Université
Constantine 1

Faculté des Sciences
de la Technologie

Département Électronique

Programmation en Langage C++ : Cours et Travaux Pratiques

Boubekeur BOUKHEZZAR

Laboratoire d'Automatique
et de Robotique de Constantine (LARC)

boubekeur.boukhezzar@umc.edu.dz

Année universitaire 2013-2014

Avant-propos

Ce document regroupe les notes de cours, textes de TP et mini-projets présentés aux étudiants . Il s'agit d'un cours très court qui comporte 1,5 heures de cours et 1,5 heure de TP par semaine sur 07 semaines.

L'objectif du cours est de fournir en un temps minimum aux étudiants une formation minimale pour écrire et comprendre des programmes en C++. Compte tenu du volume horaire très restreint, le cours se limite au strict minimum. Il ne prétend donc ni à l'exhaustivité ni à servir de manuel complet pour une formation autodidacte. Il s'agit simplement d'un outil pédagogique qui peut être utilisé pour un premier cours présentiel cours d'initiation à la programmation en langage C++.

Au temps très restreint impartis à ce cours, s'ajoute le fait que les étudiants auxquels il est destiné ne possèdent généralement pas un prérequis suffisant en programmation structurée, ni un prérecquis suffisant en algorithmique ou en strictures de données.

Pour toutes ces raisons, ce cours constitue seulement une initiation à la programmation en langage C++ :

1. Il ne s'agit pas d'un cours d'algorithmique : L'expérience à montrer que le choix d'exemples d'algorithmes, même simples, pour illustrer des structures ou concepts de la programmation faisait perdre les étudiants dans ce qu'il considèrent comme la difficulté de l'exemple perdant de vue l'objectif initial qui est de traduire un algorithme en un programme.
2. IL ne s'agit pas d'un cours de structures de données : Seuls les tableaux et les pointeurs sont traités. Les autres structures (piles, files, listes, arbres, ...) ainsi que les élément de la bibliothèque standard ne sont pas abordés.
3. Il ne s'agit pas d'un cours sur la conception logicielle : Un tel cours est généralement destiné à des informaticiens après des cours de POO et de génie logiciel, soit dans les années supérieures.
4. Il ne s'agit pas d'un cours sur les paradigmes de la programmation :Un tel cours est généralement destiné à des informaticiens. De plus, pour comparer deux choses au moins, il faut en avoir une certaine connaissance préalable. Ce n'est évidement pas le cas des étudiant auxquels s'adresse ce cours.

Les exemples de programmes présentés dans les notes de cours sont les plus simples possible et sont une illustration directe des concepts et des instructions introduites. Ceci est dû comme mentionné plus haut aussi bien au temps très restreint de l'UE qu'au manque de prérequis des étudiants ; des exemples plus complexes ferait perdre l'objectif initial : Assimiler l'usage une notion ou une instruction en programmation C++. Pour les

mêmes raisons, les travaux pratiques, qui eux aussi ne durent qu'une heure et demi, sont présentés sous forme d'exercices relativement court.

Toutefois, des mini-projets sont proposés aux étudiant si l'avancement est suffisant pour mettre en pratique, autour d'un petit projet concret, les notions vues en cours et en TP. Une assistance personnalisée est proposée aux étudiant par l'usage de l'e-mail.

Le document s'est voulu aussi court et aussi concis que possible afin de faciliter sa lecture et de pouvoir atteindre un seul objectif en un temps très court : Avoir les connaissance de base pour écrire et comprendre un programme en C++.

Néanmoins,il existe un bibliographie abondante en C++ aussi bien en version imprimé (ouvrages) qu'en version électronique sur la toile. Des ouvrages de références dépassant souvent les 1000 sont assez exhaustifs sur ce sujet et sont conseillé aux étudiants en guise de complément, particulièrement lors de la préparation de leurs mini-projets Le document est divisée en trois parties :

1. La première contient les notes de cours mises à la disposition des étudiants et utilisées lors du cours présentiel.
2. La seconde contient la majorité TP présentés aux étudiants. Ces derniers sont quelque peu modifiés d'année en année. La version présentée est celle de l'année 2013-2014.
3. La dernière partie regroupe les mini-projets proposés aux étudiants durant les années passées.

La bibliographie utilisée est présentée à la fin du document.

Table des matières

Première partie

Notes de cours

Chapitre 1

Éléments de Base

1.1 Aperçu général

Le langage C++ est une extension du langage C. Il n'a pas été appelé langage D car il n'est pas une nouvelle mise à jour, qui corrige les bugs d'une ancienne version du langage C.

1.1.1 Bref historique du langage C++

Les principales dates dans la création et l'évolution du langage C++ peuvent êtres résumée comme suit :

- 1972 : Création du langage C par Denis Ritchie.
- 1980 : Création du *C avec classes* par Bjarne Stroustrup.
- 1983 : Rick Mascitti donne le nom C++ au langage (qui veut dire C incrémenté dans la syntaxe du langage).
- 1998 : Ratification du standard ISO C++ (ISO IEC/14882).

1.1.2 Liens du C++ avec les autres langages

Le langage C++ a été crée dans le but de rajouter au langage C, dont il hérite de la syntaxe, des fonctionnalités qui lui permettent de prendre en charge les concepts de la POO (Programmation orientée objet). Les concepts de classe, classe dérivée et fonction virtuelle sont inspirés par le langage Simula67. Les facilités offertes par le C++ pour la surcharge des opérateurs, le libre emplacement des opérateurs et des déclarations ressemblent à celles du langage Algol68. L'utilisation des templates (prototypes) a été inspirée par les génériques d'Ada et par les modules paramétrés de Clu. Les mécanismes de traitement des exceptions sont dus aux langages Ada, Clu et ML. La figure 1.1 schématise une partie des dépendances du langage C++ avec ses prédécesseurs

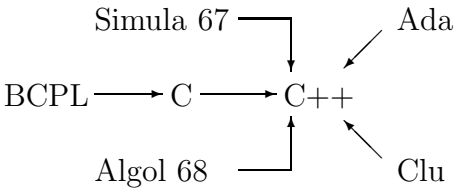

Figure 1.1 – Dépendances du langage C++ de ses prédécesseurs

D'autres concepts tels que l'héritage multiple, les fonctions virtuelles pures et les namespaces sont propres au langage C++.

1.2 Structure d'un programme en C++

L'exemple 1.1 montre une structure minimale d'un programme en C++ :

Exemple 1.1 (Programme Hello World)

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main()
6 {
7     cout << "Hello world!" << endl;
8     return 0;
9 }
```

Listing 1.1 – Exemple de base d'un programme en C++

Un programme est un ensemble de déclarations et d'instructions. Le programme doit être placé entre accolades {} à la suite du nom de la fonction principale `main()`. En C++, toute déclaration ou fonction doit se terminer par un point virgule `;`.

`main()` est la fonction principale. C'est le point d'entrée du programme. Elle est donnée ici sans arguments, mais on peut lui transmettre des arguments en écrivant par exemple `main (int nbarg)` où `nbarg` est un paramètre entier.

La première ligne `#include <iostream.h>` est une *directive*. Elle est prise en charge par le préprocesseur avant la compilation. Elle permet d'utiliser des bibliothèques C++.

1.3 Création d'un programme en C++

La création d'un exécutable en C++ passe par trois étapes principales :

1. Édition du programme.
2. Compilation.
3. Edition des liens.

1.3.1 Édition du programme

L'édition du programme consiste en la saisie du texte du programme par un éditeur. Ce texte est appelé *programme source* ou *source* tout simplement. Il est sauvegardé dans un fichier appelé *fichier source*. L'éditeur de l'environnement de développement sauvegarde le fichier source avec une dénomination possédant une extension spécifique. Pour les programmes C++, l'extension est généralement `.cpp`.

1.3.2 Compilation

Le programme source est écrit dans un *langage évolué*. La compilation consiste à traduire le programme source en langage machine. La compilation est effectué par un programme appelé *compilateur*. La compilation est précédée par le traitement par le préprocesseur.

1.3.2.1 Traitement par le préprocesseur

Le préprocesseur exécute les directives qui le concernent. Il les reconnaît par le fait qu'elles commencent par le caractère #. Le résultat est un programme C++ pur, sans directives.

1.3.2.2 Compilation

Elle produit un programme en langage machine. Ce résultat est appelé *module objet* et porte généralement l'extension `.obj`. Le module objet n'est pas directement exécutable.

1.3.3 Edition de liens

Pour s'exécuter, le module objet principal à besoin des fonctions auxquels il fait appel. L'éditeur de liens (linker) va chercher dans les bibliothèques concernées les modules objet de ces fonctions. Une bibliothèque est une collection de modules objets (fonctions compilées) organisés en plusieurs fichiers. Grâce à la compilation séparée, on peut rassembler lors de l'édition des liens plusieurs modules objet compilés de façon séparée. Le résultat de l'édition des liens est un programme exécutable qui porte généralement l'extension `.exe`. Le programme exécutable peut être exécuté sans faire appel à l'environnement de développement C++.

1.4 Les fichiers d'entête

Ils sont traités par le préprocesseur et sont introduits grâce à la directive `#include`. Le listing 1.2 montre l'inclusion du fichier d'entête `iostream`.

```
1 #include <iostream
```

Listing 1.2 – Inclusion du fichier d'entête `iostream`

Ces fichiers comportent les déclarations relatives aux fonctions prédéfinies. Le préprocesseur copie le contenu de ce fichier dans le programme qu'il produit à l'endroit de la directive. Le code compilé de ces fonctions est contenu dans les modules objet et il est rangé dans des bibliothèques. Le fichier d'entête peut être nommé de trois façons :

Par son chemin absolu : Le préprocesseur recherche le fichier à cet emplacement même, comme le montre la ligne de code ci-dessous :

```
1 #include "D:\Program Files\CodeBlocks\include\entete"
```

Listing 1.3 – Inclusion d'un fichier d'entête par son adresse absolue

à partir du répertoire courant : Le préprocesseur recherche le fichier d'entête dans le répertoire courant, à titre d'exemple :

```
1 #include "header"
```

Listing 1.4 – Inclusion d'un fichier d'entête contenu dans le répertoire courant

à partir d'un répertoire prédéfini : C'est les répertoires standard correspondant à l'installation du compilateur, si le nom de fichier est entouré par un inférieur et un supérieur. C'est dans ces répertoires que le préprocesseur recherchera le fichier d'entête. Dans ce cas, le nom du fichier est inséré entre les symboles `<` et `>` comme dans l'exemple suivant :

```
1 #include <iostream>
```

Listing 1.5 – Inclusion du fichier d'entête contenu dans un répertoire prédéfini

1.5 Le préprocesseur

Les fonctions du préprocesseur peuvent être résumées dans les points suivants :

1. Traiter le fichier source avant le compilateur.
2. Retirer les commentaires (compris entre `/*` et `*/` ou après `//`).
3. Prend en compte les lignes commençant par `#`.

1.6 Les commentaires

Ils peuvent êtres faits à n'importe quel endroit du programme. Ce sont des textes explicatifs destinés aux lecteurs du programme. Ils ne sont pas pris en compte par le compilateur. On distingue deux types de commentaires :

1. Les commentaires libres (par bloc).
2. Les commentaires de ligne (de fin de ligne).

1.6.1 Les commentaires libres (par bloc)

Ils commencent par `/*` et se termine par `*/`. Ils peuvent s'étendre sur plusieurs lignes. Un exemple est donné ci-dessous

```
/*+-------------------------------------------------------------+
  |                    Mon premier programme en C++             |
  +-------------------------------------------------------------+*/
```

Listing 1.6 – Commentaire par bloc

1.6.2 Les commentaires de ligne (de fin de ligne)

Ils sont introduit par les deux caractères `//`. Ils sont généralement placés en fin de ligne après une instruction. Tout ce qui se situe entre `//` et la fin de la ligne est un commentaire. Un exemple est donné dans le listing 1.7

```
cout << "Hello \n" ; // bonjour
```

Listing 1.7 – Commentaire de ligne

1.7 Les variables

Les variables servent à stocker les données manipulées par les programmes en mémoire. Une variable a une adresse, un type et une valeur. Le nom est appelé *identificateur*, quand au contenu d'une variable, il appartient à son type. Les variables peuvent êtres partagés en deux :

Les variables scalaires : Ce sont les entiers, réels, pointeurs, ...

Les variables structurées : Tableaux, structures, ...

1.7.1 Déclaration des variables

Le langage C++ est déclaratif : Toutes les variables doivent êtres déclarées avant leur première utilisation. La syntaxe de la déclaration d'une variable est la suivante :

```
<type> <identificateur>
```

Listing 1.8 – Syntaxe de la déclaration d'une variable

1.7.2 Type des variables

Il existe trois types de base pour les variables en C++ : entiers, flottants et caractères

1.7.2.1 Les types entiers

Selon la taille en mémoire des variables, on distingue trois types entiers :
- `short int (short)`
- `int`
- `long int (long)`

Chaque type permet de représenter les données sur un certain nombre d'octets. Le nombre d'octet définit la plage des valeurs prise par les variables de chaque type.

Lorsque la plage des valeurs est strictement positive, on utilise des variables non signés (unsigned). On a lors les autres types suivants.
- `unsigned short int`
- `unsigned long int`
- `unsigned int`

1.7.2.2 Les types réels ou flottants

Un réel est composé d'un signe, d'une mantisse et d'un exposant. On distingue trois types flottants :
- `float`
- `double`
- `long double`

1.7.2.3 Les types caractère

Il s'agit du type `char`. Une variable de ce type stocke un seul caractère qui est codé sur un octet (8 bits). Il peut être signé ou non signé :
- `char :` prend les valeurs entre -127 à +127.
- `unsigned char :` prend les valeurs positives de 0 à 255.

Un caractère peut être une lettre, un chiffre, ou un autre caractère du code ASCII. Il peut être un caractère non imprimable (dit de contrôle). A chaque caractère du code ASCII correspond une valeur entière comprise entre `0` et `255`. Ceci permet d'utiliser une variable de type `char`, codé seulement sur un octet, au lieu d'une variable de type `int` qui occupe plus de mémoire pour stocker un entier compris entre 0 et 255. Cette opération permet de réduire la mémoire utilisée.

1.7.2.4 Le type booléen

Il contient deux valeurs `true` et `false`.

L'exemple suivant récapitule la déclaration de variables des différents types cités.

Exemple 1.2 (Déclaration de variables)

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main()
6  {
7      int i = 2;
8      float x = 4.3;
```

```
9       float y = 54e-01;
10      double z = 5.4;
11      unsigned char c = 'A';
12      return 0;
13  }
```

Listing 1.9 – Exemple de déclaration de variables de différents types

1.7.3 Règles d'écriture des identificateurs

Les règles d'écritures des identificateurs sont les suivantes :

1. Ne commence pas par un chiffre.
2. Il y a une différence entre majuscules et minuscules (C++ est sensible à la case ou case-sensitive).
3. Ne pas utiliser les caractères accentués (é,è,ê,ù, ...).
4. Ne pas utiliser les mots réservés du langage.

