PROGRAMMATION ORIENTÉE OBJETS C++

MASTER 1

ROLAND RUELLE

ANNÉE 2018/2019

UN MODULE, DEUX LANGAGES

- 7 COURS ET 7 TPS DE C++
- 5 COURS ET 5 TPS DE PYTHON
- → 12 SEMAINES DE PROGRAMMATION ORIENTÉE OBJETS

PRÉSENTATION DU MODULE

- 7 SEMAINES : 7 COURS / 7 TPS
- PLAN
- VALIDATION DU MODULE
- OBJECTIFS
- BIBLIOGRAPHIE

CONTACT C++

ROLAND RUELLE

AU LJAD:
BUREAU 612 SUR RENDEZ-VOUS

PAR MAIL :
RRUELLE@UNICE.FR
ÉCRIRE AVEC LE PRÉFIXE [M1 POO] DANS L'OBJET DU MAIL

SUPPORTS DE COURS :

HTTPS://MATH.UNICE.FR/~RRUELLE
PLATEFORME JALON/MOODLE DE L'UNS (À SUIVRE …)

CONTACT PYTHON

Master 1 - POO / C++

GILLES SCARELLA

PAR MAIL :
AVEC [M1 POO] DANS L'OBJET DU MAIL
GILLES.SCARELLA@UNICE.FR

SUPPORTS DE COURS :
PLATEFORME MOODLE DE L'UNS

PLAN DES COURS C++

Master 1 - POO / C++

- 7 cours sur le semestre (1h30 / cours)
- 7 Tps de 2h30
- 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
  - Entrées/sorties
  - Standard Template Library (STL)

OBJECTIFS EN C++

- 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 EN C++

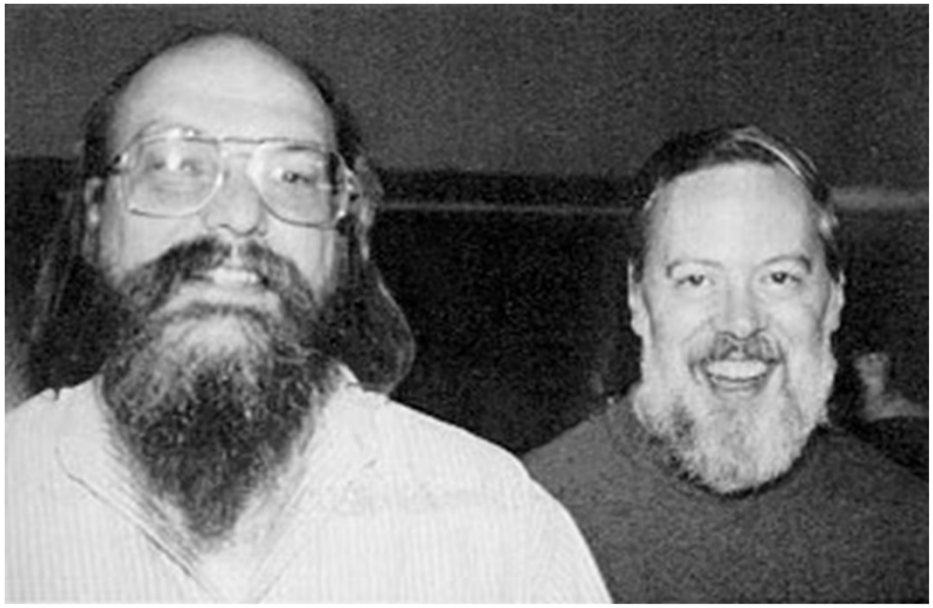
- **APPRENDRE LE C++** DE CLAUDE DELANNOY (SUR LEQUEL S'APPUIE EN PARTIE CE COURS)
- **POUR LES PURISTES : LE LANGAGE C++** DE BJARNE STROUSTRUP
- **POUR LES CURIEUX : LE LANGAGE C. NORME ANSI** DE BRIAN-W KERNIGHAN, DENIS-M RITCHIE
- LE SITE DE RÉFÉRENCE : HTTP://WWW.CPLUSPLUS.COM/

VALIDATION

- POUR VALIDER LE COURS :
- CONTRÔLE DES CONNAISSANCES :
- DEUX TPS NOTÉS (C++) : (½ + ½) * 0,6 DE LA NOTE FINALE
- DEUX TPS NOTÉS (PYTHON) : (½ + ½) * 0,4 DE LA NOTE FINALE

