Éléments de programmation en C++

1. Un bref historique

En 1972, Dennis Ritchie créa le C ancêtre du C++. Ce langage peut être qualifié à la fois de bas niveau car il permet de faire appel à toutes les ressources de la machine mais aussi de langage évolué. En effet, il introduit une syntaxe assez complexe par rapport aux langages précités ainsi que de nombreuses fonctions évoluées. Ces fonctions sont regroupées en librairies, considérée comme des boîtes à outils.

La mise à jour la plus marquante du C fut apportée par Bjarne Stroustrup en 1982. Il y intégra la programmation objet. Le C++ est en fait une surcouche du C (C++ signifie une incrémentation du C). Il hérite donc de tous les outils du langage C.

Bjarne Stroustrup, l'inventeur du C++, a en effet décidé d'ajouter au langage C les propriétés de l'approche **orientée objet**. Ainsi, vers la fin des années 80 un nouveau langage, baptisé *C with classes* (traduisez "C avec des classes"), apparaît. Celui-ci a ensuite été renommé en *C++*, clin d'oeil au symbole d'incrémentation ++ du langage C, afin de signaler qu'il s'agit d'un langage C amélioré.

Le langage C++ a été donc conçu pour :

1. être mieux que le C,
2. supporter l'abstraction des données,
3. supporter la programmation orientée objet.

Tout programme C est donc un programme C++, mais quelques incompatibilités existent dues aux règles de typage plus strictes en C++. Il est toujours possible toutefois de se servir de la librairie C dans C++.

La structure d'un programme écrit en C++ est évidemment similaire à celle d'un programme en C. On y retrouve une fonction principale, des fichiers d'en-tête et des fichiers source, et une syntaxe très proche du langage C. Les fichiers d'en-tête (extensions : .H, .h, .hxx, .hpp) regroupent les déclarations de types (typedef) et de classes, les prototypes des fonctions, etc.

Les fichiers sources (extensions : .C, .cxx, .cpp) comportent le code proprement dit : le corps des fonctions et des méthodes, la déclaration de variables, d'objets.

La syntaxe des expressions est très proche du C pour le *for*, *while*, *do*, *if*, *switch*, etc. À la différence du C, dans le corps d'une fonction ou d'une méthode, on peut déclarer un objet (ou une variable) à tout moment.

Les commentaires du C restent, le compilateur ignore le texte entre /*...*/. Le C++ introduit un nouveau type de commentaire : le compilateur ignore le texte situé après // jusqu'à la fin de la ligne.

```
int i = 0; // Ceci est un commentaire sur une ligne
/* Ceci est un commentaire qui s'étend sur plusieurs lignes */
```

2. Structure d'un programme C++

Considérons le petit programme ci-dessous :

```
#include <iostream.h>

# define TVA 18.6

main(void) {

        float HT, TTC;

        cout << " Veuillez entrer le prix H.H.T";

        cin >> HT;

        TTC = HT*(1+(TVA/100));

        cout << "Le prix T.T.C. est  " << TTC;

        } // fin du programme
```

Comme on peut le constater, on trouve dans ce programme :

A. des directives du pré processeur (commençant par #)

#include : inclus le fichier définissant (on préfère dire déclarant) les fonctions standards d'entrées/sorties (en anglais STanDard In/Out), qui feront le lien entre le programme et la console (clavier/écran). Cette instruction permet d'inclure dans un programme la définition de certains objets, types ou fonctions. Le nom du fichier peut être soit à l'intérieur des < et >, soit entre guillemets :

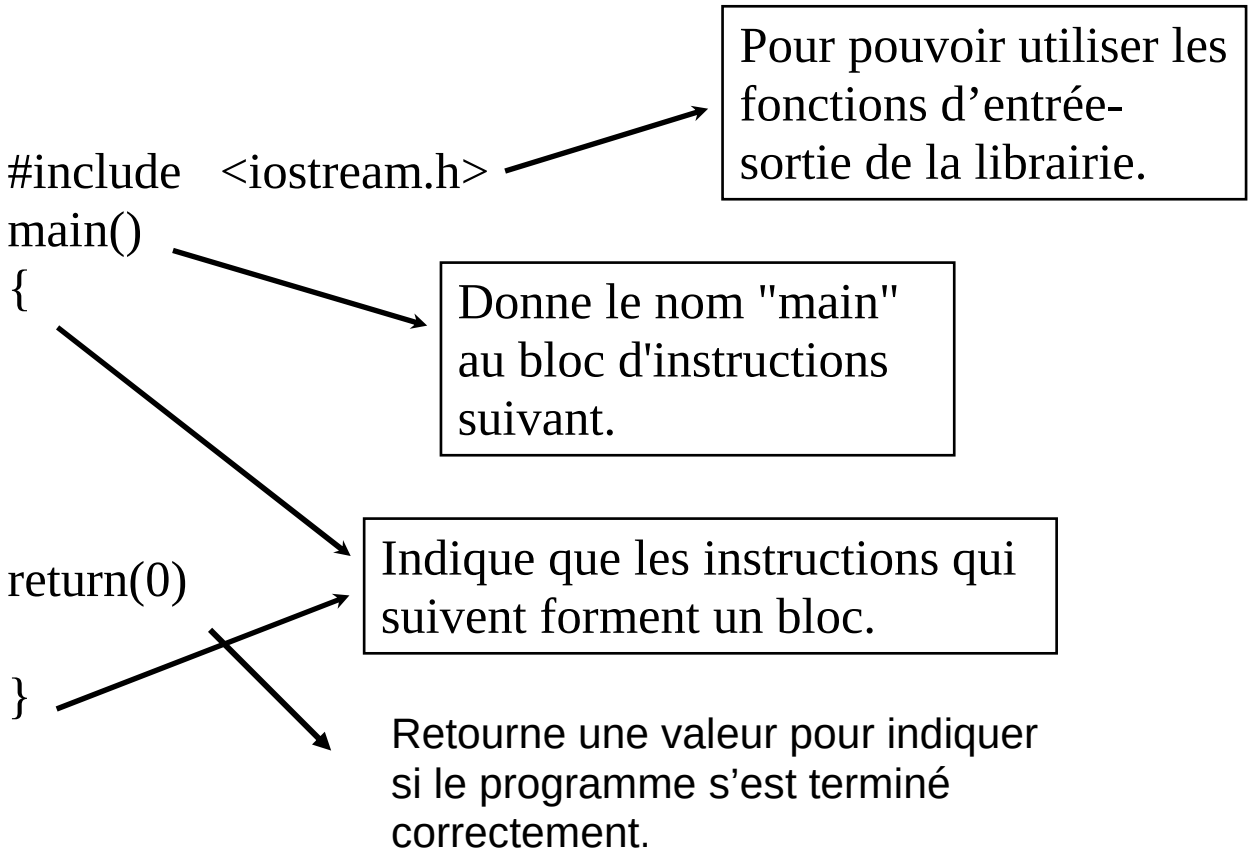
- `#include <nom_fichier>` : inclut le fichier nom_fichier en le cherchant d'abord dans les chemins configurés, puis dans le même répertoire que le fichier source,
- `#include "nom_fichier"` Inclut le fichier nom_fichier en le cherchant d'abord dans le même répertoire que le fichier source, puis dans les chemins configurés.

#define : définit une constante. A chaque fois que le compilateur rencontrera, dans sa traduction de la suite du fichier en langage machine, le mot TVA, ces trois lettres seront remplacées par le nombre 18.6. Ces transformation sont faites dans une première passe (appelée pré compilation), où l'on ne fait que du "traitement de texte", c'est-à-dire des remplacements d'un texte par un autre sans chercher à en comprendre la signification.

B. Une entête de fonction. Dans ce cas, on ne possède qu'une seule fonction, la fonction principale (**main function**). Cette ligne est obligatoire en C++, elle définit le "point d'entrée" du programme, c'est-à-dire l'endroit où débutera l'exécution du programme. Tous programmes en C++ nécessitent un point d'entrée qui constitue la première fonction appelée dans un programme: c'est le rôle de la fonction **main()**. Elle est appelée automatiquement lors de l'exécution du programme. Comme **void** l'indique notre fonction ne renvoie rien. Nous aurions pu choisir de renvoyer un entier en écrivant int main() par exemple. De plus, main() est suivi de parenthèses. C'est le cas pour toutes fonctions. Les parenthèses servent à passer des arguments à la fonction, c'est-à-dire des données. Pour plus de détails et d'exemples concrets sur ce point, consulter le fichier suivant. Dans notre cas, les parenthèses sont vides, cela veut dire que notre fonction ne prend pas d'arguments, elle n'a besoin d'aucune donnée extérieure pour fonctionner.

C. Un "bloc d'instructions" : délimité par des accolades {}, et comportant des déclarations de variables et une suite d'instructions. Noter qu'il est préférable lors de l'écriture du code de fermer une accolade juste après l'avoir ouverte. Ainsi, on évitera de se poser les questions du type *Pourquoi ça ne marche pas ?* alors qu'il suffisait de fermer l'accolade! Comme dans l'exemple, l'alignement vertical et le décalage du code par rapport à celles-ci permettent aussi une lecture et un déboggage plus aisés du code. Toutefois, cette convention n'est pas obligatoire, mais elle nous facilite la vie!

Remarquez qu'à l'intérieur d'un bloc d'instructions, chaque ligne se termine par un point virgule (;). Chaque ligne représente alors une "phrase" indépendante des autres, interprétée séparément par le compilateur, appelée une instruction. La structure d'un programme écrit en C++ est donc comme suit :



Dans ce chapitre, nous allons survoler le langage C++. Il est supposé que les étudiant(e)s aient une certaine expérience de programmation, de préférence dans le langage C-C++.

3. Survol du langage C++

3.1. Type de variables

Par défaut, plusieurs types simples sont prédéfinis comme les entiers, les caractères ou les décimaux. Toutefois, il est possible de définir son propre type, comme on le verra par la suite.

Type de donnée	Signification	Taille (en octets)	Plage de valeurs acceptées
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	-2 147 483 648 à 2 147 483 647
unsigned long int	Entier long non signé	4	0 à 4 294 967 295
float	Flottant (réel)	4	$-3.4*10^{-38}$ à $3.4*10^{38}$
double	Flottant double	8	$-1.7*10^{-308}$ à $1.7*10^{308}$
long double	Flottant double long	10	$-3.4*10^{-4932}$ à $3.4*10^{4932}$
bool	Booléen	Même taille que le type *int*, parfois *1* sur quelques compilateurs	Prend deux valeurs : *'true'* et *'false'* mais une conversion implicite (valant 0 ou 1) est faite par le compilateur lorsque l'on affecte un entier (en réalité toute autre valeur que 0 est considérée comme égale à *True*).

De façon générale, un caractère est représenté par sa valeur entre guillemets ou sa valeur numérique correspondante: voir ASCII. Le tableau des types exposés plus haut n'est pas exhaustif. En effet, on peut tout à fait combiner un mot clef du type short (court), unsigned (valeur absolue), long (long) avec un des types primaires char, int, float ou double. De plus, il n'est pas nécessaire de tous les retenir. Le choix de type doit dépendre de la nature de l'information à stocker et de l'intervalle des valeurs possibles.

3.2. Variables

Une variable est une case mémoire de l'ordinateur, que l'on se réserve pour notre programme. On définit le nom que l'on choisit pour chaque variable, ainsi que son type. On ne peut jamais utiliser de variable sans l'avoir déclarée auparavant.

Une erreur de frappe devrait donc être facilement détectée, à condition d'avoir choisi des noms de variables suffisamment différents (et de plus d'une lettre).

Toute variable correspond à un type. Toutefois, avant de les utiliser, il y a lieu de les déclarer précédées de leur type.

La déclaration des variables se fait selon le modèle suivant:

type variable1, variable2, ... ;

Remarquez la **virgule** séparant deux variables ainsi que le **point-virgule** à la fin.

Exemple 1

int nbre;	définit nbre comme une variable entière
int a,b,c;	définit trois variables entières dont les noms respectifs sont a, b et c.
double x;	définit x comme une variable réelle.

Nous pouvons aussi initialiser une variable lors de sa déclaration, par exemple :

```
int var1=10, var2=0;
```

On dispose aussi de *qualificateurs*: unsigned, long, const, …

Exemple 2

```
char x = -1; /* Un caractère est en fait de type entier

unssigned int y = 255;

const int u = 10;

const int v; /* légal */
```

Les constantes de type entier

décimales:	28	6	-67	
octales:	02	0100	-0107	(commence par un zéro)
hexadécimales:	0x21	0X7	-0x46	(commence par 0x ou 0X)

Les constantes de type réel

point flottant:	1200.0		3.1416	-.00067
avec exposant:	12e2	3	1416E-4	-67e-5
mixtes:	1.2e3		3.1416e0	-6.7E-4

Le type booléen est représenté à l'aide du type **bool**. La valeur 0 signifie *faux* et toute autre valeur signifie *vrai*.