Le compilateur `C++` n'impose aucune limite sur la longueur des identificateurs. Toutefois, certaines parties de l'environnement de développement (comme l'éditeur de liens) peuvent imposer une limitation. Cette limite peut varier en fonction des environnements utilisés.

1.8 Les constantes

Les constantes sont déclarées avec le mot clé `const`. Comme les variables, elles possèdent un nom (identificateur) et un type. Elles ne changent pas de valeur au long du programme et doivent êtres initialisées au moment de leur création. Dans l'exemple suivant, on déclare une constante entière et deux constantes caractère et chaîne de caractères.

```
1  const int a=1 ;//constante entière
2  const char c='A' ; constante caractère
3  const char z[10]= "C++" ;//constante chaîne de caractère
```

Listing 1.10 – Exemple de déclaration de constantes

1.8.1 Définition d'une constante à l'aide du préprocesseur

Le préprocesseur effectue dans tout le code source une substitution de l'identificateur par la valeur indiquée. La syntaxe de cette déclaration est la suivante

```
1  #define < identificateur > < valeur >
```

Listing 1.11 – Syntaxe de la définition d'une constante par le préprocesseur

Un exemple

```
1  #define PI 3.14159
```

Listing 1.12 – Exemple de la déclaration de constantes en faisant appel au préprocesseur

1.8.2 Les constantes énumérées

Le mot clé `enum` permet de définir des types puis de définir des variables de ce type comme dans l'exemple suivant

```
1 enum couleur {vert,rouge,bleu,noir,blanc};
2 couleur c1,c2 ;
```

Listing 1.13 – Exemple de la déclaration de constantes énumérées

Les variables `c1` et `c2` peuvent avoir cinq valeurs possibles : `vert`, `rouge`, `bleu`, `noir` et `blanc`. Chaque constante énumérée possède une valeur numérique par défaut. Si aucune valeur n'est spécifiée, la première constante va avoir la valeur 0, la seconde 1, etc .... Si des valeurs sont spécifiées comme dans l'exemple suivant

```
1 enum couleur {vert=2,rouge,bleu=6,noir,blanc=50};
```

Listing 1.14 – Exemple de déclaration énumérée

Les valeurs prises par ces constantes énumérés `vert`, `rouge`, `bleu`, `noir` et `blanc` sont respectivement 2, 3, 6, 7, 50.

1.9 Les opérateurs

La syntaxe d'utilisation d'un opérateur est la suivante

```
1 < opérande1> < opérateur ><opérande2>
```

Listing 1.15 – Syntaxe d'utilisation d'un opérateur

On définira maintenant les principaux opérateurs en C++ classés par catégories

1.9.1 Opérateurs d'affectation

Ils permettent de simplifier la notation des opérations arithmétiques. Chaque opérateur arithmétique possède un opérateur simplifié.

1.9.1.1 L'opérateur d'affectation

Le symbole " = " est utilisé pour assigner (affecter) la valeur d'une expression à un identificateur.

```
1 j=2*i+1 ; //x reçoit le résultat de l'expression 2*i+1
```

Listing 1.16 – Utilisation de l'opérateur d'affectation

1.9.1.2 Autres opérateurs

Opération	Résultat
`x+=y;`	`x=x+y ;`
`x*=y ;`	`x=x*y ;`
`x/=y ;`	`x=x/y ;`
`x%=y ;`	`x=x%y ;`

Table 1.1 – Autres opérateurs

1.9.2 Les opérateurs arithmétiques

Opérateur	Rôle
`+`	Addition
`-`	Soustraction
`*`	Multiplication
/	Division
`%`	Modulo (Reste de division entière)

Table 1.2 – Opérateurs arithmétiques

1.9.3 Les opérateurs logiques

Le résultat de l'utilisation d'un opérateur logique est un booléen (true ou false).

Opérateur	Rôle
`&&`	Et
`||`	Ou
`!`	Non (Not)

Table 1.3 – Opérateurs logiques

1.9.4 Les opérateurs de comparaison

Opérateur	Rôle
`==`	Égal
`!=`	Différent
`<`	Strictement inférieur
`<=`	Inférieur ou égal
`>`	Strictement supérieur
`>=`	Supérieur ou égal

Table 1.4 – Les Opérateurs de comparaison

1.9.5 Les Opérateurs d'incrémentation et de décrémentation

Opérateur	Exemple d'utilisation	Résultat
Incrémentation	`x++ ;`	`x=x+1 ;`
Décrémentation	`x-- ;`	`x=x-1 ;`

Table 1.5 – Les opérateurs d'incrémentation et de décrémentation

- Dans le cas de l'opérateur préfixé, le `++` ou `--` est placé avant la variable. Dans ce cas, la variable est incrémentée (ou décrémentée) avant l'utilisation de la valeur.
- Dans le cas de l'opérateur postfixé, le `++` ou `--` est placé après la variable. Dans ce cas, la variable est incrémentée (ou décrémentée) après l'utilisation de la valeur.

L'exemple suivant montre la différence entre les deux cas

```
1 int x=1 ;
2 int y=0 ; y=x++ ; // dans ce cas y=1 et x=2
3 x=1 ; // On réinitialise x
4 y=++x ; // dans ce cas x=2 et y=2
```

Listing 1.17 – Opérateurs préfixés et postfixés

1.9.6 Opérateurs binaires

Opérateur	Rôle
&	Et binaire bit à bit
—	Ou binaire bit à bit
∧	Ou exclusif bit à bit
∼	Complément à 1
<<	Décalage de n bits à gauche
>>	Décalage de n bits à droite

Table 1.6 – Opérateurs binaires

1.9.7 Opérateurs ternaire

Appelé aussi opérateur conditionnel, c'est le seul opérateur à prendre trois opérandes (ternaire). Il utilise deux symboles (`?` et `:`). Le premier opérande est une condition :

1. Si la condition est vraie, on retourne ce qui suit le point d'interrogation (`?`).
2. Si la condition est fausse, on retourne ce qui suit les deux points (`:`).

A titre d'exemple, le test suivant

```
1 if (a>b) z=a ;
2 else
3     z=b;
```

Listing 1.18 – Utilisation de l'opérateur d'affectation

est équivalent à

```
z=(a>b) ?a :b ;
```

Listing 1.19 – Utilisation de l'opérateur ternaire

1.9.8 Opérateurs sizeof

Il s'utilise pour obtenir l'espace occupé par une variable ou un type de données. Cette taille est indiquée en octets. L'exemple suivant illustre l'utilisation de cet opérateur

```
int i=1 ;
cout << sizeof(i)<<"\n";//affiche 4
cout << sizeof(int)<<"\n";//affiche 4
```

Listing 1.20 – Utilisation de l'opérateur sizeof

1.10 Les entrées-sorties

Les entrées sorties en langage C++ font intervenir la notion de flot qui sera vue en détail au chapitre 8. Les fonctions `printf` et `scanf` héritées de la bibliothèque standard du langage `C` peuvent aussi être utilisées en incluant le fichier d'entête `<stdio.h>`

1.10.1 Les entrées : `cin`

Pour le moment, il est suffisant de considèrer le flot `cin` comme une instruction Permet de saisir soit des valeurs numériques, des caractères simples ou des chaînes de caractères. Sa syntaxe est la suivante :

```
cin>>variable1>>variable2>>...>>varaibleN ;
```

Listing 1.21 – Syntaxe de `cin`

`variable1`, `variable2`,..., `variableN` sont les noms des variables qui vont contenir les données saisies à partir du clavier. L'exemple suivant illustre son utilisation

```
int i,j;
float z ;
char s,nom[12] ;
cin>>i>>j>>z>>s>>nom ;
```

Listing 1.22 – Exemple de l'utilisation de `cin`

1.10.2 Les sorties : `cout`

Le flot `cout` permet d'afficher le contenu des variables, mais aussi des messages à l'écran.

1.10.2.1 Afficher un message

Le message est placé entre deux guillemets après un `cout`

```
cout << "Bonjour " ;
```

Listing 1.23 – Exemple de l'utilisation de `cout`

1.10.2.2 Affichage d'une variable avec message

A titre d'exemple, le programme suivant

```
int i=1 ;
int j=2 ;
cout<<i<<"+"<<j<<"="<<z ;
```

Listing 1.24 – Affichage d'une variable avec message

affichera `1+2=3`.

Le changement de ligne se fait par `\n` et la tabulation par `\t`.

Plus de détails sur le concept et l'utilisation des flots seront donnés au chapitre 8

Chapitre 2

Structures de Contrôle

2.1 L'instruction `if-else`

Cette instruction permet d'effectuer un test pour choisir entre deux alternatives. En fonction du résultat du test, l'une des deux alternatives est choisie. La syntaxe de cette instruction est la suivante

```
if (condition) {
//Bloc d'instructions 1
} else {
//Bloc d'instructions 2
}
```

Listing 2.1 – Syntaxe de l'instruction `if-else`

L'expression `condition` est évaluée. Si elle est vraie (non nulle), le bloc d'instructions 1 est exécuté. Si elle est fausse (nulle), le bloc d'instructions 2 est exécuté. Un exemple est donné dans le listing 2.2

Exemple 2.1 (Instruction `if-else`)

```
#include <iostream>

using namespace std;

int main() {
    int a;
    cout<<"Entrez un entier : ";
    cin>>a;
    if (a>=0)
    {
        cout<<"\n"<<a<<" est positif";
    }
    else
    {
        cout<<"\n"<<a<<" est négatif";
    }
    return 0;
}
```

Listing 2.2 – Exemple d'utilisation de l'instruction `if-else`

Remarque 2.1
Si le bloc d'instructions 1 ou 2 est composé d'une seule instruction, les accolades ne sont pas nécessaires, mais il est conseillé de les mettre.

Remarque 2.2
L'instruction `if` peut être utilisée toute seule sans le `else`. Dans ce cas, l'instruction (ou le bloc d'instructions) est exécuté si la condition est vraie. Sinon, le programme passe à l'instruction suivante.

S'il existe plus de deux alternatives, elles s'expriment avec des `else if (condition)`. Le listing 2.3 illustre ceci

Exemple 2.2 (Instruction `if-else if`)

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

using namespace std;

int main() {
    int a;
    cout<<"Entrez un entier : ";
    cin>>a;
    if (a>0)
    {
        cout<<"\n"<<a<<" est positif";
    }
    else if (a==0)
    {
        cout<<"\n"<<a<<" est nul";
    }
    else
    {
        cout<<"\n"<<a<<" est négatif";
    }
    return 0;
}
```

Listing 2.3 – Exemple d'utilisation de l'instruction `else if`

2.2 La boucle `while` (tant que)

La syntaxe de cette instruction est donnée par le listing 2.4

```
while (condition) {
//Bloc d'instructions
}
```

Listing 2.4 – Syntaxe de l'instruction `while`

Le bloc d'instruction est répété tant que condition est vérifiée (non nulle). Pour la boucle `while`, la condition est vérifiée avant l'entrée dans le bloc. La boucle peut ne jamais être exécutée si la condition n'est initialement pas vérifiée. Le programme 2.5 met en œuvre l'utilisation de l'instruction `while`

Exemple 2.3 (Instruction `while`)

```
//Affiche les lettres de l'alphabet en majuscules
#include <iostream>

using namespace std;

int main() {
    char i='A';
    while (i<='Z')
    {
        cout<<i<<"\n";
        i++;
    }
}
```

Listing 2.5 – Exemple d'utilisation de l'instruction `while`

2.3 La boucle `do-while` (faire-tant que)

La syntaxe de cette instruction est donnée par le listing 2.6

```
do
{
//Bloc d'instructions
}while(condition) ;
```

Listing 2.6 – Syntaxe de l'instruction `do-while`

Le test de la condition se fait à la fin de la boucle. Si la condition est vraie, la boucle est réexecutée. Contrairement à la boucle while, le bloc d'instruction est exécuté au moins une fois. Le listing 2.7 donne un exemple de l'utilisation de cette instruction.

Exemple 2.4 (Instruction `do-while`)