INTRODUCTION

PETITE HISTOIRE DU C/C++

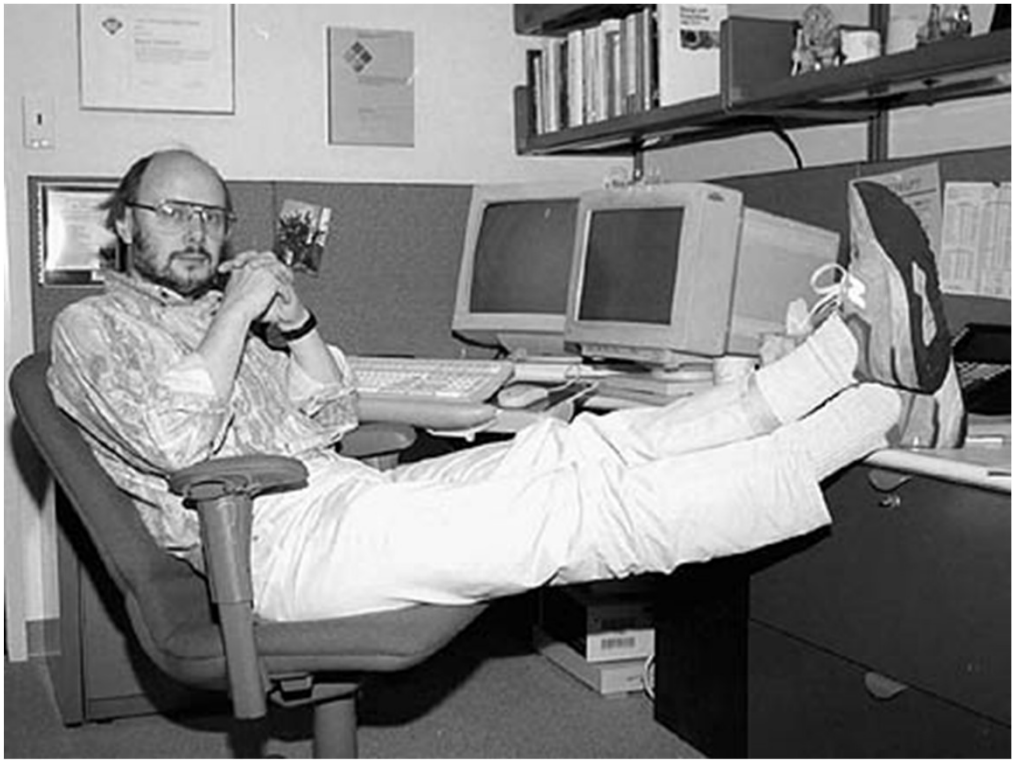


Ken Thompson (à gauche) et Dennis Ritchie (à droite).
(source Wikipédia)

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.

Impératif : les instructions se suivent dans un ordre précis et transmis au processeur de la machine dans cet ordre.

Impératif et objet ne se contredisent pas, C++ est un langage multi-paradigmes. Il respecte à la fois le paradigme objet et impératif.

On va donc commencer par faire du C++ impératif.

HELLOWORLD.CPP

```
#include <iostream>

using namespace std;

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

Comme dans la plupart des cours de programmation, on commence par un HelloWorld : Il s'agit d'un programme très simple qui affiche un message à l'écran.

Dans un éditeur de texte quelconque, on écrira donc le texte ci-contre.

(notez ici la coloration syntaxique qui permet de lire plus facilement le code)

COMPILATION ET EXÉCUTION DU HELLOWORLD

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello World !" << endl;
}
```

On « compile » ensuite le fichier HelloWorld.cpp pour créer un exécutable.

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

LE HELLOWORLD LIGNE À LIGNE

```
#include <iostream>

using namespace std;


int main()
{
    cout << "Hello World !" << endl;
}
```

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

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

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello World !" << endl;
}
```

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

Cette ligne sera considérée comme « magique » au début de ce cours.

La notion d'espace de noms ne sera expliquée que beaucoup plus tard.

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

LE HELLOWORLD LIGNE À LIGNE

```
#include <iostream>


using namespace std;


int main()
{
    cout << "Hello World !" << endl;
}
```

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

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

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

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

LE HELLOWORLD LIGNE À LIGNE

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello World !" << endl;
}
```

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

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

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

LE HELLOWORLD LIGNE À LIGNE

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello World !" << endl;
}
```

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

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

LE HELLOWORLD LIGNE À LIGNE

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello World !" << endl;
}
```

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

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 ;**

`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.

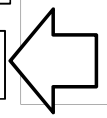
« `Hello World` » est une chaîne de caractère, c'est-à-dire un emplacement mémoire contigüe contenant un caractère par octet de mémoire, et se terminant conventionnellement par le caractère nul. Nous verrons cela plus en détail lorsque nous reparlerons des types de données.

