!s <-> !s.good()
```

LES FLOTS

LES FICHIERS EN LECTURE

```
#include <fstream>
#include <iostream>

using namespace std;

int main()
{
    ifstream fichier("test.dat");

    int i;
    while (fichier){
        fichier >> i;
        cout << i << endl;
    }
}
```

Dans le fichier header fstream, il est déclaré une classe ifstream.

Cette classe permet de créer un flot en lecture à partir d'un fichier.

On créée donc un objet de type ifstream.

Le constructeur de cet objet prend le nom du fichier en paramètre.(Le fichier doit alors se trouver dans le même répertoire que l'exécutable)

Ici, on ouvre en lecture un fichier 'test.dat'.

Il existe d'autres paramètres possibles pour le constructeur.

LES FLOTS

LES FICHIERS EN LECTURE

```
#include <fstream>
#include <iostream>

using namespace std;

int main()
{
    ifstream fichier("test.dat");

    int i;
    while (fichier){
        fichier >> i;
        cout << i << endl;
    }
}
```

Ensuite, on boucle tant que le fichier a un statut good, comme on l'a vu dans les diapositives précédentes.

Si on sait que le fichier contient des entiers au format texte, (c'est-à-dire dans un format lisible par un humain, pas en binaire), on peut utiliser l'opérateur pour la lecture des int.

On va donc lire entier par entier le contenu du fichier.

L'opérateur >> va utiliser comme séparateur les espaces blancs comme on l'a déjà dit. (y compris le caractère '\n' de passage à la ligne suivante).

LES FLOTS

LES FICHIERS EN LECTURE

```cpp
#include <fstream>
#include <iostream>

using namespace std;

int main()
{
    ifstream fichier("test.dat");

    int i;
    while (fichier){
        fichier >> i;
        cout << i << endl;
    }
}
```

```
123
456
789
0
1
42
```

test.dat

```
$ ./a.exe
123
456
789
0
1
42
42
```

Que se passe t il ici ?

Le nombre 42 n'apparait qu'une fois dans le fichier test.dat, mais il apparait deux fois quand on exécute le programme !?

En fait, il manque quelque chose.

On lit le fichier tant qu'il reste des caractères à lire dedans. Seulement pour le savoir on est obligé de lire une dernière fois et que le flot ne renvoie rien, c'est à ce moment là que le bit de fin de fichier se met à 1 et qu'on sort de la boucle.

LES FLOTS

LES FICHIERS EN LECTURE

```
#include <fstream>
#include <iostream>

using namespace std;

int main()
{
    ifstream fichier("test.dat");

    int i;
    while (fichier){
        fichier >> i;
        cout << i << endl;
    }
}
```

```
123
456
789
0
1
42
```

test.dat

```
$ ./a.exe
123
456
789
0
1
42
42
```

Comme on ne remet pas à 0 la valeur de i à chaque tour de boucle, lorsque le flot ne renvoie rien, sa valeur ne change pas et il garde la valeur de tour de boucle précèdent.

D'où le deuxième '42' qui apparait en fin de fichier.

Comment résoudre ce problème ?

LES FLOTS

LES FICHIERS EN LECTURE

```
#include <fstream>
#include <iostream>

using namespace std;

int main()
{
    ifstream fichier("test.dat");

    int i;
    while (fichier){
        fichier >> i;
        if (fichier)
            cout << i << endl;
    }
}
```

```
123
456
789
0
1
42
```

test.dat

```
$ ./a.exe
123
456
789
0
1
42
```

On rajoute dans la boucle une vérification.

En fin de fichier la lecture qui ne renvoie rien met le bit de fin de fichier à vrai.

À ce moment là, on n'affiche pas la dernière valeur de i.

Et le problème est résolu.

LES FLOTS

LES FICHIERS EN LECTURE

```
#include <fstream>
#include <iostream>

using namespace std;

int main()
{
    ifstream fichier("test2.dat");

    if (!fichier)
    {
        cerr << "Ce fichier n'existe pas !" << endl;
        return 1;
    }

    int i;
    while (fichier){
        fichier >> i;
        if (fichier)
            cout << i << endl;
    }
}
```

En toute rigueur, on ne peut pas faire confiance à l'existence du fichier « test.dat » dans l'exemple.

Il faudrait donc une vérification préliminaire qui n'essaierait pas de lire un fichier et par exemple, qui quitterait le programme.

Ici, on écrit donc sur la sortie d'erreur un message explicite, et on quitte la fonction main, donc le programme.

LES FLOTS

LES FICHIERS EN LECTURE

```
#include <fstream>
#include <iostream>
#include <cstring>

using namespace std;
int main()
{
    ifstream fichier("lorem.txt");
    if (!fichier)
    {
        cerr << "Ce fichier n'existe pas !" << endl;
        return 1;
    }

    int i;
    char txt[20+1];
    while (fichier){
        bzero(txt, 20+1);
        fichier.read(txt, 20);
        cout << "on a lu : [" << txt
        <<"]"<< endl;
    }
}
```

Comme sur le flot cin, il existe aussi les fonctions non formatées sur les flots de fichiers.

Ici, on lit un fichier texte 20 caractères par 20 caractères. En effet, la fonction read lit 20 octets (des caractères) qu'elle place dans le tableau 'txt'.

On se sera assurés de plusieurs points :

- Le tableau txt peut contenir 21 éléments.
- À chaque tour de boucle, on met les éléments de 'txt' à 0 en utilisant la fonction du header <cstring> bzero.

LES FLOTS

LES FICHIERS EN LECTURE

