La programmation scientifique en **C++**

Christophe Labourdette

Octobre 2014

1 La programmation

Avant de rentrer dans les détails de syntaxe du langage **C++**, arrêtons nous pour nous questionner sur la programmation, de quoi s'agit-il au juste ?
Ajourd'hui on dit plus volontier, "développement logiciel" que programmation, pourtant au fond il s'agit bien de la même chose : donner à l'ordinateur des instructions pour lui permettre d'exécuter un algorithme.
Pour cela, il faut choisir un langage de programmation, dans lequel on va transcrire l'algorithme. La programmation est donc trés liée à l'algorithmique. Il existe de nombreux langages de programmation, mais même s'ils sont parfois trés différents, le principe reste le même, le langage fournit :

- des structures de données simples,
- des itérateurs,
- des tests,
- des opérateurs d'entrées/sorties,
- la possibilité de construire des types de données complexes.

Au fil du temps, les gens construisent des bibliothèques qui permettent d'enrichir le langage notamment en ajoutant des traitement pour des données plus spécialisées et plus complexes.
Par exemple, il est nécessaire d'utiliser des bibliothèques graphiques, lorsque l'on veut dessiner ou afficher des résultats, de même, il est trés utile de disposer de bibliothèques numériques, qui implémentent de manière efficace des routines d'algèbre linéaire.
Le langage **C++** est un langage plutôt générique, orienté objet qui permet de construire et manipuler des structures de données abstraites et complexes.

2 Quelques règles

Le **C++** est un langage fortement typé, il faut définir de manière assez précise, les objets (variables) sur lesquels on va effectuer des opérations. Il s'agit également d'un langage compilé, qui nécessite que le code source soit transformé par un compilateur (puis par un éditeur de liens) en un objet exécutable. La compilation se déroule en deux phases, dans un premier temps il faut *compiler* les différents fichiers qui constituent le programme, puis effectuer une édition de liens qui va rassembler les divers éléments compilés ainsi que les bibliothèques pour constituer un fichier exécutable.

2.1 Compilation séparée

Les grosses applications comportent souvent des dizaines ou des centaines de milliers de ligne de code. Il est impossible de manipuler des fichiers qui contiennent trop de lignes. Une règle élémentaire de programmation est donc de ne traiter que des petits fichiers. On découpe pour cela le programme en petites entités (fonctions, classes, etc). Mais on se retrouve alors trés vite avec des quantités de fichiers, il est nécessaire de bien organiser son code, parfois de définir une nomenclature et des règles de nommages des fichiers. La compilation n'est du coup pas facile et l'on utilise des outils comme Make, pour automatiser le processus de compilation. La séparation des entités du programme pose aussi le problème du partage de définitions de certains objets ou des variables. On utilise pour cela des fichiers dits à entète, qui ne contiennent théoriquement que des définitions

et qui permettent de partager entre les différents fichiers certains informations. Cette manière de travailler est en accord avec les recommandations données par le génie logiciel. Les fichiers de définitions sont appelés "interfaces".

2.2 Les mots réservés

Chaque langage possède un certain nombre de mots clés, ceux-ci seront interprétés par le compilateur comme élément du langage. Voici un tableau désignant les mots reservés par le **C++** :

TABLE 1 – Les mots clés du **C++**

and	and_eq	asm	auto	bitand	bitor
bool	break	case	catch	char	class
compl	const	const_cast	continue	default	delete
do	double	dynamic_cast	else	enum	explicit
export	extern	false	float	for	friend
goto	if	inline	int	long	mutable
namespace	new	not	not_eq	operator	or
or_eq	private	protected	public	register	reinterpret_cast
return	short	signed	sizeof	static	static_cast
struct	switch	template	this	throw	true
try	typedef	typeid	typename	union	unsigned
using	virtual	void	volatile	wchar_t	while
xor	xor_eq				

Comme tout langage, le **C++** va manipuler des expressions comportant des variables. Pour nommer les variables ou les types créés par l'utilisateur, on utilise des **identificateurs**. Un **identificateur** est constitué d'une séquence de lettres et de chiffres, il doit nécessairement commencer par une lettre. L'alphabet admissible comprend l'ensemble des lettres de l'alphabet (minuscule et majuscule) ainsi que le caractère souligné. Il n'y a pas, théoriquement, de limite au nombre de caractères d'un nom. Voici quelques exemples d'**identificateurs** :

```
A1
b
B                    // cet identificateur est different du precedent
a23
alamaison
une_phrase
Vu
NombreIterations
point_de_vue
```

ATTENTION il ne peut pas y avoir de blanc dans un **identificateur**.
ATTENTION les majuscules sont différentes des minuscules.

3 Les types intégraux

Il apparait tout à fait naturel que nous commencions notre revue de détails des éléments du langage par ce qui est, réellement, pour le programmeur, "le nerf de la guerre" : les données. Comme tous les langages de programmation, le **C++**, comprend un certain nombre de types de base, sur lesquels nous allons pouvoir effectuer des opérations. Ce sont ces entités, qui vont permettre de préciser le stockage des données ainsi que les méthodes utilisés lors d'opérations.
Le **C++** est un langage fortement typé dans lequel il est nécessaire de déclarer toute variable, afin que son type soit fixé. Il existe, lorsque cela est possible des mécanismes de conversion entre les différents types, leur

utilisation peut être implicite ou bien forcé (nous verrons les conversions plus en détail un peu plus loin). Les types booléens, caractères et entiers sont dit de type **intégral**.

3.1 Les booléens

Un booléen ne peut avoir comme valeur que *true* (vrai) ou *false* (faux). Après la conversion d'un booléen en entiers *true* correspond a la valeur 1 et *false* à la valeur 0. Inversement l'entier 0, sera converti en *true* et toute valeur non nulle sera convertie en *false*.
Dans les expressions arithmétiques et logiques, les booléens sont convertis en entiers pour les opérations, puis le résultat est reconvertis ensuite. Voici quelques exemples de déclarations et d'expressions :

```
bool b; //une declaration de la variable b toute simple
bool a = true; // la declaration de la variable a avec une
               // initialisation a true
bool a2 = false; // une declaration de a2 en l'initialisant a false
bool a3 = 4; // une declaration de a3 en l'initialisant a false
```

Le plus naturel avec les booléens, consiste à effectuer des opérations logiques, en voici les règles :

A	B	A ou B	A et B	Non A
0	0	0	0	1
0	1	1	0	1
1	0	1	0	0
1	1	1	1	0

3.2 Les caractères

Une variable de type **char** est un caractère, elle est généralement stockée sur 8 bits. Il existe de trés nombreux jeux de caractères spécifiques à la langue considérée. En général pour la langue Française, on utilise le jeu iso-8859-15. Depuis quelques temps, le jeu UTF-8 pour les caractères unicode est normalisé et il est utilisé de plus en plus souvent. Chaque constante de type caractère, possède une valeur entière. On peut obtenir la valeur entière du caractère 'c' avec la commande **int**(c).
REMARQUE, nous venons d'effectuer à l'aide de la commande **int**(c), notre première conversion, d'un caractère vers un entier.
ATTENTION, il existe deux façon d'implémenter les caractères. Avec des **unsigned char** qui fournira des entiers de -127 à 127 ou bien avec des **unsigned char** qui fournira des valeurs de 0 à 255. Il existe également un type **wchar_t** qui permet de stocker des jeux de caractères plus étendu comme Unicode, il sera donc stocké sur 16 bits et non plus 8.

```
char c;  // la declaration du caractere c
char ch='j'; // la declaration et l'initialisation du
             // caractere ch e   j

int i = int(ch); // la declaration d'un entier j, initialise avec le
                 // code du caractere ch pr cedemment defini
wchar_t uu; // declaration d'un caractere du jeu etendu
```

Comme le type caractère est un type intégral, il est possible de lui appliquer des opérations logiques ou arithmétiques.

3.3 Les entiers

Comme le type **char**, le type entier dispose de trois formes :
int, **signed int** et **unsigned int**.
Mais il peut également disposer de trois tailles :
short int, **int** et **long int**.

On peut en fait utiliser des raccourcis :
short pour **short int**, **long** pour **long int**, **signed** pour **signed int** et **unsigned** pour **unsigned int**.
ATTENTION, en général les trois tailles ne sont pas réellement utilisées, on a **short** = **int** ou bien **long** = **int**.
ATTENTION, selon les processeurs les implémentations des short, int et long peuvent être différentes !.
Voici quelques exemples :

```
int i;              // un entier
short ii;           // un short int
long jj;            // un long int
unsigned kk;        // un unsigned int
```

3.3.1 Les valeurs

Il existe plusieurs façons d'affecter une valeur à un entier, celle-ci peut être, caractère, décimale, octale ou hexadécimale.
Bien entendu les décimales sont les plus courantes, mais il n'est pas rare tout de même de rencontrer des valeurs octales (en base 8), elles commençent par un 0 ou hexadécimales (en base 16), elles commençent par un x ou 0x.
Les lettres a,b,c,d,e,f ou A,B,C,D,E,F représentent respectivement les valeurs 10,11,12,13,14,15.
Les représentations octales ou hexadécimales sont utilisées quelques fois pour des applications liées à l'administration du système ou pour des manipulation de champs de bit.
On peut également utiliser le suffixe U pour déclarer explicitement un "unsigned".
Le suffixe L permet de déclarer un type long int.
Lorsque l'utilisateur déclare une valeur trop longue pour être représentée, le compilateur affichera un message d'erreur.

```
int i1 = 2343677; // une valeur decimale
int i2 = 066;     // la valeur decimale 54 declaree en octal
int i3 = 0x4ef3;  // la valeur decimale 20211 declaree en
                  // hexadecimal
int i4 = 111111111U; // un unsigned
int i5 = 787878787L; // un long
```

3.4 Les flottants

Les types flottants représentent les nombres en virgule flottante. Il s'agit de représenter les nombres Réels des Mathématique, donc chaque fois qu'un calcul ne peut se satisfaire d'un opérande ou d'un résultat entier. Le système est normalisé et suit les règles IEEE. Il existe trois tailles :

- **float** simple précision,
- **double** double précision,
- **long double** précision étendue.

La représentation choisie est celle d'une mantisse accompagnée d'un exposant, autrement dit on manipule des nombres de la forme :

- le signe $s \in \{0, 1\}$
- l'exposant e est un entier
- la mantisse m est un réel positif du type

$$m = d_1 10^{-1} + d_2 10^{-2} + \cdots + d_p 10^{-p}$$

Le nombre réel est donné par $x = (-1)^s 10^e m$
Cette représentation à "virgule flottante" est ce que les fabricants de calculatrice appellent la "notation scientifique".