`endl` demande au flux de passer à la ligne suivante.

COMPILATION ET EXÉCUTION DU HELLOWORLD

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello World !" << endl;
}
```

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



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

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

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

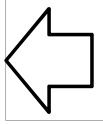
`-o HelloWorld` est une **option transmise au compilateur** lui demandant de créer un fichier exécutable portant ce nom là. Il s'agit d'un argument optionnel. Notre programme se nommerait `a.out` (sous Linux, a.exe sous windows), sans cet argument.

COMPILATION ET EXÉCUTION DU HELLOWORLD

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello World !" << endl;
}
```

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

```
$ ./HelloWorld
Hello World !
```



`./HelloWorld` 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 parler, mais plutôt d'informations qui permettront au compilateur de vérifier la cohérence du code écrit ensuite. Cela peut être assez long, et, souvent le programmeur 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 parler, dans un fichier suffixé en .cpp pour le C++, .c pour le C.

ORGANISATION D'UN PROGRAMME EN C++

```
#include <iostream>
using namespace std;
/*
    Déclaration de la fonction somme
*/
double somme(double a, double b);

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

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

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++

```
#include <iostream>

using namespace std;

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

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

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

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

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

Nous verrons tout cela dans la suite de ce cours.

ORGANISATION D'UN PROGRAMME EN C++

```
#ifndef __SOMME_HH__
#define __SOMME_HH__

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

#endif
```

somme.hh

Un tel fichier header ressemblerait à celui-ci .

On voit apparaitre 3 nouvelles **directives préprocesseurs** ainsi que la déclaration de la fonction somme.

On remarquera aussi que ce fichier NE CONTIENT PAS DE CODE, seulement des déclarations.

ORGANISATION D'UN PROGRAMME EN C++

```
#ifndef __SOMME_HH__
#define __SOMME_HH__

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

#endif
```

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 classes).

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 telle constante, et on déclare notre fonction.

Enfin, on ferme la condition `#ifndef` par `#endif`

DÉCLARATIONS

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

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

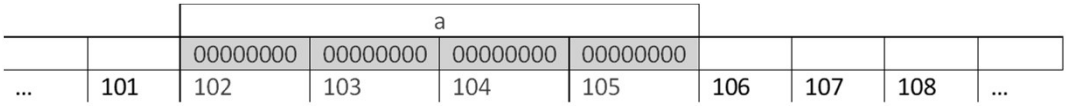
Ci-dessus on peut voir l'exemple de trois déclarations.

VARIABLES

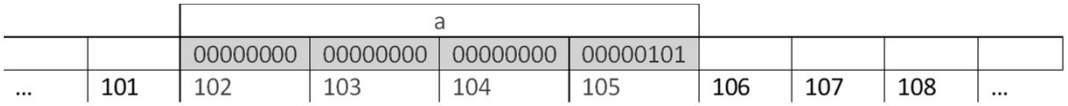
```
#include <iostream>

using namespace std;

int main()
{
  int a;

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

}
```

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

Représentation en mémoire de a = 0



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



TYPES DE DONNÉES

TYPES DE DONNÉES

Le C++ est un langage « fortement typé »

La compilation permet de détecter des erreurs de typage.

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

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

TYPES DE DONNÉES NUMÉRIQUES

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

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

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

TYPES DE DONNÉES NUMÉRIQUES

Les entiers peuvent être déclarés en utilisant les types suivants :

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

TYPES DE DONNÉES NUMÉRIQUES

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

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

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

Et la mémoire coute chère.

TYPES DE DONNÉES NUMÉRIQUES

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

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

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

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

TYPES DE DONNÉES NUMÉRIQUES

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

TYPES DE DONNÉES NUMÉRIQUES

Le type bool permet de représenter les valeurs true et false. C'est-à-dire 0 ou 1.

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

En pratique, le type bool prend autant de place mémoire que le type int.

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 de nombreux opérateurs en C++

Tout d'abord sont définis des opérateurs classiques :

- Arithmétiques
- Relationnels
- logiques

Moins classiques comme les opérateurs de manipulation de bits

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

OPÉRATEURS ARITHMÉTIQUES

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

Addition +, Soustraction -, multiplication * et division /

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

OPÉRATEURS ARITHMÉTIQUES

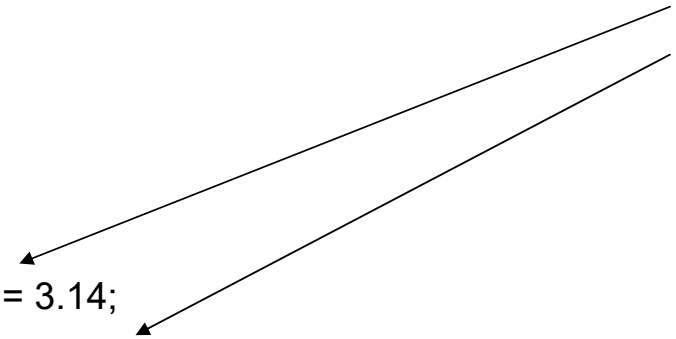
Ces opérateurs binaires ne sont à priori définis que pour des opérandes de même type parmi