```
#include <fstream>
#include <iostream>
#include <cstring>

using namespace std;
int main()
{
    ifstream fichier("lorem.txt");
    if (!fichier)
    {
        cerr << "Ce fichier n'existe pas !" << endl;
        return 1;
    }

    int i;
    char txt[20+1];
    while (fichier){
        bzero(txt, 20+1);
        fichier.read(txt, 20);
        cout << "on a lu : [" << txt
        <<"]"<< endl;
    }
}
```

Pourquoi 21 et pas 20 éléments ?

La fonction read va remplir les 20 octets qu'on lui passe en paramètres. Ici, il s'agit de chaines de caractères. Donc il faut, si on veut pouvoir les afficher, s'assurer qu'il reste le caractère nul en fin de chaine, et donc lui prévoir systématiquement une place !

LES FLOTS

LES FICHIERS EN LECTURE

```
#include <fstream>
#include <iostream>
#include <cstring>

using namespace std;
int main()
{
    ifstream fichier("lorem.txt");
    if (!fichier)
    {
        cerr << "Ce fichier n'existe pas !" << endl;
        return 1;
    }

    int i;
    char txt[20+1];
    while (fichier){
        bzero(txt, 20+1);
        fichier.read(txt, 20);
        cout << "on a lu : [" << txt
        <<"]"<< endl;
    }
}
```

Pourquoi utiliser la fonction bzero (ou memset) ?

Il s'agit d'une fonction qui met à 0 la valeur de chaque élément du tableau 'txt'.

Le nombre de caractères à lire dans un fichier n'est évidement pas forcément un multiple du nombre de caractères par read.

Il y a donc, en fin de fichier, un certains nombres de caractères qui ne suffisent pas à remplir le tableau.

En mettant le tableau à 0, on s'assure de pouvoir correctement afficher la dernière chaine.

LES FLOTS

LES FICHIERS EN LECTURE

```
#include <fstream>
#include <iostream>
#include <cstring>

using namespace std;
int main()
{
    ifstream fichier("lorem.txt");
    if (!fichier)
    {
        cerr << "Ce fichier n'existe pas !" << endl;
        return 1;
    }

    int i;
    char txt[20+1];
    while (fichier){
        bzero(txt, 20+1);
        fichier.read(txt, 20);
        cout << "on a lu : [" << txt
        <<"]"<< endl;
    }
}
```

```
$ cat lorem.txt
Lorem ipsum dolor sit amet, consectetur
adipiscing elit. Praesent auctor vel enim
vel ullamcorper.
```

```
$ ./a.exe
on a lu : [Lorem ipsum dolor si]
on a lu : [t amet, consectetur ]
on a lu : [adipiscing elit. Pra]
on a lu : [esent auctor vel eni]
on a lu : [m vel ullamcorper. ]
```

LES FLOTS

LES FICHIERS EN LECTURE

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main()
{
    ifstream fichier("test.dat");

    while (fichier){
        string s;
        getline(fichier, s);
        cout << "LU : " << s << endl;
    }
}
```

Il existe aussi la fonction getline, déclarée dans le header <string>, qui prend un flot en paramètre et une string, et qui va extraire la ‘ligne’ suivante du flot dans la chaine.

Il existe également la possibilité d’un troisième paramètre pour définir le délimiteur. Par défaut il s’agit du caractère de fin de ligne ‘\n’.

LES FLOTS

LES FICHIERS EN ÉCRITURE

```
#include <fstream>

using namespace std;

int main()
{
    ofstream fichier("test.bak");

    fichier << 123 << endl;
    fichier << 456 << endl;
    fichier << 789 << endl;
    fichier << "FIN" << endl;
}
```

Comme on ouvre un fichier en lecture, on peut également créer un flot en sortie vers un fichier.

On va donc créer un objet de type ofstream.

Comme pour les flots de sortie, on peut utiliser l'opérateur '<<' pour écrire dans le fichier.

Attention :

Lorsqu'on ouvre un fichier en écriture, par défaut, si ce fichier existe, son contenu est écrasé. Tout contenu précèdent sera perdu !

LES FLOTS

LES FICHIERS EN ÉCRITURE

```
#include <fstream>

using namespace std;

int main()
{
    ofstream fichier("test.bak", ios::app);

    fichier << 123 << endl;
    fichier << 456 << endl;
    fichier << 789 << endl;
    fichier << "FIN" << endl;
}
```

Pour éviter ce comportement par défaut, on pourra utiliser une option lors de la construction du flot.

Il s'agit de ios::app, donc une option définie dans la classe ios.

Elle permet d'ouvrir le flot en précisant que les écritures supplémentaires auront lieu à la fin du fichier.

LES FLOTS

LES FICHIERS EN ÉCRITURE

```
#include <fstream>

using namespace std;

int main()
{
    ofstream fichier("test.bak", ios::app);

    fichier << 123 << endl;
    fichier << 456 << endl;
    fichier << 789 << endl;
    fichier << "FIN" << endl;
    fichier.flush();
}
```

Il peut arriver que l'on veuille s'assurer que tous les éléments contenus dans le tampon d'écriture soient écrits dans le fichier, avant la fermeture du fichier.

Pour cela on peut utiliser la fonction membre flush() qui va vider les tampons dans le fichier.

C'est à ce moment là que l'on est sûr que le fichier contiendra effectivement les données que l'on a passées à l'opérateur '<<'.

LES FLOTS

LA FONCTION CLOSE

```
#include <fstream>

using namespace std;

int main()
{
    ofstream fichier("test.bak", ios::app);

    fichier << 123 << endl;
    fichier << 456 << endl;
    fichier << 789 << endl;
    fichier << "FIN" << endl;
    fichier.close();
}
```

Pour les fichiers en écriture comme en lecture, jusqu'à présent, la connexion au fichier était fermée en même temps que la destruction de l'objet (i/o)fstream.

Lorsqu'on a lu le contenu d'un fichier, ou que l'on a écrit ce qui devait l'être, on doit vouloir libérer les ressources dés qu'elles sont devenues inutiles.

On utilisera la fonction membre close() sur le flot.

Cette fonction videra les tampons qui doivent l'être et fermera l'accès vers le fichier.