3.4.1 Les valeurs

les déclarations peuvent se faire de plusieurs façons, mais par défaut une valeur est un **double**, si l'on veut utiliser un **float** il faut utiliser le suffixe f :

```
3.6    .9808    3.    7.11e21    9.5e-11    -3.21 // quelques valeurs litterales
65.8f    32.767767F                              // pour des float
```

Si l'utilisateur entre une valeur trop grande pour pouvoir être représentée, le compilateur affichera un message d'erreur.
Voici quelques exemples de déclarations de flottants :

```
float          f1;                    // un float f1
float          f2 = 3.71;             // un float f2 initialise
double         x = 3.71;              // un double x initialise
double         y = 3.5e-4;            // y est un double
long double    z = .0;                // z est un long double initialise    0
```

3.5 Le type void

Nous l'avons dit à plusieurs reprises, il est nécessaire de déclarer tous les types de variable. Cela est parfois trés contraignant et pour pallier à cette rigidité, le langage **C++** possède un type "non-typé", le type **void**.
De la même façon que l'utilisation des pointeurs permet de contourner la réservation de la place mémoire pour les variables, le type **void** permet de déclarer une variable ou un pointeur non-typé mais qui pourra l'être par la suite à l'aide d'une conversion de type.
Il est également trés utile pour certaines fonctions.

```
 void x;                  // on definit une variable void
(int) x =  233;           // on type la variable pour lui donner une valeur
```

Il est cependant plus utilisé pour des fonctions que pour des variables.
Nous reviendrons sur son utilisation plus précisément un peu plus tard.

4 Quelques propriétés

4.1 Les déclarations

Pour qu'un identificateur soit utilisé dans un programme, il faut le déclarer. Le type de celui-ci doit donc être spécifié afin que le compilateur réserve la place mémoire nécessaire à son stockage. Contrairement au langage **C**, le **C++** permet de déclarer, dans tout le corps du programme et non pas seulement au début. Mais il est souvent plus lisible de déclarer les variables importantes au début du fichier et de ne disséminer dans le corps du programme que des variables temporaires.

4.1.1 extern

Le mot clé **extern** permet de déclarer une variable sans la définir, un petit peu comme un prototype pour les fonctions. Elle n'entraine pas d'allocation de mémoire. Comme il est d'usage lorsque l'on écrit des applications en **C++** de pratiquer la compilation séparée : les sources de l'applications se trouvent dans de nombreux fichiers différents et le compilateur examine ces fichiers les uns aprés les autres, il faut donc lui donner des indications lorsque une variable est utilisée dans un fichier autre que celui où elle est déclarée. Le fait de définir une variable comme **extern** permet de dire au compilateur qu'elle est définie par ailleurs.

4.1.2 virtual, static, register

Ces instructions ne sont pas utiles pour débuter, elles concernent plutôt l'optimisation. **static** demande au compilateur de garder une entité en activité, autrement dit de ne pas effacer les données la concernant, même lorsque l'on sort de la portée, car elle va être réutilisée.

register demande au compilateur d'utiliser, si possible un registre du processeur pour cette variable qui va être massivement utilisée.

4.1.3 portée

La portée est un concept trés important, qu'il est indispensable de comprendre avant de se lancer dans l'écriture d'applications complexes. Une variable, a un durée de vie qui dépend trés fortement de l'endroit où elle est déclarée. C'est cette durée de vie que l'on appelle la portée. Autrement dit, la portée est la partie du programme où l'on peut employer le nom déclaré.
Il est possible d'avoir des variables *globales* dont la portée est le programme dans son entier. mais la majorité des portées sont locales, depuis la déclaration, jusqu'a la fin du bloc dans lequel elle se trouve. Un bloc est encadré par des accolades { }.
ATTENTION, une déclaration peut en masquer une autre, par défaut la variable sera toujours considérée comme la plus locale.
On peut cependant enlever le masquage pour accéder à une variable globale en utilisant les : : comme par exemple,

```
int      i;                          // le i global

float calcul()
{
 int     i = 0;                      // ce i local masque le i global
 i = 1;                              // c'est le i local qui est affecte

   {
        int     i = 2;               // masque le i local par un autre
        i = i + 1;                   // le i le plus local prend la valeur 3
        ::i += 8;                    // le i global est incremente de 8
   }
}

i += 5;                              // le i global est incremente de  la valeur 5
```

4.2 Les constantes

Il est possible en **C++**, de définir des constantes gràce au mot clé **const**, après sa définition, il sera impossible de modifier la valeur de la constante. A part son affectation, la constante pourra être utilisée comme une variable habituelle.

```
const int taille = 32;      // on definit taille comme un entier
                            // constant valant 32
const float pi = 3.1415;    // pi est un float constant valant 3.14.15

int tableau[taille];        // la constante taille peut servir a
                            // dimensionner le tableau

taille = 64;                // ERREUR cette operation est impossible
```

Une constante doit toujours être initialisée.

5 Les types collections

Les types collections sont souvent construits explicitement ou implicitement à partir des types intégraux, mais peuvent l'être aussi à partir de types collection.

5.1 Les énumérations

Il s'agit d'un type dans lequel on peut stocker des valeurs spécifiées par l'utilisateur. L'utilisation est trés proche de celle des entiers. Il est assez peu utilisé en pratique. Chaque énumération représente un type distinct, par exemple si l'on définit :

```
enum truc {ZERO, UN, DEUX, TROIS};
```

l'expression précédente instancie en fait quatre constantes auxquelles on a associe des valeurs croissante en partant de 0, comme ci-dessous,

ZERO == 0, UN == 1, DEUX == 2, TROIS == 3

Il est possible d'initialiser un énumérateur à l'aide d'une expression-constante d'un type intégral. L'intervalle de l'énumération est un peu plus compliqué à déterminer, il contient toutes les valeurs de l'énumération arrondies à la puissance binaire immédiatement supérieure moins 1. Si le plus petit énumérateur est positif, il part de 0. Si le plus petit énumérateur est négatif, il part de la puissance binaire négative la plus proche.

```
enum p1{z,u};                    // intervalle 0:1
enum p2{a=7, b=12};              // intervalle 0:15
enum p3{u1=-4, u2=129};          // intervalle -256:255
```

Les énumérations sont converties par défaut en entiers pour les opérations arithmétiques.

5.2 Les tableaux

Le tableau de type T, T[n] est une collection de n éléments de type T. Ses éléments sont indicés de 0 à n−1. **Remarque**, En Mathématique, le ou les objets les plus proches des tableaux sont les vecteurs et les matrices, en langage **C**, ils sont souvent fondamentaux dans tous les algorithmes de calcul. En **C++** on utilise plus volontier l'encapsulation fournie par la librairie standard le type **vector**.

```
int ti[3];       // un tableau de 3 entiers,
                 // ti[0], ti[1], ti[2]
char *a[2];      // un tableau de 2 pointeurs de type caracteres
                 // a[0], a[1]
double x[50];    // un tableau x de 50 flottants de type double
                 // x[0], ..., x[49]
```

Il est bien entendu possible d'initialiser les tableaux, en utilisant des accolades.

```
int tii[5]={89,45,2,9,12};   // un tableau de 5 entiers
char ta[]={'a','e','i'};     // un tableau de 3 caracteres
```

Il n'est pas nécessaire d'initialiser toutes les valeurs, dans le cas où le nombre d'initialiseur est trop faible on suppose que les valeurs restantes sont nulles.

```
int tgi[8]={3,8,9};          // equivalent a
                             // int tgi[8]={3,8,9,0,0,0,0,0};
```

Rien n'empêche bien entendu de construire des tableaux à plusieurs dimensions, mais il faut faire attention leur maniement n'est pas aussi facile que losqu'il s'agit d'un vecteur.

```
int tabi[2][2];              // tabi est donc un tableau de tableau
                             // on peut faire une analogie avec une
                             // matrice 2x2 dont les elements seront
                             // tabi[0][0], tabi[0][1], tabi[1][0],
                             // tabi[1][1]
```

L'initialisation des tableaux multidimensionnel se fait de la même façon en plaçant les valeurs dans l'ordre lexicographique.

```
int A[3][3]={1,2,3,3,1,2,2,3,1}; // le tableau A initialise comme suit
                                 // A[0][0]=1 ,A[0][1]=2, A[0][2]=3
                                 // A[1][0]=3, A[1][1]=1, A[1][2]=2
                                 // A[2][0]=2, A[2][1]=3, A[2][2]=1
```

On peut également pour être plus clair initialiser ligne par ligne.

```
int A[3][3]={{1,2,3},
             {3,1,2},
             {2,3,1}}; // le tableau A est initialise ligne par ligne
```

Attention, il n'est pas possible d'utiliser l'affectation de tableau au cours du code, celle-ci n'est valable que pour l'initialisation. Autrement dit on ne peut pas écrire :

```
int U[4];
U={1,2,3,4};        // ERREUR
```

6 Les pointeurs

La nécessité de manipuler la mémoire plus ou moins directement est apparue de manière trés claire avec le langage **C**, dont la fonction était avant tout d'écrire un système d'exploitation (Unix). En **C++**, c'est essentiellement pour gérer de manière dynamique les variables et les types et donc prendre d'une certaine façon la main sur le compilateur que l'on utilise un type particulier, le **pointeur**

6.1 Accéder à la mémoire

Pour un type donné T, on appelle T* le type pointeur de T. Comme son nom l'indique, il va s'agir de adresse mémoire contenant la variable du type T. Nous sommes bien en présence d'un **pointeur** vers une variable.

```
int i = 6;              // soit i une variable de type int
int *pi = &i;           // pi est un pointeur qui contient
                        // l'adresse de i
int j = *pi;            // l'entier j est initialise avec la valeur 6

char lettre = 'l';      // le caractere lettre contient l
char *pl = &lettre;     // le pointeur pl contient l'adresse de lettre
char ll = (*pl) + 1;    // le caractere ll est initialise avec le
                        // caractere m
```

Rien n'empêche évidemment de construire des pointeurs de pointeurs et nous verrons même dans la suite que cela peut être trés utile, mais il faut faire attention la manipulation des pointeurs demande une certaine habitude.

```
int   **pi;             // pi est un pointeur de pointeur
```

Le pointeur, contient donc une adresse mémoire, par son intermédiaire il est donc possible d'accéder directement ou de modifier la valeur de la variable pointée.
L'opérateur fondamental du type pointeur est l'indirection, c'est l'opérateur * , il permet de faire référence à l'objet pointé.
Il est possible d'effectuer des opérations arithmétique avec les pointeurs, nous verrons cela un peu plus loin.