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

```
#include <iostream>

using namespace std;

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

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



Ainsi, le programme suivant :

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

Mais, a et b ne sont pas de même type ?

Quel va être alors le type de a+b ? Comme b est un réel; pour ne pas perdre d'information, il faut que a+b soit aussi « grand » que b.

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

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

OPÉRATEURS ARITHMÉTIQUES
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 !

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

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 connaître la taille en mémoire de l'objet passé en paramètre.

Opérateurs de manipulation de bit :

& → ET bit à bit
| → OU bit à bit
^ → OU Exclusif bit à bit
<< → Décalage à gauche
>> → Décalage à droite
~ → Complément à un (bit à bit)

STRUCTURES DE 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 langages 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.

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

Les instructions placées entre l'accolade ouvrante { et fermante } font parties du même bloc d'instructions.

STRUCTURES DE CONTRÔLES

LES CONDITIONS – L'INSTRUCTION « IF »

```
#include <iostream>

int main()
{
  int a;

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

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

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

Sa syntaxe est :

if (expression)

    instruction_1

else

    instruction_2

STRUCTURES DE CONTRÔLES

LES CONDITIONS – L'INSTRUCTION « IF »

```
#include <iostream>


int main()
{
  int a;


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

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

LES CONDITIONS – L'INSTRUCTION « IF »

```
#include <iostream>


int main()
{
  int a;


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

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

Son utilisation est facultative.

STRUCTURES DE 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]
}
```

L'instruction switch 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 testée 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 SWITCH – SYNTAXE

```
#include <iostream>
using namespace std;
int main()
{
  unsigned int a;
  cout << "Valeur de a ?" << endl;
  cin >> a;

  switch (a)
  {
    case 0 :
      cout << "a est nul "   << endl;
      break;
    case 1 :
      cout << "a n'est pas très grand ..." << endl;
      break;
    default:
      cout << "a est un nombre > 1"  << endl;
      break;
  }
}
```

Voici un exemple d'utilisation de l'instruction switch.

Tout d'abord le programme demande à l'utilisateur d'entrer un nombre (entier non signé) au clavier.

Ensuite, en fonction des valeurs de a, un affichage différent est provoqué.

STRUCTURES DE CONTRÔLES

L'INSTRUCTION DO … WHILE

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

L'instruction do … while permet de répéter une ou plusieurs instructions tant que la condition expression est 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
```

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

A noter que :

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

STRUCTURES DE CONTRÔLES

UN EXEMPLE

```
#include <iostream>

using namespace std;

int main()
{
  int a = 0;
  while (a < 10)
  {
    cout << a << endl;
    a = a + 1;
  }
  do
  {
    cout << a << endl;
    a = a - 1;
  }
  while (a > 0);
}
```

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

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

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

Que vaut a lorsque la deuxième boucle se termine ?

STRUCTURES DE CONTRÔLES

L'INSTRUCTION FOR – « BOUCLE AVEC COMPTEUR »

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

L'instruction for 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.

Instruction → il s'agit d'une instruction simple, d'un bloc, d'une structure de contrôle …

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 obtenir ce résultat avec une boucle **while**.

```
$ g++ ex.cpp

$ ./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

SYNTAXE

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

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

Une fonction peut être déclarée dans un premier temps – c'est-à-dire que le compilateur sait que cette fonction existe et qu'il connait les différents types de ses arguments et son type de retour – et être définie dans un second temps : c'est-à-dire l'ensemble des instructions qui vont lui donner un comportement particulier.

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

LES FONCTIONS

SYNTAXE

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

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

LES FONCTIONS

SYNTAXE

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

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

LES FONCTIONS

SYNTAXE

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

Paramètres → Il peut s'agir d'un ou de plusieurs paramètres, séparés par des virgules, et qui doivent être précédés par leurs types respectifs.

LES FONCTIONS

SYNTAXE

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

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

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

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

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

LES FONCTIONS

VOID

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

Il existe un type de retour un peu particulier : void

Ce type signifie que la fonction ne renvoie rien. C'est par exemple le cas si celle-ci doit uniquement provoquer un affichage etc.

Dans ce cas, l'instruction return est autorisée (par exemple pour sortir de la fonction avant d'atteindre la dernière instruction) mais ne doit pas retourner de valeur.

LES FONCTIONS

UN EXEMPLE DE FONCTION : LA FONCTION PUISSANCE

Fonction my_pow

Fonction main

```
#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.