LES FLOTS

LA FONCTION CLOSE

```
#include <fstream>

using namespace std;

int main()
{
    ofstream fichier("test.bak", ios::app);

    fichier << 123 << endl;
    fichier << 456 << endl;
    fichier << 789 << endl;
    fichier << "FIN" << endl;
    fichier.close();
}
```

Evidemment, utiliser un fichier après sa fermeture conduira à une erreur.

LES *EXCEPTIONS*

LES EXCEPTIONS

Pour gérer les situations inattendues lors de l'exécution d'un programme, il n'y a pas de solution simple.

On peut, lors de chaque instruction qui peut provoquer une erreur d'exécution, faire des vérifications systématiques. Cela reviendrait à de fastidieuses vérifications. Et ensuite ?

En général dans un programme relativement complexe, l'erreur a lieu dans une fonction, et le mécanisme de gestion de l'erreur a lieu ailleurs dans le code.

Comment faire alors pour gérer ces comportements le mieux possible ?

LES EXCEPTIONS

Il existe en C++ (comme en Java et dans d'autres langages) un mécanisme de gestion des erreurs possibles dans un programme.

Ce mécanisme est appelé 'Exceptions'.

Par exemple :

Lorsqu'on construit un vecteur, et qu'on demande à l'utilisateur de choisir un indice de ce vecteur, rien de lui interdit de rentrer un indice négatif, ou un indice trop grand.

De même, dans le choix d'un fichier à ouvrir, il se peut que ce fichier n'existe pas. On doit alors gérer le comportement du programme pour qu'il s'adapte à cette possibilité : on n'a pas envie que word plante si on ouvre un fichier qui n'existe pas !

LES EXCEPTIONS

UN EXEMPLE

```
#ifndef __VECTOR_HPP__
#define __VECTOR_HPP__
#include <string>
using namespace std;
template <typename T>
class Vector
{
   T* _datas;
   int _size;
public:
   Vector(int n){ _datas = new T[n];_size = n;}
   ~Vector() { delete [] _datas;}
   T& operator[] (int i){
      if (i < 0 || i >= _size)
         throw (string("Indice hors limite : ") + to_string(i));
      return _datas[i];
   }
};
#endif
```

Dans cet exemple, on retrouve la classe template Vector.

On a juste surchargé l'opérateur [], avec une nouveauté à l'intérieur de celui-ci.

Il y a maintenant une condition sur l'indice passé à l'opérateur.

Lorsque cet indice est négatif, ou que celui-ci est trop grand, on 'lance une exception.'

Le mot clé qui permet de lancer une exception est 'throw'. Celui-ci s'utilise conjointement avec une expression d'un type quelconque, utilisateur ou natif.

LES EXCEPTIONS

UN EXEMPLE

```
#ifndef __VECTOR_HPP__
#define __VECTOR_HPP__
#include <string>
using namespace std;
template <typename T>
class Vector
{
   T* _datas;
   int _size;
public:
   Vector(int n){ _datas = new T[n];_size = n;}
   ~Vector() { delete [] _datas;}
   T& operator[] (int i){
      if (i < 0 || i >= _size)
         throw (string("Indice hors limite : ") + to_string(i));
      return _datas[i];
   }
};
#endif
```

Ici, throw est suivi d'une string, une chaîne de caractère que l'on construit pour décrire l'erreur qui a eu lieu.

Même si cette utilisation de throw est possible, elle n'est néanmoins pas recommandée.

En effet, on verra qu'il vaut mieux utiliser des classes déjà présentes dans la librairie standard.

LES EXCEPTIONS

UN EXEMPLE

```
#ifndef __VECTOR_HPP__
#define __VECTOR_HPP__
#include <string>
using namespace std;
template <typename T>
class Vector
{
   T* _datas;
   int _size;
public:
   Vector(int n){ _datas = new T[n];_size = n;}
   ~Vector() { delete [] _datas;}
   T& operator[] (int i){
      if (i < 0 || i >= _size)
         throw (string("Indice hors limite : ") + to_string(i));
      return _datas[i];
   }
};
#endif
```

```
#include <iostream>
#include <string>
#include "Vector.hpp"
using namespace std;
int main(){
   Vector<double> v(5);
   try {
      v[3] = 3;
   }
   catch (string s){
      cout << s << endl;
   }
   try {
      v[5] = 3;
   }
   catch (string s){
      cout << s << endl;
   }
}
```

Ce n'est pas tout de lancer une exception, il faut aussi pouvoir l'intercepter quelque part.

Pour cela, on va utiliser un bloc dit try {....} catch {... }.

On place dans le bloc 'try' les instructions qui sont susceptibles de lever une exceptions.

Le bloc catch lui prend en paramètre une expression d'un type donné.

Si l'exception levée est de ce type, alors les instructions de son blocs seront exécutées.

LES EXCEPTIONS

EXCEPTIONS DE TYPE UTILISATEUR

A.hh

```
#ifndef __A_HH__
#define __A_HH__
#include <string>
using namespace std;
class Exc
{
    public:
        Exc(string);
        string _message;
};

class A
{
public:
    void f();
};

#endif
```

A.cpp

```
#include "A.hh"

Exc::Exc(string s)
: _message(s)
{}

void A::f()
{
    throw Exc("KO !");
}
```

main.cpp

```
#include <iostream>
#include "A.hh"
using namespace std;
int main()
{
    A a;
    try{
        a.f();
    }
    catch (Exc e){
        cout << e._message << endl;
    }
}
```

On a dit que les exceptions levées pouvaient être de n'importe quel type.

Aussi, afin de bien faire la distinction entre les différentes exceptions qui peuvent se déclencher au cours de l'exécution, on peut créer des types différents pour chacune d'elles.