```
#include <iostream>
//Affiche les 10 chiffres
using namespace std;

int main()
{
    int i=0;
    do
    {
        cout<<i<<"\n";
        i++;
    }
    while(i<=9);
    return 0;
}
```

Listing 2.7 – Exemple d'utilisation de l'instruction `do-while`

2.4 La boucle `for`

La syntaxe de cette instruction est donnée par le listing 2.8

```
for (expression1 ;expression2 ;expression3)
{
//Bloc d'instructions
}
```

Listing 2.8 – Syntaxe de l'instruction `for`

La signification des différentes expressions apparaissant dans la syntaxe de l'instruction `for` est la suivante :

expression1 : C'est l'expression de démarrage. Elle consiste généralement en l'initialisation d'une ou plusieurs variables entières appelées compteurs. Elle est effectuée avant d'entrer dans la boucle.

expression 2 : C'est la condition d'arrêt de la boucle. Elle est évaluée avant l'entrée dans la boucle. La boucle est alors exécutée tant que la condition est vérifiée.

expression 3 : Elle est exécutée à la fin de chaque itération de la boucle. Il s'agit généralement de la mise à jour (augmentation ou diminution) des valeurs des compteurs.

L'exemple suivant utilise une boucle `for` pour afficher le carré et la racine des cinq premier chiffres non nuls (de 1 à 5)

Exemple 2.5 (Instruction `for`)

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

using namespace std;
int main()
{
    for (int i=1; i<6; i++)
    {
        cout<<i<<" puissance 2 = "<<pow(i,2)<<endl;
        cout<<"Racine de "<<i<<" = "<<sqrt(i)<<endl;
    }
    return 0;
}
```

Listing 2.9 – Exemple d'utilisation de l'instruction `for`

2.5 L'instruction `switch-case`

Elle permet de sélectionner un groupe d'instructions parmi plusieurs. Ceci permet d'éviter l'imbrication de plusieurs instructions `if else`. Le groupe d'instruction à exécuter est alors choisis en fonction de la valeur d'une expression. Sa syntaxe est donnée par le listing 2.10

```
switch (expression)
{
case val1 : //instruction1 ou bloc d'instructions1
break;
case val2 : //instruction2 ou bloc d'instructions2
break;
...
```

```
default : //instruction ou bloc d'instructions par défaut
}
```

Listing 2.10 – Syntaxe de l'instruction `switch-case`

L'expression est évaluée à l'entrée de la boucle `switch`. Elle doit donner un résultat de type entier. `val1`, `val2` sont des constantes de type entier. L'exécution de l'instruction se fait à partir du `case` dont la valeur correspond à celle de l'expression évaluée. L'exécution peut continuer pour les autres `case` jusqu'à rencontrer un `break`. Si la valeur de l'expression ne correspond à aucune des valeurs des `case`, c'est les instructions de `default` qui sont exécutée. Un exemple d'utilisation de l'instruction `switch-case` est donné par le programme 2.11

Exemple 2.6 (Instruction `switch-case`)

```
#include <iostream>

using namespace std;
int main()

{
    int i;
    cout<<"Entrez 1 à 4 pour afficher une lettre de l'alphabet ";
    cin>>i;
    switch (i)
    {
    case 1 :
        cout<<"A";
        break;
    case 2 :
        cout<<"B";
        break;
    case 3 :
        cout<<"C";
        break;
    case 4 :
        cout<<"D";
        break;
    default:
        cout<<"Erreur";
    }
    return 0;
}
```

Listing 2.11 – Exemple d'utilisation de l'instruction `switch-case`

2.6 L'instruction `break`

Elle permet de sortir d'une boucle (`while`, `do-while`, `for`) ou d'une instruction de sélection (`switch-case`). Elle figure ainsi dans les quatre instructions : `while`, `do-while`, `for` et `switch-case`.

2.7 L'instruction `continue`

Elle permet de suspendre les instructions restantes dans une boucle et de revenir au début de la boucle (`while`, `do-while`, `for`).

2.8 L'instruction `exit`

Elle termine l'exécution d'un programme. Elle permet de retourner une valeur `n` au système d'exploitation par `exit(n)`.

Chapitre 3

Tableaux, pointeurs, structures

3.1 Les tableaux

Un tableau est un ensemble de données de même type. Les éléments d'un tableau sont rangés consécutivement en mémoire et occupent chacun le même nombre de cases mémoires.

3.1.1 Tableaux à une dimension

La syntaxe de déclaration d'un tableau à une dimension est la suivante :

```
type <nom tableau>[dimension]
```

Listing 3.1 – Syntaxe de la déclaration d'un tableau

3.1.1.1 Déclaration du tableau

La déclaration d'un tableau comporte les éléments suivants :

type : C'est le type des données que contient le tableau.

nom tableau : C'est l'identificateur du tableau.

dimension : C'est le nombre d'éléments que contient le tableau (entre crochets).

L'exemple suivant montre la déclaration de trois tableaux de types différents

```
int tab[10] ;//tableau de dix entiers
char a[6] ;//tableau de caractères
float x[8] ;//tableau de huit réels
```

Listing 3.2 – Exemples de tableaux à une dimension

Remarque 3.1
La taille d'un tableau doit être une constante entière. Elle doit être connue au moment de la compilation. Une déclaration de la forme `int tab [n]`, où `n` est une variable entière génère une erreur à la compilation, car le compilateur ne connaît pas la taille de l'espace nécessaire qu'il doit réserver pour stocker les éléments du tableau.

Il est commun d'utiliser une directive pour déclarer une constante qui définira la taille de tous les tableaux d'un programme comme montré ci-dessous

```
#define dim 10
int tab[dim]; //tableau de dix entiers
float x[dim];//tableau de dix réels
```

Listing 3.3 – Utilisation d'une directive pour la déclaration de la dimension d'un tableau

3.1.1.2 Accès aux éléments d'un tableau

Chaque élément d'un tableau à une dimension est identifié par la position à laquelle il se trouve dans le tableau. Cette position est appelée *indice*. On accède à cet élément par l'intermédiaire de cet indice.

L'indice peut être une valeur entière, une expression arithmétique, ou une variable.

- Le premier élément est repéré par l'indice `0`. Premier élément : `tab[0]`.
- Le dernier élément est repéré par l'indice `dim-1`, où `dim` est la taille du tableau. Dernier élément : `tab[dim-1]`.

```
int i=5;
tab[0];//accès au premier élément du tableau
tab[i];//accès au 6ème élément du tableau
tab[2*i-1];//accès au 10ème élément du tableau
```

Listing 3.4 – Accès aux éléments d'un tableau

Remarque 3.2
Le compilateur ne signale pas d'erreur s'il y a un dépassement de bornes. Par exemple, si `tab` est un tableau de dimension `10`, écrie `tab[15]` n'est pas une erreur que le compilateur détectera.

3.1.1.3 Initialisation d'un tableau unidimensionnel

Un tableau peut être initialisé au moment de sa création en énumérant une liste de données. La liste figure entre deux accolades. Ces éléments sont séparés par des virgules. Dans l'exemple suivant, on initialise un tableau de cinq entiers.

```
int tab[5]={1,2,3,4,5} ;
```

Listing 3.5 – Initialisation d'un tableau

Les règles suivantes sont suivies lors de l'initialisation d'un tableau :

- Si la liste comporte un nombre insuffisant d'éléments, seuls les premiers éléments seront initialisés. Les autres seront à zéro.
- Si la liste comporte plus d'éléments que la dimension du tableau, la compilation déclarera une erreur.

Un exemple est donné ci-dessous :

```
int tab[5]={1,2} ; // affecte à tab[0] et tab[1] les valeurs 1 et 2,
                   // les autres sont à 0.
```

Listing 3.6 – Initialisation des premiers éléments d'un tableau

Lorsque la taille du tableau n'est pas indiquée, et que le tableau est initialisé par énumération, le compilateur peut déterminer la taille du tableau en comptant les valeurs.

```
int tab[]={1,2,5,8,9} ; // équivalent à int tab[5]={1,2,5,8,9}
```

Listing 3.7 – Initialisation d'un tableau sans indiquer sa taille

3.1.1.4 Détermination de la taille d'un tableau

L'opérateur `sizeof()` appliqué à un tableau permet de déterminer l'espace occupé par celui-ci-en mémoire. Cet espace est égal au nombre d'éléments du tableau (sa dimension) multiplié par la taille d'un élément. Pour obtenir la dimension (nombre d'éléments) d'un tableau d'entiers, on peut écrire `(sizeof(tab)/sizeof(tab[0]))` ou `(sizeof(tab)/sizeof(int))` .

3.1.2 Tableaux à plusieurs dimensions (multidimensionnels)

Un tableau à deux dimensions est un tableau à une dimension dont les éléments sont aussi des tableau à une dimension. En bref, c'est un tableau de tableaux. Un tableau à trois dimensions est un tableau à une dimension dont les éléments sont des tableaux à deux dimensions, c'est un tableau, de tableaux, de tableaux.

La déclaration d'un tableau à plusieurs dimensions se fait de la façon suivante :

```
type <nom tableau>[dimension1] [dimension2]...[dimensionn]
```

Listing 3.8 – Déclaration d'un tableau à plusieurs dimensions

Le principe de manipulation d'un tableau à plusieurs dimensions est le même que celui de la manipulation d'un tableau à une dimension. 0n peut initialiser un tableau à deux dimensions par énumération en plaçant les lignes entres des accolades. Dans l'exemple suivant, L'élément `tab[0][0]` prend la valeur `1`. L'élément `tab[0][1]` prend la valeur `2`.

```
int tab[3][2] = {{1, 2} , {4, 5},{5,7}};
```

Listing 3.9 – Initialisation d'un tableau à deux dimension

3.1.3 Les chaînes de caractères

Une chaîne de caractères est stockée sous forme d'un tableau de caractères. Ce tableau se termine avec le caractère `NULL` '\0'. Le tableau qui stocke une chaîne de caractère doit avoir au moins un élément en plus que la taille de la chaîne pour stocker le caractère '\0'.

3.1.3.1 Déclaration d'une chaîne de caractères

Cette déclaration peut se faire de deux façons

a Par une liste de constantes caractères

```
1 char ch[3] = {'A', 'B', 'C'};
```

Listing 3.10 – Déclaration d'une chaîne de caractères par une liste de constantes caractères

b Par une chaîne littérale

```
1 char ch[8] = "bonjour";
```

Listing 3.11 – Déclaration littérale d'une chaîne de caractères

3.2 Les pointeurs

Un pointeur est une variable qui contient l'adresse d'une autre variable. Une adresse est un nombre qui définit de façon unique une case mémoire (un octet en mémoire).

3.2.1 Syntaxe de déclaration

La syntaxe de déclaration d'une variable de type pointeur fait précéder le nom de la variable par un astérisque *.

```
1 type *nom
```

Listing 3.12 – Syntaxe de la déclaration d'un pointeur

type : correspond au type de la variable pointée.

nom : correspond au nom du pointeur.

Un exemple est donné ci-dessous

```
1 int *p; // déclaration d'un pointeur p sur un entier
2 float *x ;// déclaration d'un pointeur a sur un réel
3 char *c ;// déclaration d'un pointeur c sur une chaîne de caractères
```

Listing 3.13 – Exemples de déclarations de pointeurs

3.2.2 Utilisation des pointeurs

Il existe deux façons d'accéder à une variable :
- Par son nom (identificateur).
- Par son adresse, en utilisant un pointeur vers cette variable.

Les pointeurs offrent donc la possibilité d'accéder à une variable par son adresse.

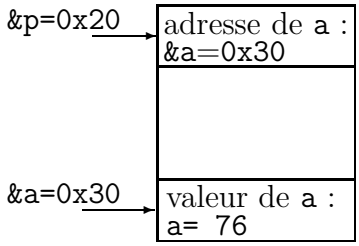

Figure 3.1 – Pointeur et variable pointée.

3.2.2.1 Opérateur *

Cet opérateur permet d'accéder au contenu de la variable pointée.

3.2.2.2 Opérateur &

Si cet opérateur est placé avant une variable, il permet de désigner son adresse.

```
1 int a, *p; // a : variable entière , p : pointeur vers un entier.
2 a = 76; //initialisation de a dont l'adresse est 0x30
3 p = &a; // initialisation du pointeur p
```

Listing 3.14 – Exemple de l'utilisation de l'opérateur &

Comme le montre la figure 3.1, le pointeur `p` pointe l'entier `a` dont l'adresse est `0x30`.

`a`	:	Désigne le contenu de `a`.
`&a`	:	Désigne l'adresse de `a`.
`p`	:	Désigne le contenu du pointeur, c'est-à-dire l'adresse de `a`
`*p`	:	Désigne le contenu pointé par le pointeur `p`, c'est à dire la variable `a`.
`&p`	:	Désigne l'adresse du pointeur `p`.

3.2.3 Arithmétique des pointeurs

L'adresse d'une variable qui occupe plusieurs octets correspond à l'adresse du premier octet occupé par cette variable. Quand on rajoute `1` au contenu d'un pointeur, il se déplace du même nombre d'octets qu'occupe la variable pointée. L'exemple suivant montre l'incrémentation d'un pointeur vers un entier.