Lorsque des variables sont de types différents, le compilateur pourra effectuer une conversion implicite:

Exemple 3

```
int i = 12345;
```

```
char c;
float x = 2.2;
c = i; // Danger! Le compilateur vous avertira.
       // Sinon, changez de compilateur!
i = c; // Ok.
c = x; // Attention.
```

On peut aussi effectuer la conversion d'une manière explicite (cast en anglais, transtypage en français). Cette technique permet de forcer une conversion. L'opérateur cast consiste à indiquer le type entre parenthèses devant la variable à convertir:

type1 variable1= (type1) expression variable2 ;

Cette instruction permet de convertir le type de ***expression*** à type1.

Exemple 4

```
int c = ((int) pow(2, 3)) % 4;
sqrt((double)c);
```

int entier = (int)3.14;	//cast d'un decimal en entier
double decimal =6.56;	//cast d'un decimal en entier
int entier2 = (int)decimal;	
char caractere = (char)56;	//cast d'un entier en caractère

Opérateurs

- opérateurs arithmétiques usuels (+, *, -, /, %),
- opérateurs de comparaison (<, >, <=, >=, = =, !=)
- opérateurs logiques (&&, ||, !, ~),
- opérateurs d'*incrémentation* et de *décrémentation* (++, --).
  1. ++a : ajoute 1 à la variable a. Le résultat de l'expression est la valeur finale de a (c'est-à-dire après incrémentation). On l'appelle incrémentation préfixée.
  2. a++ : ajoute 1 à la variable a. Le résultat de l'expression est la valeur initiale de a (c'est-à-dire avant incrémentation). C'est l'incrémentation postfixée.

Exemple 5

```
i = 1;
j = ++i ;
j = i++ ;
```

Dans les deux cas, la valeur de i est incrémentée de 1; c'est-à-dire, elle devient égale à 2. Seulement, dans j = ++i, la valeur de j est de 2, et dans j = i++, la valeur de j est de 1.

Mot Clef	Syntaxe	Equivalence
++	x++;	x=x+1;
++	y=x++;	y=x; **ensuite** x=x+1;
++	y=++x;	x=x+1; **ensuite** y=x;
--	x--;	x=x-1;
--	y=x--;	y=x; **ensuite** x=x-1;
--	y=--x;	x=x-1; **ensuite** y=x;

Variable chaîne de caractères

Une chaîne de caractère (appelée *string* en anglais) est une suite de caractères, c'est-à-dire un ensemble de symboles faisant partie du jeu de caractères, défini par le code ASCII. Dans le langage C++, une chaîne de caractères est un tableau, comportant plusieurs données de type char, dont le dernier élément est le caractère nul '\0', c'est-à-dire le premier caractère du code ASCII (dont la valeur est 0). Ce caractère est un caractère de contrôle (donc non affichable) qui permet d'indiquer une fin de chaîne de caractères. Ainsi une chaîne composée de *n* éléments sera en fait un tableau de *n+1* éléments de type char.
De nombreuses fonctions de manipulation de chaîne de caractères sont directement fournies. Ces fonctions se trouvent dans le fichier d'en-tête <string>, c'est la raison pour laquelle il y a lieu d'ajouter la ligne suivante en début de programme:

#include <string>

Exemple 6

```
#include <iostream>
using namespace  std   /* certains compilateurs supportent la déclaration <iostream>, au lieu de <iostream.h>. Dans ce cas, pour utiliser les entrées/sortie std ::cin et std ::cout, on doit auparavant faire la déclaration using namespace std. */

#include <string>

int main()
{
  string   nomDuClient = "Robert";
  string message = "Bonjour, M. " + nomDuClient;
  message += '.';
  cout << message << endl;
}
```

Noter l'utilisation du type string au lieu de char*.

Sur ce point-là, le C++ a mis à notre disposition un outils réellement plus pratique que les tableaux de type **char***. Bien sûr, la copie de chaînes de caractères est tout aussi simple :

```
string str1 = "Coucou";
string str2 = str1;
cout << str2 << endl
```

Ici encore, un simple opérateur d'affectation suffit pour effectuer la copie.

3.3. Les flux d'entrée - sortie

Les flux d'entrée et de sortie vont nous permettre d'entrer des valeurs au clavier ou bien d'en afficher à l'écran.

- **Écriture sur la sortie standard**

Pour écrire sur le flux de sortie (et en particulier sur la sortie standard : cout), on utilise l'opérateur << . Cela nous permet de faire des affichages sur l'écran:

```
// Ecriture d'une chaîne de caractères
cout << "bonjour!" << endl;

// Déclaration d'un entier, initialisé à 10
int i = 10;

// Déclaration d'un réel initialisé à 3.14159
double pi = 3.14159;
// Ecriture formattée d'un entier
cout << "i = " << i << endl;

// Ecriture formattée d'un double
cout << "pi = " << pi << endl;
```

La commande **endl** correspond au caractère de fin de ligne, et force l'écriture du buffer sur le flux. Autrement dit à sauter de ligne.

Pour lire sur un flux d'entrée (en particulier sur l'entrée standard cin), on utilise l'opérateur >> Autrement dit, à introduire les données via le clavier.

```
int i, j;       // Déclaration de 2 entiers
cin >> i >> j;  // Lecture de 2 entiers sur l'entrée standard
```

// l'espace et la fin de ligne sont utilisés comme séparateurs

Si on introduit "10 20↵", alors, la lecture effectuera i=10 et j=20.

Les opérateurs << et >> sont définis pour les types de base. Pour les types plus complexes (structures, objets), ils peuvent être redéfinis.

- **cout** est un flot de sortie prédéfini associé à la sortie standard (stdout du C).
- << est un opérateur (nommé opérateur d'insertion de flux) dont l'opérande de gauche (cout) est un flot et l'opérande de droite, une expression de type quelconque.

```
#include <iostream.h>   // indispensable pour utiliser cout et cin

// ou alors #include <iostream>
//              using namespace  std  instruction nécessaire pour utiliser le cout et cin

int main(){
        int n = 25;
        cout << "valeur: ";
        cout << n;
        return 0;
}
```

Le résultat est : valeur: 25

Exemple 7 : Les instructions

```
cout << "valeur: ";
cout << n;
```

sont équivalentes à

```
cout << "valeur: " << n;
```

Exemple 8

```
#include <iostream.h>

int main()
{
        int n = 25;
        char c = 'a';
```

```
        char *ch = "bonjour";
        double x = 12.3456789;

        cout << "valeur de n : " << n << "\n";
        cout << "valeur de c : " << c << "\n";
        cout << "chaîne ch : " << ch << "\n";
        cout << "valeur de x : " << x << "\n";
        return 0;
}
```

L'exécution devrait donner (sur Sun et sur PC) :

valeur de n : 25
valeur de c : a
chaîne ch : bonjour
valeur de x : 12.3457

- **Lecture sur l'entrée standard**

```
#include <iostream.h>
…
cin >> Var1 >> Var2 >> … >> VarN;
```

cin est le flot d'entrée connectée à l'entrée standard, le clavier. Il correspond au fichier prédéfini stdin du langage C.

Exemple 9

```
#include <iostream.h>

int main () {
        float valeur1, valeur2, valeur3;
        cout << "entrez 3 valeurs : ";
        cin >> valeur1 >> valeur2 >> valeur3;
        return 0;
}
```

Exemple 10

```
#include <iostream.h>

int main (){
        int n;
```

```
        float x;
        char t[80+1];

        do {
                cout << "donner un entier, une chaîne et un flottant : ";
                cin >> n >> t >> x;
                cout << "merci pour " << n << ", " << t << " et " << x << "\n";
           }
        while (n);
        return 0;
}
```

Exécution

```
donner un entier, une chaîne et un flottant : 15 bonjour 8.25
merci pour 15, bonjour et 8.25
donner un entier, une chaîne et un flottant : 15 bonjour  8.25
merci pour 15, bonjour et 8.25
donner un entier, une chaîne et un flottant : 0 bye 0
merci pour 0, bye et 0
```

Noter l'usage des séparateurs.

```
#include <iostream.h>

int main ()
{
        char t[80+1];   // pour conserver les caractères lus sur cin
        int i = 0;          // position courante dans le tableau t
        cout << "entrez une suite de caractères terminée par un point.\n";
        do
                cin >> t[i];
        while ( t[i++] != '.');
        cout << "\n\nVoici les caractères effectivement lus :\n";

        i=0;
        do
          cout << t[i];
        while ( t[i++] != '.');
        return 0;
}
```

Exécution

entrez une suite de caractères terminée par un point.
Voyons ce que fait C++ à la lecture d'une "suite de caractères".

Voici les caractères effectivement lus :
VoyonscequefaitC++àlalectured'une"suitedecaractères".

- **Fonction getline() et gets()**

Ces fonctions permettent de lire des chaînes de caractères qui contiennent plusieurs mots.

```
#include <iostream.h>
…
cin.getline (Var_chaine, Nb_max, Car_special);
```

La fonction getline() prend trois arguments : Var_chaine est une variable de type chaîne de caractères. Nb_max est le nombre maximum de caractères à lire moins 1 (Nb_max – 1). Ce nombre correspond au nombre maximal de caractères que la chaîne peut contenir. Car_special, le troisième argument qui est facultatif, permet de spécifier un caractère dont la rencontre va interrompre la lecture. Lorsque cet argument est absent c'est le caractère de fin de ligne qui sera le caractère de fin de lecture.

Exemple 11

```
char chaine[81];
cin.getline( chaine , 81);


char chaine[81];
cin.getline(chaine, 81, '%');
```

- **Les manipulateurs**

Pour préciser les formats des données, des manipulateurs peuvent être insérés dans les instructions d'extraction et d'insertion dans un flot. Ces manipulateurs conservent leur état jusqu'au prochain changement, sauf pour setw, qui revient à 0 après chaque opération. Pour utiliser les manipulateurs, il faut inclure les fichiers <iostream.h> et <iomanip.h>.

Variables numériques

setw (i) : i est la largeur minimale de la prochaine donnée à afficher.

```
int var = 12;
```

cout << setw(7) << var;

Cette instruction permet d'inscrire 5 espaces devant la valeur 12

L'instuction cout << setw(7) << -12; provoque l'ajout de 4 espaces devant –12.

setiosflags (ios :: mode1 | ios :: mode2 | ios :: mode3) : précise un ou des modes d'affichage
L'affichage d'un réel peut se faire en mode point flottant, *fixed* (ex : 3.1415) ou en notation scientifique, *scientific* (ex : 3.1415E+00). On précise le mode d'affichage d'un réel à l'aide de la fonction setiosflags(). ios :: fixed, comme argument de la fonction, précise le mode point flottant. La notation scientifique est obtenue en utilsant ios :: scientific dans l'argument. On peut préciser plus d'un argument en les séparant par l'opérateur | (ou binaire).

setprecision (i) : nombre de chiffres significatifs pour float et double

Exemple 12

	Affichage
cout << setiosflags (ios::fixed);	
cout << 3.14;	3.140000
cout << setw(4) << setprecision(1) << 3.14159;	3.1
cout << setw(2) << setprecision(6) << 3.14159;	3.14159

Si on veut par exemple afficher six décimales après le point même si le nombre à afficher n'en comporte pas autant dans sa partie fractionnaire, on utilise l'expression ios :: showpoint.

cout << setiosflags(ios::showpoint|ios::fixed);	
cout << 3.14;	3.140000
cout << setw(3) << setprecision(1) << 3.14159;	3.1
cout << setw(2) << setprecision(6) << 3.14159;	3.141590

hex : nombres en base 16
dec : nombres en base 10
oct : nombre en base 8

Exemple 13

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


int main (){
        int n = 12000;
        cout << "par défaut : " << n << "\n";
        cout << "en hexadécimal : " << hex << n << "\n";
        cout << "en décimal : " << dec << n << "\n";
        cout << "en octal : " << oct << n << "\n";
```