LES EXCEPTIONS

EXCEPTIONS DE TYPE UTILISATEUR

A.hh

```
#ifndef __A_HH__
#define __A_HH__
#include <string>
using namespace std;
class Exc
{
    public:
        Exc(string);
        string _message;
};

class A
{
public:
    void f();
};

#endif
```

A.cpp

```
#include "A.hh"

Exc::Exc(string s)
: _message(s)
{}

void A::f()
{
    throw Exc("KO !");
}
```

main.cpp

```
#include <iostream>
#include "A.hh"
using namespace std;
int main()
{
    A a;
    try{
        a.f();
    }
    catch (Exc e){
        cout << e._message << endl;
    }
}
```

Ici, la classe Exc va servir à créer un type exception pouvant contenir un message.

Ce message sera initialisé lors de la levée de l'exception.

LES EXCEPTIONS

EXCEPTIONS DE TYPE UTILISATEUR

A.hh

```
#ifndef __A_HH__
#define __A_HH__
#include <string>
using namespace std;
class Exc
{
    public:
        Exc(string);
        string _message;
};

class A
{
public:
    void f();
};

#endif
```

A.cpp

```
#include "A.hh"

Exc::Exc(string s)
: _message(s)
{}

void A::f()
{
    throw Exc("KO !");
}
```

main.cpp

```
#include <iostream>
#include "A.hh"
using namespace std;
int main()
{
    A a;
    try{
        a.f();
    }
    catch (Exc e){
        cout << e._message << endl;
    }
}
```

Ici, par exemple, lors de la levée (appel de l'instruction throw) on construit une exception en lui passant une chaine de caractère en paramètre.

Cette chaine de caractère va pouvoir être affichée dans le bloc catch correspondant à cette exception.

LES EXCEPTIONS

PLUSIEURS CATCH

A.hh

```
#ifndef __A_HH__
#define __A_HH__
#include <string>
using namespace std;
class Exc
{
    public:
        Exc(string);
        string _message;
};

class A
{
public:
    void f();
    void f2();
};

#endif
```

A.cpp

```
#include "A.hh"

Exc::Exc(string s)
: _message(s)
{}

void A::f()
{
    throw Exc("KO !");
}

void A::f2()
{
    throw string("Encore KO");
}
```

main.cpp

```
#include <iostream>
#include "A.hh"
using namespace std;
int main()
{
    A a;
    try {
        a.f2();
    }
    catch (Exc e){
        cout << e._message << endl;
    }
    catch (string s){
        cout << s << endl;
    }
}
```

À la suite d'un bloc try, on peut mettre plusieurs blocs catch.

Le bloc catch correspondant au type de l'exception qui a été levée sera celui qui sera exécuté. Il recevra en paramètre l'exception (passée par valeur depuis le throw).

LES EXCEPTIONS

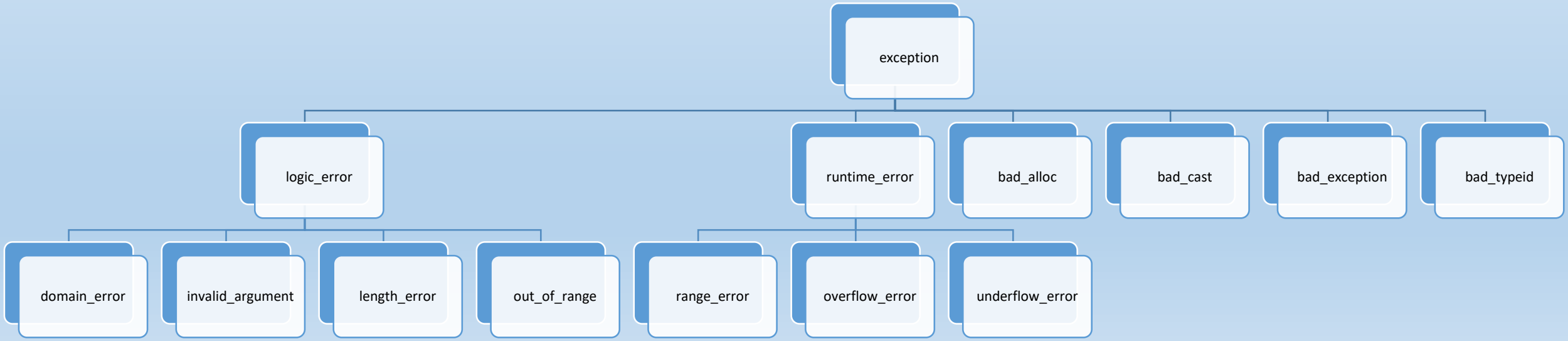
LES EXCEPTIONS STANDARD

Il existe des classes d'exceptions déjà disponibles dans la librairie standard.

Elles dérivent toutes d'une même classe « exception ».

Cette classe fournit une fonction virtuelle « what » qui renvoie une chaine de caractères de type C (char *) décrivant l'erreur.

Au niveau de l'héritage, elles s'articulent de la façon suivante :