Exemple 3.1 (Arithmétique des pointeurs)

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main()
6 {
```

```
7       int a=1;
8       int *p=&a;
9       cout<<"Adresse de a :"<<&a<<'\n';
10      cout<<"Adresse p    :"<<p<<'\n';
11      cout<<"Adresse p+1:"<<p+1<<'\n';
12      return 0;
13  }
```

Listing 3.15 – Exemple sur l'arithmétique des pointeurs

Ce programme affiche

```
1  Adresse de a :0x23ff74
2  Adresse p    :0x23ff74
3  Adresse p+1  :0x23ff78
```

Le pointeur `p` s'est décalé de `4` octets en lui rajoutant `1` qui est la taille de l'espace mémoire occupé par une variable de type `int`.

3.2.4 Pointeurs et tableaux

Le nom d'un tableau est un pointeur constant (non modifiable) vers celui-ci

$$\textit{nom du tableau} = \textit{adresse du tableau} = \textit{adresse du } 1^{er} \textit{ élément}$$

```
1  int A[10];
2  int *p;
3  p = A; //est équivalente à p = &A[0];
```

Listing 3.16 – Pointeurs et tableaux

Dans cet exemple, après l'instruction `p = A`, le pointeur `p` pointe sur `A[0]`. De façon générale, soit `p` un pointeur : si `p` pointe sur une composante quelconque d'un tableau, alors

- `(p+1)` pointe sur la composante suivante.
- `(p + i)` pointe sur la $i^{ème}$ composante devant `p`.
- `(p - i)` pointe sur la $i^{ème}$ composante derrière `p`.
- `*(p)` désigne le contenu de `A[0]`.
- `*(p + 1)` désigne le contenu de `A[1]`.
- `*(p + 2)` désigne le contenu de `A[2]`.
- `*(p + i)` désigne le contenu de `A[i]`.

Un tableau peut donc être déclaré de deux façons :

```
1  type identificateur [dimension ] ; // tableau statique
2  type *identificateur ; // pointeur seul
```

Listing 3.17 – Déclarations d'un tableau

L'analogie entre tableau et pointeur est illustrée dans l'exemple suivant :

Exemple 3.2 (Analogie pointeur-tableau)

```cpp
#include <iostream>

using namespace std;

int main()

{
    int tab[10];
    int *p ;
    p = tab;    //  équivaut à p = &tab[0];
    *p =1;      //  équivaut à tab[0] = 1;
    *(p+1) =2; //  équivaut à tab[1]=2;
    return 0;
}
```

Listing 3.18 – Exemple illustrant l'analogie entre pointeurs et tableaux

3.2.5 Pointeurs et chaînes de caractères

Les chaînes de caractères sont des tableaux spéciaux de caractères se terminant par le caractère de fin de chaîne '\0'.

Il est possible de définir une chaîne de caractères constante grâce à un pointeur vers le type `char` comme dans l'exemple suivant. Une chaîne de caractère définie de cette manière ne peut et ne doit être modifiée.

```cpp
char *p=  "Bonjour" ;
```

Listing 3.19 – Exemple de la déclaration d'une chaîne de caractères par un pointeur

3.2.6 Tableaux de pointeurs

Un tableau de pointeurs est un tableau dont les éléments sont de pointeurs vers des variables du type spécifié.

```cpp
int *p[5] ; //tableau de cinq pointeurs vers de int
```

Listing 3.20 – Exemple d'un tableau de pointeurs

Il ne faut pas confondre un tableau de pointeurs et un pointeur vers un tableau. Cette notation est justifié par la priorité de l'opérateur `[ ]` par rapport à l'opérateur `*`.

```cpp
int *p[5]  ; //tableau de cinq pointeurs vers des int int (*p)[5];
//pointeur vers un tableau int de 5 éléments
```

Listing 3.21 – Différence entre un tableau de pointeurs et un pointeur vers un tableau

3.2.7 Pointeurs vers des pointeurs

La syntaxe de déclaration est la suivante :

```
type **nom
```

Listing 3.22 – Syntaxe de la déclaration d'un pointeur vers un pointeur

avec deux opérateurs d'indirection * pour désigner que c'est un pointeur vers un pointeur vers une donnée du type spécifié. L'exemple suivant déclare un double pointeur vers un entier initialisé avec l'adresse d'un pointeur vers un entier lui même initialisé avec l'adresse d'un entier.

```
int i=1 ;
int *p=&i ;   //pointeur vers i
int **pp=&p ; //pointeur vers pointeur vers i
```

Listing 3.23 – Exemple d'un pointeur vers un pointeur

3.3 Allocation dynamique de la mémoire

3.3.1 L'opérateur `new`

L'opérateur `new` permet d'allouer de l'espace mémoire dynamiquement ; c.à.d. au moment de l'exécution du programme en fonction des besoins. Sa syntaxe est la suivante :

```
<nom pointeur> = new type [taille]
```

Listing 3.24 – Syntaxe de l'allocation dynamique de la mémoire

L'exemple suivant montre l'allocation d'un espace mémoire pour 10 variables de type `int`.

```
int *p ;
p=new int [10] ; //alloue de la mémoire pour 10 variables int
// et initialise le pointeur p avec le début
// du bloc mémoire
```

Listing 3.25 – Exemple d'une allocation dynamique de la mémoire

On alloue de l'espace mémoire pour 10 entiers et on récupère l'adresse de cet espace dans un pointeur de même type.

- Il est possible de faire l'allocation directement au moment de la déclaration du pointeur.
- Si l'allocation mémoire échoue faute d'espace, la valeur du pointeur sera égale à `NULL` (`NULL` est une constante prédéfinie du C++ qui vaut zéro).
- Il est conseillé de vérifier que le pointeur n'est pas égal à `NULL` après l'utilisation de `new`.

L'exemple suivant montre l'allocation mémoire au moment de la déclaration et le test si l'allocation mémoire est réussie.

```
int *p =new  int [10] ;
if (p==NULL) cout<< "L'allocation mémoire a échoué " ;
```

Listing 3.26 – Test de la réussite de l'allocation dynamique de la mémoire

L'allocation dynamique de la mémoire permet de définir des *tableaux dynamiques* dont la dimension est spécifiée au moment de l'exécution comme le montre l'exemple suivant

```
int n;
cout<< "Nombre d'éléments ";
cin>>n ;
int *p =new  int [n] ;
```

Listing 3.27 – Choix en-ligne de l'espace mémoire à allouer

3.3.2 L'opérateur `delete`

L'opérateur `delete` permet de libérer l'espace mémoire alloué par `new`.

```
delete p ;
p=0 ;
```

Listing 3.28 – Libération de l'espace mémoire alloué

Même si on appel l'opérateur `delete` sur un pointeur, celui-ci continue d'avoir la valeur de l'adresse de ce espace mémoire. Après l'appel de `delete` sur un pointeur, il faut le remettre à `0`.

3.4 Les structures

Une structure est un ensemble d'éléments de types différents réunis sous un même nom.

3.4.1 Déclaration d'une structure

La syntaxe de la déclaration d'une structure est la suivante :

```
struct NomStructure { type1 champ1 ; type2 champ2 ; } ;
```

Listing 3.29 – Syntaxe de la déclaration d'une structure

La déclaration d'une structure correspond à la déclaration d'un nouveau type de données ; on définit ainsi un modèle. Les constituants d'une structure sont appelés *champs* (ou *membres*). Après la définition du modèle de la structure, il est possible de déclarer des variables de ce nouveau type par la même syntaxe utilisée pour les types prédéfinis.

Dans l'exemple suivant, on déclare une structure `article` et deux variables de cette structure

```
struct article
{
    int numero ;
    int qte ;
    float prix ;
} ; //Définition de la structure article
article art1, art2 ; // déclaration de 2 variables structre
```

Listing 3.30 – Exemple de définition d'une structure

3.4.2 Manipulation d'une structure

L'accès à un champ d'une variable structure se fait en suivant d'un point ('.') le nom de la variable structure, suivi du nom du champ. Ce champ peut alors être manipulé comme n'importe quelle variable du type correspondant.

```
art1.numero = 15 ;
cin >> art2.prix ;
art1.numero++ ;
```

Listing 3.31 – Exemple de manipulation d'une structure

3.4.3 Utilisation globale d'une structure

Il est possible d'affecter à une structure le contenu d'une autre structure du même type. On peut donc écrire

```
art1 = art2 ;
```

Listing 3.32 – Exemple d'affectation globale d'une structure

au lieu de

```
art1.numero = art2.numero ;
art1.qte = art2.qte ;
art1.prix = art2.prix ;
```

3.4.4 Initialisation d'une structure

Comme pour un tableau, une structure peut être initialisée au moment de sa création en énumérant ses champs

```
article art1 = { 100, 285, 200 } ;
```

Listing 3.33 – Initialisation par énumération d'une structure

Une structure peut être aussi initialisée avec les éléments d'une structure du même type

```
struct article
{
    .....
} ;
main()
{
    article e1 = { ..... } ;
    Article e2 = e1 ; // les valeurs des champs de e1 sont recopiés
    // dans ceux de e2
    ...
}
```

Listing 3.34 – Initialisation d'une structure avec les éléments d'une autre structure

Chapitre 4

Les fonctions

4.1 Généralités

L'utilisation des fonctions a pour but de décomposer un problème en sous-problèmes moins complexes et les traiter séparément. L'autre avantage des foncions est la réutilisabilité du code. Si une fonction est suffisamment générique, elle peut être utilisée dans un autre contexte.

4.1.1 Définition et avantages des fonctions

Une fonction est un tout homogène réalisant une tâche définie. En résumé, les fonctions permettent d'atteindre les objectifs suivants :
- Simplifier la réalisation et la maintenance des programmes.
- Éviter les répétitions.
- Favoriser la réutilisabilité des codes.

4.1.2 Caractéristiques des fonctions

En C++, une fonction est caractérisée par :

1. **Son emplacement :** Elle peut être définie dans le fichier du programme principal (`main()`) ou dans un autre fichier.
2. **Son nom :** par lequel on peut l'appeler à partir du `main()`, d'une autre fonction ou à partir d'elle-même (fonctions récursives).
3. **Ses arguments :** C'est les données qu'on lui transmet au moment de son appel.
4. **Le type de sa valeur de retour :** Si la fonction n'a pas de retour, son type de retour est `void`.
5. **Ses variables locales :** Elles sont définies et connues seulement à l'intérieur de la fonction.

Une fonction est constituée d'un *entête* et d'un *corps*. Ces éléments seront détaillés dans la section 4.4.

4.2 Déclaration d'une fonction, prototype

Il est nécessaire de faire la différence entre la *déclaration* et la *définition* d'une fonction. Ces deux opérations peuvent être faites séparément ou en même temps.

4.2.1 Prototype d'une fonction

Toute fonction utilisée dans le programme principal doit être déclarée avant le `main()`. La déclaration d'une fonction définit son *prototype* ou sa *signature*. Un prototype donne la règle d'usage de la fonction : nombre et type d'arguments d'entrée ainsi que le type de la valeur retournée par la fonction.

La déclaration (prototypage) d'une fonction consiste à indiquer :

1. Le nom de la fonction, qui est son *identificateur*.
2. Le type de la valeur retournée (type de la fonction) : C'est le type de la variable ou de l'expression que la fonction retourne. Si la fonction ne retourne aucune valeur, elle est de type `void`.
3. Le type des paramètres (arguments) de la fonction. Ils sont donnés entre parenthèses et séparés par des virgules.

La déclaration d'une fonction se termine toujours par un point virgule `;`.

4.2.1.1 Syntaxe de déclaration

```
1 <type fonction> <nom fonction> (type 1, type 2,...) ;
```

Listing 4.1 – Syntaxe de la déclaration d'une fonction

Trois exemples de déclaration de fonctions sont présentés ci-dessous

Premier exemple

```
1 int fct1(int,int,float) ;
```

Listing 4.2 – Déclaration d'une fonction : Premier exemple

La fonction `fct1()` admet trois paramètres : Deux entiers et un réel. Elle retourne en résultat un entier. Elle est donc de type `int`.

Deuxième exemple

```
1 float fct2() ;
```

Listing 4.3 – Déclaration d'une fonction : Deuxième exemple

La fonction `fct2()` ne reçoit aucun paramètre (parenthèses vides) et retourne un réel (de type `float`). On aurait pu également écrire `float fct2(void)` ;

Troisième exemple

```
1 void fct3(int) ;
```

Listing 4.4 – Déclaration d'une fonction : Troisième exemple

La fonction `fct3()` reçoit en paramètre un entier et ne retourne aucune valeur (`void`).

4.3 Appel d'une fonction

L'appel d'une fonction peut se faire dans le `main()` ou dans une autre fonction. L'appel d'une fonction se fait en mentionnant son *nom* et ses *paramètres* entre parenthèses. Si la fonction ne possède pas de paramètres, les parenthèses sont vides.

Syntaxe d'appel

```
1 <nom fonction> (paramètre1, paamètre2,...) ;
```

Listing 4.5 – Syntaxe d'appel d'une fonction

Les paramètres de la fonction doivent avoir une valeur au moment de l'appel de la fonction. Ces valeurs peuvent êtres des variables, des constantes ou des expressions.

4.4 Définition d'une fonction

La définition d'une fonction consiste à écrire les instructions qui la composent. Elle se fait généralement après le `main()`. Elle comporte les éléments suivants :

1. **L'entête :** Elle est compatible à son prototype (déclaration).
2. **Le corps de la fonction :** C'est un bloc délimité par deux accolades { et } contenant les instructions qui composent la fonction. Les variables déclarées dans le corps de la fonction sont appelées variables locales. Leur portée est limitée au corps de la fonction (elles sont détruites après l'appel de la fonction).
3. **L'instruction `return` :** Elle permet la sortie immédiate de la fonction et de transmettre la valeur.

4.4.1 Syntaxe de définition

La syntaxe de définition d'une fonction est la suivante