6.2 Les références

Le type référence, utilise l'opérateur d'adressage &, après le type référent, il désigne en quelque sorte un alias. Comme les constantes, les références doivent obligatoirement être initialisées.

```
float joe = 75.5;        // une variable float initialisee     75.5

,float &poids = joe;      // poids est une reference a joe
poids++;                  // joe = joe + 1 = 76.5
float truc;               // une variable float
truc = poids;             // truc = 76.5
int i = 10;               //

int &ref;                 // ERREUR il faut initialiser ref
```

Une référence ne peut être initialisée qu'avec une "lvalue" du même type.

6.3 Les chaînes de caractères

Un chaîne littérale est une suite de caractères encadrée par des doubles guillemets :

```
''Bienvenue au pays des chaines''
```

La chaîne en question est composée de 30 caractères, car elle se termine par le caractère nul '\0'. Le type de la chaîne est donc un tableau const de char de la bonne taille : ''Coucou'' est donc de type **const char**[7]; Anciennement en C et en C++ le type de la chaîne de caractère était **char** $*$, il était alors possible de modifier la chaîne par l'intermédiaire du pointeur. ATTENTION ce n'est plus possible !
Il existe des caractères que l'on ne peut pas écrire, comme par exemple l'apostrophe ou le guillemet, on utilise des séquences particulières pour les utiliser.

newline	\n
horizontal tab	\t
vertical tab	\v
backspace	\b
carriage return	\r
formfeed	\f
alert(bell)	\a
backslash	\\
question mark	\?
single quote	\'
double quote	\''

Il est également possible lorsque l'on connait la valeur numérique en octal d'utiliser une séquence du type \ooo, où ooo est la valeur en notation octale du caractère désiré. Par exemple en ASCII on a :

```
\7(bell)
\0 (null)
\12 (newline)
```

6.4 Un rapport étroit entre pointeur et tableau

De fait dans la machine les données des tableaux sont plaçées de manière contigüe, il est donc possible d'utiliser des pointeurs pour parcourir les tableaux. L'adresse d'un tableau est l'adresse de la première "case" de ce tableau. L'identificateur lui même représente cette adresse. On peut alors mélanger pointeurs et tableaux.

```
char tab[] = ''Heureusement'';  // l'identificateur du tableau, tab
                                // renvoie l'adresse du premier element
                                // autrement dit l'adresse du tableau
                                //
                                // tab == &tab[0] == &tab;
char *pt = tab;                 // le pointeur sur un caractere pt est
                                // initialise avec l'adresse du tableau
```

Lorsqu'un pointeur pointe sur une l'adresse d'un tableau on peut l'utiliser pour parcourir ce dernier. Par exemple $*(pt+5)$ est équivalent à tab[5].
ATTENTION le type du pointeur et du tableau doivent être le même, il est bien entendu impossible de parcourir un tableau de caractère avec un pointeur de type float.

6.5 Allocation mémoire

Mais si les tableaux sont en réalité des pointeurs, en quoi sont-ils différents ? La réponse est d'importance :

- les tableaux impliquent par leurs déclarations une allocation mémoire de la part du compilateur qui permet de contenir le nombre spécifié d'éléments.
- La déclaration d'un pointeur implique l'allocation par le compilateur d'une adresse mémoire qui n'est bien souvent même pas suffisante pour stocker un seul élément du type spécifié. Tout se passe bien si ce pointeur ne sert que de façon "passive", à parcourir la mémoire, mais si l'on commençe à modifier les valeurs de ces adresses, il est parfaitement possible que la zone mémoire affectée ne soient pas allouée à ce programme.

Il est donc nécessaire lorsque l'on veut utiliser les pointeurs pour instancier des variables ne disposant pas encore d'une allocation, de demander, par l'intermédiaire de la commande **new** une allocation mémoire, aussi bien pour une variable simple que pour un tableau.

```
float *xx = new float;          // le pointeur sur un float xx est
                                // initialise avec une adresse demandee au
                                // systeme, c'est l'equivalent de la
                                //  creation d'une variable
float *yy = new float[10]; // le pointeur sur un float yy est
                                // inititalise avec l'adresse d'une zone
                                // memoire contenant au moins 10 float
                                // c'est l'equivalent de la creation d'une
                                // variable de type tableau de float de
                                // taille 10, float yy[10];

int *pi = new int[8];           // ici le pointeur sur un entier est
                                // initialise avec une adresse contenant au
                                // moins 8 entiers equivalent a la
                                // declaration du tableau int pi[8];
```

ATTENTION, la déclaration d'un tableau, permet au compilateur de prévoir la place nécessaire lors de la phase de compilation, dans le cas d'allocation de pointeur il s'agit, d'allocation dynamique qui se déroule au cours de l'exécution du programme.
Pour cette raison, il existe une instruction, **delete**, permettant de libérer la place réservée pour le pointeur. Il est trés important de libérer la place mémoire qui n'est plus utilisée. Par exemple, pour libérer les variables précédemment déclarées :

```
delete xx;
delete[] yy;
delete[] pi;
```

ATTENTION, la création d'un objet à l'aide de **new** est indépendant de la portée !
La place mémoire allouée n'est rendue que par la commande **delete** ou bien par la fin du programme.

7 Les structures

Alors que les tableaux permettent d'agréger des éléments de même type les structures permettent elles, d'agglomérer des éléments de types arbitraires. Ce type existait déjà en langage **C**. Nous verrons dans la suite qu'il s'agit en **C++** d'une **classe** un peu particulière. Dans ce chapître nous nous attacherons essentiellement aux aspects présent dans le langage **C**.

7.1 Des objets multiformes

A l'aide du mot clé **struct** on construit un objet multiforme composé d'un agrégat d'éléments :

```
struct eleve{
    char *nom;
    char *prenom;
    float   notes_maths[15];
    float   notes_physique[15];
};
```

La structure définit donc un élève comme l'agglomération d'un nom, d'un prénom ainsi que des tableaux de notes en maths et en physique. On va pouvoir ensuite se servir du type eleve pour créer des variables. ATTENTION, dans l'exemple, il n'y a pas eu de mémoire allouée pour les chaînes de caractères, cela doit être fait !

```
eleve joe = {                   // on cree une variable joe du type
                                // eleve
             ''Calliari'',      // on initialise le nom
             ''Joe'',           // le prenom
             {12.5,11.3},       // les deux premieres notes de maths
             {14.0,13.7}        // les deux premieres notes de physique
}

eleve arthur;                   // declaration d'une variable arthur
                                // de type eleve
arthur.nom=''Fulbert'';         // instanciation du champ nom de
                                // arthur
arthur.notes_maths[1] = 9.6;    // la deuxieme note de maths est 9.6
```

Il est courant de manipuler les pointeurs sur les structures. On utilise alors l'opérateur d'adressage indirect -> plutèt que (*pointeur). , par exemple :

```
struct mensuration{                 // on declare une structure
    float taille;                   // mensuration
    float poids;
    int pointure;
};

mensuration  *tom = new mensuration;  // on alloue le pointeur tom
tom->taille = 1.83;             // utilisation de l'operateur
tom->poids = 88.5;               // -> pour fixer les valeurs
(*tom).pointure = 45;           // (*tom). equivalent a tom->
```

Il n'est pas possible de définir de façon récursive un objet du type de la structure avant que la déclaration soit finie, car le compilateur ne connait pas encore la taille nécessaire à la structure, mais cela est possible avec des pointeurs :

```
struct liste{               // une liste chainee comportant un
  liste* precedent;         // pointeur vers le precedent et
  liste* suivant;           // un pointeur vers le suivant
};

struct aie {
  aie essai;                // ERREUR, le compilateur ne connait
};                          // pas la taille de aie !
```

Lorsque deux structures se font références, il faut déclarer un nom comme étant du type structure, avant les déclarations :

```
struct liste;                  // on declare liste comme etant de type
                               // structure, la vraie declaration se
                               // fera plus tard
struct feuille{                // la structure feuille
  feuille* avant;              //
  feuille* apres;              //
  liste* membre;               // contient un pointeur sur une structure liste
};

struct liste{                  // vraie declaration de la structure liste
  feuille* racine;             // contient un pointeur sur une structure
};                             // feuille
```

En fait l'objet **struct** est une forme simplifiée de l'objet de type **class**, nous verrons tout ceci plus en détail plus tard(13).

7.2 Les unions

Une **union** est une instance spéciale d'une **classe**. La mémoire alloué à l'union correspond à l'espace nécessaire pour sa donnée la plus large. Il s'agit en quelques sortes d'une **struct** dans laquelle tous les membres commençent à la même adresse.

```
union value{           // declaration d'une union Value
  char cvalue;         //
  char* pvalue;        //
  int ivalue;          //
  double dvalue;       //
};
```

value définie précedemment va occuper en mémoire, la place d'un double. On ne peut affecter une valeur qu'a un seul membre à la fois. L'utilisation des unions est assez complexes et nous laisserons les lecteurs intéressés chercher ailleurs des exemples sophistiqués.

7.3 Les champs de bit

Difficile de parler réellement d'un type particulier pour les champs de bit, il s'agit surtout de la possibilité de préciser le nombre de bits que l'on utilise pour le stockage d'une variable.
Cette possibilité est utilisé lorsque l'on veut contrôler, surtout à bas niveau la composition d'un *mot* machine (en général 32 ou 64 bits). Il est donc possible d'associer plusieurs variables dans une classe ou une structure, sous la forme de champs en précisant le nombre de bit occupé pour chaque membre :

```
stuct AddIP{                       // declaration d'une struct decrivant
                                   // une adresse IP
  unsigned int champ1 : 8;         // le premier champ comprend 8 bits
  unsigned int champ2 : 8;         // le deuxieme champ comprend 8 bits
  unsigned int champ3 : 8;         // le troisieme champ comprend 8 bits
  unsigned int chmap4 : 8;         // le quatrieme champ comprend 8 bits
};
```

8 Opérations et instructions

Après avoir passé en revue les types de base du C++, il faut maintenant comprendre comment construire des expressions pour manipuler des données.

8.1 Une expression

Une *expression* comprend une ou plusieurs *opérations*.Les *opérations* sont représentées par des *opérateurs* qui agissent sur des *opérandes*. Les *opérateurs* qui s'appliquent sur un seul opérande sont dénommés *unaires*, ceux qui mettent en oeuvre deux *opérandes* sont dit *binaires*, on distingue alors *l'opérande gauche* de *l'opérande droit*.
a + b représente l'opérateur binaire d'addition, il effectue l'addition de l'opérande gauche, a, et de l'opérande droit, b.
∗pointeur repésente l'opérateur unaire de déréférence il retourne la valeur stockée à l'adresse du pointeur.
En général une expression nécessite une ou plusieurs opération et renvoie une *rvalue*. Le type du résultat est déduit du type de opérandes. Dans les cas compliqués où les types sont trés différents, il y a de nombreuses conversions réalisées. Pour déterminer l'ordre des opérations lorsqu'il y en a plusieurs, il existe des lois *d'associativité* et de *priorité*.