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

LES FONCTIONS

UN EXEMPLE DE FONCTION : LA FONCTION PUISSANCE

Fonction my_pow

Fonction main

```
#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;
}
```

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 noter 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 évidemment 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;
}
```

Quand on tente de compiler ce programme, le compilateur termine en erreur.

Pourquoi ?

A la lecture du message, on comprend qu'on ne fournit pas à la fonction un paramètre du bon type.

En effet, on ne peut pas modifier la constante 2 ou 3 ! Heureusement !

```
$ g++ 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 &);
```

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;
}
```

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.

```
$ 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 &);
```

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 elles peuvent rendre un code compliqué à comprendre et à maintenir.

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

LES FONCTIONS

VARIABLES LOCALES

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

SURCHARGE – 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

EXEMPLE

```
#include <iostream>

using namespace std;

int main()
{
  int t[10];

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

}
```

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 RÈGLES

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 RÈGLES

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

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

```
t[n-3], t[3*p-2*k+j%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 RÈGLES

En C ANSI et en iso C++, la dimension d'un tableau (son nombre d'éléments) ne peut être qu'une **constante**, ou une expression constante. Certains compilateurs 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

```
#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-dessous, ce code ne compile pas. En effet, par défaut, un compilateur fera son possible pour compiler un programme, quitte à ne pas respecter scrupuleusement la norme du langage.

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

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

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

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

TABLEAUX & 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;
}
```

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`.

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

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.

TABLEAUX & POINTEURS

ALLOCATION STATIQUE ET DYNAMIQUE

<table>
<tr><td rowspan="6">MÉMOIRE</td><td rowspan="3">PILE
(stack)</td><td>main</td><td>variables de la fonction main</td></tr>
<tr><td>fct_1</td><td>variables et arguments de la fonction fct_1
appelée dans main</td></tr>
<tr><td>fct_2</td><td>variables et arguments de la fonction fct_2
appelée dans fct_1</td></tr>
<tr><td colspan="3">La pile peut grandir en occupant la mémoire libre</td></tr>
<tr><td colspan="3">mémoire libre</td></tr>
<tr><td colspan="3">Le tas peut grandir en occupant la mémoire libre</td></tr>
<tr><td></td><td>TAS
(heap)</td><td colspan="2">Le tas offre un espace de mémoire dite d'allocation dynamique. C'est un espace mémoire qui est géré par le programmeur, en faisant appel aux opérateurs d'allocation new pour allouer un espace et delete pour 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 dont on a alloué l'espace 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 dû entrer une valeur constante pour sa taille qui aurait pu être soit trop grande, soit trop petite.

On notera

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

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

```
#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);
}
```

POINTEURS SUR FONCTIONS

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.out 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

```
#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;
}
```

```
#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

```
#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;
}
```

```
#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

```
#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;
}
```

```
#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

```
#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;
}
```

```
#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. Ce n'est qu'une déclaration, une possibilité d'existence.

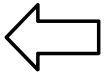
C'est une fois qu'elle a été **instanciée**, c'est-à-dire qu'une variable aura été créé à partir de sa déclaration, que la structure existe vraiment en mémoire pour cette variable.

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 bâtie 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;
}
```

Tableau de 10 char

On l'initialise ici

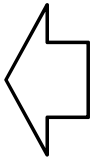
Une structure peut contenir un tableau. La taille de celui-ci sera réservée en mémoire quand une variable issue de cette structure sera créée.

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

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

```
#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;
}
```

STRUCTURES IMBRIQUÉES

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

```
#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;
}
```

STRUCTURES ET FONCTIONS

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 déclare 3 fonctions membres de la structure

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 définit 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 crée 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

```
#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

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

On inclut 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.