```
1 <type fonction> <nom fonction> (type 1 p1, type 2 p2,...)
2 {
3 ...;// instructions de la fonction
4 }
```

Listing 4.6 – Syntaxe de la définition d'une fonction

`type 1`, `type 2` : désignent les types des paramètres de la fonction.
`p1`, `p2` : désignent les noms des paramètres (arguments) de la fonction.

4.4.2 Argument muets (fictifs) et arguments effectifs

Les arguments figurant dans l'en-tête de la définition de la fonction sont appelés *arguments muets*, *arguments formels* ou *paramètres formels*. Ce sont des variables qui ont une portée locale et qui sont initialisées avec les arguments fournis à la fonction lors de son appel.

Les arguments fournis lors l'appel de la fonction se nomment des *arguments effectifs* ou *paramètres effectifs*. Ces arguments peuvent êtres des variables, des constantes ou des expressions. C'est cette expression qui sera alors transmise à la fonction.

Exemple 4.1 (Fonction `somme`)

```
#include <iostream>

using namespace std;

int somme(int,int);//Déclaration de la fonction somme
int main()
{
    int a,b,c;
    a=3;
    b=4;
    c=somme(a,b);//appel de la fonction somme
    cout<<a <<" + " <<b <<" = " <<c;
    return 0;
}
//Définition de la fonction somme
int somme(int x,int y)
{
    int z ;//variable locale
    z=x+y;
    return z;//retourne la valeur de z
}
```

Listing 4.7 – Déclaration, définition et appel d'une fonction

Dans cet exemple, on déclare d'abord une fonction `somme` avant le `main` qui reçoit deux entiers et revoie leur somme. La définition de la fonction est faite après le `main`. Dans le `main`, on appelle la fonction `somme` avec comme argument effectif les deux variables `a` et `b` et on affecte son résultat à la variable entière `z` qui est une variable locale.

4.4.3 Variables locales

Une fonction peut avoir ses propres variables qui sont déclarées lors de sa définition. La portée des variables est *locale* à la fonction qui les contient. Les variables déclarées dans le `main` sont locales au `main` et les variables déclarées dans la fonction sont locales à la fonction, même si elles portent le même nom.

4.4.4 Variables globales

Les variables globales sont déclarées avant le `main`. Elles sont connues au sein de toutes les fonctions qui sont compilées au sein du même programme source. L'exemple suivant montre l'utilisation d'une variable globale par une fonction

Exemple 4.2 (Fonctions et variables globales)

```
#include <iostream>

using namespace std;

void fct();//Déclaration de la fonction fct
int i;//varaible globale
int main()
{
    i=5;//pas beson de declarer i, variable globale
    int j=6;//variable locale
    cout<<"\n Avant appel de fct(): i="<<i;
    cout<<"\n Avant appel de fct(): j="<<j;
    fct();
    cout<<"\n Après appel de fct(): i="<<i;
    cout<<"\n Après appel de fct(): j="<<j;
    return 0;
}

void fct()
{
    int j=3;//varaible locale
    i=i+5;
    j=j+5;
    cout<<"\n Dans fct(): i="<<i;
    cout<<"\n Dans fct(): j="<<j;
}
```

Listing 4.8 – Exemple d'une fonction utilisant des variables globales

Ce programme affiche le résultat suivant

```
Avant appel de fct(): i=5
Avant appel de fct(): j=6
Dans fct(): i=10
Dans fct(): j=8
Après appel de fct(): i=10
Après appel de fct(): j=6
```

Le contenu de la variable locale `j` reprend sa valeur initiale dans `main()` après l'appel de la fonction. Par contre, la variable globale `i` est modifiée et garde cette modification après l'appel de la fonction.

4.5 Passage d'arguments aux fonctions

Toute expression valide du même type que l'argument muet peut être passé en paramètre à une fonction. Il peut s'agir d'une constante, d'une expression mathématique ou logique ou d'une fonction retournant une valeur.

Les arguments peuvent êtres passés à une fonction de trois manières :

1. Par valeur.
2. Par pointeur.
3. Par référence.

4.5.1 Le passage par valeur

Le passage par valeur consiste à fournir à la fonction appelée une copie d'une variable de la fonction appelante. Il s'agit de deux variables différentes, l'une étant simplement initialisée avec la valeur de l'autre. L'exemple suivant est celui d'une fonction recevant ses arguments par un passage par valeur

Exemple 4.3 (Passage d'arguments par valeur)

```
#include <iostream>

using namespace std;

void PassageParValeur(int i) //Déclaration et définition
{
    i=10;
}
int main()
{
    int i=0;
    PassageParValeur(i);
    cout<<"Valeur de i :"<<i;
    return 0;
}
```

Listing 4.9 – Exemple d'une fonction recevant ses arguments par valeur

Le programme affiche

```
Valeur de i : 0
```

La variable `i` n'est pas modifiée par l'appel de la fonction `PassageParValeur`, car les variables `i` dans `PassageParValeur` et `main` sont différentes même si elles portent le même nom.

4.5.2 Le passage par pointeur

Appelé aussi *passage par adresse*, le passage par pointeur permet de transmettre à une fonction l'adresse d'une variable. Celle-ci est stockée dans un argument de type pointeur qui peut être par la suite utilisé pour modifier la variable pointée. Les changements effectués à partir de cette adresse affectent aussi la variable de la fonction appelante. L'exemple qui suit illustre le passage d'arguments par pointeur

Exemple 4.4 (Passage d'arguments par adresse)

```
#include <iostream>

using namespace std;

void PassageParPointeur(int *var) //Déclaration et définition
{
    *var=10;//modification du contenu pointé
}
int main()
{
    int i=0;
    PassageParPointeur(&i);
    cout<<"Valeur de i :"<<i;
    return 0;
}
```

Listing 4.10 – Exemple d'une fonction recevant ses arguments par adresse

Le programme affiche

```
Valeur de i : 10
```

La variable `i` est modifiée suite à l'appel de la fonction `PassageParAdresse`.

4.5.3 Le passage par référence

Les références permettent de désigner des variables déclarées dans une autre fonction. Dans d'autres langages, les références sont nommées *alias*, c'est à dire un nom de variable qui désigne une autre variable. Les références fournissent un concept proche de celui des pointeurs et peuvent s'utiliser dans le même contexte.

La déclaration d'une référence se fait en plaçant l'opérateur & entre le type de la référence et son nom. Les références **doivent** être initialisées au moment de leur déclaration. Le passage par référence est illustré par cet exemple

Exemple 4.5 (Utilisation des références)

```
#include <iostream>

using namespace std;

int main()
{
    int a;
    int & ref=a; //declaration d'une reference vers a
    ref=10;//equivalent à a=10
    cout<<"a = "<<a<<endl;
    cout<<"ref ="<<ref;
    return 0;
}
```

Listing 4.11 – Exemple de l'utilisation des références

Le passage d'argument par référence constitue une solution hybride entre le passage par valeur et par pointeur. C'est la variable et non l'adresse qui est passée à la fonction appelée, cependant, la référence permet de modifier la variable de la fonction appelante.

Exemple 4.6 (Passage d'arguments par référence)

```
#include <iostream>

using namespace std;

void PassageParReference(int & var)//Déclaration et définition
{
    var=10;//modification du contenu pointé
}
int main()
{
    int i=0;
    PassageParReference(i);
    cout<<"Valeur de i :"<<i;
    return 0;
}
```

Listing 4.12 – Exemple d'une fonction recevant ses arguments par référence

Le programme affiche

```
Valeur de i : 10
```

4.6 Passage et retour d'un tableau à une fonction

Une fonction peut recevoir en argument et retourner des tableaux.

4.6.1 Passage d'un tableau à une fonction

Une fonction peut recevoir en argument un tableau de n'importe quel type.

4.6.1.1 Syntaxe de déclaration :

```
<type fonction><nom fonction> (<type tableau> [])
```

Listing 4.13 – Syntaxe d'une fonction recevant un tableau en argument

4.6.1.2 Syntaxe d'appel

```
<nom fonction> (<nom tableau>)
```

Listing 4.14 – Syntaxe d'appel d'une fonction recevant un tableau en argument

L'exemple suivant montre des fonctions utilisant des tableaux

Exemple 4.7 (Fonctions et tableaux)

```
#include <iostream>

using namespace std;

void affiche_num(int []);//Déclaration

int main()
{
    int entiers[]= {4,5,8,9};
//cout<<sizeof(entiers)/sizeof(int);
    affiche_num(entiers);
    return 0;
}
void affiche_num(int tab[])//Définition
{
    int i;
    for (i=0; i<4; i++)
        cout<<tab[i]<<" ";
}
```

Listing 4.15 – Exemple d'une fonction utilisant les tableaux

Il est également possible de passer l'adresse du premier élément du tableau à travers son identificateur, en faisant un passage par pointeur. Ceci est montré sur l'exemple suivant

Exemple 4.8 (Fonctions, tableaux et pointeurs)

```cpp
#include <iostream>

using namespace std;

void affiche_num(int*);//Déclaration

int main()
{
    int entiers[]= {4,5,8,9};
    affiche_num(entiers);
    return 0;
}
void affiche_num(int* tab)//Définition
{
    int i;
    for (i=0; i<4; i++)
        cout<<tab[i]<<" ";
}
```

Listing 4.16 – Exemple de passage d'un tableau par pointeur

4.6.2 Retour d'un tableau par une fonction

Une fonction peut retourner un tableau de n'importe quel type à la fonction appelante. Le retour d'un tableau par une fonction se fait par adresse, soit en retournant un pointeur vers le premier élément du tableau.

4.6.2.1 Syntaxe de déclaration :

```
<type fonction> * <nom fonction> (<types parametres> )
```

Listing 4.17 – Syntaxe de l'appel d'une fonction renvoyant un pointeur

4.6.2.2 Syntaxe d'appel

```
<nom fonction>(parametres)
```

Listing 4.18 – Syntaxe de l'appel d'une fonction renvoyant un pointeur

4.7 Passage d'une structure à une fonction

Le passage d'une structure peut se faire soit par valeur ou par adresse, à travers un pointeur vers le type structure. Dans ce deuxième cas, l'accès aux champs de la structure se fait à travers l'opérateur `->` appliqué au pointeur. Il est nécessaire cependant de déclarer la structure avant la déclaration des fonctions qui l'utilisent. L'exemple suivant montre des fonctions qui manipulent des structures

Exemple 4.9 (Fonctions et structures)

```cpp
#include <iostream>

using namespace std;

struct etudiant
{
    char* nom;
    char* prenom;
    int age;
};
void AfficheEtudiantValeur(etudiant);//Déclaration
void AfficheEtudiantAdresse(etudiant*);//Déclaration
int main()
{
    etudiant personne1= {"NOM_ETUDIANT","PRENOM_ETUDIANT",22};
    etudiant* ptr_personne1=&personne1;
    cout<<"PASSAGE PAR VALEUR\n\n";
    AfficheEtudiantValeur(personne1);
    cout<<"PASSAGE PAR ADRESSE\n\n";
    AfficheEtudiantAdresse(ptr_personne1);
    return 0;
}
void AfficheEtudiantValeur(etudiant etd)//Définition
{
    cout<<"Nom : "<<etd.nom<<endl;
    cout<<"Prenomom : "<<etd.prenom<<endl;
    cout<<"Age : "<<etd.age<<"\n\n";
}
void AfficheEtudiantAdresse(etudiant *etd)//Définition
{
    cout<<"Nom : "<<etd->nom<<endl;
    cout<<"Prenomom : "<<etd->prenom<<endl;
    cout<<"Age : "<<etd->age<<"\n\n";
}
```

Listing 4.19 – Exemple de passage d'une structure à une fonction

4.8 Pointeur vers une fonction

Un pointeur de fonctions pointe vers l'adresse du code de la fonction en mémoire. Il contient l'adresse de cette fonction. Une fois initialisé, le pointeur permet d'appeler la fonction de la même manière que son nom.

Syntaxe de déclaration :

```
<type retour> (* <nom poniteur>) (<types parametres> )
```

Listing 4.20 – Syntaxe de la déclaration d'un pointeur vers une fonction

Dans cette ligne de code, on déclare un pointeur vers une fonction retournant un réel et ayant pour arguments deux entiers

```cpp
float (*pfct) (int, int) ;
```

Listing 4.21 – Exemple de déclaration d'un pointeur vers une fonction

L'exemple suivant illustre l'utilisation des pointeurs vers des fonctions

Exemple 4.10 (Pointeur vers une fonction)