8.2 L'arithmétique

Pour effectuer des opérations d'arithmétique, on utilise les opérateurs les plus classiques :

Fonction	Opérateur	Utilisation
multiplication	$*$	expr $*$ expr
division	/	expr / expr
modulo (reste de la division)	%	expr % expr
addition	$+$	expr $+$ expr
soustraction	$-$	expr $-$ expr
négatif	$-$	$-$ expr

ATTENTION, dans le cas où les deux opérandes sont des entiers le résultat de la division est entier. Il faut utiliser l'opération modulo pour obtenir le reste de la division. Les opérations entières se font en règles générales de façon modulaire, comme sur un cercle. Lorsque le résultat de l'opération est plus grand que le plus grand entier (respectivement plus petit), il utilise les plus petits (respectivement les plus grands entiers) par exemple :

```
const int plus_grand_entier;
const int plus_petit_entier;

int i = plus_grand_entier;
int j = plus_petit_entier;

i+1 == plus_petit_entier;
j-1 == plus_grand_entier;
```

8.3 Logique et comparaison

Les opérateurs logiques ont pour valeur, vrai ou faux. la valeur vrai vaut 1, alors que faux vaut 0.
- L'opérateur ET logique && est vrai, si les deux opérandes sont vrais.
- L'opérateur OR logique || est vrai, si l'un des opérandes est vrai.

On traite toujours les opérandes de la gauche vers la droite, on cesse d'évaluer dés que la valeur est établie. Il n'est donc pas toujours nécessaire d'évaluer toutes les expressions. Par exemple, dans l'expression expr1 && expr2 on n'évaluera pas expr2 si expr1 est faux.

Fonction	Opérateur	Utilisation
Négation logique	!	!expr
inférieur à	<	expr < expr
inférieur ou égal	<=	expr <= expr
supérieur à	>	expr > expr
supérieur ou égal	>=	expr >= expr
égalité	==	expr == expr
inégalité	!=	expr != expr
et logique	&&	expr && expr
ou logique	||	expr || expr

8.4 L'affectation

L'opérande gauche de l'opérateur d'affectation ("=") doit être ce que l'on appelle une lvalue. L'affectation a pour effet de placer une nouvelle valeur dans la mémoire associée à cette lvalue, cette valeur est le résultat de l'expression située à droite. Le type résultat est également celui de l'opérande gauche.

```
int i = 34 + 4 - 1;
int *pi =    ;
```

Il est possible de concaténer plusieurs opérateurs d'affectation si les opérandes sont du même type.

```
float x1, x2, x3;
x1 = x2 = x3 = 0.0;
```

Il existe également une notion d'opérateur composé, on peut écrire : a op= b; en utilisant comme opérateur op=, +=, −=, *=, /=, %=, <<=, >>=, &=, ^=, |=, qui s'interprète comme a = a op b.

8.5 Incrémentation et décrémentation

Il est possible d'utiliser des opérateurs, d'incrémentation ++ et de décrémentation −−, pour ajouter ou soustraire 1 à une variable. Il s'agit d'un opérateur d'affectation, qui peut être utilisée comme *préfixe* ou comme *suffixe*.
ATTENTION le résultat peut être extrèmement différents dans les deux cas.

- la forme préfixée ++i incrémente la variable i avant qu'elle ne soit utilisée dans l'expression.
- la forme suffixée i++ n'incrémentera la variable i qu'après le calcul de l'expression.

```
int i,j, k;
i=0;
j=1;
k=i++;                    // k=0 puis i=i+1
k=++j;              // j=j+1 et k=2
```

8.6 Un opérateur ternaire

Il existe un opérateur **if** ternaire, expr1 ? expr2 : expr3; . Si expr1 est vrai (une valeur non nulle quelconque) alors expr2 est évaluée dans le cas contraire c'est expr3 qui est évalué.

```
int a,b,c;
a=2;
b=7;
c = (a>b) ? a : b;        // c=7 car b>a
```

8.7 La virgule

Lorsque des expressions sont séparée par des virgules elles sont examinées de la gauche vers la droite et le résultat de l'expression est donnée par la valeur la plus à droite.

```
int a,b,c;
a = (b=3, b+2);          // d'abord b=3 puis a=3+2 donc a=5
```

8.8 Manipulation de bits

Il est possible de manipuler directement les bits a l'aide d'opérateurs particuliers, les opérandes sont obligatoirement entiers (ou char, bien entendu). On dispose des fonctions suivantes :

Fonction	Opérateur	Utilisation
NOT	$\sim$	$\sim$expr
décalage gauche	<<	expr1 << expr2
décalage droit	>>	expr1 >> expr2
AND	&	expr1 & expr2
XOR	$\wedge$	expr1 $\wedge$ expr2
OR	|	expr1 | expr2

Il est préférable, d'utiliser des opérandes non signé, car le traitement du bit de signe dépend des implémentations.
La négation remplace les 0 par des 1 et les 1 par des 0 :

```
unsigned char octet = 0257;     // 1 0 1 0 1 1 1 1
octet = ~octet;                 // 0 1 0 1 0 0 0 0
```

Les opérations de décalage déplacent les bits de l'opérande gauche d'un certain nombre de position soit vers la gauche, soit vers la droite. Les bits insérés sont des 0.

```
unsigned char octet = 1;        // 0 0 0 0 0 0 0 1
octet <<= 3;                    // 0 0 0 0 1 0 0 0
octet >>=  1;                   // 0 0 0 0 0 1 0 0
```

Le & prend deux opérandes entiers et pour chaque bit renvoie 1 si les deux bits sont à 1 dans les deux opérandes et 0 sinon.
Le OR, | renvoie 1, si l'un des deux bits est à 1 et 0 sinon.
Le XOR, $\wedge$ renvoie 1, si l'un des deux bits est à 1 mais pas les deux et 0 sinon.

```
unsigned char octet1 = 0257;    // 1 0 1 0 1 1 1 1
unsigned char octet2 = 0121;    // 0 1 0 1 0 0 0 1

unsigned char octet_and;
unsigned char octet_or;
unsigned char octet_xor;
octet_and = octet1 & octet2;        // 0 0 0 0 0 0 0 1
octet_or = octet1 | octet2;         // 1 1 1 1 1 1 1 1
octet_xor = octet1 ^ octet2;        // 1 1 1 1 1 1 1 0
```

8.9 Des priorités

Lorsque les expressions sont composées de nombreux opérateurs, il est important de connaître l'ordre dans lequel les opérations vont s'effectuer. Il existe des règles de priorité et d'associativité entre les opérateurs qui rendent déterministe le calcul d'une expression. Voici un tableau indiquant les ordres de priorité et d'associativité associé à chaque opérateur.

Niveau	Associativité	Opérateur	Fonction
17	gauche à droite	::	portée globale (unaire)
17	droite à gauche	::	portée de classe (binaire)
16	droite à gauche	−>,.	sélecteur de membre
16	droite à gauche	[]	indice de tableau
16	droite à gauche	()	appel de fonction
16	droite à gauche	()	construction de type
15	gauche à droite	**sizeof**	taille en octets
15	gauche à droite	++,−−	incrémentation,décrémentation
15	gauche à droite	∼	opérateur de bit NOT
15	gauche à droite	!	opérateur logique NOT
15	gauche à droite	+,−	plus,moins (unaire)
15	gauche à droite	∗,\&	déréférence, adressage
15	gauche à droite	()	coercition de type
15	gauche à droite	**new**,**delete**	gestion mémoire
14	droite à gauche	−>∗,.∗	sélecteur de pointeurs membre
13	droite à gauche	∗,/,%	opérateurs multiplicatifs
12	droite à gauche	+,−	opérateurs arithmétiques
11	droite à gauche	<<,>>	opérateurs de décalage de bit
10	droite à gauche	<,<=,>,>=	opérateur de relation
9	droite à gauche	==,=!	égalité,inégalité
8	droite à gauche	&	opérateur de bit AND
7	droite à gauche	∼	opérateur de bit XOR
6	droite à gauche	|	opérateur de bit OR
5	droite à gauche	&&	opérateur logique AND
4	droite à gauche	||	opérateur logique OR
3	droite à gauche	?:	if arithmétique
2	gauche à droite	=,∗=,/=	opérateurs d'affectation
2	gauche à droite	%=,+=,−=,<<=	opérateurs d'affectation
2	gauche à droite	>>=,&=,|=,^=	opérateurs d'affectation
1	droite à gauche	,	opérateur virgule

8.10 Les conversions

Nous avons déjà abordé ce problème précedemment, il s'agit d'un point important, la conversion permet de passer d'un type à un autre lorsque cela est possible. Elle peut également typer une variable.

```
int i;
float f = 3.14;
i = (int) f;              // le float f (3.14) est converti en entier (3) donc i=3
```

9 Les tests

Comme dans la plupart des langages, il est possible de tester une valeur (résultatnt d'une instruction à l'aide des commandes **if** ou **switch**.