```
$ g++ ex_struct_fct_membres.cpp Point2D.cpp

$ ./a.exe
x : 1.1 y : 0.9
```

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

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

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

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

Point2D 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
```

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 dès le début de la vie de notre instance.

```
#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;
}
```

CLASSES & OBJETS

CONSTRUCTEURS

```
#ifndef __POINT2D_HH__
#define __POINT2D_HH__

class Point2D
{
private:
  double x;
  double y;

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

};
#endif
```

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 foulée le constructeur avec les paramètres 3 et 4.

```
#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;
}
```

CLASSES & OBJETS

CONSTRUCTEURS

```
#ifndef __POINT2D_HH__
#define __POINT2D_HH__

class Point2D
{
private:
  double x;
  double y;

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

};
#endif
```

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.

```
#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;
}
```

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 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

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

```
#ifndef __CERCLE_HH__
#define __CERCLE_HH__

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

#endif
```

```
#include "Cercle.hh"

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

Soit le code ci contre :

On déclare deux classes. Une classe Point ayant deux données membres et deux constructeurs.

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

```
$ ./a.out
Constructeur sans arguments de Point
```

CLASSES & OBJETS

OBJETS MEMBRES

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

```
#ifndef __CERCLE_HH__
#define __CERCLE_HH__

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

#endif
```

```
#include "Cercle.hh"

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

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

```
$ ./a.out
Constructeur sans arguments de Point
```

CLASSES & OBJETS

OBJETS MEMBRES

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

```
#ifndef __CERCLE_HH__
#define __CERCLE_HH__

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

#endif
```

```
#include "Cercle.hh"

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

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

Ici, on remarque d'ailleurs que le constructeur par défaut de Cercle fait appelle au constructeur sans argument de Point.

Pourquoi ?

```
$ ./a.out
Constructeur sans arguments de Point
```

CLASSES & OBJETS

OBJETS MEMBRES

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

```
#ifndef __CERCLE_HH__
#define __CERCLE_HH__

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

#endif
```

```
#include "Cercle.hh"

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

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

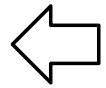
```
$ ./a.out
Constructeur sans arguments de Point
```

CLASSES & OBJETS

OBJETS MEMBRES

```
#ifndef __CERCLE_HH__
#define __CERCLE_HH__

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

#endif
```

```
#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;
}
Point::Point()
{
  cout << "Constructeur sans"
  << " argument de Point" << endl;
}
Cercle::Cercle(double x, double y, double r) :
  _centre(x, y)
{
  _rayon = r;
  cout << "Constructeur de Cercle" << endl;
}
```



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

On modifie le code :

On ajoute un constructeur à la classe Cercle qui possède 3 paramètres . On va passer 2 de ces paramètres au constructeur de la classe Point.

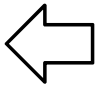
```
#include "Cercle.hh"

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

CLASSES & OBJETS

OBJETS MEMBRES

```
#ifndef __CERCLE_HH__
#define __CERCLE_HH__

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

#endif
```

```
#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;
}
Point::Point()
{
  cout << "Constructeur sans"
  << " argument de Point" << endl;
}
Cercle::Cercle(double x, double y, double r) :
  _centre(x, y)
{
  _rayon = r;
  cout << "Constructeur de Cercle" << endl;
}
```



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

On va utiliser une syntaxe particulière :

Entre l'entête de la fonction et son corps, on insère

« : » et l'la création de la donnée membre : _centre(x, y)

```
#include "Cercle.hh"

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

CLASSES & OBJETS

```
#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;
}
```

Déclaration

Initialisation de la donnée static

MEMBRES STATIQUES

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.

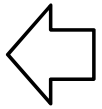
Et la valeur de cette donnée est partagée par toutes les instances ainsi que par la classe elle-même.

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

CLASSES & OBJETS

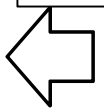
MEMBRES STATIQUES

```
#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;
}
```



Déclaration

Initialisation de la donnée static



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

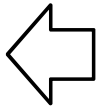
Celle-ci ne peut pas être initialisée par un constructeur : en effet celui-ci est intimement lié au cycle de vie d'un objet et donc à une instance elle-même.

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.

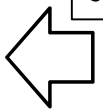
CLASSES & OBJETS

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;
}
```



Déclaration



Initialisation de la donnée static

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;
}
```

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.

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

CLASSES & OBJETS

ACCESSEURS & MUTATEURS

```
#ifndef __O_HH__
#define __O_HH__

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

#endif
```

```
#include "Objet.hh"

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

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

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

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

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

```
$ g++ main.cpp Objet.cpp

$ ./a.out
4
```

CLASSES & OBJETS

ACCESSEURS & MUTATEURS

```
#ifndef __O_HH__
#define __O_HH__

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

#endif
```

```
#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

```
#ifndef __O_HH__
#define __O_HH__

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

#endif
```

```
#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

```
#ifndef __O_HH__
#define __O_HH__

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

#endif
```

```
#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

```
#ifndef __O_HH__
#define __O_HH__

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

#endif
```

```
#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.

Nous verrons cela plus en détail quand nous parlerons des flots.

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.

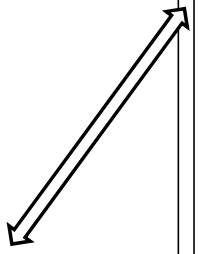
Ainsi, même si nous les redéfinissons, l'opérateur * de la multiplication sera plus prioritaire que l'addition +.

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

SURCHARGE D'OPERATEURS

UN EXEMPLE

```
#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();
}
```

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

```
#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();
}
```

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 cette 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-constructor

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
```

REMARQUE : 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

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
```

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 affichages définis dans le corps de la fonction.