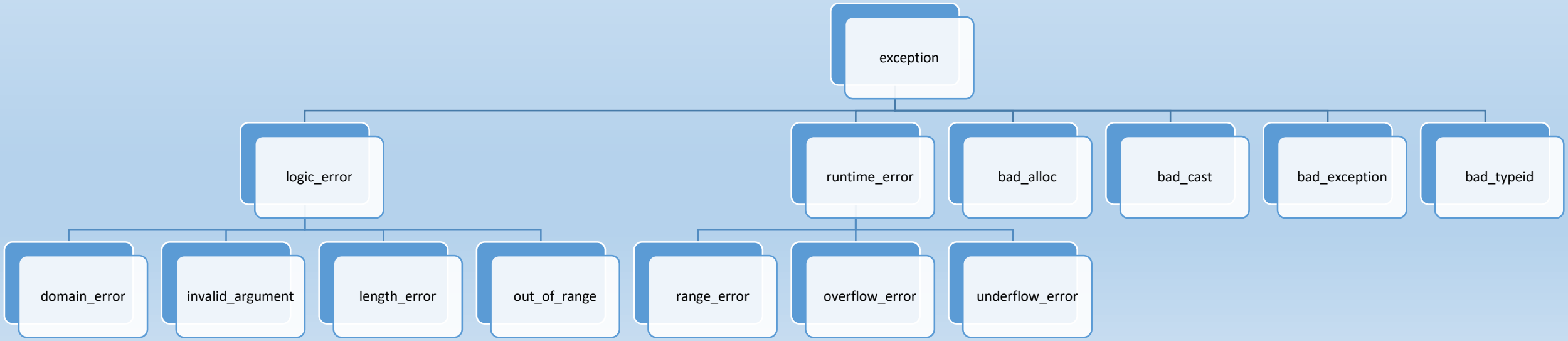


LES EXCEPTIONS

LES EXCEPTIONS STANDARD

On peut soit réutiliser ces classes dans le code, en fonction de l'utilisation de cette classe. Par exemple, ce n'est pas déraisonnable d'utiliser la classe range_error dans le cas d'un mauvais indice d'un tableau.

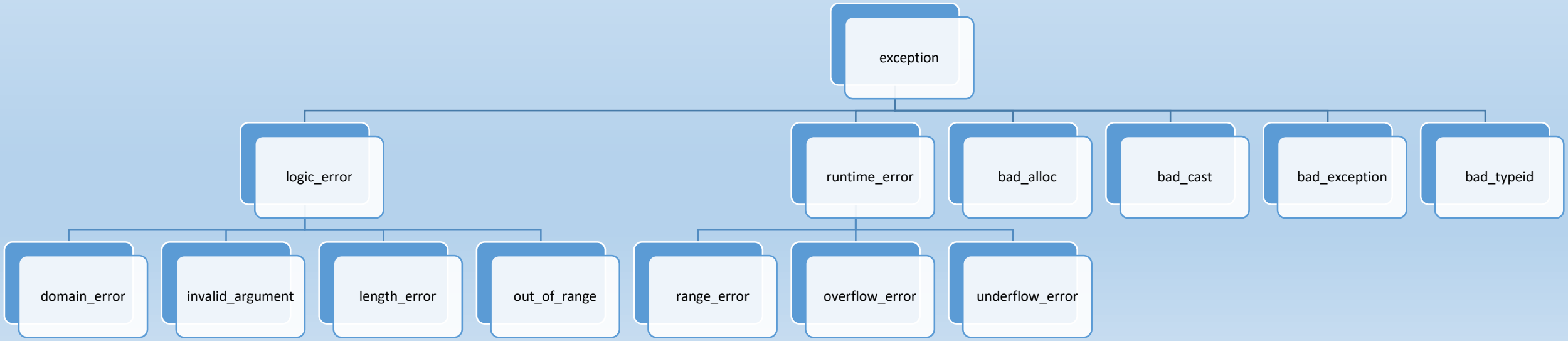
Par contre, utiliser bad_alloc pour une erreur non liée à de la mémoire serait malvenue.



LES EXCEPTIONS

LES EXCEPTIONS STANDARD

On peut également créer soi même des classes qui hériteraient de la classe exception, afin qu'elle puisse être gérée comme les autres.



LES EXCEPTIONS

LES EXCEPTIONS STANDARD

```
#include <stdexcept>
#include <iostream>
using namespace std;
int main()
{
    try
    {
        int *tab = new int[10000000000];
    }
    catch (bad_alloc &e)
    {
        cout << "Error" << endl;
        cout << e.what() << endl;
    }
}
```

```
$ ./a.exe
Error
std::bad_alloc
```

- bad_alloc est l'exception levée par l'opérateur new dans le cas d'un échec de l'allocation mémoire.(Comme dans l'exemple ci contre)
- bad_cast est levée dans le cas d'un dynamique cast qui serait en erreur.
- bad_typeid est levée lors d'une erreur de l'opérateur typeid.

POUR FINIR

POUR FINIR

Pour finir ce cours, nous allons présenter dans ce chapitre quelques notions qui n'ont pas été, volontairement ou par manque de temps, abordées dans ce cours.

Nous allons également aborder quelques notions présentes dans les normes récentes de C++, les normes C++11 et C++14.

POUR FINIR

LES ESPACES DE NOMS

Lors de la création de gros programmes ou de bibliothèques, on peut avoir envie de regrouper les symboles (noms de fonctions/de classes/de variables globales) sous une même entité, un nom, qui permettrait de qualifier et de regrouper ces symboles.

On va alors utiliser des espaces de noms. Nous en avons déjà rencontré un lors de ce cours, que nous avons utilisé. Il s'agissait de l'espace de nom de la bibliothèque standard : std.

POUR FINIR

LES ESPACES DE NOMS

```
#ifndef __MYNAMESPACE_HH__
#define __MYNAMESPACE_HH__
namespace MyName
{
   class A
   {
      /* déclarations de
      la classe A*/
   };

   class B
   {  /* déclarations de
      la classe B*/
      public :
         B();
   };
   void fct(); // une fonction
} // pas de ;
#endif
```

```
#include "MyNamespace.hh"

namespace MyName{
   void fct()
   {
      // instructions
   }

   B::B()
   {
      // contructeur de B.
   }
}
```

Pour déclarer un nouvel espace de nom, on va utiliser le mot clé « namespace » suivi d'un bloc d'accolades ouvrantes et fermantes.

Dans ce bloc, on pourra déclarer des classes, des fonctions, des variables…

Dans le fichier de code .cpp correspondant, on fait la même chose pour la définition des objets déclarées.