9.1 if

Il existe deux façons d'utiliser le **if**, simplement avec une seule condition ou bien avec plusieurs cas possibles :

- **if** (condition) instruction1
- **if** (condition) instruction1 **else** instruction2

Bien entendu les instructions et les conditions peuvent être composées. instruction1 est exécutée si la condition est non nulle, si ce n'est pas le cas et si il y a un **else** alors instruction2 est exécutée. En particulier toute expression arithmétique ou de pointeur peut être utilisée comme condition.

```
float x,y,xx,yy;
float *px, *py;
.
.
.
if (x<0)
{
  x-=6;
  xx *= .6;
}
else
{
  x += 7.1;
  xx *= 3.2;
}
.
.
.

if (px) py = px;
```

Donc si j est un entier,

```
if (j) // ...
```

est la même chose que

```
if (j != 0) // ...
```

Lorsque les conditions sont composées, en général à l'aide de && ou || l'ordre des conditions est important puisque l'on a vu que les conditions ne sont évaluées que si nécessaire. Pour optimiser il est donc important de réfléchir, de façon à évaluer le moins de conditions possible.
Attention, lorsque l'on veut limiter l'utilisation (et donc la portée), de variables servant uniquement pour un test, il est possible de les déclarer dans les conditions. Dans ce cas la portée de la variable est exactement le test (autrement dit jusqu'à la fin du else).

```
double yy;
double zz;

.
.
.

if (double xx = premier(z)) {    // xx est declare puis initialise
  yy = xx + 1.0;                 // Si xx est different de 0, on
  zz = xx * xx;                  // calcule yy et zz
}
else                             // si xx est nul
{
  yy = yy + 1.0;                 // on calcule yy et zz      partir
  zz = (zz + 1.0) * zz;          // d'une autre valeur mais
                                 // xx existe toujours
```

```
}
```

9.2 switch

La fonction switch permet de remplacer un ensemble d'instructions, **if**

```
int truc;              // une variable entiere
float bidule;          // une variable float

.
.
.

switch(truc) {
case 1:                // si (truc == 1) alors
  bidule = 1241.7;     // bidule = 1241.7
  break;
  case 2:              // si (truc == 2) alors
  bidule = 56.4;       // bidule = 56.4
  break;
case 3:                // si (truc == 3) alors
  bidule = 89.65;      // bidule = 89.65
  break
default:               // sinon
  bidule = 1.0;        // bidule = 1.0
  break;
}
```

est équivalent (mais beaucoup plus explicite et lisible que :

```
int truc;              // une variable entiere
float bidule;          // une variable float

.
.
.
if (truc == 1) bidule = 1241.7;
  else
    if (truc == 2) bidule = 56.4;
      else
        if (truc == 3) bidule = 89.65;
          else
             bidule = 1.0;
```

10 Itérations

Il existe trois forme itératives de boucle, elles utilisent les mots clés, **for**, **while**, **do**.

10.1 for

Il s'agit de la boucle la plus simple et donc la plus utilisée, le principe en est simple, il faut disposer d'une valeur de début d'une condition de fin ainsi que d'une fonction d'incrémentation :

```
for (initialisation;expression1;expression2)
  expression ;
```

L'initialisation peut être à la fois une déclaration et une expression, elle sert en général à initialiser le compteur de boucle. Traditionnelement expression1 est une règle d'arrêt et expression2 le passage à l'itération suivante :

```
for (int i=0; i<64; i++)     // l'initialisation declare et initialise
                             // le compteur de boucle i a 0
                             // la boucle s'arretera quand l'expression1
                             // ne sera plus vraie  ==> i>= 64
                             // on passe a l'iteration suivante en
                             // incrementant i  de 1
{
  x += i;
}
```

Il est parfaitement licite de ne pas mettre d'expression, on peut par exemple faire une boucle infinie :

```
for (;;)
{
  // ...
}
```

Il faudra pour en sortir utiliser explicitement l'une des instructions **break**, **return**, **goto**, **throw** ou exit()

10.2 while

Comme son nom l'indique on teste ici l'exécution d'une boucle sur une condition de fin, *tant que* la condition est vraie on boucle :

```
while (condition)
  expression;
```

On exécute l'expression tant que la condition est vraie :

```
int   i=0;
int   j=0;
.
.
.
while (i<7){
  j = 3 * ++i;
}
```

Va donner

$$j = 1*3 + 2*3 + 3*3 + 4*3 + 5*3 + 6*3 + 7*3 = \sum_{i=1}^{i=7} 3*i = 69$$

10.3 do

La boucle faire, tant que est utile lorsque l'on ne connait pas la contion au début de la boucle. Cette est la plus dangereuse car elle prête souvent à confusion, pour que la condition soit évaluée, il faut que le corps de la boucle soit exécuté au moins une fois :

```
do
  expression;
while (condition);
```

Elle est à éviter au début il vaut mieux utiliser les deux autres (**for** et **while**).

10.4 break

L'instruction **break** sort immédiatement de la boucle ou du test courant. Elle permet de sortir, aussi bien des boucles que des tests, l'exécution reprend immédiatement après la fin de la boucle ou du test.

```
int i;
int j;

.
.
.

for (;;)
{
  i+=4;                    // on incremente i de 4
  j++;                     // on incremente j de 1
  if (i>j) break;          // si i est plus grand que j on sort de la
                           // boucle
}
```

10.5 continue

Continue permet de passer à l'itération suivante sans se préoccuper des instructions restantes dans la boucle. Il peut s'agir d'une boucle **for**, **while** ou **do**

```
int i;
double x, xx, y, yy;
.
.
.
for (i=0;i<MAX; i++)       // une boucle de 0 a MAX avec un
                           // increment de 1
{
  x += i * x;              // on incremente x
  xx -= 2.5 *y;            // on decremente xx
  if (i > INTERMEDIAIRE)   // si i depasse la valeur INTERMEDIAIRE
    continue;              // on passe a l'iteration suivante
  y += i;                  // increment de y
  yy = yy * xx *(y - 1);   // calcul de yy
}
```

Dans l'exemple précédents les incréments de y et le calcul de yy ne se font que jusqu'à ce que i dépasse INTERMEDIAIRE.

10.6 goto

En **C++** il est possible d'utiliser **goto** pour se brancher sur un label donné quelque part dans le code. Cela permet de quitter plusieurs boucles imbriquées par exemple.

```
goto label;
.
.
.
label: instruction;
```

Cette instruction est rarement utilisée, car elle permet de construire des programmes totalement illisibles. Mais elle peut être trés pratique lorsque le programme est construit automatiquement.

11 Fonctions

La fonction est un peu la cheville ouvrière du C++. Lorsque l'on a besoin d'exécuter une tâche dans un programme on appelle une fonction.

11.1 Déclaration

Comme les variables, toute fonction utilisée dans un programme doit être déclarée et définie (en même temps ou séparément).
La déclaration de la fonction indique :

- le nom de la fonction,
- le type de la valeur de retour,
- le nombre et les types des arguments.

La déclaration simple sans définition est appelée prototype.

```
int pgcd(int i1, int i2);              // une fonction pgcd qui prend
                                       // deux arguments entiers et
                                       // renvoie un entier

int plusgrand (int tableau[]);         // une fonction plusgrand qui prend
                                       // en argument un tableau d'entier
                                       // et renvoie un entier

char* copie(char*, const char*);       // une fonction copie qui prend en
                                       // argument un pointeur sur une
                                       // chaine de caracteres et un
                                       // pointeur constant sur une
                                       // chaine de caracteres et qui
                                       // retourne un pointeur sur une
                                       // chaine de caracteres
```

On voit dans le dernier exemple qu'il n'est pas nécessaire de nommer les arguments, s'il s'agit d'une simple déclaration, les noms des arguments ne font pas partis du type de la fonction.
La définition est une déclaration dans laquelle apparaît le corps de la fonction. Bien entendu la définition et les déclarations d'une fonction doivent spécifier le même type. Lorsque la fonction ne renvoie aucune valeur elle est de type **void**, il s'agit alors d'une **procédure** :

```
void erreur (const char*);     // typiquement une fonction affichant
                               // un message d'erreur et quittant le
                               // programme, elle ne renvoie pas de
                               // valeur. Elle est donc de type void

double val_absolue(double x)   // la fonction valeur absolue retourne
                               // un double et prend un double comme
                               // argument
{
  if (x < 0) return -x;        // si x est negatif on renvoie -x
  else return x;               // sinon on renvoie x
}
```

Il est possible de définir des fonctions récursives (autrement dit une fonction qui s'appelle elle même), voici la fonction factorielle :

```
unsigned long factorielle (int i1)
{
  if (i1 > 1)
```

```
    return i1 * factorielle (i1 -1);

  return i1;

}
```

Il est possible d'utiliser le mot clé **extern** comme pour les variables, si la déclaration est séparée de la définition :

```
extern double* calcul(int, float, double *);
.
.
.
double* calcul(int ii, float xx, double *val)
{
  int imin;
  double x;
  double min = val[1] - val[0];
  for (int i=1;i<ii;i++)
  {
    x = val[i] - val[i-1];
    if (x < min)
    {
      min = x;
      imin = i;
    }
  }
  return &val[imin];
}
```

Comme pour les variables le compilateur vérifie les types des arguments et de la valeur de retour pour chaque appel de fonction.
Attention la valeur de retour ne peut être, ni un tableau, ni une fonction, mais elle peut être un type prédéfini, un type dérivé comme un pointeur ou bien un type défini par l'utilisateur.
Dans la nature, il existe bien entendu des fonctions qui ne sont pas typées, autrement dit elles ne renvoient pas de valeurs. C'est le cas par exemple de fonctions d'initialisations, qui ne font pas de calculs mais instancient simplement des variables. Elles sont appelées **procédures**, c'est l'utilisation du type **void** pour la fonction, qui signifiera qu'elle ne retourne pas de valeurs.

11.2 Le return

L'action de l'instruction est immédiat, il stoppe l'exécution de la fonction en cours, il existe deux façons de l'utiliser :

```
return;                 // dans ce cas la fonction ne renvoie pas de valeur

return expression; // la valeur retour sera celle de l'expression
```

Il n'est pas obligatoire d'utiliser un **return** dans le cas d'une fonction **void**, car l'appel est implicite à la fin des instructions, mais cela peut permettre l'arrêt prématuré de la fonction et améliore la lisibilité du programme.

```
void copie_char(char *u, char *v, int t) // copie de la chaine v dans
                                         // u on suppose qu'elles ont
                                         // la meme taille t
{
  if (u == 0 || v == 0) return;          // dans le cas ou l'un des
                                         // pointeurs est indefini
```

```
                                        // on quitte la fonction

  if (u == v) return;                   // si les deux pointeurs
                                        // sont identiques on quitte

  for (int i=0;i<t;++i)                 // on copie v dans u
    u[i] = v[i];                        // le return est implicite
}
```

11.3 Le passage des arguments

Lors de l'appel d'une fonction, les arguments formels se voient réserver une place mémoire correspondante à la taille définie dans le prototype de la fonction. Les arguments courants initialisent les arguments formels. La fonction n'a ensuite accès qu'aux valeurs qui se sont transmisent par l'initialisation.