```
#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 <<

```
#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

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, 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

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

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

LES PATRONS DE FONCTIONS

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 définit 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 template 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 classes que nous déclarons d'habitude, ici tout est dans le même fichier !

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

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 déclaré qu'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

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;
}
```

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
```

On utilise la surcharge de << ici

```
#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

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 les notions de type expression, ni la spécialisation de classe template, ou la spécialisation partielle.

Nous ne parlerons pas non plus de meta-programmation ou de template récursifs.

Bien qu'utiles et intéressantes, ces notions sortent de l'objectif de ce cours qui est une introduction au C++et à la Poo.

HÉRITAGE

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 définit 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 changer la valeur de la chaine _nom 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&);
};
```

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

```
#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

```
#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 "Rond.hh"

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

Rond.cpp

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

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

Carre.cpp

On déclare de même une classe Carre, héritant également de la classe Forme.

Ces deux classes ont chacune leurs spécificités, comme on peut le voir, le rond ayant un 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.");
}
```

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

```
#ifndef __FORME_HH__
#define __FORME_HH__

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

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

```
#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 "Rond.hh"

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

Rond.cpp

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

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

Carre.cpp

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

Forme.cpp

On définit une fonction affiche dans la classe Forme.

Celle-ci se contente d'afficher le nom de la forme.

Cette fonction est alors utilisable par toutes les classes filles, qui afficheront 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;
}
```

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();
}
```

Forme.cpp

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

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

L'HÉRITAGE

REDÉFINITION D'UNE FONCTION HÉRITÉE

```
#ifndef __FORME_HH__
#define __FORME_HH__

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

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

#endif
```

Forme.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.hh

```
#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;
}
```

Rond.cpp

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

Carre.hh

```
#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;
}
```

Carre.cpp

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

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

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

C'est pas très orienté objet ca …

L'HÉRITAGE

REDÉFINITION D'UNE FONCTION HÉRITÉE

```
#ifndef __FORME_HH__
#define __FORME_HH__

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

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

#endif
```

Forme.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.hh

```
#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 <<
}
```

Rond.cpp

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

Carre.hh

```
#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;
}
```

Carre.cpp

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

```
#ifndef __FORME_HH__
#define __FORME_HH__

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

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

#endif
```

Forme.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.hh

```
#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;
}
```

Rond.cpp

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

Carre.hh

```
#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;
}
```

Carre.cpp

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 !

```
$ ./a.out
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
```

L'HÉRITAGE

REDÉFINITION D'UNE FONCTION HÉRITÉE

```
#ifndef __FORME_HH__
#define __FORME_HH__

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

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

#endif
```

Forme.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.hh

```
#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;
}
```

Rond.cpp

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

Carre.hh

```
#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;
}
```

Carre.cpp

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();

```
$ ./a.out
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
```

L'HÉRITAGE

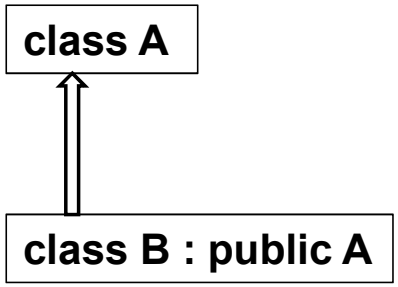
SURCHARGE ET REDÉFINITION

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.

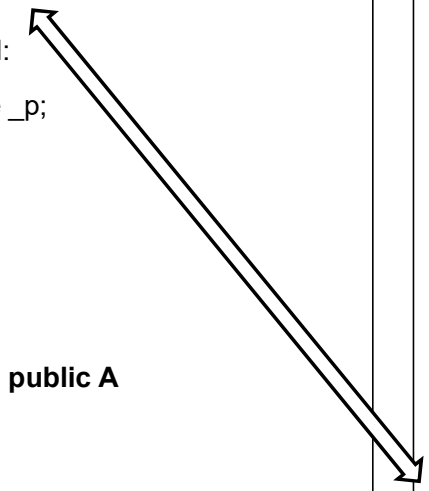
Il va permettre à un développeur qui souhaite étendre les fonctionnalités d'une classe d'avoir plus de pouvoirs qu'un utilisateur extérieur de la classe.