POUR FINIR

LES ESPACES DE NOMS

```
#ifndef __MYNAMESPACE_HH__
#define __MYNAMESPACE_HH__
namespace MyName
{
   class A
   {
      /* déclarations de
      la classe A*/
   };

   class B
   {  /* déclarations de
      la classe B*/
      public :
         B();
   };
   void fct(); // une fonction
} // pas de ;
#endif
```

```
#include "MyNamespace.hh"

namespace MyName{
   void fct()
   {
      // instructions
   }

   B::B()
   {
      // constructeur de B.
   }
}
```

```
#include "MyNamespace.hh"

int main()
{
   MyName::fct();

   MyName::A a;
}
```

Pour utiliser les symboles déclarés dans un espace de nom, on va utiliser l'opérateur de résolution de portée « :: ».

Pas de surprise ici, donc.

POUR FINIR

LES ESPACES DE NOMS

```
#ifndef __MYNAMESPACE_HH__
#define __MYNAMESPACE_HH__
namespace MyName
{
   class A
   {
      /* déclarations de
      la classe A*/
   };

   class B
   {  /* déclarations de
      la classe B*/
      public :
         B();
   };
   void fct(); // une fonction
} // pas de ;
#endif
```

```
#include "MyNamespace.hh"

namespace MyName{
   void fct()
   {
      // instructions
   }

   B::B()
   {
      // constructeur de B.
   }
}
```

```
#include "MyNamespace.hh"

int main()
{
   MyName::fct();

   MyName::A a;
}
```

On peut également créer un espace de nom, au fur et à mesure.

C'est-à-dire qu'on peut déclarer des symboles comme faisant parti d'un espace de nom dans plusieurs fichiers entête.

POUR FINIR

LA DIRECTIVE USING

```
#ifndef __MYNAMESPACE_HH__
#define __MYNAMESPACE_HH__
namespace MyName
{
   class A
   {
      /* déclarations de
      la classe A*/
   };

   class B
   {  /* déclarations de
      la classe B*/
      public :
         B();
   };
   void fct(); // une fonction
} // pas de ;
#endif
```

```
#include "MyNamespace.hh"


class A
{
public:
   double x;
};

int main()
{
   MyName::fct();

   // A est synonyme de
   // MyName::A
   using MyName::A;
   A a;    //
   //a.x = 0; // NE COMPILE PAS !!
   // l'espace de nom a masqué la class
   // A de l'espace global.

   // pour atteindre l'espace global.
   ::A b;
   b.x = 0;
}
```

On peut utiliser la directive using que nous connaissons déjà, de deux façons.

La première est de l'utiliser directement avec un espace de nom. Nous reviendrons sur cette utilisation plus tard.

On peut aussi utiliser « using » symbole par symbole.

Ici, A devient synonyme de MyName::A dans la suite du code.

POUR FINIR

LA DIRECTIVE USING

```
#ifndef __MYNAMESPACE_HH__
#define __MYNAMESPACE_HH__
namespace MyName
{
   class A
   {
      /* déclarations de
      la classe A*/
   };

   class B
   {  /* déclarations de
      la classe B*/
      public :
         B();
   };
   void fct(); // une fonction
} // pas de ;
#endif
```

```
#include "MyNamespace.hh"


class A
{
public:
   double x;
};

int main()
{
   MyName::fct();

   // A est synonyme de
   // MyName::A
   using MyName::A;
   A a;    //
   //a.x = 0; // NE COMPILE PAS !!
   // l'espace de nom a masqué la class
   // A de l'espace global.

   // pour atteindre l'espace global.
   ::A b;
   b.x = 0;
}
```

Deux points sont à noter :

- On n'est pas obligés de placer using à l'extérieur d'une fonction. Il peut être placé n'importe où dans le code, et n'a d'effet que sur la portée locale : c'est-à-dire jusqu'à la fin du bloc d'instruction courant.
- Dans cet exemple, une classe A apparait dans le namespace MyName. Mais une autre classe A est déclarée dans le namespace global.

POUR FINIR

LA DIRECTIVE USING

```
#ifndef __MYNAMESPACE_HH__
#define __MYNAMESPACE_HH__
namespace MyName
{
   class A
   {
      /* déclarations de
      la classe A*/
   };

   class B
   {  /* déclarations de
      la classe B*/
      public :
         B();
   };
   void fct(); // une fonction
} // pas de ;
#endif
```

```
#include "MyNamespace.hh"


class A
{
public:
   double x;
};

int main()
{
   MyName::fct();

   // A est synonyme de
   // MyName::A
   using MyName::A;
   A a;    //
   //a.x = 0; // NE COMPILE PAS !!
   // l'espace de nom a masqué la class
   // A de l'espace global.

   // pour atteindre l'espace global.
   ::A b;
   b.x = 0;
}
```

- Comment lever l'ambiguité ?

Ici, la directive using masque le symbole globale A.

Pour y accéder quand même, on va utiliser l'opérateur de résolution de portée sur l 'espace globale c'est-à-dire sans préciser d'espace de nom.

On lève ainsi l'ambiguïté qui avait été levée.

POUR FINIR

LA DIRECTIVE USING

```
#ifndef __MYNAMESPACE_HH__
#define __MYNAMESPACE_HH__
namespace MyName
{
   class A
   {
      /* déclarations de
      la classe A*/
   };

   class B
   {  /* déclarations de
      la classe B*/
      public :
         B();
   };
   void fct(); // une fonction
} // pas de ;
#endif
```

```
#include "MyNamespace.hh"
using namespace MyName;

class A
{
public:
   double x;
};


int main()
{
   fct();

   B b;
   A a;
}
```

On peut bien sur utiliser la directive comme on l'a déjà fait pour l'espace de nom std.

Dans cet exemple, cela ne pose pas de problème.