```
    return 0;
}
```

Exécution

par défaut : 12000
en hexadécimal : 2ee0
en décimal : 12000
en octal : 27340

3.4. Variables de type chaîne de caractères

cout << setw(i) << chaîne_de_caractères;

Si le nombre de caractères (i) spécifié est insuffisant pour que s'inscrivent tous les caractères, l'opérateur << ignore ce nombre et affiche la chaîne au complet. Si le nombre i dépasse la longueur de la chaîne, la chaîne est complétée avec des blancs par la gauche.

Autres manipulateurs

endl : insère une fin de ligne

cout << "merci" << endl; équivalente à cout << "merci" << "\n";

setfill(c) : Fixe le caractère de remplissage (blanc par défaut)

Exemple 14

```
cout << setfill('*') << setw(6) << 12;
```

L'affichage sera : ****12
On peut écrire nous-mêmes des manipulateurs, il suffit de faire des fonctions qui prennent un flot en entrée et qui retournent un flot en sortie, les deux par référence (nous reviendrons sur ce point lors de l'étude des fonctions).

3.5 Entrées /sorties sur fichiers

Sous Unix, il est possible de rediriger les entrées/sorties standards à l'aide des opérateurs < et > pour les remplacer par des fichiers ordinaires.
Si l'exécutable de notre programme est dans a.out :

a.out < entree > sortie

Le programme a.out fera sa lecture standard (cin) dans le fichier entree et sa sortie standard (cout) dans le fichier sortie. La redirection des entrées/sorties est commode mais elle force à choisir entre l'utilisation d'un fichier ordinaire et celle du clavier ou de l'écran. On pourrait vouloir utiliser les deux.

En C++, l'accès à des fichiers en entrée et en sortie se fait à l'aide des flots de fichiers d'entrée (*ifstream*), de sortie (*ofstream*), d'entrée/sortie (*fstream*).

Exemple 15

```
#include <fstream>
…
```

Déclaration de fichiers :

```
ifstream entree;          // fichier d'entrée
ofstream sortie;          // fichier de sortie
fstream entree_sortie; // fichier d'entrée/sortie
```

Ouverture des fichiers : fonction open()

```
entree.open("donnees");
sortie.open ("resultats");
entree_sortie.open ("Fichier", ios::in|ios::out);
```

On peut tester si l'ouverture a réussi à l'aide de la fonction fail()

```
if ( entree.fail() )
        cout << "problème d'ouverture";

if ( entree.eof() )
        cout << "fin de fichier atteinte";
```

À l'ouverture, on peut spécifier comme deuxième paramètre certains modes particuliers. Voici les modes d'entrée-sortie possibles :

ios::in	lecture au début
ios::out	écriture au début
ios::app	écriture à la fin du fichier
ios::nocreate	le fichier doit exister à l'ouverture (ne sera pas créé si absent)
ios::noreplace	le fichier ne doit pas exister (ne sera pas écrasé si présent)
ios::trunc	efface un fichier existant.
ios::ate	se positionne à la fin du fichier

```
#include <iostream.h>
#include <fstream>

int main (){
        fstream sortie;
```

```
        char nom[10];

        cout << "nom du fichier : ";
        cin >> nom;
        sortie.open(nom,ios::out|ios::noreplace);
        if (!sortie)
                cout << "le fichier existe déjà!" << endl;
        return 0;
}
```

Note : if (!sortie) est équivalente à if (sortie.fail())

- **Lecture/écriture de fichiers**

Les opérations d'entrée se font comme pour le clavier à l'aide de l'opérateur >>
Les opérations de sortie se font comme pour l'écran à l'aide de l'opérateur <<

```
entree >> i >> x;                   // lecture
sortie << setw(4) << x << endl;     // écriture
entree_sortie >> j;
entree_sortie << x;                 // Lecture suivie d'écriture
```

Fermeture des fichiers

```
        sortie.close();
        entree.close();
```

Autres procédures d'entrées-sorties

Il existe un grand nombre de procédures d'entrées/sorties. Comme pour les fonctions open() et close(), l'appel se fait en préfixant le nom de la fonction appelée par le nom du flot concerné, les deux séparées par un ".".

- char get() : lit le prochain caractère.
- put (char c) : écrit le caractère c.
- read ((char*) &buf, int nb) : lit nb caractères et les place à l'adresse buf en mémoire
- putback (char c) : recule d'une position et remplace le caractère par c.

4. Instructions

Une instruction est un ordre élémentaire que l'on donne à la machine, qui manipulera les données (variables) du programme. Les instructions constituant un programme C sont séparées par des

points-virgules (;). Dans le langage C, une instruction est simple ou composée (dans ce dernier cas on parle aussi d'un bloc d'instructions).

4.1. Instructions simples

```
int x = 2;
x = x + 1;
y = cos(x);
x = cos(x / y - 8) * 2;
```

4.2. Blocs d'instructions

```
{
  x = x + 1;
  y = cos(x);
  x = cos(x / y - 8) * 2;
 }
```

Autre exemple

```
int x = 1;
{ /* Début du bloc */
  int z = 0; /* Variable locale au bloc.*/
  x = z + 2;
  z = sin(x);
} /* Fin du bloc */
z = 0; /* Erreur, z est déclarée dans le bloc. */
```

a. **Instruction de test :** on en distingue trois formes.

Forme 1 : if (condition)
inst0;

L'instruction est évaluée si et seulement si la valeur de la condition est différente de 0. La valeur 0 signifie la valeur FAUX. Une valeur différente de 0 signifie la valeur VRAI

Forme 2: if (condition)
inst0;
else inst1;

L'instruction inst0 est exécutée si la valeur de la condition est différente de 0, sinon inst1 est exécutée.

Forme 3:

```
switch ( expression  ){
     case const0: inst0;
     case const1: inst1;
     ……
    case constn: instn;
   default: Inst;
 }
```

Notes

- expression est une expression arithmétique;
- inst, inst0,… instn représentent chacune une instruction simple ou un bloc d'instructions et,
- const0, … constn des constantes.
- le **default** est facultatif. Toutefois, s'il est utilisé, il doit être mis à la fin.
- La variable ou le terme *expression* est d'abord évaluée, puis le cas correspondant parmi consta0, const1,…, constn ou le **default** est exécuté selon le résultat de cette évaluation.

Notez que les instructions suivant l'exécution du cas correspondant sont également exécutées. Pour éviter que toutes les possibilités d'un **switch** ne soient exécutées, l'instruction break peut être utilisée.

Exemple 16

```
switch (n){
  case 0: cout << '0';
  case 1: cout << '1';
  case 2: cout << '2';
  default: cout << '3';
}
```

Si n vaut 1 alors le programme affichera 123

```
switch (n)
{
 case 0: cout << '0'; break;
 case 1: cout << '1'; break;
 case 2: cout << '2'; break;
 default: cout << '3';
}
```

Si n vaut 1 alors le programme affichera 1.

b. Instruction de répétition : elle consiste à répéter une instruction ou un bloc d'instructions tant qu'une condition est vraie. On en distingue trois formes. Noter que si ct un bloc d'instructions qui est exécuté à l'intérieur d'une boucle, il ya lieu de mettre ce dernier entre **{ }**.

b-1. Boucle while: la syntaxe est comme suit :

```
while (condition)
    < instruction ou bloc>;
```

b-2. Boucle for: la syntaxe est comme suit:

```
for (<expr1>;<expr2>;<expr3>)

    <instruction ou bloc>;
```

- <expr1> est une expression d'initialisation exécutée une seule fois (au début de l'exécution de la boucle.
- <expr2> est un test (d'arrêt) exécuté avant chaque itération.
- <expr3> est une expression, modifiant au moins une variable de <expr2>, exécutée à la fin de chaque itération.
- Noter que <expr1>, <expr2> ou <expr3> peuvent être vides.

```
for ( ; ; ) /* boucle infinie ! */
   ;
```

b-3. Boucle do … while: la syntaxe est comme suit:

```
do
    < instruction ou bloc>;
 while (condition);
```

Pour éviter toute ambiguité (la dernière ligne peut être interprétée par le lecteur comme étant une boucle **while** ne comportant pas d'instructions), on utilise la fome suivante :

```
do {
      instruction(s);
    }
while (condition);
```

Note: prenez garde au ; de la fin de cette instruction.

Exemple 17

```
int s = 0; i = 1;
while (i < n){
   s = s*i;
   i++;
  }
```

```
int s =0;
for (i=1; i<n; i++)
      s = s*i;

int s =0; i=1;     ou alors  int s = 0; I = 1;
do                                do {
  s = s*i;                          s = s*i;
  i++;                             }
while (i< n);                       while (++i < n);
```

Saut inconditionnel dans une boucle

Dans certaines situations, faire sauter à la boucle une ou plusieurs valeurs sans pour autant mettre fin à celle-ci peut être nécessaire. On peut le faire avec l'instruction « *continue;* » qui doit se placer dans une boucle. Cette commande est associée à une structure conditionnelle, autrement les lignes situées entre cette commande et la fin de la boucle sont obsolètes.

Exemple 18

```
n =1;
while (n <= 10) {
          if (n == 1) {
                    cout << "Ooops : il ya une division par zéro !";
                    continue;
          }
         double a = 1/(n-1);
         cout << a;
         n++;
}
```

Si la condition n=1 est vraie, avec le continue, on s,en va directement à la fin de la boucle. Toutefois, comme c'est écrit ci-dessus, quand n = 1, le compteur ne s'incrémente plus, il reste constamment à la valeur 1. Il aurait donc fallu écrire :

```
n = 1;
while (n <= 10) {
          if (n == 1) {
                    cout << "ooops : division par 0";
                    n++;
                    continue;
          }
          double a = 1/(n-1);
          cout << a;
          n++; }
```

Arrêt conditionnel dans une boucle

À l'inverse de ce qui a été dit ci-dessus, dans certaines situations, il peut être voulu d'arrêter prématurément une boucle. L'instruction ***break*** permet de le faire. Par exemple, pour éviter une division par zéro, on aurait procéder comme suit :

```
n = 1;
while (n <= 10) {
        if (n == 1) {
                cout << "ooops : division par 0";
               break ;
        }
        double a = 1/(n-1);
        cout << a;
        n++; }
```

C. Fonctions: Une fonction est un petit programme qui consiste à résoudre une tâche bien spécifique. La description d'une fonction se divise en deux parties: l'entête et le corps:

```
int modulo(int x, int y){
        return x % y;
 }
```

- L'entête spécifie les paramètres d'entrée et le type retourné par la fonction.
- Le corps de la fonction définit le travail qu'elle effectue.
- L'instruction return est l'instruction de sortie (qui détermine la valeur retournée).
- return (expression) est exécutée de la façon suivante:
    1. L'expression est d'abord évaluée.
    2. Le résultat est retourné à la fonction appelante.
    3. Toutes les variables créées après l'appel de la fonction sont détruites.
    4. Le contrôle est redonné à la fonction appelante.

Pour retourner donc le résultat de la fonction, on utilise le mot clé return suivi du nom de la variable. Dès la rencontre de return, la fonction évalue la valeur qui la suit, et la renvoie au programme appelant. Une fonction peut contenir plusieurs instructions *return*, ce sera toutefois la première instruction *return* rencontrée qui provoquera la fin de la fonction et le renvoi de la valeur qui la suit Si la fonction ne retourne aucun résultat (en précédant la fonction d'un **void**), la commande return peut être omise. La syntaxe de l'instruction return est simple :

return valeur_ou_variable;

Le type de valeur retourné doit correspondre à celui qui a été précisé dans la définition (et le prototype)

```
int addition(int a, int b) {
  int resultat;
  resultat = a+b;
  return (resultat);
}
```

```
// déclaration d'un entier

// opération arithmétique utilisant les arguments
// renvoi du résultat
```

```
void affiche(void){
cout << " pas de return ici" << endl;
// ou alors ajouter return;
}
```

```
// ou void affiche()
// affichage d'un message à l'écran
// le return est omis
```

Il existe deux types de paramètres pour une fonction :

- **Paramètres formels**: Ceux utilisés dans la définition de la fonction.

- **Paramètres d'appel**: Ceux utilisés lors de l'appel de la fonction.

Passage de paramètres

Il est possible de passer des arguments (paramètres aussi) à une fonction donnée (lui fournir une valeur ou le nom d'une variable pour que cette dite fonction puisse effectuer des opérations). Ce passage se fait au moyen d'une liste d'arguments. Un argument peut-être :

- Une constante.
- Une variable.
- Une expression.
- Une autre fonction.

Lorsque l'on passe une variable en paramètre d'une fonction, celle-ci utilise une copie de cette variable quand elle effectue des opérations sensées la modifier. Autrement dit, en sortie de la fonction, une variable passée en paramètre n'est pas modifiée. On dit que le passage s'est fait par copie.

1) Passage par copie (par valeur): Les paramètres d'appel sont copiés dans les paramètres formels. Il y a dans ce cas création de nouvelles variables.

Une solution qui consiste à modifier ces paramètres et de retourner la valeur modifiée et de la stocker par affectation dans la variable elle-même :