```
#include <iostream>

using namespace std;

int somme (int,int);
int produit(int,int);
int main()
{
    int a=2,b=3;
    int (* pfct)(int,int);
    pfct=somme;
    cout<<"a + b = "<<pfct(a,b)<<endl;
    pfct=produit;
    cout<<"a * b = "<<pfct(a,b)<<endl;
    return 0;
}
int somme(int x, int y)
{
    return x+y;
}
int produit(int x, int y)
{
    return x*y;
}
```

Listing 4.22 – Exemple de l'utilisation d'un pointeur vers une fonction

Remarque 4.1
Les parenthèses permettent de distinguer entre un pointeur de fonction et une fonction renvoyant un pointeur

`int (* pfct)(int,int);` : Pointeur vers une fonction renvoyant un entier et admettant deux entiers comme arguments.

`int * pfct(int,int);` : Fonction renvoyant un pointeur vers un entier et admettant deux entiers comme arguments.

Chapitre 5

Les classes d'objets

5.1 Définitions

Les notions de classe et d'objet sont à la base de la programmation orientée objet. On commencera par donner leurs définitions.

5.1.1 Classes et objets

Une *classe* est un type de données défini par l'utilisateur, qui regroupe à la fois des données (variables) et des traitements (fonctions). Une classe est le modèle à partir duquel les objets sont créés. Un objet est une *instance* de la classe à partir de laquelle il a été déclaré. Lorsqu'on crée un objet à partir d'une classe, on dit que l'on crée *une instance* de cette classe.

5.1.2 Données membres et fonctions membres

Les données membres, appelées aussi variables membres ou attributs, sont les données, de types différents, qui sont définies à l'intérieur de la classe.

Les fonctions membres, appelées aussi méthodes membres, sont les fonctions définies au sein de la classe. Elles constituent l'ensemble des traitements à mettre en œuvre avec les objets de la classe. On dit aussi qu'une fonction membre est la l'implémentation d'un traitement.

5.2 Déclaration d'une classe

La déclaration d'une classe se fait avec le mot-clé `class` suivi d'accolades à l'intérieur desquelles on déclare les variables et les fonctions membres. La déclaration se termine par un point virgule.

5.2.1 Syntaxe de déclaration

```
1 class <nom de la classe>
2 {
```

```
//déclaration des variables membres
    type1 variable1 ;
    type2 variable2 ;
    ...
//déclaration des fonctions membres
    type1 fonction1(type_arg1,type_arg2,...) ;
    type2 fonction2(type_arg1,type_arg2,...) ;
    ...
} ;
```

Listing 5.1 – Syntaxe de la déclaration d'une classe

Un exemple est donné ci-dessous

```
class ordinateur
{
    char reference[15] ;
    char marque[15];
    int disque ;
    int memoire ;
    void afficher() ;
};
```

Listing 5.2 – Exemple de déclaration de classe

On a déclaré une classe `ordinateur` qui comporte quatre données membres et une fonction membre `afficher()`.

5.3 Définition des fonctions membres

Elles représentent l'ensemble des traitements qui peuvent êtres mis en œuvre avec les objets de la classe.

5.3.1 Définition en-ligne des fonctions membres

Il est possible de définir les fonctions membres (le corps de ces fonctions) à l'intérieur de la classe, on définit alors l'entête et le corps de la fonction au moment de sa déclaration, cette définition s'appelle *définition en-ligne*.

Exemple 5.1 (Fonctions membres en-ligne)

```
#include <iostream>

using namespace std;

class chien
{
    char race[15] ;
    int age,poid ;
public:
    void creer()
    {
        cout<<"\n donner la race ";
        cin>>race ;
        cout<<"\n donner l'age ";
        cin>>age;
        cout<<"\n donner le poid ";
```

```
17          cin>>poid;
18      }
19      void afficher()
20      {
21          cout<<"\n Race : "<<race;
22          cout<<"\n Age  : "<<age;
23          cout<<"\n Poid : "<<poid;
24      }
25  };
26  int main()
27  {
28      chien dog1;
29      dog1.creer();
30      dog1.afficher();
31      return 0;
32  }
```

Listing 5.3 – Exemple de définition en-ligne des fonctions membres

On a défini une classe `chien` avec deux fonctions membres `creer()` et `afficher()` qui sont définies directement à l'intérieur de la classe.

5.3.2 Définition déportée des fonctions membres

Dans la majorité des cas, on utilise *une définition déportée* :

1. On déclare le prototype de la fonction membre à l'intérieur de la classe,
2. On définit le corps de la fonction en dehors de cette classe en utilisant le nom complet de la fonction.

La définition des fonctions membres se fait à l'extérieur de la classe en indiquant son type, le nom de la classe qui la contient, suivi de quatre points `::`, le nom de cette fonction, et enfin ses arguments entre parenthèses.

Les fonctions membres accessibles aux utilisateurs de la classe sont précédées du mot clé `public :`. Elles constituent une interface public qui définit la façon avec laquelle on peut interagir avec les objets de cette classe. L'exemple suivant illustre la définition déportée de fonctions membres

Exemple 5.2 (Fonctions membres déportées)

```
1   #include <iostream>
2
3   using namespace std;
4
5   class chien
6   {
7       char
8       race[15] ;
9       int age,poid ;
10  public:
11      void creer();
12      void afficher();
13  };
14  void chien::creer()
15  {
16      cout<<"\n donner la race ";
17      cin>>race ;
18      cout<<"\n donner l'age ";
19      cin>>age;
20      cout<<"\n donner le poid ";
```

```
21      cin>>poid;
22  }
23  void chien::afficher()
24  {
25      cout<<"\n Race  : "<<race;
26      cout<<"\n Age   : "<<age;
27      cout<<"\n Poid  : "<<poid;
28  }
29  int main()
30  {
31      chien dog1;
32      dog1.creer();
33      dog1.afficher();
34      return 0;
35  }
```

Listing 5.4 – Exemple de définition déportée des fonctions membres

L'expression `chien::afficher()` définit la fonction membre `afficher()` de la classe `chien`. L'opérateur `::` s'appelle *opérateur de résolution de portée*.

5.4 Déclaration des objets

Un objet est *une instance d'une classe*. C'est une variable dont le type est cette classe. La déclaration d'un objet se fait alors de la même façon que pour la déclaration d'une variable d'un autre type.

5.4.1 Syntaxe de déclaration d'un objet

```
1  <nom de la classe>  objet_1, objet_2, ..., objet_n ;
```

Listing 5.5 – Syntaxe de la déclaration d'un objet

5.5 Accès aux membres d'une classe

Pour un objet donné, l'accès aux membres de la classe (variables et fonctions membres) se fait en suivant d'un point, comme dans le cas des structures, le nom de l'objet, suivi du nom de la variable ou de la méthode.

Exemple 5.3 (Accès au membres d'une classe)

```
1   ##include <iostream>
2
3   using namespace std;
4
5   class client
6   {
7   public:
8       int code;
9       void initialiser(int);
10  };
11
12  void client::initialiser(int n)
13  {
14      code=n;
```

```
}
int main()
{
    client personne1;
    personne1.code=10;
    client personne2;
    personne2.initialiser(10);
    return 0;
}
```

Listing 5.6 – Exemple d'accès aux membres d'une classe

5.5.1 Membres privés et membres publics

Par défaut, les membres d'une classes sont privés, ils ne sont pas accessibles en-dehors de la classe. Le mot clé `public :` à l'intérieur de la déclaration d'une classe indique que tous les membres (variables et fonctions) qui suivent sont accessibles en dehors de la classe. A l'opposée, le mot clé `private :` indique que tous les membres qui suivent sont privés. Ils sont alors inaccessibles de l'extérieur de la classe.

On définit des fonctions membres publics qui permettent d'accéder aux variables membres privées de la classe pour les récupérer ou les modifier :

- *Les getter* (ou accesseurs) permettent d'accéder aux variables.
- *Les setter* (ou altérateurs) permettent de modifier ces variables privées.

Les mots clés `public` et `private` peuvent apparaître à plusieurs reprises dans la définition d'une classe :

```
class A
{
private :
...
public :
...
private :
...
} ;
```

Listing 5.7 – mots clés `public` et `private`

Dans cet exemple, on définit une classe `client` avec des membres publics et des membres privés.

Exemple 5.4 (Membres publics et privés d'une classe)

```
#include <iostream>

using namespace std;

class client
{

public:
    char nom[15];
    int getCode ();
    void setCode (int);
private :
    int code;
};
```

```
15
16 int client::getCode()
17 {
18     return code;
19 }
20 void client::setCode(int n)
21 {
22     code=n;
23 }
24 int main()
25 {
26     client personne1;
27 //personne1.code=10; //non autorisée : code  variable privée
28     client personne2;
29     personne2.setCode(10);
30     cout<<"Code="<<personne2.getCode();
31     return 0;
32 }
```

Listing 5.8 – Exemple de choix de visibilité des membres d'une classe

5.5.2 Encapsulation

L'encapsulation est l'un des fondements de la POO. Elle permet d'utiliser un objet en cachant son fonctionnement interne. Grâce à l'encapsulation, les données privées ne sont pas accessibles de façon directe. Ceci permet de sécuriser les données qui ne peuvent pas êtres altérées par un utilisateur extérieur.

Les utilisateurs de l'objet disposent d'une interface public pour utiliser l'objet. L'accès aux données privées se fait alors uniquement par les fonctions membres de la classe. Un autre avantage de l'encapsulation est de rendre l'implémentation de la classe transparente aux utilisateurs. Tant que l'interface public de la classe n'a pas changée, les utilisateurs peuvent continuer à utiliser la classe de la même façon même si l'implémentation de ses données a changée. Grâce à l'encapsulation, on peut changer la façon dont les données sont déclarées et stockées sans changer la façon dont elles sont utilisées.

5.5.3 L'objet courant et le mot-clé `this`

Le mot-clé `this` correspond à une variable prédéfinie par le C++. Il désigne un pointeur vers l'objet courant, c'est-à-dire l'objet pour lequel on appel une fonction membre.

La variable `this` est passée en tant qu'argument caché à toutes les fonctions membre. De cette façon, toutes les fonctions membres auront accès à tous les membres de la classe. Le compilateur modifie le code des fonctions source pour ajouter le pointeur `this` comme paramètre. Il va également référencer toutes les données membre comme des champs de l'objet courant en ajoutant l'expression `this->` devant ces champs.

Il est possible d'utiliser le pointeur `this` pour désigner l'objet courant à l'intérieur d'une fonction membre. Les méthodes `static` ne connaissent pas l'objet courant. L'argument `this` ne leurs est pas transmis et il ne peut pas être utilisé à l'intérieur d'un fonction `static`. C'est pour cette raison qu'elles sont appelées *méthodes de classe*.

5.5.4 Affectation des objets

Comme pour les structures, et contrairement aux tableaux, il est possible d'affecter un objet à un autre objet à condition qu'ils soient de la même classe. Le listing 5.9 recopie les données de l'objet `personne1` vers `personne2`

```
...
client personne1,personne2;
personne1.setCode(10);
cout<<"Code="<<personne1.getCode();
personne2=personne1;
cout<<"Code="<<personne2.getCode();
...
```

Listing 5.9 – Affectation d'objets

L'affectation recopie toutes les variables membres de l'objet à droite du = vers celle l'objet à gauche. L'affectation globale entre objet est toujours possible même s'il existe des membres privés. Par contre, l'affectation entre variables membres est possible seulement si elles sont publics. L'écriture

```
personne1.code=personne2.code;
```

va générer une erreur de compilation car l'attribut `code` est privé.

5.6 Constructeurs et destructeurs

5.6.1 Constructeurs

Un constructeur est une fonction membre spéciale qui a les particularités suivantes :

1. Il possède le même nom que la classe, et ne possède pas de type de retour, même pas `void`.
2. Si aucun constructeur n'est défini, le compilateur défini un constructeur par défaut qui ne fait rien.
3. Le constructeur par défaut est appelé chaque fois qu'un objet est déclaré. Il ne comporte pas de paramètres.
4. Il est toujours déclaré `public` :
5. Son rôle est d'initialiser les objets de la classe au moment de leur déclaration.
6. On peut définir d'autres constructeurs que le constructeur par défaut qui reçoivent des paramètres, par surcharge de celui-ci.

5.6.2 Destructeurs

Comme le constructeur, le destructeur est une fonction membre spéciale qui possède les propriétés suivantes :

1. Il porte le même nom que la classe et il est déclaré `public`.

2. Son nom est précédé du symbole tilde `~`.
3. Son rôle est de libérer l'espace mémoire alloué aux objets qui ne sont plus utilisés.
4. C'est une fonction membre de la classe qui n'a pas de type de retour (même pas `void`) et ne reçoit pas de paramètres.
5. Contrairement aux constructeurs, il n'existe qu'un seul destructeur.
6. Il est appelé automatiquement à la fin du programme.

Dans le listing 5.10, on définit une classe rectangle avec un constructeur et un destructeur

Exemple 5.5 (Constructeurs et destructeurs)

```
#include <iostream>

using namespace std;

class rectangle
{
private :
    int largeur,longueur;
public:
    rectangle();
    rectangle(int,int);//constructeur avec arguments
    ~rectangle();
};

rectangle::rectangle()
{
    cout<<"Execution du constructeur par défaut "<<endl;
}
rectangle::rectangle(int a,int b)
{
    largeur=a;
    longueur=b;
}
rectangle::~rectangle()
{
    cout<<"Destruction de l'objet\n";
}
int main()
{
    rectangle rect1;
    rectangle rect2(10,5);
    return 0;
}
```

Listing 5.10 – Exemple d'utilisation des constructeurs et destructeurs d'une classe

Le programme affiche

```
Execution du constructeur par défaut
Destruction de l'objet
Destruction de l'objet
```

Les deux objets `rect1` et `rect2` sont détruits à la fin du programme.

Chapitre 6

Classes, variables et fonctions spéciales

6.1 Les données membres statiques

On commencera d'abord par donner la définition générale de données statiques en-dehors du concept de classe. Par la suite, cette notion sera abordée dans le cas d'une classe d'objet.

6.1.1 Les variables locales statiques

Rappelons que les variables locales sont des variables définies au sein de la fonction où elles sont déclarées. Elles sont connues à l'intérieur de la fonction et sont détruites après l'appel de la fonction (voir section 4.4.3). Toutefois, il est possible d'attribuer un emplacement permanent à une variable locale et de conserver sa valeur d'un appel à l'autre. Ces variables sont dites *statiques*. Leur déclaration est précédée du mot-clé `static`.

Exemple 6.1 (Variables statiques)