Mais que se passe t il dans le cas où un symbole porte le même nom dans l'espace global ?

```
main.cpp: Dans la fonction ‘int main()’:
main.cpp:16:5: erreur : reference to ‘A’ is ambiguous
     A a;
     ^
main.cpp:4:7: note : candidats sont : class A
 class A
       ^
In file included from main.cpp:1:0:
MyNamespace.hh:6:11: note :                 class MyName::A
     class A
           ^
```

POUR FINIR

LA DIRECTIVE USING

```
#ifndef __MYNAMESPACE_HH__
#define __MYNAMESPACE_HH__
namespace MyName
{
   class A
   {
      /* déclarations de
      la classe A*/
   };

   class B
   {  /* déclarations de
      la classe B*/
      public :
         B();
   };
   void fct(); // une fonction
} // pas de ;
#endif
```

```
#include "MyNamespace.hh"
using namespace MyName;

class A
{
public:
   double x;
};


int main()
{
   fct();

   B b;
   A a;
}
```

Le compilateur ne sait plus de quel objet il s'agit et s'arrête en erreur.

Dans ce cas, il y a deux solutions :

- Ne pas utiliser la directive using.
- Lever les ambiguïtés au cas par cas, en utilisant l'opérateur :: quand cela est nécessaire.

```
main.cpp: Dans la fonction ‘int main()’:
main.cpp:16:5: erreur : reference to ‘A’ is ambiguous
     A a;
     ^
main.cpp:4:7: note : candidats sont : class A
 class A
       ^
In file included from main.cpp:1:0:
MyNamespace.hh:6:11: note :                 class MyName::A
     class A
           ^
```

POUR FINIR

LA DIRECTIVE USING

```
#ifndef __MYNAMESPACE_HH__
#define __MYNAMESPACE_HH__
namespace MyName
{
   class A
   {
      /* déclarations de
      la classe A*/
   };

   class B
   {  /* déclarations de
      la classe B*/
      public :
         B();
   };
   void fct(); // une fonction
} // pas de ;
#endif
```

```
#include "MyNamespace.hh"
using namespace MyName;

class A
{
public:
   double x;
};


int main()
{
   fct();

   B b;
   A a;
}
```

```
main.cpp: Dans la fonction ‘int main()’:
main.cpp:16:5: erreur : reference to ‘A’ is ambiguous
     A a;
     ^
main.cpp:4:7: note : candidats sont : class A
 class A
       ^
In file included from main.cpp:1:0:
MyNamespace.hh:6:11: note :                 class MyName::A
     class A
           ^
```

Cela nécessite évidemment de savoir quel objet on désire utiliser.

Facile si c’est son propre code, mais plus délicat si on n’est pas créateur du code à maintenir…

POUR FINIR

LES ARGUMENTS PAR DÉFAUT

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

int fct(int=0);

int fct(int a)
{
    cout << a << endl;
}

int main()
{
    fct();
    fct(2);
}
```

Lors de la déclaration d'une fonction, on peut définir une valeur par défaut pour certains paramètres d'une fonction.

Ces valeurs seront utilisées si aucunes valeurs n'est données par l'utilisateur.

Afin de ne pas créer d'ambiguïtés, les arguments ayant une valeur par défaut doivent être les derniers de la liste d'arguments. En effet, pour la fonction

```
void fct(int a = 0, int b, int c = 0)
{
    cout << a << b << c << endl;
}
```

Que donnerait l'appel de : fct(3, 2); ??

POUR FINIR

LES ARGUMENTS PAR DÉFAUT

```
int fct(double);
int fct(double, int=0);

int fct(double d)
{
    cout << d << endl;
}

int fct(double d, int a)
{
    cout << a << endl;
}
```

On évitera aussi de se placer dans ce genre de situation, où on utilise à la fois des valeurs par défaut pour les arguments et la surdéfinition de fonction.

En effet, quel effet aurait

fct(3);

Evidemment, le compilateur signalerait une ambiguïté et s'arrêterait en erreur.

POUR FINIR

TYPEDEF

```
#include <iostream>

using std::cout;
using std::endl;

int main()
{
    typedef int entier;

    entier i = 20;
    cout << i << endl;
}
```

Le mot clé typedef est surtout utilisé en programmation C. Cependant son usage persiste encore en C++ et nous allons donc en toucher deux mots ici.

Typedef permet de déclarer un synonyme pour un type.

En effet, les types peuvent parfois s’ avérer complexes et longs et difficile à lire.

Aussi on préfère parfois les renommer pour simplifier la lecture.

POUR FINIR

TYPEDEF

```
#include <iostream>

using std::cout;
using std::endl;

int main()
{
    typedef int entier;

    entier i = 20;
    cout << i << endl;
}
```

La syntaxe est :

```
typedef ancien_type nouveau_type;
```

Ici, on crée donc un type « entier » synonyme de int.

On peut ensuite utiliser « entier » en lieu et place de int dans le code.

Attention toutefois à ne pas rendre le code difficile à lire ou à comprendre en utilisant typedef de façon inopportune.

POUR FINIR

TYPEDEF

```
#include <iostream>
#include <utility>
#include <vector>

using std::cout;
using std::endl;

int main()
{
    typedef std::pair<std::string,int>  value_t;
    typedef std::vector<value_t>        values_t;