```
int ajouter (int a)
{
  a+=1;
  return;
}

int b = 1;

b = ajouter (b);
```

Néanmoins, il se pourrait que cette valeur de retour soit destinée à une autre opération. Dans ce cas, on pourrait utiliser un pointeur (voir plus loin) vers la variable en paramètre. On parle dans ce cas, de passage par adresse.

2) Passage par adresse : Les paramètres d'appel réfèrent par adressage aux paramètres formels**:** plusieurs noms pour une même case mémoire.

Dans ce cas, il s'agit de déclarer un paramètre de type pointeur et passer l'adresse de la variable au lieu de passer la variable elle-même. En reprenant l'exemple ci-dessus :

```
int ajouter (int *a)
{
  a+=1;
  return;
}

int b = 1;

ajouter (&b);
```

3) Passage par référence : Le langage C++ allie les avantages de passage par pointeur avec la simplicité de passage par valeur, grâce à l'emploi du concept **reference**. Les paramètres formels réfèrent aux paramètres d'appel: plusieurs noms pour une même case mémoire. Toutefois, même si le résultat est le même que précédemment, il y a une nuance. En effet, dans ce cas, on fait juste référence à une variable qui existe déjà. Autrement dit, une référence doit obligatoirement être initialisée lors de sa déclaration. La déclaration d'une référence se fait via l'utilisation de &.

En reprenant l'exemple précédent :

```
int ajouter (int &a)
{
  a+=1;
  return;
}

int b = 1;

ajouter (b);
```

La notion de référence permet au compilateur de mettre en œuvre les bonnes instructions pour assurer effectivement un transfert par adresse. C'est le compilateur qui prend en charge la transmission des arguments par adresse. Le même mécanisme pourra s'appliquer à une valeur de retour.

Lors d'un passage par référence, nous pouvons manipuler à l'intérieur de la fonction la valeur d'une variable externe passée en paramètre. Pour pouvoir passer une variable par référence, Il suffit d'ajouter le symbole **&** devant un paramètre de la fonction à la suite du type. Ce symbole permet de passer la variable elle-même et non une copie. Ainsi **&X** signifie référence à la variable **X**.

Exemple 19

```
#include <iostream.h>

int main ()
{
        void echange ( int &, int & ) ;

        int n = 10, p = 20 ;

        cout << "avant l'appel : " << n << " " << p << endl ;
        echange ( n, p ) ;                  // noter l'absence de l'opérateur &
        cout << "aprés l'appel : " << n << " " << p << endl ;
        return 0 ;
}

void echange ( int & a, int & b ){
        int c ;
        cout << "début, change : " << a << " " << b << endl ;
        c = a;
        a = b;
        b = c;
        cout << "fin, change : " << a << " " << b << endl ;
        return ;
}
```

La notation **int & a** signifie que **a** est une information de type entier transmise par référence. Noter que dans la fonction *echange(),* on utilise le symbole **a** pour désigner cette variable dont la fonction aura reçu effectivement l'adresse (non la valeur). En fait, il suffit de faire le choix de transmission au niveau de l'en-tête de la fonction pour que le compilateur prenne en charge le processus.

Pour transmettre en argument la référence à un pointeur, on écrit: **int * & adr** ; Dans ce cas, **adr** est une référence à un pointeur sur un entier. Ce mécanisme s'applique également à la valeur de retour d'une fonction.

En dehors des arguments de fonctions, il est possible de déclarer un identificateur comme référence d'une autre variable.

Exemple 20

```
int n ;
int & p = n ;
p = 5;
```

Dans ce cas, la variable *p* est une référence à la variable *n*. Les variables *n* et *p* vont désigner le même emplacement mémoire. La référence est donc un moyen d'associer un nouveau nom à un

objet déjà existant. L'utilisation des références est essentiellement dans la spécification des paramètres et valeurs de retour d'une fonction.

On ne peut pas déclarer une référence sans l'initialiser.

```
int & p ;   // est incorrect car la variable p n'a pas été initialisée.
```

Toute référence doit se référer à un identificateur : il est donc impossible de déclarer une référence sans l'initialiser. De plus, la déclaration d'une référence ne crée pas un nouvel objet comme c'est le cas pour la déclaration d'une variable par exemple. En effet, les références se rapportent à des identificateurs déjà existants.

```
int i=0;
int &ri=i;   // Référence sur la variable i.
ri=ri+i;     // Double la valeur de i (et de ri aussi).
```

Il est possible de faire des références sur des valeurs numériques. Dans ce cas, les références doivent être déclarées comme étant constantes, puisqu'une valeur est une constante :

```
const int &ri=3;  // Référence sur 3.
int &error=4;     // Erreur ! La référence n'est pas constante
```

Note

- Lorsque le paramètre d'une fonction prend beaucoup d'espace mémoire, le passer par référence évite de copier une grande quantité d'information. En ajoutant le mot clé **const**, on rend impossible la modification de ce paramètre.

- Tout programme *C++* débute par l'exécution de la fonction nommée main. Par ailleurs, une fonction peut comporter des variables locales. On distingue la *déclaration* d'une fonction de sa *définition*.

Le prototype d'une fonction est l'entête seule.

```
void modulo(int , int );
```

Lorsque le compilateur transforme le programme *C++* en langage machine, il lit de manière séquentielle ce dernier. Si une des instructions qu'il rencontre comporte l'appel d'une fonction, il doit en connaître le prototype. C'est la raison pour laquelle il faut *déclarer* les fonctions avant de les utiliser. La déclaration d'une fonction consiste simplement à écrire le prototype de celle-ci dans le programme.

Le prototype sert donc à informer le compilateur qu'une fonction donnée (ayant les paramètres spécifiés dans le prototype et retournant une variable du type spécifié) a été définie dans un des fichiers sources du programme.

Pour utiliser une variable ou une fonction, le compilateur doit connaître la convention permettant de l'appeler. Par exemple, le programme suivant produira une erreur de compilation:

```
main() {
  int x;
   x = f(0); /* Le compilateur ne connaît pas encore la  fonction f.*/
 }

int f(int x) {
  return x + 1;
}
```

En revanche, le programme suivant ne produira pas d'erreur lors de la compilation.

```
int f(int);
main() {
   int x;
   x = f(0);
 }

int f(int x){
   return x + 1;
 }
```

Un bon découpage du problème à programmer en fonctions est important. Par ailleurs, il est conseillé d'avoir des noms aux fonctions par rapport au travail qu'elles effectuent (tout comme pour les variables d'ailleurs).

Note : Plusieurs fonctions mathématiques courantes font partie de la bibliothèque standard de C++. Pour les utiliser, il suffit d'inclure au début du programme la ligne suivante:

#include <math.h>

Il est alors possible d'employer des fonctions telles que:

cos(x)	cosinus de x
pow(x,y)	x à la puissance y
sqrt(x)	racine carrée de x

et plusieurs autres fonctions.

Fonctions récursives : Il est possible pour une fonction donnée de s'appeler elle-même : soit d'une manière directe, soit d'une manière indirecte. Si c'est le cas, on dit que la fonction est récursive. Par exemple, pour calculer la somme des n premier nombres entiers, on peut procéderd'une manière récursive comme suit :

```
int mystere (int n) {

if (n = = 1)  return 1;
else return(mystere (n-1) + n);
}
```

N.B : On verra un peu plus en détails la récursivité dans le chapitre: analyse des algorithmes.

4.2 Les arguments par défaut d'une fonction

Le langage C++ permet d'attribuer des valeurs par défaut à des arguments.

Exemple 21

```
#include <iostream.h>
int main ()
{
        int n = 10, p = 20 ;

        void fonct ( int, int = 12 ) ;   // prototype avec une valeur par défaut

        fonct ( n, p ) ;  // appel normal
        fonct ( n ) ;     // appel avec un seul argument
        return 0 ;
}

void fonct ( int a, int b )
{
        cout << "premier argument : " << a << endl ;
        cout << "deuxième argument : " << b << endl ;
        return ;
}
```

Exécution

```
premier argument : 10
deuxième argument : 20
premier argument : 10
deuxième argument : 12
```

```
void fonct ( int, int = 12 ) ;
```

Ce prototype précise au compilateur qu'en cas d'absence du deuxième argument, il faut prendre la valeur 12.

void fonct (int = 0, int = 12) ;

Les appels suivants sont légaux: fonct (10, 20) ; fonct (10) ; fonct () ;
Dans le dernier cas le compilateur prend 0 pour le premier argument et 12 pour le deuxième.

Note : Seuls les derniers paramètres peuvent avoir une valeur par défaut. Par exemple la définition suivante n'est pas correcte :

```
int mult(int x=3, int y){
  int z=1;
  for (int i=0; i<y; i++) z*=x;
  return z;
}
```

4.3 La surcharge des fonctions : En C++, une même fonction peut être définie de plusieurs façons, comme le montrent les exemples suivants:

```
int moyenne(int a, int b){
    return (a+b)/2;
}

double moyenne(double a, double b){
    return (a+b)/2.0;
}

int mult(int x){
    return x*x;
}

int mult(int x, int y){
     int z=1;
    for (int i=0; i<y; i++) z*=x;
    return z;
}
```

Un appel à mult(3) retournera 9 et un appel à mult(3,3) retournera 27.

Fonction en ligne : Les fonctions en ligne permettent d'utiliser de petites fonctions lorsque, pour des raisons d'efficacité, le programmeur ne veut pas appeler une fonction pour effectuer les quelques instructions qui implémentent une opération (**le compilateur fait en fait une copie du code au lieu d'un appel).**

```
inline int concat (char ch1, char ch2) {
      return ch1+ch2 ;
}
```

Lorsqu'un programme est chargé, son exécution commence par l'appel d'une fonction spéciale du programme. Cette fonction doit impérativement s'appeler « `main` » (*principal* en anglais) pour que le compilateur puisse savoir que c'est cette fonction qui marque le début du programme. La fonction `main` est appelée par le système d'exploitation, elle ne peut pas être appelée par le programme, c'est-à-dire qu'elle ne peut pas être récursive.

Exemple 1-21. Programme minimal

```
int main()   /* Plus petit programme C/C++. */
{
    return 0;
}
```

La fonction main

Lorsqu'un programme est chargé, son exécution commence par l'appel d'une fonction spéciale du programme. Cette fonction doit impérativement s'appeler « `main` » (pour *principal* en anglais) pour que le compilateur puisse savoir que c'est cette fonction qui marque le début du programme. La fonction `main` est appelée par le système d'exploitation, elle ne peut pas être appelée par le programme, c'est-à-dire qu'elle ne peut pas être récursive.