```
#include <iostream>

using namespace std;

void fct() ;
int main()
{
    for ( int n=1 ; n<=3 ; n++)
        fct() ;
    return 0;
}
void fct()
{
    static int i=0 ;
    int j=0;
    i++ ;
    j++;
    cout << " i= " << i << "\n" ;
    cout << " j= " << j << "\n" ;
}
```

Listing 6.1 – Exemple de déclaration de variables statiques

Le programme affiche

```
i=1
j=1
i=2
j=1
i=3
j=1
```

Les variables statiques locales sont initialisées au premier appel de la fonction. Par défaut, elles sont initialisées à zéro.

6.1.2 Les variables membres statiques

Des objets différents d'une même classe possèdent des variables membres différentes. Il est aussi possible que tous les objets d'une même classe *partagent* une ou plusieurs variables membres qui seront identiques pour tous ces objets. Ces variables partagées sont appelées *variables statiques* et leur déclaration est précédé du mot-clé `static`.

Une variable statique a un emplacement permanent et existe en un seul exemplaire, même si aucune instance de la classe n'a été déclarée. Elle contient la même donnée pour toutes les instances. Les variables membres statiques peuvent êtres vues comme des variables globales dont la portée est limitée à cette classe. La syntaxe de déclaration d'une variable membre statique est la suivante

```
class nom_classe {
static type <nom variable>
} ;
```

Listing 6.2 – Syntaxe de déclaration d'une variable membre statique

Sur le listing 6.3, la variable `n` est une variable statique commune aux objets `a` et `b` de la classe `maclasse`. Par contre, chaque objet à une variable membre `x` différente qui lui est personnelle.

Exemple 6.2 (Variables membres statiques)

```
#include <iostream>

using namespace std;

class maclasse
{
    static int n ;
    double x ;
} ;
int main()
{
    maclasse a, b ;
    return 0;
}
```

Listing 6.3 – Exemple de déclaration de variables membres statiques

6.1.3 Initialisation des variables membres statiques

Les variables membres statiques sont initialisées *en dehors* de la déclaration de la classe. Ceci permet d'éviter l'allocation d'emplacements différents si des modules objets, utilisant cette classe, sont compilés séparément.

La variable membre statique est initialisée explicitement, à l'extérieur de la déclaration de la classe.

```
class nom_classe
{
    static type <nom variable>
} ;
type nom_classe ::<nom variable>=valeur initiale ;
```

Listing 6.4 – Syntaxe d'initialisation d'une variable membre statique

L'exemple suivant illustre la déclaration d'une variable membre statique `nb_instances` qui compte le nombre d'instances existantes à un instant donné

Exemple 6.3 (Utilisation des variables membres statiques)

```
#include <iostream>

using namespace std;

class objet
{
public:
    int code;
    static int nb_instances;
    objet (int x)
    {
        code=x;
        ++nb_instances;
        cout<<"Il y a "<<nb_instances<<" objets \n";
    }
    ~objet ()
    {
        --nb_instances;
        cout<<"Il reste "<<nb_instances<<" objets \n";
    }
};
int objet::nb_instances=0;
int main()
{
    objet obj1=objet(1);

    objet obj2=objet(2);
    return 0;
}
```

Listing 6.5 – Exemple d"utilisation des variables membre statiques

Le programme affiche

```
Il y a 1 objets
Il y a 2 objets
Il reste 1 objets
Il reste 0 objets
```

6.2 Les fonctions membres statiques

Comme pour une donnée membre statique, une *fonction membre statique* est partagée par toutes les instances d'une même classe. Au lieu d'être appelé avec un objet de la classe, elle est appelée avec le nom de la classe.

Une fonction membre statique ne sait pas travailler sur un objet particulier de la classe. Elle ne récupère pas le pointeur système `this` (voir section 5.5.3). Par conséquent, elle ne peut pas accéder aux données membres d'un objet et elle ne peut pas appeler une fonction membre non statique.

Les fonctions membres statiques sont généralement utilisées pour manipuler les données statiques. Elles mettent en œuvre un traitement qui n'est pas lié aux objets de la classe

```
class nom_classe {
static type <nom variable>
public :
static type fonction(){return variable ;}
} ;
type nom_classe ::<nom variable>=valeur initiale ;
```

Listing 6.6 – Syntaxe de déclaration d'une fonction membre statique

Ci-dessous un exemple d'une fonction membre statique qui affiche le nombre d'instances

Exemple 6.4 (Fonctions membres statiques)

```
#include <iostream>

using namespace std;

class objet
{
public:
    int code;
    static int nb_instances;
    objet (int x)
    {
        code=x;
        ++nb_instances;
        cout<<"Il y a "<<nb_instances<<" objets \n";
    }
    ~objet ()
    {
        --nb_instances;
        cout<<"Il reste "<<nb_instances<<" objets \n";
    }
    static int affiche()
    {
        return nb_instances;
    }
};
int objet::nb_instances=0;
int main()
{
    objet obj1=objet(1);

    objet obj2=objet(2);
    cout<<"Nombre d'objets = "<<objet::affiche()<<'\n';
    return 0;
}
```

Listing 6.7 – Exemple d'utilisation des fonctions membre statiques

6.3 Portée d'une fonction

Les fonctions membres d'une classe ont une portée limitée aux objets cette classe. Leur appel est toujours précédé du nom de l'objet ou de la classe. Les fonctions définies à l'extérieur des classes ont par contre une portée globale. Cette différence est illustrée par l'exemple suivant

Exemple 6.5 (Portée des fonctions)

```
#include <iostream>

using namespace std;

int foisdeux1(int a)
{
    return 2*a;
}
class maclasse
{
public:
    int v;
    void foisdeux2()
    {
        v*=v;
    }
};
int main()
{
    int x=2,y;
    y=foisdeux1(x);// appel de la fonction foisdeux1()
    maclasse obj;
    obj.v=2;
    obj.foisdeux2();//appel de la fonction membre foisdeux2()
    cout<<y<<'\n';
    cout<<obj.v;
    return 0;
}
```

Listing 6.8 – Exemple illustrant la portée d'une fonction

Le programme affiche

```
4
4
```

6.4 Fonctions amies

Une fonction extérieure à une classe peut être déclarée *amie de cette classe*. Dans ce cas, toutes les données et fonctions membres de cette classe, qu'elles soient publics ou privées, deviennent publics (accessibles) pour cette fonction.

La fonction amie d'une autre classe peut être soit une fonction globale, déclarée à l'extérieur d'une classe, ou fonction membre d'une autre classe.

Comme le montre la syntaxe du listing 6.9, une fonction est déclarée comme amie de la classe `A` à l'intérieur de cette classe. Ceci est fait en précédant le prototype de la fonction par le mot-clé `friend`.

Une même fonction peut être amie de plusieurs classes. La technique de la fonction amie permet le contrôle d'accès à la classe. Seules les fonctions déclarées amies à l'intérieur d'une classe peuvent accéder à ces membres.

```
class A
{
private :
    ...
public :
//Syntaxe pour une fonction amie globale
    friend type fct(arguments) ;
//syntaxe pour une fonction amie membre d'une autre classe B
    friend type B ::fct(arguments) ;
    ...
};
```

Listing 6.9 – Syntaxe de déclaration d'une fonction amie

6.5 Classes amies

Une classe `B` peut être déclarée amie d'une classe `A`. Dans ce cas, tous les membres de la classe `A` deviennent `public` pour la classe `B`. La syntaxe de déclaration d'une classe amie est donné ci-dessous

```
class A
{
private :
    ...
public :
    friend class B ;
} ;
class B
{
};
```

Listing 6.10 – Syntaxe de déclaration d'une classe amie

L'amitié n'est pas commutative, si la classe `B` est amie de la classe `A`, ça ne veut pas dire que `A` est amie de la classe `B`. De même, l'amitié entre classes n'est pas transitive. Si `A` est amie de `B` et `B` amie de C, alors on ne peut pas conclure que `A` est amie de `C`.

Chapitre 7

Héritage

L'*héritage* constitue avec *l'encapsulation* et *le polymorphisme* l'un des fondements de la POO. La technique de l'héritage permet de créer une nouvelle classe à partir d'une classe existante (déjà définie). Cette technique permet de réutiliser, d'étendre et de spécialiser les classes existantes.

Si la classe B hérite de la classe A, elle possèdera toutes les caractéristiques de la classe A en plus des siennes. La classe B possèdera les attributs (données membres) et les comportements (fonctions membres) de la classe dont elle dérive. On définit alors de nouveaux attributs (données et fonctions membres) pour la classe B afin de la modifier et la spécialiser.

L'héritage permet l'exploitation du code existant sans le modifier, contrairement à ce qui pourrai être le cas dans d'autres techniques de programmation.

7.1 Déclaration d'une classe dérivée

7.1.1 Classe dérivée

L'héritage est aussi appelé *dérivation* de classes. On dit alors qu'une classe hérite ou dérive d'une autre classe. La classe dont dérive une classe se nome *la classe de base*, *la superclasse*, *la classe parent*, *la classe ancêtre*, *la classe mère* ou *la classe père*. La classe dérivée se nome la *classe fille* ou la *sous-classe*.

7.1.2 Syntaxe de déclaration d'une classe dérivée

```
1 class A
2 {
3     ...
4 } ;
5 class B : public A
6 {
7     ...
8 };
```

Listing 7.1 – Syntaxe de déclaration d'une classe dérivée

Avec la syntaxe du listing 7.1, la classe `A` est la classe de base (classe parent) et la classe `B` est la classe dérivée (classe fille). La classe `A` doit obligatoirement être déclarée avant la classe `B`.

Grâce à l'héritage, on peut définir une hiérarchie (arborescence) de classes. Il est possible de dériver une nouvelle classe `C` à partir de la classe `B` qui elle même dérive de la classe `A`. A chaque fois qu'un objet de la classe dérivée `B` est créé, C++ crée un objet de la classe de base `A`. Un objet de la classe `B` peut être alors vu comme un objet de la classe `A` avec des éléments supplémentaires, qui sont les membres spécifiques à la classe `B` qui ne sont pas définis dans la classe `A`.

Les données membre de la classe de base sont allouées au moment de la déclaration d'un objet de la classe dérivée, de la même façon que si un objet de la classe de base a été déclaré. Ce processus de création automatique d'un objet de la classe de base si un objet de la classe dérivée est déclaré se propage tout au long de la hiérarchie des classes. Si `B` dérive de `A` et `C` dérive `B`, la déclaration d'un objet de la classe C crée des objets de la classe B et A. L'exemple suivant montre une classe parent `A` dont dérive la classe `B`

Exemple 7.1 (Classe dérivée)

```
#include <iostream>

using namespace std;

class A
{
public :
    int i;
    void afficheI();
};
class B : public A
{
public :
    int j;
    void afficheIJ();
};
void A::afficheI()
{
    cout<<"i ="<<i<<"\n";
}
void B::afficheIJ()
{
    afficheI();
    cout<<"j ="<<j<<"\n";
}
int main()
{
    B b;
    b.i=1;
    b.j=2;
    b.afficheIJ();
    return 0;
}
```

Listing 7.2 – Exemple de déclaration d'une classe dérivée

Le programme affiche

```
i=1
j=2
```

On a déclaré une classe `A` dont tous les membres (données et fonctions membres) sont déclarés sous l'étiquette `public`. La classe `A` possède une donnée membre `i` et une méthode `afficheI()` qui permet d'afficher `i`. On déclare une classe `B` qui dérive de la classe `A` avec un nouvel attribut `j` et une nouvelle méthode `afficheIJ()`. Dans ce cas, comme les membres de la classe `A` sont déclarés `public`, ils sont accessibles par les membres de la classe `B`. Ceci permet à la méthode `afficheIJ()` d'appeler la méthode `afficheI()`. Dans le `main()`, un objet `b` de la classe `B` est créé. Ceci entraîne la création d'un objet de la classe `A` et l'objet `b` possèdera l'attribut `i` de la classe `A`. Comme cet attribut est `public`, il a été affecté directement : `b.i=1`. L'application de la méthode `afficheIJ()` sur l'objet `b` permet de faire appel à la méthode de la classe de base `afficheI()` et d'afficher ainsi `i` et `j`.

7.2 Héritage et protection

La protection des membres dépend de l'étiquette avec laquelle ils ont été définis. Il existe trois étiquettes de protection : `public`, `private` et `protected`.

`private` : les membres sont accessibles par les fonctions membres (publics ou privées) et par les fonctions amies de la classe.

`public` : les membres sont accessibles par les fonctions membres, les fonctions amies et *tout utilisateur de la classe*.

`protected` : les membres sont accessibles par les fonctions membres, par la classe dérivée, mais ils ne sont pas accessibles par les utilisateurs de la classe.

Une classe *dérivée* peut accéder aux données membres publics (`public`) et protégées (`protected`) de la classe de base. Les membres privés sont accessibles seulement par les méthodes de la classe qui les a définis.

Les méthodes de la classe dérivée n'ont pas accès aux membres *privés*. Généralement, les données privées de la classe de base (une classe dont on souhaite dériver d'autres classes) sont déclarées `protected`. L'étiquette `protected` a pour effet de rendre ces données accessibles pour les sous classes et privées pour les autres classes. L'exemple suivant montre l'effet des étiquettes de protection

Exemple 7.2 (Etiquettes de protection)

```
1  #include <iostream>
2  #include <string.h>
3
4  using namespace std;
5
6  class mamifere
7  {
8  protected :
9      int age,poid;
10     void affichage1();
11 public:
12     void init1(int,int);
13 };
14 class chien : public mamifere
15 {
16     char
17     race[10];
```