11.3.1 Passage par valeur

C'est ce que l'on appelle une transmission par valeur. La conséquence est importante, la fonction n'a jamais accès aux arguments courants mais uniquement à des copies, il lui est donc impossible de modifier ces valeurs. Voici une petite illustration du concept de passage par valeur :

```
void ff(int i, float x)
{
  std::cout << ''Dans ff, i = ''<< i<<std::endl;
  std::cout<<''Dans ff, x = ''<< x<<std::endl;
  i += 21;
  x /= 3.5;
  std::cout<<''Dans ff, i = ''<< i<<std::endl;
  std::cout<<''Dans ff, x = ''<< x<<std::endl;
}
.
.
.

int main()
{

  int i = 3;
  float x = 35.0;
  std::cout<<''i = ''<< i<<std::endl;
  std::cout<<''x = ''<< x<<std::endl;

  ff(i,x);
  std::cout<<''Apr s ff, i = ''<< i<<std::endl;
  std::cout<<''Apr s ff, x = ''<< x<<std::endl;

}
```

l'exécution nous donne :
i = 3
x = 35
Dans ff, i = 3
Dans ff, x = 35

Dans ff, i = 24
Dans ff, x = 10
Après ff, i = 3
Après ff, x = 35
Autrement dit les modifications de la fonction ff n'ont eu aucun effet sur les variables passées en arguments, elles n'ont affectées que des copies temporaires.
Le passage par valeur est insuffisant car il arrive souvent que l'on veuille profiter des modifications où des calculs effectués par les fonctions. Il existe donc des exceptions.

11.3.2 Passage par référence

Le premier est le cas du type référence. En effet déclarer un argument de type référence permet d'éviter le mécanisme de passage des arguments par valeur. Par exemple si l'on modifie quelque peu notre exemple précédent :

```
void ff(int& i, float& x)
{
  std::cout << ''Dans ff, i = ''<< i<<std::endl;
  std::cout<<''Dans ff, x = ''<< x<<std::endl;
  i += 21;
  x /= 3.5;
  std::cout<<''Dans ff, i = ''<< i<<std::endl;
  std::cout<<''Dans ff, x = ''<< x<<std::endl;
}
.
.
.

int main()
{

  int i = 3;
  float x = 35.0;
  std::cout<<''i = ''<< i<<std::endl;
  std::cout<<''x = ''<< x<<std::endl;

  ff(i,x);
  std::cout<<''Apr s ff, i = ''<< i<<std::endl;
  std::cout<<''Apr s ff, x = ''<< x<<std::endl;

}
```

Voici ce que donne l'exécution à présent :
i = 3
x = 35
Dans ff, i = 3
Dans ff, x = 35
Dans ff, i = 24
Dans ff, x = 10
Après ff, i = 24
Après ff, x = 10
Les variables i et x ont bien été modifiées par la fonction ff. Les seules modifications, ont été de déclarer les argument i et x comme références.

Les pointeurs possèdent également les mêmes propriétés que les références de permettre la modification de l'objet pointé puisque ce que l'on manipule, c'est une adresse.
Attention, le passage par référence n'est pas sans danger et doit être réservé aux cas qui le nécessite. Il est possible lorsque l'on veut transmettre de trés grosses structures, mais pas mofifier les données d'utiliser judicieusement le mot clé **const** qui permet, aussi bien pour des références que pour des pointeurs d'empêcher la modification des valeurs.

Les tableaux lorsqu'ils sont donnés comme arguments se comportent comme le pointeur du premier élément. Un tableau ne peut donc pas être transmis par valeur. Un argument du type $T[]$ est donc automatiquement converti en $T*$ lors de la transmission.

11.4 Les surcharges

Il est possible en C++, lorsque les arguments sont de types différents mais que la tâche est la même de donner le même nom aux fonctions. On appelle cela surcharger la fonction, prenons l'exemple d'une fonction *min* qui calcule la plus petite valeur, elle peut s'appliquer à de nombreux arguments différents et ill parait normal de l'appeler toujours de la même façon :

```
int min(int, int);                    // le min entre deux entiers
int min(const int*, int);             // le min sur un tableau
                                      // d'entiers

int mint(const List&);                // le min dans une liste
.
.
.
```

ATTENTION le type de retour ne fait pas partie de la signature de la fonction, par exemple :

```
int min(int, int);
unsigned int min(int, int); // erreur meme signature mais le type de
                            // retour est different
```

verra le compilateur afficher une erreur annonçant la deuxième fonction comme une redéclaration de la première de façon erronée.
Ces règles indiquent bien les conditions pour l'utilisation de la surcharge, il faut s'assurer que les fonctions surchargées effectuent bien une même opération sur des données de types différents. Dans le cas contraire la surcharge va rendre le programme beaucoup plus difficile à relire.

11.5 Inline

Nous n'en avons pas parlé jusqu'à présent mais le mécanisme de mise en place d'une fonction, par le compilateur, comme il doit répondre à tous les types de fonction n'est pas forcément adapté à de toutes petites fonctions trés simples. Il existe donc un mot clé **inline** qui permet de demander au compilateur de s'affranchir du mécanisme standart, complexe, et de placer le code de la fonction directement là ou elle est appelée. Ce mécanisme est utile pour obtenir des codes plus rapide mais uniquement sur des fonctions de type macro, par exemple pour la factorielle :

```
inline int factorielle(int m)
{
  return (n<2)? 1:n*factorielle(n-1);
}
```

Ou bien la valeur absolue :

```
inline int abs(int i)
{
  return (i<0 ? -i : i);
}
```

Remarque, les fonctions inline remplaçent avantageusement les macros fournies au préprocesseur dont étaient coutumiers les utilisateurs du langage **C** mais qui avaient le gros défaut de ne pas avoir de contrôle sur les arguments.

11.6 Liste d'arguments

La liste d'argument est l'élément clé, avec le nom de la fonction, de la signature de la fonction, elle est obligatoire. Une fonction qui ne prend pas d'argument, prend un argument de type **void**. Il est possible d'utiliser ... dans la liste d'argument, obligatoirement à la dernière place, pour signifier que le nombre d'arguments est inconnu, la vérification des types est alors suspendue.

```
void truc(const double*, int, ...);    // La fonction truc prend comme
                                        // arguments, un tableau, un
                                        // entier puis des arguments
                                        // optionnels
void truc(...);                         // On ne connait pas les
                                        // arguments de truc, il peut ne
                                        // pas y en avoir
void truc();                            // truc ne prend pas d'arguments
                                        // il y a un void implicite
                                        // comme pour
void truc(void);                        //
```

Il est également possible dans la liste des arguments de préciser des valeurs par défaut, ce qui permet d'invoquer ensuite la fonction avec ou sans la présence des arguments correspondants. Si l'argument n'est pas précisé, la valeur par défaut est utilisé sinon elle est ignorée.

```
char *initialisation(char *chaine,      // une fonction initialisation
                     int taille= 256);  // qui prend en argument, une
                                        // chaine de caracteres et
                                        // un entier dont la valeur
                                        // est initialisee    256

char *tab;
tab = initialisation(tab);              // equivalent a
                                        // initialisation(tab,256);

tab = initialisation(tab, 512);         // la valeur d'initialisation
                                        // est oubliee
```

Lorsqu'il y a plusieurs arguments initialisé, il faut que ce soit les plus à droite. Car les arguments sont traités de gauche à droite. Il n'est pas possible d'initialiser seulement, un ou plusieurs arguments au milieu de la liste, seuls les plus à droite sont initialisés.

12 Espaces de noms

L'espace de noms est un mécanisme qui permet de regrouper un certain nombre de déclarations dans un espace commun.

```
namespace Optimisation
{
  int min(int, int) {/* ... */}
  int max (int, int){/* ... */}
  int min(const int*, int){/* ... */}
  int max(const int*, int){/* ... */}
}
```

Les fonctions min et max sont regroupées dans l'espace de noms Optimisation, pour les appeler ensuite on utilise l'opérateur de portée ::,

```
int i,j,k;
.
.
.
k = Optimisation::min(i, j);
```

On peut également séparer déclaration et définition dans l'espace de noms de la façon suivante :

```
namespace Outils
{
  int abs(int);
}

Outils::abs(int i) {return (i<0 ? -i : i);}
```

Un espace de noms représente une portée, on peut donc utiliser les règles habituelle, si un nom est déclarée précedemment dans la portée, il peut être utilisé tel quel, si le nom provient d'un autre espace de noms il faudra utiliser l'opérateur de portée(::). Lorsqu'un nom est utilisé fréquemment en dehors de son espace de noms, il est pénible de répéter sans cesse l'opérateur de portée, on utilise alors une directive **using** qui permet d'utiliser les noms de l'espace cité :

```
using namespace Optimisation;   // rend utilisable tous les noms de
                                // Optimisation

i = min(k, l);                  // on peut donc appeler la fonction min
                                // sans preciser son espace de noms

j = Outils::abs(i);             // on utilise l'operateur de portee
                                // pour faire appel a abs au sein de
                                // l'espace de noms Outils
```

Néanmoins ce n'est pas parce que cela n'est pas nécessaire que l'on n'a pas le droit d'utiliser, la portée explicite, souvent cela élimine des ambigüité ou simplifie la lecture du code. Il est possible de rajouter à tout moment des noms dans un espace de nom en rajoutant une instruction **namespace**

```
namespace Truc              // l'espace de noms Truc
{                           // contient les fonctions
  int ff(int);              // ff et fg
  int fg(char);
}
.
.
```

```
.
namespace Truc
{
  float gg(float);          // l'espace de noms Truc
}                           // contient a present egalement la fonction
                            // gg
```

Il est également possible d'utiliser using en sélectionnant certains noms un espace de noms, par exemple

```
using Optimisation::min;    // on peut par la suite utiliser l'un des
                            // noms min, sans utiliser ::
```

Attention comme son nom l'indique, l'espace de noms regroupe des noms, il peut bien entendu s'agir de fonctions mais aussi d'autres objet comme des classes, par exemple.

13 Les Classes

Nous voici à présent dans ce qui est vraisemblablement, le coeur du C++, nous avons vu les types prédéfinis ainsi que les fonctions qui permettent d'agir sur ces données mais la force d'un langage de programmation c'est aussi sa capacité à modéliser les données de l'utilisateur. Les classes permettent de définir de nouveaux types de données abstraits ou concret. On parlera souvent d'objet pour désigner les données ainsi créées. Nous verrons par la suite qu'il existe de fait de nombreux types d'objet, mais nous supposons ici que la classe est utilisée pour représenter un concept.