```
int main()   /* Plus petit programme C/C++. */
{
   return 0;
}
```

La fonction **`main`** doit renvoyer un code d'erreur d'exécution du programme, le type de ce code est **int**. Elle peut aussi recevoir des paramètres du système d'exploitation. Ceci est expliqué plus en détails dans le fichier « **Informations sur la fonction main »** mis sur ce site web.

Note : Il est spécifié dans la norme du C++ que la fonction `main` ne doit pas renvoyer le type void. En pratique cependant, beaucoup de compilateurs l'acceptent également. La valeur *0* retournée par la fonction **`main`** indique que tout s'est déroulé correctement. En réalité, la valeur du code de retour peut être interprétée différemment selon le système d'exploitation utilisé. La bibliothèque C définit donc les constantes *EXIT_SUCCESS* et *EXIT_FAILURE*, qui permettent de supprimer l'hypothèse sur la valeur à utiliser respectivement en cas de succès et en cas d'erreur.

5. Les structures

Le langage C/C++ permet la définition de types personnalisés construits à partir des types de base du langage. À l'aide de la notion de structure, il est possible de définir différents types de données évolués. On commence par voir les structures dans cette section, Mais, on en verra d'autres par la suite (comme les classe, tableaux, pointeurs, etc.).
Les structures sont construites en regroupant des éléments d'autres types. Elles sont définies par l'utilisateur. Sa syntaxe est la suivante :

```
struct [nom_structure]
{
   type champ;
  [type champ;
  [...]]
};
```

Exemple 22

```
struct Temps{
        int heure, minute ;
        int seconde ;
};
```

Cette définition ne réserve aucun espace en mémoire, elle crée un nouveau type. Les variables de types structures sont déclarées comme les variables des autres types.

Exemple 23

```
Temps tempsObjet, tempsTableau[10], *tempsptr,
      &tempsRef = tempsObjet ; // deux noms pour une même case mémoire
                               // (c'est ce qu'on appelle une référence);
```

En C++, il est possible que les structures contiennent des fonctions

```
struct point{
  double x;
  double y;
};
struct segment{
 point p1;
 point p2;
 double longueur(){
     double x = p1.x - p2.x;
     double y = p1.y - p2.y;
     return sqrt(x*x + y*y)
  }
}; // fin de la déclaration de segment
```

L'accès aux élément d'une structure se fait à l'aide de l'opérateur **.** , en invoquant le nom de la structure, suivi du point (.) et ensuite du champ à accéder.

L'utilisation des structures se fait comme suit :

Exemple 24

```
segment s={{1,2},{4,6}};
```

```
cout<<s.longueur()<<endl;

segment t;
t=s;
t.p1.x=3.5;
t.p2.y=7.8;
cout<<s.longueur() - t.longueur()<<endl;
```

Exemple 25

```
struct segment{
  point p1;
  point p2;
  double longueur();
 };
double segment::longueur(){
        double x = p1.x - p2.x;
        double y = p1.y - p2.y;
        return sqrt(x*x + y*y);
  }
```

Remarque: Dans le second exemple, la fonction longueur est déclarée dans la structure, mais elle est définie à part, en utilisant le symbole **::**

Le symbole :: dans segment::longueur s'appelle un opérateur de résolution de portée en C++. Il sert à modifier la portée d'un identificateur. Ici, il signifie que l'identificateur initialise dont on parle est celui défini dans point. Sans le préfixe point::, longueur serait une fonction ordinaire et non plus la fonction membre de la structure point.

Exemple 26

```
#include <iostream.h>

// déclaration du type point

struct point{    // déclaration des données
        int x ;
        int y ;
                // déclarations des fonctions membres
        void initialise ( int, int ) ;
        void deplace ( int, int ) ;
        void affiche () ;
};
// définition des fonctions membres du type point
```

```
void point::initialise ( int abs, int ord ){
        x = abs ;
        y = ord ;
        return ;
}


void point::deplace ( int dx, int dy ){
        x += dx ;
        y += dy ;
        return ;
}
void point::affiche (){
        cout << "Je suis en " << x << " " << y << endl ;
        return ; }
int main (){
        point a, b ;
        a.initialise ( 5, 2 ) ;
        a.affiche () ;
        a.deplace ( -2, 4 );
        a.affiche () ;
        b.initialise ( 1, -1 ) ;
        b.affiche () ;
        return 0 ;
}
```

Exécution

```
Je suis en 5 2
Je suis en 3 6
Je suis en 1 -1
```

Noter que l'appel des fonctions membres est toujours préfixé par le nom de la structure correspondante.

6. Notion de classe

En C++, la notion de classe permet de construire des nouveaux types abstraits. Une classe contient des données (données membres) ainsi que la série de fonctions (fonctions membres ou méthodes) qui manipulent ces données. En programmation orientée objet pure, les données sont encapsulées et leur accès ne peut se faire que par le biais des méthodes. C++ vous autorise à n'encapsuler qu'une partie des données (démarche déconseillée). Comme une classe n'est qu'un simple type défini par l'utilisateur, les objets possèdent les mêmes caractéristiques que les variables ordinaires. Les classes sont comme les structures. Elles sont utilisées afin d'isoler certains membres du reste du programme.

6.1 Déclaration d'une classe

La déclaration d'une classe s'inspire de la déclaration des structures en C. Voici la déclaration d'une classe A ayant un attribut entier **i** et une méthode **afficher** qui affiche l'objet sur la sortie standard : (attention au point virgule à la fin de la déclaration de la classe, il est obligatoire).

```
class A {
  public:
    int i;
    void afficher() {
    cout << "i = " << i << endl;
  }
};
```

Instanciation, accès aux membres (publics) : Pour déclarer une instance nommée x de la classe A, il suffit d'écrire :

```
A x;
```

L'accès aux attributs et aux méthodes (publiques) de l'objet s'effectue avec l'opérateur **.**

```
x.i = 10;     // Accès direct à l'attribut i de l'objet a (de classe A)
x.afficher(); // Appel de la fonction membre afficher de l'objet a
```

Une méthode d'instance (ou d'objet) a accès directement aux attributs de l'objet. Par exemple, dans la classe A précédente, la méthode afficher accède directement à l'attribut i de l'objet courant.

6.2 Méthodes en ligne, méthodes déportées

Il faut distinguer la déclaration d'une fonction (membre ou non) de sa définition. La déclaration consiste à signaler à une portion de code que la fonction existe. Elle est faite en générale dans les fichiers d'en-tête et elle consiste juste à donner le prototype de la fonction (qu'elle soit dans une classe, et il s'agit alors d'une méthode, soit à l'extérieur de toute déclaration de classe, et il s'agit alors d'une fonction globale).

La définition consiste à définir la fonction, c'est à dire à spécifier le corps de la méthode. Ceci ne doit apparaître qu'une seule fois dans tout le programme. Lorsque la définition d'une méthode est faite lors sa déclaration, on parle de méthode en ligne, ce qui est le cas pour la méthode afficher de la classe A précédemment déclarée.

A l'inverse, lorsque la méthode est définie séparément de sa déclaration (i.e. à l'extérieur de la déclaration de la classe), on parle de méthode déportée : cela est très courant lors de la compilation séparée. Cela permet de bien séparer la déclaration d'une classe de son implémentation (qui est transparente pour celui qui utilise la classe).

Réécrivons la classe A en déportant la définition de sa méthode **afficher** :

```
// Déclaration de la classe A
  class A {
     public:
            int i;
            void afficher();
   };

// Définition de la méthode afficher de la classe A

void A::afficher() {
  // Affichage de l'attribut i sur la sortie standard
  cout << "i = " << i << endl; }
```

Pour spécifier qu'il s'agit de la méthode **afficher** de la classe A, il faut faire précéder le nom de la méthode par le nom de la classe, les deux identifiants étant séparés par ::. Ainsi A::afficher() désigne la méthode afficher de la classe A.

6.3 Visibilité des attributs et des méthodes

L'encapsulation est un mécanisme consistant à rassembler les données et les méthodes au sein d'une structure en cachant l'implémentation de l'objet, c'est-à-dire en empêchant l'accès aux données par un autre moyen que les services proposés. L'encapsulation permet donc de garantir l'intégrité des données contenues dans l'objet. Pour se conformer à ce principe d'encapsulation, il est possible de masquer des attributs et des méthodes au sein d'un objet. On utilise pour cela les mots clés **private** et **public** : les membres (attributs ou méthodes) qui sont dans la section **private** ne sont accessibles que par les méthodes de l'objet lui-même et les méthodes des objets de la même classe, tandis que les membres **public** sont visibles par tous. Notons qu'une structure est classe où tous ses attributs sont dans la section public.

```
class B {
private:
    int attribut_prive;
    void methode_privee();
public:
    int attribut_publique;
    void methode_publique();
};
int main() {
  B b;                  // Déclaration d'un objet b de classe B
  b.attribut_publique = 0; // OK
  b.attribut_prive = 0;    // Erreur de compilation
  b.methode_publique();    // OK
  b.methode_privee();      // Erreur de compilation
}
```

Ainsi, un attribut **public** pourra être modifié par tous et l'objet ne pourra finalement pas contrôler les modifications effectuées sur son propre attribut. Pour éviter ce manque de contrôle, il est préférable de toujours protéger les attributs et de mettre en place une méthode permettant de récupérer la valeur d'un attribut et une méthode permettant de modifier la valeur d'un attribut (si l'on veut modifier cet attribut bien sûr). Notons que les mots clés **private** et **public** peuvent apparaitre plusieurs fois dans une classe. Un attribut (membre ou donnée) est **private** ou **public** jusqu'à la rencontre d'un autre mot clés. Par défaut (i.e. si rien n'est mentionné), c'est **private** qui est utilisé.

Note

- Si aucun de **private** et **public** n'apparaît dans une déclaration de classe, c'est **private** qui est pris par défaut.
- Si l'on rend **public** tous les membres d'une classe, on obtient l'équivalent d'une structure, discutée précédemment.
- Il existe également un troisième mot clé ***protected***. Qui est en rapport avec le concept d'héritage; ce concept est discuté dans un autre cours qu'est celui de la programmation orientée objet.
- Dans la description d'un nouveau type abstrait de données, le compilateur a besoin de deux types d'information : la représentation de l'objet et son comportement, c'est à dire l'ensemble des fonctions pouvant manipuler cette représentation. Dans le concept de classe, la représentation est déclarée dans la partie privée de la classe (données) et le comportement correspond à la partie des fonctions membres. Tout accès aux données se fait via les fonctions d'accès. L'intérêt de cette encapsulation est que la modification de la représentation d'un objet n'affecte pas le code qui utilise ces objets.

Écrivons à nouveau la classe B :

```
// Déclaration de la classe B
class B {
private:
  int attribut1;
  double attribut2;
public:
  // Méthode qui renvoie la valeur de l'attribut1
  int getAttribut1();

  // Méthode qui affecte la valeur passée en paramètre à l'attribut1
  void setAttribut1(int v);
  // Méthode qui renvoie la valeur de l'attribut2
  double getAttribut2();
  // Méthode qui affecte la valeur passée en paramètre à l'attribut2
  void setAttribut2(double v);
};
// Corps des méthodes déportées (définition des méthodes)
int B::getAttribut1() {
  return attribut1;
}
```

```
void B::setAttribut1(int v) {
  attribut1 = v;
}

double B::getAttribut2() {
  return attribut2;
}
void B::setAttribut2(double v) {
  attribut2 = v;
}
```

6.4 Les fonctions amies

Il est parfois nécessaire de donner la possibilité à des fonctions ou classes non membres d'une classe d'accéder à des données privées de cette classe. Pour cela, on dispose de l'opérateur **friend.** Ces fonctions ou classes sont définies dans la section publique de la classe.

```
class Moi {
public:
    ...
   friend class MonAmie ;
   frient int fonctionAmie () ;
private:
    ...
} ;
```

6.5 Construction et destruction des objets

6.5.1 Le constructeur

À la déclaration d'un objet, les attributs ne sont pas initialisés. Il faut donc initialiser les attributs de l'objet avant de pouvoir manipuler l'objet (comme en C, il ne faut pas manipuler une variable sans qu'elle soit initialisée). Cette phase d'initialisation des attributs est effectuée par une méthode particulière : le constructeur. Le constructeur est une méthode appelée lors de la construction de l'objet.

Le constructeur est une méthode ayant le même nom que la classe et qui n'a pas de type de retour. Il peut prendre des paramètres, il faudra alors spécifier un paramètre lors de la déclaration d'un objet. Une classe peut définir plusieurs constructeurs avec des paramètres différents (il y a surcharge de méthode).

```
// Déclaration de la classe Etudiant avec
// pour seul attribut la moyenne de l'etudiant

class Etudiant {
private:
        double moyenne;
```

```
public:
  // Constructeur sans paramètre (constructeur par défaut)
  Etudiant() {
    moyenne = 0;
  }
  // Constructeur avec un paramètre
  Etudiant(double note) {
    moyenne = note;
  }



 // Méthodes d'accès à l'attribut moyenne
  double getMoyenne() { return moyenne; }
  void setMoyenne(double note) { moyenne = note; }
};

int main() {
  // Déclaration d'un objet avec appel du constructeur par défaut
  Etudiant bob;      // La moyenne de l'étudiant est initialisé à 0
  // Déclaration d'un objet avec appel du constructeur avec un paramètre
  Etudiant john(10); // La moyenne de l'étudiant est initialisée à 10
}
```

6.5.2 Le destructeur : C'est est une fonction spéciale, membre d'une classe, dont le nom est le nom de la classe précédé du caractère tilde (~). Le destructeur d'une classe est appelé lorsque l'exécution du programme quitte la portée dans laquelle un objet de cette classe a été instancié.

Le destructeur est une méthode appelée lorsque l'objet est détruit. Le destructeur doit être impérativement écrit pour libérer la mémoire allouée de façon dynamique pendant la vie d'un objet. Cependant ce n'est pas le destructeur qui libère la mémoire prise par l'objet lui-même : celle-ci sera libérée automatiquement après l'exécution du destructeur.

Le destructeur est désigné par le nom de la classe précédé de ~ , il n'a pas de paramètre, ni de type de retour. Un destructeur ne reçoit aucun paramètre et ne renvoie aucune valeur. Il n'y a qu'un seul destructeur par classe.