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 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

```
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 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

```
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 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.out
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.out
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.

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.out
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
```

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

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.out
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
```

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

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.

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.out
A: 2
```

Dans cet exemple, nous rappelons le problème.

Dans le main, on créée 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.out
A: 2
B: 3
```

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

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

Que s'est il passé ?

POLYMORPHISME

FONCTIONS VIRTUELLES

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

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

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.out
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.out
Des B
Des A
```

On déclare maintenant le destructeur de A comme étant virtuel.

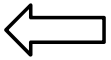
Lors de l'exécution, c'est donc bien le destructeur de B qui est appelé.

POLYMORPHISME

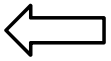
CLASSES ABSTRAITES – FONCTIONS VIRTUELLES PURES

```
#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 d'hériter de celle-ci en créant le code supplémentaire si besoin.

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

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;
               ^
```

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 fonction des besoins.

LA STL

LES ITERATEURS

Pour homogénéiser les actions possibles sur les différents conteneurs, il est apparu la notion d'itérateurs. Un itérateur, qui peut être vu comme une généralisation de la notion de pointeur, permet de parcourir un conteneur.

Il possède ces propriétés :

- À chaque instant, un itérateur pointe vers une valeur qui désigne un élément donné du conteneur.
- Un itérateur peut être incrémenté par l'opérateur ++.
- Il peut être déréférencé, c'est-à-dire que l'utilisation de l'opérateur * permet (comme sur un pointeur) d'accéder à la valeur courante de l'itérateur.
- Deux itérateurs sur un même conteneur peuvent être comparés.

LA STL

ITERATEUR SUR VECTEUR

```
#include <vector>
#include <iostream>
int main()
{
  std::vector<double> v(5);
  std::vector<double>::iterator iv;
  double i = 0;
  for (iv = v.begin();
        iv != v.end();
        ++iv)
  {
    *iv = i++;
  }
  for (iv = v.begin();
        iv != v.end();
        ++iv)
  {
    std::cout << *iv << ";";
  }
}
```

Un iterateur (sur vecteur) 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++.

En effet, 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++;
  }
  for (iv = v.begin();
       iv != v.end();
       ++iv)
  {
    std::cout << *iv << ";";
  }
}
```

Pour accéder à la valeur pointée par l'itérateur, on le déréférence grâce à l'opérateur « * ».

On peut ainsi modifier ou accéder à cette valeur.

Pour les vecteurs, il existe aussi une surcharge de l'opérateur « [] » permettant d'accéder immédiatement à un élément donné.

Cela 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

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.out
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.out
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.out
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.out
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.out 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.out
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.out
 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.out
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. Celui-ci est automatiquement ajouté, comme cela est mentionné dans la documentation.

http://www.cplusplus.com/reference/istream/istream/operator-free/

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;
}
```

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();
}
```

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).

```
$ ./a.out
test
test 5
tom;tom
tom 4
```

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();
}
```

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.

```
$ ./a.out
test
test 5
tom;tom
tom 4
```

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();
}
```

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'.

```
$ ./a.out
test
test 5
tom;tom
tom 4
```

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

```
#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.out
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.out
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 du 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.out
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.out
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 :

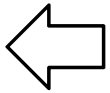
Lorsqu'on ouvre un fichier en écriture, par défaut, si ce fichier existe, **son contenu est écrasé**.

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.

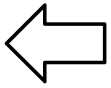
Pour cela on peut utiliser la fonction membre flush() qui va vider les tampons dans le fichier.

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.

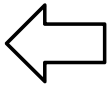
On utilisera la fonction membre close() sur le flot.

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**.

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()
   {
      // contructeur 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 : c'est la même syntaxe que pour l'espace de nom 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.
   }
}
```

```
#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"
using namespace MyName;

class A
{
public:
   double x;
};

int main()
{
   fct();

   B b;
   A a;
}
```

Pour utiliser l'espace de nom que nous avons créé, nous pouvons utiliser la directive using (comme ce fût le cas pour std)

```
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;
}
```

Mais que se passe-t-il lorsqu'un symbole porte déjà le même nom qu'un symbole de notre namespace ?

Erreur de compilation !

```
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;
}
```

Il faut alors utiliser l'opérateur de résolution de porté « :: » pour lever l'ambiguité.

Ici, ::A sans espace de nom avant ::, permet d'utiliser « l'espace de nom global »

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.

FIN !
MERCI D'AVOIR SUIVI CE COURS