13.1 Introduction

Prenons un exemple géométrique simple, une classe point qui représente le concept de point dans le plan (un espace à deux dimensions) :

```
class Point
{
  public:
  float x;                  // abscisse
  float y;                  // ordonnee

  Point(){};                  // constructeur par d faut
  Point (float a, float b);   // un constructeur qui initialise les
                              // coordonnees
  ~Point(){};                 // destructeur par defaut
  float Distance(Point X);  // distance a un autre point
};
```

C'est une déclaration des plus simple, Point possède des données membres, ainsi qu'une fonction membre (Distance()). Cette déclaration est en général écrite dans un fichier d'entête, par exemple "Point.hpp" et sert d'interface pour la classe. Les définitions de fonctions membres sont en général décrites au sein du fichier "Point.cpp" qui contiendra dans ses premières lignes l'inclusion du fichier interface :

```
#include ''Point.hpp''
#include <cmath>

Point:Point(float x, float y)
{
  this.x = x;    // pour faire reference a la donnee membre on utilise
                 // le pointeur  this
```

```
  this.y = y;    //
}

float Point::Distance(Point p)
{
  float a, b;
  a = x - p.x;                  // comme il n'y a pas d'ambiguite et que
                                // Distance est une fonction membre
                                // il est inutile d'utiliser this
  b = y - p.y;
  return sqrt(a*a + b*b);
}
```

Pour déclarer des fonctions membres en dehors de la classe on utilise l'opérateur de portée ::, pour faire référence à des données membres de la classe il est souvent utile de le faire explicitement à l'aide de **this** .

13.2 Les constructeurs et les destructeurs

Nous avons vu dans l'exemple précédent deux constructeurs pour la classe Point, le premier par défaut, au contenu vide qui ne faisait rien et un deuxième constructeur, prenant en arguments les coordonnées du Point. Les constructeurs sont essentiels lorsque la classe contient ou manipule des données dynamique, à l'aide de pointeurs par exemple. Moralement, l'attribution de mémoire se fera dans le constructeur et la libération dans le destructeur. Dans l'exemple, le constructeur et le destructeur par défaut ne font rien et leur écriture n'est pas obligatoire, mais c'est une bonne chose de les écrire systématiquement. Les constructeurs répondent aux mêmes règles de surcharge que les autres fonctions, il est donc possible d'en déclarer plusieurs, la condition étant que les listes d'arguments qui caractérisent leurs signatures soient suffisamment différentes pour qu'il n'y ait pas d'ambigüité et que le compilateur puisse choisir. Dans notre exemple de Point nous pouvons imaginer par exemple en sus des deux constructeurs déjà définis un nouveau constructeur initialisant un point comme intersection de deux droites (bien entendu nous supposons la classe Droite définie quelque part) :

```
class Point
{
  /* ... */

  Point(){};                    // constructeur par defaut
  Point (float a, float b);     // un constructeur qui initialise les
                                // coordonnees

  Point (Droite d1, Droite d2); // un constructeur construisant le
                                // point, intersection de deux droites
                                // si les droites sont paralleles
                                // il retournera l'origine


  ~Point(){};                   // destructeur par defaut

  /* ... */

};
```

13.2.1 Constructeur par recopie

Dans la liste des bonnes pratiques, il est essentiel de rajouter la définition systématique d'un constructeur par recopie. Il s'agit simplement de définir la construction d'un objet de la classe considérée en supposant disposer

d'un objet de cette même classe et en recopiant les données. Dans notre cas :

```
class Point
{
  /* ... */

  Point(const Point &p);

  /* ... */
}
```

Dans la déclaration précédente, le mot clé **const** garantie l'impossibilité de modifier le point p que l'on copie, mais il déclare surtout que l'on ne veut pas le faire. Il est important de spécifier ce genre de chose dans l'interface car elle précise si l'on à l'intention de modifier ou non l'argument en question.
Bien entendu, en l'absence de données dynamiques les choses sont bien plus simples et souvent les constructeurs par défaut suffisent, mais encore une fois, c'est une bonne habitude à prendre que de s'astreindre à définir de manière systématique un constructeur par recopie.

13.2.2 Destructeur

Le destructeur joue un rôle fondamental également, son rôle est de nettoyer les classes à la fin de leur vie. Comme par exemple libérer la mémoire alloué dynamiquement, généralement dans le constructeur.

13.3 Les restrictions d'accès

Dans notre exemple précédent, le mot clé **public** indique que les données membres et les fonctions membres sont accessibles sans restriction. Une fonction quelconque utilisant la classe Point peut donc par exemple, modifier directement les coordonnées du point.

```
float truc (Point& p)
{
  float u,v;
  /* ... */


  p.x = 3.0 - u;          // on manipule directement les champs x et y du
  p.y = 2*v -1.6;         // point, p
}
```

Mais il est souvent utile d'interdire l'accès aux données par des fonctions non membres, pour un meilleur contrôle des données. On peut ainsi limiter l'accès à travers des fonctions particulières qui ne permettent par exemple que la lecture et non l'écriture.

```
class Point
{
  private:
  float x;                    // abscisse
  float y;                    // ordonnee


  public:
  Point(){};                    // constructeur par defaut
  Point (float a, float b);     // un constructeur qui initialise les
```

```
                                    // coordonnees
  ~Point(){};                       // destructeur par defaut
  float Distance(Point X);          // distance a un autre point
  float getx(void){return x;}       // la fonction getx permet de lire le
                                    // champ x du point
  float gety(void){return y;}       // de meme gety donne la valeur de y
};
```

Le mot clé **private** indique les données ou fonctions qui ne sont accessibles qu'aux fonctions membres ou aux amis (**friend**) de la classe (nous verrons plus tard la notion d'amis).
Dans l'exemple précédent, il n'est plus possible d'accéder directement aux champs x et y de la classe point, donc telle qu'elle est définie, il n'est plus possible de modifier les coordonnées d'un point. Si cela est nécessaire il faut alors définir par exemple, des fonctions membres qui modifieront les champs x et y :

```
void Point::setx(float a){x=a;};
void Point::sety(float b){y=b;};
```

Il existe également une autre catégorie de restriction, le mot clé **protected** défini un membre comme publique pour une classe dérivée et privée pour les autres (cette notion aussi sera vue en détail lors de l'héritage).
Attention, les restrictions d'accès sont bien entendu contournables à l'aide de manipulation d'adresses, mais nous supposons ici que personne ne cherche à tricher. ;-)

13.4 Surcharge des opérateurs

En C++, il est possible de surcharger pour une classe données, les fonctions particulières que sont les opérateurs :

+	-	*	/	%	∧	&
|	∼	!	=	<	>	+=
− =	*=	/=	%=	∧ =	& =	| =
<<	>>	>>=	<<=	==	! =	<=
>=	&&	||	++	− −	− > *	,
− >	[ ]	( )	*new*	*new*[ ]	*delete*	*delete*[ ]

Par contre, il est impossible de définir de nouveaux opérateurs, ceux-ci sont les seuls utilisables.
Attention, tout de même, les opérateurs peuvent être binaires ou unaires, voire unaires et binaires, comme le − par exemple qui peut représenter la soustraction et le négatif. Il est impossible de changer le type unaire en binaire ou inversement binaire en unaire.
Pour surcharger un opérateur, il est nécessaire que ce soit une fonction membre ou bien que l'opérateur prenne comme argument, au moins un type défini par l'utilisateur.
Il est important d'être cohérent avec l'utilisation des opérateurs, il faut que ceux-ci soient sans ambigüité pour l'utilisateur et surtout pas utilisés à contre-emploi.

13.4.1 L'exemple des complexes

Tiré du livre de Bjarne Stroustrup, l'exemple de la classe des complexes est un trés bon exemple de l'intérêt et de l'utilisation de la surcharge des opérateurs. On veut écrire une classe de complexe grace à laquelle on veut pouvoir faire des choses comme :

```
.
.
.
Complexe z1 = Complexe(5,-1);
Complexe z2 = z1 + 1.7;
Complexe z3 = z2 - 2.1;
Complexe z4;
```

```
z4 = 2*z1*z3;
.
.
.
```

Pour les opérateurs, il existe essentiellement deux manière de voir les choses, l'opérateur peut être une fonction membre ou une fonction non-membre. Si on veut minimiser l'accès direct à la représentation il faut également minimiser les fonctions membres. On peut alors couper la poire en deux, déclarer membre des fonctions du type $+=$ et non membre l'opérateur $+$:

```
class Complexe
{
  private:
  double re, im;

  public :

  Complexe(){re=im=0.0;};
  Complexe(double a, double b)
  {
    re = a;
    im = b;
  };
  Complexe& operator+=(Complexe a);
};


 Complexe operator+(Complexe a, Complexe b);
  {
    Complexe z = a;
    return z += b;
  }
```

On peut bien entendu faire de même avec $-=$, $*=$ et $/=$. Il ne faut pas également oublier de permettre les opérations avec des types différents, voici une nouvelle version de l'interface :

```
class Complexe
{
  private:
  double re, im;

  public :

  Complexe(){re=im=0.0;};

  Complexe(double a, double b)
  {
    re = a;
    im = b;
  }

  Complexe& operator+=(Complexe a)
  {
```

```
    re += a.re;
    im += a.im;
    return *this;
  }

  Complexe& operator+=(double x)
  {
    re += x;
    return *this;
  }
};

 Complexe operator+(Complexe a, Complexe b);
  {
    Complexe z = a;
    return z += b;
  }

 Complexe operator+(Complexe a, double x);
  {
    Complexe z = a;
    return z += x;
  }

 Complexe operator+(double x,Complexe a);
  {
    Complexe z = a;
    return z += x;
  }
```

13.4.2 Les conversions

Pour pouvoir effectuer des initialisations ou des affectuations, il faut souvent disposer de conversions, par exemple pour écrire Complex z = 8.0; il faut pourvoir convertir un scalaire (8.0) vers un Complexe (z). Un constructeur qui ne prend qu'un seul argument précise la conversion de son type d'argument vers le type de la classe :

```
class Complexe
{
  private:
  double re;
  double im;

  public:
  Complexe(double x):re(x), im(0){}
  .
  .
  .
};
```

Il est d'ailleurs possible, en utilisant les arguments par défaut de reécrire le tout en un seul constructeur :

```
class Complexe
{
```

```
  private:
  double re, im;

  public :

  Complexe(double x=0, double y=0): re(x),im(y){}
  .
  .
  .
};
```

13.4.3 La copie

Le constructeur de copie permet l'initialisation par un complexe déja défini. Nous avons vu précedemment qu'il est trés important lors de l'utilisation de structures de données dynamiques. Même si ici ce n'est pas le cas, il est d'usage de le déclarer explicitement :

```
class Complexe
{
  private:
  double re, im;

  public :

  Complexe(const Complexe& z):re(z.re),im(z.im){}
  Complexe(double x=0, double y=0): re(x),im(y){}
  .
  .
  .
```

13.4.4 Quelques opérateurs particuliers

14 Template

En français on traduit par modèle, à défaut d'autre chose ! Les **templates** sont essentiels en **C++** surtout lorsqu'il s'agit de calculs.
Les templates permettent de faire de la programmation générique, autrment dit de traiter les types comme des paramêtres. Le mécanisme des modèles est à la base de la librairie standard, celle-ci s'est d'abord appelée STL comme standart template library.