    values_t values;    //  vector de pair de string/int
}
```

Voici un exemple un peu plus réaliste de l'utilisation de typedef.

On voit que l'on crée deux nouveaux types, value_t et values_t permettant de raccourcir et simplifier le code qui suit.

POUR FINIR

LES NORMES MODERNES DE C++

Comme de nombreux langages de programmation (sinon tous) le C++ est un langage vivant qui se construit et se définit au fil du temps.

Ainsi, les techniques de programmation évoluant, de nouveaux besoins l'amènent à évoluer.

Afin que le langage conserve une cohérence et que chaque éditeur de compilateur ne fasse « son C++ à lui », un comité met en place une norme, qui vient écrire dans le marbre les spécificités du langage.

Il y a jusqu'à maintenant eu 4 normes de C++, 98, 03, 11 et 14. Une norme 17 est également prévue dans un avenir proche.

Ces normes peuvent amener de profonds changements ou, au contraire, venir peaufiner la norme précédente.

POUR FINIR

LES NORMES MODERNES DE C++

Il ne faut pas croire que le fait qu'une fonctionnalité soit écrite dans la norme permette de l'utiliser. En effet, cela dépend grandement du compilateur (et de sa version !) que vous utilisez.

Ainsi, les versions anciennes de g++ ne permettent pas d'utiliser toutes les fonctionnalités du standard C++11 etc.

Avec le compilateur g++, pour activer la prise en charge de ces normes, on peut utiliser les options de compilation :

`--std=c++11` pour C++11

`--std=c++14` pour C++14

POUR FINIR

LES NORMES MODERNES DE C++

Sans entrer dans les détails de toutes les possibilités offertes par les versions récentes de C++, nous allons examiner ici quelques possibilités offertes qui peuvent être pratiques lors de l'écriture d'un code, ou que vous pouvez être amené à rencontrer lors de la lecture de code écris par un tiers.

POUR FINIR

INFÉRENCE DE TYPE - AUTO

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    auto i = 0;
    auto d = 3.14;
    auto s = string("Ok");

    i++;
    d = d +2;
    s += "Ko";
    cout << s << endl;
}
```

C++11 met en place un mécanisme dit d'inférence de type qui permet de déduire le type d'une variable en fonction de la valeur de son initialisation.

Pour l'utiliser, on dispose du mot clé « auto » qui permet de ne pas préciser de type lors de la déclaration d'une variable.

Il est par contre nécessaire d'initialiser la variable en même temps, comme dans les exemples ci contre.

POUR FINIR

INFÉRENCE DE TYPE - AUTO

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


int main()
{
    double tab[] = {1, 2, 3};
    vector<double> v(tab, tab+3);
    for (auto iv = v.begin(); iv != v.end(); ++iv)
        cout << *iv << endl;
}
```

L'inférence de type prend tout son intérêt lorsqu'on doit travailler avec des types longs à écrire.

Par exemple, ici, avec les itérateurs de vecteur.

POUR FINIR

LISTE D'INITIALISATION

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

int main()
{
    // inutile en C++11
    //double tab[] = {1, 2, 3};
    vector<double> v = {1, 2, 3}; // liste
                                  // d'initialisation
                                  // pour les vecteurs
    for (auto iv = v.begin(); iv != v.end(); ++iv)
        cout << *iv << endl;
}
```

Un autre point qui est désormais possible est d'utiliser les listes d'initialisation.

En effet, pour reprendre l'exemple des vector ci contre, initialiser le vector à partir d'un tableau, est assez désagréable.

Sans entrer dans les détails, elles permettent d'initialiser les conteneurs de la STL plus simplement, comme ici.

POUR FINIR

RANGE-BASED FOR

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

int main()
{
    // inutile en C++11
    //double tab[] = {1, 2, 3};
    vector<double> v = {1, 2, 3}; // liste
                                  // d'initialisation
                                // pour les vecteurs
    for (auto iv = v.begin(); iv != v.end(); ++iv)
        cout << *iv << endl;

    for (auto d : v)
        cout << d << endl;
}
```

Une nouvelle syntaxe pour les boucles for a également été introduite.

Elle permet de parcourir de manière plus simple un conteneur.

En voici un exemple ci contre.

Ici, d prend automatiquement le type des éléments du vecteur. A chaque tour de boucle, il prendra successivement les valeurs du conteneur v.

POUR FINIR

RANGE-BASED FOR

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

int main()
{
    // inutile en C++11
    //double tab[] = {1, 2, 3};
    vector<double> v = {1, 2, 3}; // liste
                                  // d'initialisation
                                  // pour les vecteurs
    for (auto iv = v.begin(); iv != v.end(); ++iv)
        cout << *iv << endl;

    for (auto d : v)
        cout << d << endl;
    for (auto i : {1, 2, 3, 5})
        cout << i << endl;
}
```

Cette syntaxe est aussi compatible avec les liste d'initialisation.

POUR FINIR

LAMBDA FONCTIONS

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


int main()
{
    vector<double> v = {1, 2, 3, 5};
    for (auto d : v) cout << d << endl;

    sort(v.begin(), v.end(),
        [](double a, double b){return a > b;});
    for (auto d : v) cout << d << endl;
}
```

On peut aussi maintenant utiliser des fonctions anonymes ou lambda-fonctions.

Elles permettent de définir des fonctions à la volée et de les utiliser directement en paramètre de fonctions.

Ici, par exemple, on se souvient qu'il fallait qu'on définisse une fonction ou un foncteur pour l'utiliser avec l'algorithme sort de la STL. Maintenant, on définit directement la fonction de comparaison lors de l'appel de l'algorithme.

POUR FINIR

LAMBDA FONCTIONS

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


int main()
{
    vector<double> v = {1, 2, 3, 5};
    for (auto d : v) cout << d << endl;

    sort(v.begin(), v.end(),
        [](double a, double b){return a > b;});
    for (auto d : v) cout << d << endl;
}
```

La syntaxe est la suivante :

```
[ liste de capture ] ( paramètres ) retour { code }
```

avec :

- liste de capture : liste de variables déclarées hors de la lambda et qui seront accessibles dans la lambda ;
- paramètres (optionnel) : paramètres qui seront envoyés par l'algorithme ;
- retour (optionnel) : type retourné par la lambda ;
- code : corps de la fonction lambda.

POUR FINIR

LAMBDA FONCTIONS

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


int main()
{
    vector<double> v = {1, 2, 3, 5};
    for (auto d : v) cout << d << endl;

    sort(v.begin(), v.end(),
        [](double a, double b){return a > b;});
    for_each(v.begin(), v.end(),
                [](auto d){cout << d <<endl;});

}
```

On peut également utiliser auto dans le type de la fonction lambda.

Ici, un autre exemple, avec l'algorithme for_each.

FIN !
MERCI D'AVOIR
SUIVI CE COURS