```
class A {
  private:
     ...
  public:
    A();   // constructeur par défaut
    ~A();  // destructeur
     ...
 };
```

Note : Lorsque ces fonctions (constructeurs et destructeurs) ne sont pas incluses dans une classe, le compilateur en génère pour nous par défaut. Cependant, la version par défaut ne convient pas toujours; notamment dans le cas d'objets utilisant l'allocation dynamique.

6.5.3 Durée de vie d'un objet

Le destructeur est appelé lorsque l'objet est détruit. Quand un objet est-il détruit ? Dans le cas d'une instanciation automatique (c'est à dire non dynamique), l'objet est détruit à la fin du bloc dans lequel il a été déclaré. Dans la portion de code suivante :

```
int main() {
  A a1;
  for(int i = 0; i<10; i++) {
    A a2;
    a2.afficher();
  } // Destruction de a2
  a1.afficher();
} // Destruction de a1
```

L'objet a1 est détruit après la dernière instruction du main, et l'objet a2 est détruit à la fin de chaque itération de la boucle for.

6.5.4 Les templates : elles permettent de définir des types génériques. L'objectif est de rendre les fonctions et les structures indépendantes du ou des types de données manipulés. Un patron est utilisé par le compilateur pour l'instancier sur des types particuliers.
Par exemple, une fonction qui calcule le maximum sur un tableau. Pour l'utiliser sur plusieurs types, deux choix sont possibles : (a.) surcharger la fonction pour chaque type nécessaire, (b.) créer un patron, le compilateur instanciera alors celui-ci pour chaque type employé dans le programme.

Pour créer un template, on préfix la déclaration de la structure ou de la fonction par

```
template<class T1, ..., class Tn>
```

où T1, ... Tn représentent les types qui seront employés dans les instanciations.

Dans le reste de la fonction ou de la structure, on empoie T1, ... Tn comme n'importe quel type '**normal**'. Pour instancier un template, on précise les types à employer après le nom de celui-ci.

Exemples 27

```
Point<int> p;
afficher<double>(10.2);
```

Pour une fonction, on peut omettre la spécification des types quand cela est non-ambigu. Le compilateur va instancier les types adéquats.

```
template<class T>
```

```
struct Complexe{
  T re;
   T img;
};
Complexe<int> cpxEntier = {1, 2};
Complexe<double> cpxDouble = {1.0, 2.0};



// Sur une fonction
template<class T> T max(const T &a, const T &b){
   return (a < b) ? b : a;
}
int a = max<int>(3, 2);
// Type explicite

double b = max(10.0, 12.0);
// Type implicite
```

Exemple 28

```
#include <iostream.h>

template <class T>

T GetMax (T a, T b) {

  T result;

  result = (a>b)? a : b;

  return (result);

}

int main () {

  int i=5, j=6, k;

  long l=10, m=5, n;

  k = GetMax<int>(i,j);

  n = GetMax<long>(l,m);

  cout << k << endl;

  cout << n << endl;

  return 0;

}
```

Exemple 29

```
class pileEntier {
       int *ppile ;
       int *base ;
       int taille ;
public:
  pileEntier(const int t=0) ;
  ~pileEntier(void) ;
  int depile(void) ;
  void empile(const int valeur) ;
  int estVide(void) const ;
  int estPleine(void) const ;
} ;
inline pileEntier::pile(const int t)
{ base = ppile = new int [t] ; }

inline pileEntier::~pile(void) { delete[] base ; }

int pileEntier::depile(void)  { return *(--ppile) ; }


void pileEntier::empile(const int valeur)
    { *(ppile(++)=valeur ; }

int pileEntier::estVide(void) const { return (ppile=base) ;}

int pileEntier::estPleine(void) const  {
  return (ppile - base)==taille ; }
```

La classe pileEntier est définie comme ci-dessus. Le principe reste identique quelque soit le type de donnée à empiler. Avec cette structure, si on a besoin d'une pile de caractères, il est nécessaire d'écrire une nouvelle classe pileCar dont le code sera pratiquement identique à celui de pileInt. La généricité répond à ce problème, en offrant une classe pile générique indépendant du type de données :

```
template<class T>
class pile {
       T *ppile ;
       T *base ;
       int taille ;
public:
  pile(const int t=0) ;
  ~pile(void) ;
  T depile(void) ;
  void empile(const T valeur) ;
  int estVide(void) const ;
```

```
int estPleine(void) const ;
} ; // fin de la déclaration

template<class T>
inline pile<T>::pile(const int t)
{ base = ppile = new T [t] ; }

template<class T>
inline pile<T>::~pile(void) { delete[] base ; }

template<class T>
T pile<T>::depile(void)  { return *(--ppile) ; }

template<class T>
void pile<T>::empile(const T valeur) { *(ppile(++)=valeur ; }


template<class T>
int pile<T>::estVide(void) const { return (ppile=base) ;}

template<class T>

int pile<T>::estPleine(void) const
   { return (ppile - base)==taille ; }
int main(void){
  pile<int> pi ;
  pile<char> pc ;
  while (! pi.estPleine() ) pi.empile(5) ;
  ...
}
```

On peut retrouver la généricité pour des fonctions génériques telle la fonction d'échange de deux valeurs.

```
template<class T>

void echange(T &a, T &b) {
  T temp = a ;
  a = b ;
  b = temp ;
}
```

7. Allocation dynamique

Il est possible en C++ d'allouer des variables de manière dynamique. L'allocation dynamique permet d'allouer des variables au moment de l'exécution du programme pour tenir compte d'événements non déterministes. Par exemple, un programme d'inversion de matrices de taille quelconque lues à partir d'un fichier pourra réserver une zone mémoire dont la taille est suffisante pour contenir la matrice à inverser. Le tas est une zone mémoire attribuée au programme à cette fin.

Pour exécuter un programme, ce dernier doit nécessairement se situer en mémoire vive de la machine. Lorsqu'il est chargé en mémoire vive pour y être exécuté, un programme se fait attribuer un *espace mémoire* par le système d'exploitation. L'espace mémoire est une suite de cellules mémoires dont la taille varie selon le système sur lequel l'utilisateur travaille. Sur la plupart des ordinateurs fonctionnant en 32 bits, l'espace mémoire peut être vu comme un tableau continu d'octets dont les indices sont des entiers de 32 bits. L'espace mémoire alloué à un programme se situe physiquement dans les puces de mémoire RAM de l'ordinateur et dans le disque dur (mémoire virtuelle).

L'espace mémoire sert à stocker le programme compilé (en langage machine), les variables, etc. On accède aux cellules mémoire à l'aide de leur *adresse*. L'adresse d'une cellule est une valeur numérique qui sert à l'identifier.

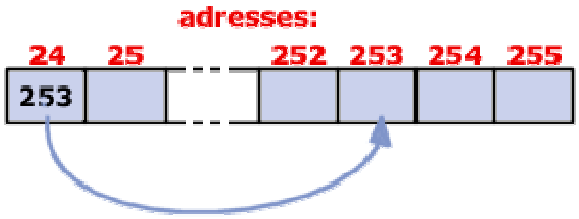
Chaque variable d'un programme est stockée dans certains nombres de cellules successives. Un des rôles du compilateur est de "convertir" la variable en adresse numérique. On pourrait écrire un programme en ne travaillant qu'avec des adresses numériques plutôt que des variables (en fait c'est ce que le processeur fait). Mais ce ne serait pas trop pratique.

Chaque programme a accès à son propre espace mémoire (l'adresse *a* d'un programme P1 correspond à une cellule différente de celle qui correspond à l'adresse *a* d'un programme P2).

La position des variables mémoire est déterminée par le compilateur. De plus, les variables ne sont pas initialisées lorsque le programme démarre. Il faut le faire explicitement.

Dans le langage *C++*, il existe un type qui permet de manipuler les adresses : c'est le type *pointeur*.

Un pointeur est une variable qui sert à identifier (*pointer vers*) une cellule mémoire. Les pointeurs sont *typés*; ce qui veut dire que l'on doit spécifier le type de la variable contenue dans la cellule à laquelle réfère le pointeur. Il suffit donc de stocker l'adresse de la variable dans un pointeur (il est prévu pour cela) afin de pouvoir accéder à celle-ci (on dit que l'on « pointe vers la variable »).



Le schéma ci-dessus montre par exemple par quel mécanisme il est possible de faire pointer une variable (de type *pointeur*) vers une autre. Ici le pointeur stocké à l'adresse 24 pointe vers une variable stockée à l'adresse 253 (les valeurs sont bien évidemment arbitraires).

En réalité on n'aura pas à écrire l'adresse d'une variable, d'autant plus qu'elle change à chaque lancement de programme étant donné que le système d'exploitation alloue les blocs de mémoire qui sont libres, et ceux-ci ne sont pas les mêmes à chaque exécution. Ainsi, il existe une syntaxe permettant de connaître l'adresse d'une variable, connaissant son nom : il suffit de faire précéder le nom de la variable par le caractère & (« ET commercial ») pour désigner l'adresse de cette variable : &Nom_de_la_variable. Pour obtenir le "contenu" d'un pointeur, on utilise * .

Exemple 30

char *pc; /* pc contient une adresse qui pointe vers une cellule qui contient un caractère */

double *pt; /* pt contient une adresse qui pointe vers une cellule qui contient un nombre réel */

Si la variable y est un caractère, alors y = *pc assigne à y le contenu de l'adresse pointée par pc.

Exemple 31

```
int x, y;
int *p, *q;
p = &x; /* L'adresse de x est mise dans p. */
*p = 0; /* le contenu pointé par p (donc par x) change; c'est-à-dire la valeur de x  est mise à 0. */
 p = &y;
*p = 1;
q = &x;
*p = *q;
```

Il faut être prudent dans l'utilisation des pointeurs.

Exemple 32

```
int x, *p;
p = &x;
*(p + 1) = 0; /* peut faire "planter" le programme. */ pourquoi?
```

Tableaux à une dimension: Un tableau correspond à une suite continue de variables d'un type donné.

```
int T1[10]; /* Tableau de 10 entiers */

char T2[314]; /* Tableau de 314 caractères. */

for ( i=0; i<10; i++ )
    T1[i] = 0;

char c = T2[2];

c = T2[500]; /* Danger! */

T2[500] = 0; /* Danger!! */
```

Remarque: Un tableau est en fait un pointeur **constant**.

Exemple 33

```
int tab[10], *p;
p = tab;
/* Les expressions suivantes sont équivalentes. */
tab[3] = 100;
*(tab + 3) = 100;
p[3] = 100;
*(p + 3) = 70;

int tab[10], *p;
p = tab;
p = tab;        /* Valide */
tab = p;        /* Non valide */ pourquoi?
```

Les tableaux à deux dimensions (matrices)

La déclaration d'une matrice à N lignes et M colonnes d'entier se fait comme suit : int mat[N][M]; l'adresse de mat[i][j] est mat+ i*M + j

Pour calculer l'adresse de Tab[i][j], il n'est pas nécessaire de connaître N, mais il est nécessaire de connaître M. Cela explique pourquoi il est nécessaire de préciser M dans les paramètres formels de fonctions.

Exemple 34 : fonct(char mat[N][M]) ou fonct(char mat[][M])

Note : mat correspond à l'adresse &mat[0][0], mais mat[1] est aussi un tableau (une ligne), donc désigne l'adresse &mat[1][0]. En fait, une matrice est un tableau de lignes.

Le C++ met à la disposition du programmeur deux opérateurs *new* et *delete* pour remplacer

- **L'opérateur *new*** réserve l'espace mémoire demandé et l'initialise. Il retourne l'adresse de début de la zone mémoire allouée.
- Tandis que ***delete*** libère l'espace mémoire correspondant déjà alloué.

Exemple 35

```
   int *ptr1, *ptr2, *ptr3; // déclaration de de pointeurs de type entier

   // allocation dynamique d'un entier
   ptr1 = new int;

   // allocation d'un tableau de 10 entiers
   ptr2 = new int [10];

   // allocation d'un entier avec initialisation
   ptr3 = new int(10);

   struct date {
     int jour, mois, an;
    };

 date *ptr4, *ptr5, *ptr6, // déclaration de pointeurs de type date
date* d = {25, 4, 1952};  // déclaration et initialisation d'une variable de type date

// allocation dynamique d'une structure

   ptr4 = new date;

//  allocation dynamique d'un tableau de 10 pointeurs chacun pointant vers un  éléments
//  de type de structure

   ptr5 = new date[10];

 // allocation dynamique d'une structure avec initialisation
   ptr6 = new date(d);
```

L'allocation des tableaux à plusieurs dimensions est possible. Elle peut se faire comme suit :

```
char **m_table; // Déclaration du tableau de caractères à 2 dimensions m_table
int m_nbLig;      // Variable représentant le nombre de lignes
int m_nbCol;     // Variable représentant le nombre de colonnes
```

```
// Allocation mémoire de m_table

m_table = new char *[m_nbLig]; // allouer un tableau de m_nblig pointeurs et chacun de ses
                                //  pointeurs va pointer sur des un tableau de caractères.

for(int i=0; i<m_nbLig; i++)
    m_table[i] = new char [m_nbCol];
```

Exemple 36

```
CString **m_table; // Déclaration du tableau à 2 dimensions m_table
int m_nbLig; // Variable représentant le nombre de lignes
int m_nbCol; // Variable représentant le nombre de colonnes

// Allocation mémoire de m_table
m_table = new CString*[m_nbLig];
for(int i=0; i<m_nbLig; i++)
m_table[i] = new CString[m_nbCol];

// Traitement
// ...
```

La suppression d'espaces mémoires se fait comme suit, selon le type du pointeur en question.