14.1 Les fonctions templates

Reprenons notre exemple de la fonction *min* de la section 11.4. Supposons donc disposer des fonctions *min* suivantes :

```
int min( int i1, int i2)
{
  return i1 < i2 ? i1 : i2;
}

float min( float x1, float x2)
{
  return x1 < x2 ? x1 : x2;
```

```
}

double min( doule xx1, xx2)
{
  return xx1 < xx2 ? xx1 : xx2;
}
```

Il n'est pas difficile de remarquer quelques similitudes entre ces fonctions, la seule chose qui les différencie c'est le type des opérandes et du résultat. Le mécanisme des **templates** pour les fonctions, permet de définir une fonction pour un ou plusieurs type générique :

```
template <class Type>  Type min( Type a1, Type a2)
{
  return a1 < a2 ? a1 : a2;
}
```

La déclaration d'une fonction **template** commence toujours par le mot clé **template** suivi d'une liste des paramètres du template séparés par des virgules et encadrés par < >. La liste de paramètres ne peut être vide. Un élément de cette liste est soit constitué du mot clé **class** ou du mot clé **typename** suivi d'un identificateur, soit d'une declaration de type ordinaire. Si nous voulions par exemple une fonction prenant le min des éléments d'un tableau, on peut ajouter le paramètre taille du tableau comme suit :

```
template <class Type, int taille>        // la liste est composee de deux declarations
   Type min( const Type (&a) [taille])   // on travaillera donc sur un tableau
{

  Type valeur_min = a[0];                // Type peut etre bien sur etre utilisee dans la fo
  for (int i = 0; i < taille ; i++)
    if (a[i] < valeur_min)
      valeur_min = a[i];

  return valeur_min;

}
```

Remarque, bien évidemment comme pour une fonction ordinaire, il est possible de surcharger une fonction **template**, mais nous ne détaillerons pas plus ici.
Pour en savoir plus on pourra consulter par exemple, C++ primer de Stanley B. Lippman et Josée Lajoie.

14.2 Les classes templates

Dans la même veine que les fonctions, il est possible en **C++** de construire des classes **templates**. Comme dans le cas des fonctions la déclaration commençe par le mot clé **template** suivi d'une liste de paramètres encadrée par < >. prenons par exemple le cas trés classique de l'algèbre linéaire et des matrices carrés.
Plutôt que de créer une classe de matrices float, une classe de matrices double, une classe de matrices complexe, etc
En utilisant les templates on écrira :

```
template <class Type, int taille> class Matrice {

  public:
    Matrice();
    Matrice(Type, int);
    ~Matrice()
```

```
  private:
    Type don[taille][taille];
    int nbligne = taille;

}
```

Ensuite son utilisation se fait en indiquant le Type et la taille, par exemple, en supposant avoir surchargé les opérateurs +,*, - ,

, de manière à faire du calcul matriciel :

```
int main()
{
    int i,j;
    const int n = 4;
    Matrice<double, n>  A, B, C, D;

    for (i=0;i<n;i++)
        for (j=0;j<n;j++)
          {
                A[i][j] = i / 2.0 + j / 3.0;
                B[i][j] = i / 5.0 + j / 7.0;
          }
    C = A - B;
    D = A * B;
}
```

Attention, cette classe est un exemple et ne contient pas grand chose, pour qu'elle soit utile il y a encore un petit peu de travail. Il existe des classes disponibles qui font ce genre travail.

15 Bibliothèque standard

Le langage C++ est accompagné d'une bibliothèque appelée, bibliothèque standard qui permet, non seulement de faire des entrées sorties mais surtout de manipuler des données génériques de haut niveau. Elle est devenu un élément essentiel pour le programmeur. La librairie est définie dans l'espace de noms *std*.

15.1 Les entrées sorties

Les entrées-sorties utilisent en C++ des classes particulières, définies dans une interface iostream. Il s'agit de flux de données. Par exemple pour écrire une petite phrase à l'écran on fera :

```
#include <iostream>

int main()
{
  std::cout << ''Rien ne sert de courir, il faut partir     point'' << std::endl;
}
```

On notera l'utilisation de l'opérateur de portée :: pour signifier que l'on travaille dans l'espace de noms *std*. La bibliothèque fournit les sorties pour tous les types intégrés, par défaut les valeurs sont envoyées vers la sortie standard comme des chaînes de caractères, autrement dit, std:cout << 38; enverra successivement les caractères 3 et 8 sur le flux de la sortie standard. Le code suivant donnera le même résultat :

```
int j = 38;
std::cout << j;
```

La possibilité de surcharger l'opérateur << permet d'adapter à une classe définie par l'utilisateur les flots de sortie, dans le cas de la classe complexe, par exemple.

```
class Complexe
{
  .
  .
  .
  public :
  double reel() const{return re;}
  double im() const{}return il;}
  .
  .
  .
}
```

On peut redéfinir << pour la classe Complexe comme suit :

```
ostream& operator<<(ostream& sortie, const Complexe& z)
{
  return sortie << '(' << z.reel() << ',' << z.im() << ')';
}
```

L'utilisation de << se fera alors pour un Complexe, comme pour un type intégré.

15.2 Les containers

Cette notion est à la base de la librairie standard, on va considérer des classes susceptibles de contenir des objets, un petit peu comme une extension des types collections (5). Un **container** pourra être, une liste, un vecteur, une pile, un ensemble, ...
Chaque structure possède des avantages et des inconvénients et c'est en général l'application considérée qui guide le choix de l'utilisateur. L'utilisation des templates permet une abstraction et la manipulation d' objets beaucoup plus complexes que par le passé. On se promène au sein de la structure à l'aide d'un **iterator**. l'idée est trés simple on veut parcourir une collection aussi facilement qu'un tableau avec une boucle **for**. Ce type de **containers** est du type **séquentiel**. Mais la librairie fournit également des **containers associatifs** qui permettent l'accès des éléments à l'aide de clés. Nous insisterons ici beaucoup plus sur les **containers** de type **séquentiels**.
Attention, pour que l'on puisse manipuler des **containers** d'un type d'objet, il faut pouvoir prendre des copies de ce dernier, autrement dit, de disposer par exemple d'un constructeur par recopie et d'un opérateur d'assignation(=).
Associée aux type génériques que sont les **containers**, il existe une notion d'**algorithme** appliqué sur ces mêmes types de données. Il existe un certain nombre d'algorithmes disponibles dans la librairie standard, mais cette dernière fournie surtout les mécanismes pour en concevoir de nouveaux.

15.3 Utiliser vector

La classe **vector** est devenue incontournable pour la programmation. Elle résout un certain nombre de problèmes que pose l'utilisation directe des tableaux. Voici quelques exemples de son utilisation. Imaginons que nous souhaitions écrire une petite application traitant d'un répertoire téléphonique.

```
struct entree {
        string prenom;
        string nom;
        int numero;
}

.
.
.

std::vector<entree> repertoire(500);     // on declare un vecteur contenant
                                         // 500 objets de type entree

int taille=repertoire.size();            // la fonction size permet de r cup rer
                                         // la taille du vecteur

repertoire.resize(taille+100);           // au besoin la fonction resize
                                         // permet de modifier la taille du vecteur
```

Attention, la classe **vector** ne contient pas de mécanisme de contrôle de dépassement. Il est nécessaire si vous en souhaitez un de l' implémenter.
Il est possible de parcourir le vecteur comme un tableau, voici par exemple une boucle qui affiche tous les numéros :

```
for (int i = 0; i < repertoire.size();i++)
        std::cout << "repertoire[" << i << "] = "
                << repertoire[i].numero << "\n";
```

On peut également utiliser deux fonctions plus génériques qui permettent d' obtenir le premier et le dernier élément, ainsi qu'un **iterator**. Cet exemple est censé fonctionner sur tous les types séquentiels :

```
vector<entree>::const_iterator i;

for (i = repertoire.begin();i != repertoire.end(); ++i)
        std::cout << i->prenom <<"-"<< i->nom <<"-" << i->numero << "\n";
```

Attention, on le voit bien dans l'exemple précédent, l'**iterator** est un pointeur.

15.4 Utiliser list

comme le type **vector** précédent le type **list** est l'un des plus couramment utilisé. en reprenant notre exemple précédent on peut décider d'implémenter notre répertoire sous forme de **list**.

```
std::list<entree> repertoire;    // pas de notion de taille ici
                                 // cette liste ne contient pas d'element
```

La manière la plus simple pour parcourir notre répertoire est donc un **iterator**

```
list<entree>::const_iterator i;

for (i = repertoire.begin();i != repertoire.end(); ++i)
        std::cout << i->prenom <<"-"<< i->nom <<"-" << i->numero << "\n";
```

On peut également ajouter des éléments, au début ou à la fin d'une liste :

```
entree truc;
truc.prenom = "philippe";
truc.nom = "Le␣Bel";
truc.numero = 0140567834;

repertoire.push_front(truc);      // ajout d'une entree au debut
repertoire.push_back(truc);       // ajout d'une entree a la fin
```

15.5 string

Il ne s'agit pas à proprement parler d'un container mais plutôt d'une encapsulation des chaînes de caractères. Il permet de manipuler des chaînes de caractères beaucoup plus facilement, avec des fonctions comme la concaténation, les comparaisons, etc.

```
std::string s1 = "un";
std::string s2 = "si";
std::string s3 = "beau";
std::string s4 = "chapeau";
std::string espace = "␣";
std::string pt = ".";

std::string s;

s = s1 + espace + s2 + espace + s3 + espace + s4 + pt;
std::cout << s << "\n";
```

L'exécution du code précédent nous affiche la phrase :
un si beau chapeau.

16 Références

Quelques références :

- livres C++
  - Le langage C++, Bjarne Stroustrup
  - Accelerated C++, Andrew Koenig, Barbara E. Moo
  - C++ Primer, Lippman
- livres langage C
  - C primer plus, Stephen Prata
  - Le langage C norme ansi, B. Kernighan, D. Ritchie
- livres plus générique
  - Concepts of programming languages, Robert W. Sebesta
  - The java programmer's guide to numerical computing, Ronald Mak
  - Analyse & conception orientées objets, Grady Booch
  - Numerical Computing with IEEE floating point arithmetic, Michael L. Overton
  - Numerical computation (Vol 1 & 2), Christoph W. Ueberhuber
- Documents pour le C
  - le poly de Patrick Corde de l'Idris
  - celui de Christian Bac
- Documents pour le C++
  - les transparents de Francois Laroussinie

— le C++ pour les pros, Ours blanc des Carpathes ISIMA 1999
— Le langage, C++ Henri Garreta Luminy

Table des matières