```
// libération d'un pointeur  entier
   delete ptr1;

// libération d'un tableau d'entier
   delete[] ptr2;


// La désallocation mémoire du tableau m_table se fait comme suit :

  for (int i=0; i<m_nbLig; i++)
     delete [] m_table[i];
  delete [] m_table;
```

Attention: ne pas confondre

```
      int *ptr = new int[10]; // création d'un tableau de 10 entiers
```

avec :

```
     int *ptr = new int(10); // création d'un entier initialisé à 10.
```

Pointeur null

On utilise une valeur spéciale pour indiquer qu'un pointeur ne désigne pas une adresse valide. C'est la valeur 0. Si la valeur d'un pointeur est 0, il ne désigne aucune cellule. C'est une convention intégrée dans le langage. En particulier, ceci implique que la cellule à l'adresse 0 ne peut contenir de variable. C'est une bonne pratique d'initialiser les pointeurs avec la valeur 0.

```
int *p = 0;
```

Exemple 37

```
int *p = 0;
/* si le pointeur ne pointe pas vers une cellule valide, */
/* il ne se passera rien. */
if ( p )
   cout << *p;
```

L'application de l'opérateur *delete* à un pointeur nul est légale et n'entraîne aucune conséquence fâcheuse (l'opération est tout simplement ignorée).

Remarque : Si **p** est un pointeur sur une structure et **c** un champ de cette structure, on utilise l'opérateur -> qui est un opérateur de déréférencement spécialisé. Son opérande gauche doit être un pointeur sur une structure. Son opérande droit doit être un champ de cette structure. Nous avons alors l'équivalence suivante:

p->c <=> (*p).c

(*p).c mérite une explication : **p** est un pointeur vers une structure. Quand on écrit ***p**, on manipule ce vers quoi pointe **p**, à savoir la structure elle-même. Ce qui devrait permettre d'écrire ***p.c** Toutefois, l'opérateur **.** a une priorité supérieure à celle de *****, Donc, le compilateur lui comprend : ***(p.c)**, ce qui ne veut rien dire ici. Du coup il y a lieu de mettre des parenthèses pour lui dire ce qu'on veut faire. Comme c'est lourd à écrire, le C/C++ définit un autre l'opérateur -> . Ainsi, **p->c** est strictement équivalent à **(*p).c**.

Pour l'exemple ci-dessus avec ptr5, pour accéder au champ jour du $i^{ème}$ élement, on peut écrire invariablement :

pPtr5[i]->jour ou *(ptr5+i).jour : cela revient au même.

Le pointeur this et les classes

On a parfois besoin de désigner à l'intérieur d'une fonction membre l'objet qui est manipulé par la méthode. Comment le désigner cependant alors qu'il n'existe aucune variable le représentant dans la fonction membre ? Les fonctions membres travaillent en effet directement sur les attributs de classes : ceux qui sont atteints correspondent alors à ceux de l'objet courant. Pour ce faire, on

utilise le mot-clé this, qui permet à tout moment dans une fonction membre d'accéder à un pointeur (et non à une référence) sur l'objet manipulé.

Dans toute fonction d'une classe, disons A, le pointeur this est implicitement déclaré comme suit :

A* const this ;

Il est initialisé à pointer sur l'objet pour lequel la fonction membre est invoqué. Comme ce pointeur est déclaré comme une constante, il ne peut être changé. Toutefois, l'objet pointé le peut. Ainsi, la classe suivante :

```
class A {
private :
 int donnee ;
Public :
    int lecture ( ) {
      return donnee ;
} ;
```

Cette classe est implicitement déclarée comme suit :

```
class A {
private:
int donnee ;
Public :
    int lecture ( ) {
      return this -> donnee ;
} ;
```

L'utilisation de this pour référencer les membres d'une classe n'est pas nécessaire. Son utilisation est essentiellement circonscrite aux fonctions utilisant des pointeurs.

```
class exemple {
 private
      exemple* avant ;
       exemple* suivant :
public :
    void attacher (exemple*) ;
} ;

void exemple ::attacher (exemple* pointeur) {
pointeur -> suivant = suivant ; // pour pointeur -> suivant = this-> suivant
pointeur-> avant =  this ;
suivant->avant = pointeur ; // this->suivant = pointeur
suivant = pointeur ;
}
```

Arithmétique des pointeurs

Toutes les opérations arithmétiques sur les pointeurs tiennent compte de la taille du *type pointé*. Exemple : si p est de type int *, p + 1 ne pointe pas vers la case mémoire suivante, mais vers l'entier suivant. Si la taille du type int est de 4 octets, par exemple, p + 1 correspond à l'adresse contenue dans p plus 4.

Les opérations valides sur les pointeurs sont :

```
pointeur + entier     -->  pointeur
pointeur  - entier    -->  pointeur
pointeur  - pointeur -->   entier
```

- Les deux premières opérations permettent de faire *avancer* ou *reculer* un pointeur. La troisième opération calcule le nombre d'éléments entre les deux pointeurs.
- L'addition de deux pointeurs n'existe pas car elle n'a pas de sens.
- L'opération *différence* sur les pointeurs n'est valide que si les deux opérandes sont de même type.

À retenir

- A chaque instruction **new** doit correspondre une instruction **delete**.
- Il est important de libérer l'espace mémoire dès que celui ci n'est plus nécessaire.
- La mémoire allouée en cours de programme sera libérée automatiquement à la fin du programme.
- Tout ce qui est alloué avec **new []**, doit être libéré avec **delete[].**

Les priorités

Un dernier mot sur la priorité des opérateurs avant passer à une application de pointeurs et clore ce survol du langage C++. Lorsque l'on associe plusieurs opérateurs, il faut que le compilateur sache dans quel ordre il va les traiter. Ci-dessous, dans **l'ordre décroissant,** les priorités des différents opérateurs du langage C++. Notez, cependant, que l'analyse des expressions peut être modifiée en changeant les priorités à l'aide de parenthèses.

Opérateur	**Niveau de priorité**
() [ ] . -> ::	15
*(contenu d'un pointeur) ! ~ ++ -- & (type) sizeof New delete -(moins unaire) + (plus unaire)	14
.* ->*	13
* / %	12
+ -	11
<< >>	10
< <= > >=	9
== !=	8
&	7
^	6
|	5
&&	4
||	3
? :	2
= *= /= %= += -= &= ^= |= <<= >>=	1
,	0

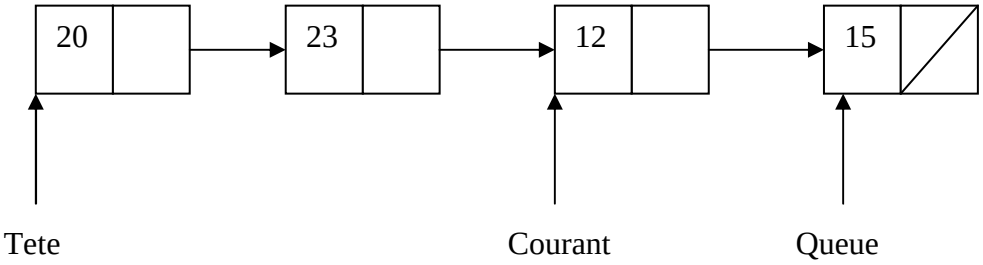
Applications des pointeurs

- Implanter le passage de paramètres par référence
- Implanter le passage de tableaux en paramètres
- Utiliser des indices négatifs aux tableaux
- Fondamental en structure de données

Une application aux listes chaînées

Une liste chaînée est constituée d'une série d'éléments, appelés nœuds, reliés entre eux par des pointeurs. L'exemple ci-dessous est une liste chaînée dont chaque élément est constitué d'une donnée entière et d'un pointeur vers un autre élément de même type.

Exemple 38



Le nœud étant un objet distinct, il est intéressant de définir une classe pour les nœuds.

```
// fichier nœud.h
// déclaration de la classe noeud
```

Note : Pour faire appel à la classe noeud au sein d'un programme, l'utilisateur devra avoir la directive #include "noeud.h" et il incorporera le module objet au moment de l'édition de liens. Cependant, il peut exister des circonstances qui peuvent amener l'utilisateur à inclure plusieurs fois le même fichier en-tête, ce qui peut donner des erreurs de compilation. Pour protéger systématiquement tout fichier en-tête des inclusions multiples, on utilisera la compilation conditionnelle comme suit :

```
#ifndef noeud.h /* raccourci pour if !defined (noeud.h) */
#define nœud
// déclaration de la classe noeud
#endif
```

Par exemple , soient un fichier A.h et un autre fichier B.h. Le fichier A.h contient un #include du fichier B.h. Le fichier B est donc inclus dans le fichier A. Mais, supposez que le fichier B.h contienne à son tour un #include du fichier A.h. Le premier fichier a besoin du second pour fonctionner, et le second a besoin du premier. On imagine vite ce qu'il va se passer :

1. La machine lit A.h et voit qu'il faut inclure B.h
2. Il lit B.h pour l'inclure, et là il voit qu'il faut inclure A.h
3. Donc il inclut A.h dans B.h, mais dans A.h on lui indique qu'il doit inclure B.h !
4. Rebelote, il va voir B.h et voit à nouveau qu'il faut inclure A.h
5. etc …

À force de faire trop d'inclusions, le préprocesseur s'arrêtera et du coup votre compilation plantera**!** Pour éviter cela, l'astuce est comme suit :

#ifndef NOMDUFICHIER // Si la constante n'a pas été définie le fichier n'a jamais été inclus
#define NOMDUFICHIER // définit la constante pour que la prochaine fois le fichier ne soit plus inclus.

/* Mettre ici le contenu du fichier .h */

#endif

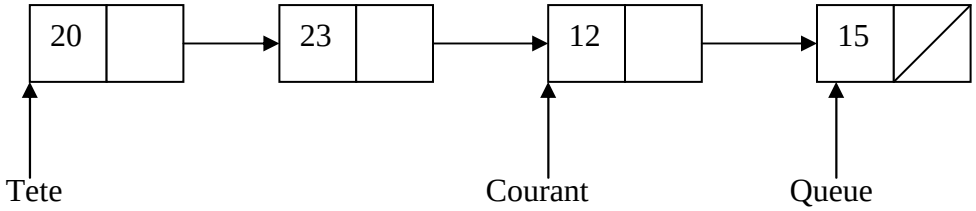
Revenons à notre exemple !

La classe nœud est très simple. Elle contient deux constructeurs : un qui prend une valeur initiale pour la partie information et l'autre non. Elle contient également un destructeur qui ne fait rien de spécial.

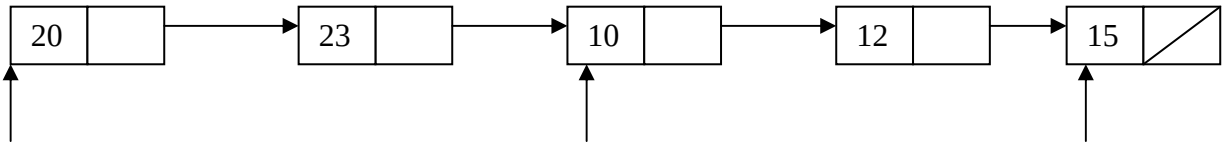
```
#ifndef noeud.h
#define noeud.h

typedef int TElement ; // TElement est de type entier

class noeud{
  public :
        TElement element ;
        noeud * Suivant ;
        noeud ( const TElement & info, noeud * suiv = NULL ) {    // constructeur1
                element = info ;
                Suivant = suiv ;
        }
        noeud ( noeud * suiv = NULL ) { // constructeur 2
                Suivant = suiv ;
        }
        ~noeud () {

        }
} ; // fin de la déclaration
#endif
```

Remarquez le champ **Suivant** qui pointe vers un objet nœud … qui est de même type que celui dans lequelle il est défini (voir exemple 15 ci-dessous).

Dans cette implantation, nous utilisons trois pointeurs : Tete, Queue et Courant qui pointent respectivement vers le premier élément, dernier élément et l'élément courant de la liste.



Si on veut insérer une nouvelle valeur, à la position courante, le nœud de valeur 10, on obtient la liste suivante :

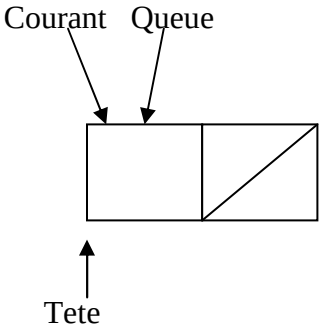


Tete Courant Queue

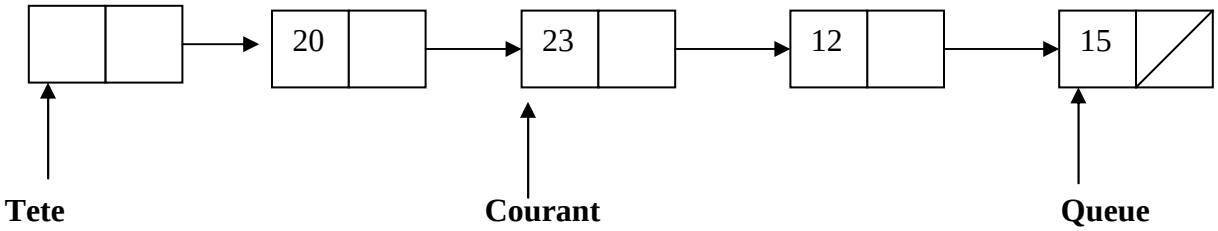
Problème ! Le nœud 23 doit pointer vers le nouveau nœud. Une solution est de parcourir la liste depuis le début jusqu'à rencontrer le nœud qui précède le nœud pointé par Courant.
Une autre solution est de copier le nouvel élément dans le nœud courant et de recopier l'ancien élément courant dans un nouveau nœud qui sera le suivant du nœud courant. Cette solution pose des problèmes similaires lors de la suppression du dernier élément de la liste.

Une autre alternative est de faire pointer Courant vers l'élément qui précède l'élément courant. Dans notre exemple si 10 est l'élément courant, on ferait pointer Courant sur 23. Un autre problème surgit avec cette convention. Si la liste contient un seul élément, il n'y a pas de précédent pour Courant.

Les cas spéciaux peuvent être éliminés si on utilise un nœud *entête*, comme le premier élément de la liste. Ce nœud est un nœud quelconque, sa partie information est complètement ignorée, et il pointe vers le premier élément de la liste. Ce nœud nous évitera de considérer les cas où la liste n'a qu'un seul élément, les cas où elle est vide et le cas où le pointeur courant est sur le début de la liste. Dans ce cas, initialement on a :



Si on reprend notre exemple on obtient ce qui suit:



Dans ce cas, l'insertion du 10 se fera normalement entre le 20 et le 23.

Cette opération qui se fait en trois étapes :

1. créer un nouveau nœud,
2. mettre son suivant vers l'élément courant actuel,
3. faire pointer le champ suivant du nœud précédent le nœud courant vers le nouveau nœud.

La classe Liste avec cette implantation est alors comme suit:

```
// fichier LISTE.H
// Déclaration de la classe Liste

#ifndef LISTE_H
#define LISTE_H
#include "noeud.h"

class Liste{

   public :
        Liste () ;                      // Constructeur
        ~Liste () ;                     // Destructeur
        void ViderListe () ;            // Vide la liste
        void Inserer ( const TElement & ) ;    // insère un élément à la position courante
        void InsererQueue ( const TElement & ) ;     // insère un élément à la fin de la liste
        TElement Supprimer () ;    // Supprime et retourne l'élément à la position courante
        void FixerTete () ;   // met la position courante à la tête de la liste
        void Precedent () ;   // Déplace la position courante à la position précédente
        void Suivant () ;      // Déplace la position courante à la position suivante
        int Longueur () const ;  // retourne la longueur courante de la liste
        void FixerPosition ( const int ) ;  // met position courante à position donnée
        void FixerValeur ( const TElement & ) ;   // met à jour la valeur à la position courante
        TElement ValeurCourante () const ;//retourne la valeur d'élément à la position courante.
        bool ListeVide () ;       // retourne vrai si la liste est vide
        bool EstDansListe () const ; // retourne vrai si position courante est dans la liste
        bool Trouver ( const TElement & ) ;// recherche une valeurs dans la
                                           // liste à partir de la position courante.

private :
        noeud * Tete ;                  // position du premier élément
        noeud * Queue ;                 // position du dernier élément
        noeud * Courant ;               // position de l'élément courant
} ;
#endif
```

Note : remarquez que le mot clé **const** est utilisé à la fin des déclarations de fonctions membres en mode lecture seule, à savoir EstDansListe (), ValeurCourante () et Longueur (). En effet, afin de mieux protéger un programme, il y a tout intérêt à déclarer comme constantes les fonctions qui ne sont pas censées modifier l'état des objets auxquels elles sont liées. On bénéficie ainsi

d’une aide supplémentaire du compilateur qui empêchera tout ajout de code modifiant l’état de l’objet dans la fonction.

```
// Fichier LISTE.CPP
#include <cassert>   //contient la macro assert  pour faciliter les deboggages de programmes
#include "LISTE.H"


Liste::Liste () {                    // Constructeur

        Queue = Tete = Courant = new noeud ;                // crée le noeud entête
  }

Liste::~Liste () {                                          // Destructeur
      while ( Tete != NULL ){
                  Courant = Tete ;
                  Tete = Tete->Suivant ;
                  delete Courant ;
       }
 }

void Liste::ViderListe () {
        // libère l'espace alloué aux noeuds, garde l'entête.
        while ( Tete->Suivant != NULL ){
                  Courant = Tete->Suivant ;
                  Tete->Suivant = Courant->Suivant ;
                  delete Courant ;
        }
        Courant = Queue = Tete ;   // réinitialise
}

// Insère un élément à la position courante

void Liste::Inserer ( const TElement & element ){
        assert ( Courant != NULL ) ;// s'assure que Courant pointe vers un noeud
        Courant->Suivant = new noeud ( element, Courant->Suivant ) ;
        if ( Queue == Courant )
                  Queue = Courant->Suivant ; //l'élément est ajouté à la fin de la liste.
}

TElement Liste::Supprimer (){                    // supprime et retourne l'élément courant
  assert ( EstDansListe() );// Courant doit être une position valide  sinon le programme se termine
  TElement temp = Courant->Suivant->element ; //Sauvegarde de l'élément courant
  noeud * ptemp = Courant->Suivant ; // Sauvegarde du pointeur du noeud Courant
  Courant->Suivant = ptemp->Suivant ;  // suppression de l'élément
  if ( Queue == ptemp )
        Queue = Courant ; // C'est le dernier élément supprimé, mise à jour de Queue
```

```
 delete ptemp ;
 return temp ;
}

void Liste::InsererQueue ( const TElement & element ){   // insère en fin de liste

        Queue = Queue->Suivant = new noeud (element, NULL) ; }

void Liste::FixerTete (){    // rend la tête comme position courante

        Courant = Tete ;
}

void Liste::Precedent (){  // met la position courante à la position précédente

        noeud * temp = Tete ;
        if ( ( Courant == NULL ) || ( Courant == Tete) ) // pas d'élément précédent
        {
                Courant = NULL ;
                return ;
        }
        while ( (temp != NULL)  && (temp->Suivant != Courant) )
                temp = temp->Suivant ;
        Courant = temp ;
}

void Liste::Suivant (){
        if ( Courant != NULL )
                Courant = Courant->Suivant ;
}



int Liste::Longueur () const{
        int cpt = 0 ;
        for ( noeud * temp = Tete->Suivant; temp != NULL; temp = temp->Suivant )
                cpt++ ;   // compte le nombre d'éléments
        return cpt ;
}

void Liste::FixerPosition ( const int pos ){
        Courant = Tete ;
        for ( int i = 0 ; ( Courant != NULL ) && ( i < pos ); i++ )
                Courant = Courant->Suivant ;
}
```

```
void Liste::FixerValeur ( const TElement & valeur ){
        assert ( EstDansListe() ) ;
        Courant->Suivant->element = valeur ;
        return ;
}

TElement Liste::ValeurCourante () const {
        assert ( EstDansListe() ) ;
        return Courant->Suivant->element ;
}

bool Liste::EstDansListe () const {
        return ( Courant != NULL ) && ( Courant->Suivant != NULL ) ;
}

bool Liste::ListeVide () {
        return Tete->Suivant == NULL ;
}

bool Liste::Trouver ( const TElement & valeur ) {    // recherche la valeur à partir
                                                     // de la position courante
        while ( EstDansListe() )
                if ( Courant->Suivant->element == valeur )
                        return true ;
           else
                        Courant = Courant->Suivant ;
        return false ;
}
#endif
```

Avertissement

Nous venons de terminer la révision du langage C++. Même si l'illustration des exemples se fera à l'aide du langage C++, dans la suite de ce cours, l'accent sera plus mis plus sur les concepts de structures de données et d'algorithmique que sur les technicalités de la programmation.

Un petit lexique utile à connaître

Code ASCII: C'est un tableau de 128 lignes faisant l'association d'un chiffre avec un caractère du clavier. Cette association permet de faire référence à un caractère soit par son numéro soit par sa valeur en guillemet. Voir le tableau.

Bit: Un bit est une donnée pouvant prendre 2 valeurs 0 ou 1. Le bit est l'élément de base de toute information numérique et donc de toute variable en mémoire. Toute information est en fait une suite de 0 et de 1: 0111100110. Le nombre de bits que prend une variable ou un fichier est donc sa taille prise en mémoire.

Booléen: C'est une variable pouvant prendre deux états Faux ou Vrai (en électronique -5v ou +5v, en C++ 0 ou 1). L'algèbre de Boole est régit par des lois: la commutativité, l'associativité, la distributivité et par des applications internes: l'addition (ET logique: &&) la multiplication (Ou logique: ||) et la négation (Différent de: !=).

C#: C'est la prochaine évolution Microsoft du C++. À prononcer "C Sharp". Ce langage tout objet, a pour vocation de concurrencer Sun avec son langage Java. À la différence de Java, le C# sera compilé en code assembleur ce qui ne permettra pas sa compatibilité entre les autres systèmes d'exploitation et les différentes machines. Pour de plus amples informations consulter: http://msdn.microsoft.com

Classe: Une classe est un type regroupant à la fois des données membres et des fonctions (appelées méthodes) ainsi que des règles d'accès. Une classe correspond à la définition d'un nouveau type. La variable créée grâce à ce type s'appelle un objet. Une classe ainsi que tous ses objets créés peuvent "hériter" des données et des méthodes de classes parentes. Une classe s'intègre donc dans une hiérarchie: elle peut posséder des mères et des filles.

Concaténation: La concaténation de deux chaînes de caractères correspond à la fusion de ses deux chaînes. On peut comparer la concaténation à l'addition pour les tableaux de caractères.

Directives Préprocesseurs: Les directives préprocesseurs sont des instructions qui dépendent du compilateur et non du langage. Ce sont en fait des instructions de compilation que l'on introduit dans le code C++. Ces instructions commencent toujours par le caractère #. Par exemple, #include <stdio.h> donne l'ordre au compilateur d'inclure tout le code de la librairie "stdio.h" au début du fichier.

Donnée membre: Une donnée membre est une variable contenue dans une classe et dans les objets créés grâce à celle ci. Elle peut appartenir à tous les objets; dans ce cas elle est unique pour tous les objets d'une même classe et elle est déclarée "static". Elle peut être aussi différente pour chaque objet d'une même classe.

Encapsulation: L'encapsulation est la technique qui consiste à réunir au sein d'une même classe des données et des fonctions.

Fonction: Une fonction est une suite d'instructions possédant un nom, une ou plusieurs variables d'entrées (paramètres de la fonction) et une variable de sortie. On appelle, par convention, fonction une procédure réalisant une opération sur les variables d'entrées et rendant le résultat en sortie.

Héritage: L'héritage est la technique qui permet à des classes filles de posséder les données membres et les méthodes d'une classe mère. Cette méthode permet de spécialiser une classe déjà existante sans avoir à la réécrire.