```
18 public:
19     void init2( char*);
20     void affichage2();
21 };
22 void
23 mamifere::init1(int a,int b)
24 {
25     age=a;
26     poid=b;
27 }
28 void mamifere::affichage1()
29 {
30     cout<<"Age :"<<age<<"\n";
31     cout<<"Poid :"<<poid<<"\n";
32 }
33
34 void chien::init2(char*c )
35 {
36     strcpy(race,c);
37 }
38 void chien::affichage2( )
39 {
40     affichage1();
41     cout<<"Race :"<<race<<"\n";
42 }
43 int main()
44 {
45     chien c;
46     c.init1(2,20);
47     c.init2("bulldog");
48     c.affichage2();
49     return 0;
50 }
```

Listing 7.3 – Exemple d'utilisation des étiquettes de protection dans les classes dérivées

Le programme affiche

```
1 Age : 2
2 Poid : 20
3 Race : bulldog
```

Les variables membres `age`, `poid` et la méthode `affiche1()` sont protégées. Elles peuvent êtres utilisées par les méthodes de la classe `mammifere` ou celle de la classe `chien` mais ne peuvent pas êtres accédées par un utilisateur de la classe.

Lors de l'appel d'une méthode par un objet de la classe dérivée, le compilateur vérifie d'abord si c'est une méthode de la classe dérivée, s'il ne la trouve pas il regarde dans les méthodes publics et protégées de la classe mère et ainsi de suite en remontant la hiérarchie des classes.

7.3 Spécification de l'héritage

Lors de la déclaration d'un classe dérivée, il est nécessaire d'indiquer une *spécification* entre les deux points : et le nom de la classe de base. Les trois choix possibles sont les mêmes que ceux utilisés pour la protection des membres : `public`, `protected` et `private`. Dans la majorité des cas c'est la spécification `public` qui est utilisée. Le choix effectué permet de spécifier les changements dans les droits d'accès à la classe de base. Cependant,

si `B` dérive de `A`, cette spécification n'aura pas d'effet sur l'accessibilité des membres de `A` dans `B` mais sur les classes dérivant de `B`. Cette spécification agit de la façon suivante :

public : Les étiquettes de protection des membres de la classe `A` restent inchangées dans la classe `B`.

protected : Les membres `public` et `protected` de `A` deviennent `protected` dans `B`.

private : Les membres `public` et `protected` de `A` deviennent `private` dans `B`. Les classes dérivées de `B` ne pourront plus accéder aux membres de `A`.

7.4 Appel des constructeurs et destructeurs

La création d'un objet de la classe dérivée entraîne la création d'un objet de la classe de base. Or, comme la création de l'objet se traduit par l'appel au constructeur de la classe, le constructeur de la classe dérivée va appeler le constructeur de la classe de base. Les objets de base sont alors créés avant les objets dérivés.

Pour les destructeurs, c'est le destructeur de la classe dérivée qui est appelé en premier. Cet exemple illustre cet état de fait

Exemple 7.3 (Constructeurs et destructeurs)

```
1  #include <iostream>
2
3  using namespace std;
4
5  class mamifere
6  {
7  public:
8      mamifere();
9      ~mamifere();
10 };
11 class chien : public mamifere
12 {
13 public:
14     chien();
15     ~chien();
16 };
17
18 mamifere::mamifere()
19 {
20     cout<<"Constructeur mamifere "<<this<<"\n";
21 }
22
23 mamifere::~mamifere()
24 {
25     cout<<"Destructeur mamifere "<<this<<"\n";
26 }
27 chien::chien()
28 {
29     cout<<"Constructeur chien "<<this<<"\n";
30 }
31
32 chien::~chien()
33 {
34     cout<<"Destructeur chien "<<this<<"\n";
35 }
36 int main()
37 {
38     cout<<"Creation d'un objet de type mamifere \n";
39     mamifere
```

```
40      *m=new mamifere();;
41      cout<<"Destruction d'un objet de type mamifere \n";
42      delete m;
43      cout<<"Creation d'un objet de type chien \n";
44      chien
45      *c=new chien();
46      cout<<"Destruction d'un objet de type chien \n";
47      delete c;
48      return 0;
49  }
```

Listing 7.4 – Constructeurs et destructeurs dans les classes dérivées

Le programme affiche

```
1  Creation d'un objet de type mamifere
2  Constructeur mamifere 0X32C10
3  Destruction d'un objet de type mamifere
4  Destructeur mamifere 0X32C10
5  Creation d'un objet de type chien
6  Constructeur mamifere 0X32C10
7  Constructeur chien 0X32C10
8  Destruction d'un objet de type chien
9  Destructeur mamifere 0X32C10
10 Destructeur chien 0X32C10
```

7.4.1 Dérivation d'une classe possédant un constructeur avec arguments

Dans ce cas, l'enchainement des constructeurs est le même : C'est le constructeur de base qui est exécuté en premier. Si le constructeur de base possède des paramètres, le constructeur de la classe dérivée doit lui transmettre ces paramètres.

Exemple 7.4 (Constructeurs avec arguments)

```
1  #include <iostream>
2
3  using namespace std;
4
5  class parc
6  {
7  protected:
8      int nombre;
9  public:
10     parc (int n);
11     int get_nombre()
12     {
13         return nombre;
14     }
15 };
16 class parcvoiture
17     : public parc
18 {
19 public :
20     parcvoiture(int n);
21 };
22 parc::parc(int n)
23 {
24     nombre=n;
25 }
26 parcvoiture::parcvoiture(int n)
```

```
27      :parc(n)
28  {
29  }
30
31  int main()
32  {
33      parcvoiture p(20);
34      cout<<"Nombre de places ="<<p.get_nombre();
35      return 0;
36  }
```

Listing 7.5 – Dérivation d'une classe possédant un constructeur avec arguments

7.4.2 Quelques règles sur les constructeurs

1. Si aucun constructeur n'est déclaré explicitement, un constructeur par défaut (sans paramètres et qui ne fait rien) est automatiquement créé par le compilateur.
2. Si un constructeur est explicitement déclaré, le constructeur par défaut n'est pas créé.
3. Si l'emploi du constructeur par défaut de la classe de base est nécessaire, il faut alors le créer explicitement (corps vide) dans la classe de base.
4. La définition d'un constructeur par défaut est nécessaire si :
   (a) Un constructeur par défaut est utilisé pour l'instanciation d'un objet de la classe dérivée, il faut alors créer un constructeur par défaut dans la classe de base.
   (b) On a déjà déclaré explicitement un constructeur et on souhaite déclarer des objets dans le programme sans les initialiser au moment de leur déclaration.

   L'exemple ci-dessous contient une classe dérivée avec des constructeurs avec et sans arguments

Exemple 7.5 (Constructeurs et destructeurs)

```
1   #include <iostream>
2
3   using namespace std;
4
5   class parc
6   {
7   protected:
8       int nombre;
9   public:
10      parc()
11      {
12          nombre=0;
13      };
14      parc (int n);
15      int get_nombre()
16      {
17          return nombre;
18      }
19
20  };
21  class parcvoiture : public parc
22  {
23  public :
24      parcvoiture() {};
```

```
25      parcvoiture(int n);
26  };
27  parc::parc(int n)
28  {
29      nombre=n;
30  }
31  parcvoiture::parcvoiture(int n)
32      :parc(n)
33  {
34  }
35
36  int main()
37  {
38      parcvoiture p;
39      cout<<"Nobre de places ="<<p.get_nombre()<<"\n";
40      parcvoiture m(20);
41      cout<<"Nobre de places ="<<m.get_nombre()<<"\n";
42  return 0;
43  }
```

Listing 7.6 – Constructeurs et destructeurs dans les classes dérivées

Le programme affiche

```
1  Nombre de places = 0
2  Nombre de places = 20
```

Chapitre 8

Les flots, les fichiers

8.1 Les flots

8.1.1 Définitions

Un flot est un canal qui véhicule de l'information. Si le canal *reçoit* de l'information à partir du programme, il s'agit d'un *flot de sortie*. Si le canal *fournit* de l'information au programme, il s'agit d'un *flot d'entrée*.

8.1.2 Transfert de flot

Un flot peut être connecté (transféré) à un périphérique d'entrée (clavier, fichier,...) ou de sortie (écran, imprimante, fichier,...) grâce aux opérateurs `<<` et `>>`. Ces opérateurs servent au transfert de l'information et à son formatage.

Le flot prédéfini `cout` est connecté à la *sortie standard* : La sortie standard correspond par défaut à l'écran.

Le flot prédéfini `cin` est connecté à *l'entrée standard* : L'entrée standard correspond par défaut au clavier. L'entrée et les sorties standards peuvent êtres redirigées vers un fichier. On peut définir d'autres flots et les connecter à un fichier.

8.1.3 Les classe ostream-istream

Un flot est un objet de la classe :

1. `ostream` pour un flot de sortie.
2. `istream` pour un flot d'entrée. L'utilisation des deux classes nécessite l'inclusion du fichier d'entête `iostream.h`.

8.2 Entrées-sorties standards

Le flot `cout` est un flot prédéfini de la classe `ostream`. Il existe deux autres flots prédéfinis :

1. `cerr` : C'est un flot de sortie, sans buffer, connecté à la sortie standard d'erreur.

2. `clog` : C'est un flot de sortie, avec buffer intermédiaire, connecté à la sortie standard d'erreur.

8.2.1 L'opérateur <<

```
ostream & operator << (expression)
```

Listing 8.1 – Syntaxe d'utilisation de l'opérateur <<

Il reçoit deux opérandes :

1. L'objet l'ayant appelé : le flot.
2. Une expression d'un type de base qu'il transmet au flot en la formatant.

Le résultat de cet opérateur est la référence au flot appelant. Ce qui permet de *chaîner* cet opérateur, *i.e.* l'appliquer plusieurs fois de suite.

```
cout << "n= " << n << '\n' ;
```

Listing 8.2 – Exemple d'utilisation de l'opérateur <<

Les types acceptés par l'opérateur << sont les suivants :

1. Tous les types de base : (`signed unsigned`) `int`, (`signed unsigned`) `char`, `long`, `float`, `double`, ...
2. Types `bool` et `short`.
3. Type `char*` : C'est la chaîne qui est a cette adresse qui est transmise au flot. Pour afficher la valeur d'un type `char*` on le convertit explicitement en `void*`.
4. Pointeur sur un autre type que `char` : On obtient la valeur du pointeur (adresse de la donnée pointée).
5. Les tableaux sont acceptés tout en étant convertit au pointeur correspondant. C'est l'adresse du tableau qui est affichée.

8.2.2 Formatage avec l'opérateur <<

8.2.2.1 Choix de la base de numération

Le formatage permet d'exprimer la valeur écrite sur un flot de sortie dans une base donnée : décimale, hexadécimale, octale. booléenne (soit sous forme de `0,1` ou `true , false`). Ceci est réalisé grâce à des manipulateurs : Ce sont des opérateurs à un seul opérande de type `flot`, fournissant en retour le même flot, après l'avoir réécrit dans la base souhaitée.

Exemple 8.1 (Formatage avec de l'opérateur <<)

```
#include <iostream>

using namespace std;

using namespace std ;
int main()
{
    int n = 10 ;
    cout << "en décimal (par defaut :) " << n << "\n" ;
    cout << "en hexadecimal : " << hex << n << "\n" ;
    cout << "en decimal : " << dec<< n << "\n" ;
    cout << "en octal : " << oct << n << "\n" ;
    cout <<  "sans manimulateur : " << n << "\n" ;
    bool b = 1 ;
    cout << "par defaut : " << b << "\n" ;
    cout << "avec noboolalpha : "<<noboolalpha << b << "\n" ;
    cout << "avec boolalpha : " <<boolalpha << b << "\n" ;
    cout << "sans manipulateur : " <<b<<"\n";
    return 0;
}
```

Listing 8.3 – Exemple de formatage avec de l'opérateur **<<**

le programme affiche

```
en décimal (par defaut :) 10
en hexadecimal : a
en decimal : 10
en octal : 12
sans manimulateur : 12
par defaut : 1
avec noboolalpha : 1
avec boolalpha : true
sans manipulateur : true
```

La valeur du flot de sortie reste la même tant qu'on ne la modifie pas. Le manipulateur `boolalpha` demande d'afficher les valeurs booléennes sous forme alphabétique (true et false). Le manipulateur `noboolapha` demande d'afficher les valeurs booléennes sous forme numérique 0 ou 1.

8.2.2.2 Choix du gabarit de l'information écrite

Le gabarit est la forme (largeur) du texte écrit. On utilise l'opérateur paramétrique (admettant un paramètre) `setw` (pour set width en anglais : fixer largeur). Cet opérateur définit le gabarit de la prochaine information à écrire. Par défaut, sans l'utilisation d'opérateurs, l'information est écrite en utilisant le minimum d'espace (emplacements).

Exemple 8.2 (Formatage avec de l'opérateur <<)

```
#include <iostream>
#include <iomanip>

using namespace std ;

int main()
{
    char* a = "ABC" ;
    int i ;
```

```
10      for (i=0 ; i<12 ; i++)
11          cout << setw(2)<< i << " : "<< setw(i) << a << "\n" ;
12      return 0;
13  }
```

Listing 8.4 – Exemple de choix du gabarit

le programme affiche

```
 0 : ABC
 1 : ABC
 2 : ABC
 3 : ABC
 4 :  ABC
 5 :   ABC
 6 :    ABC
 7 :     ABC
 8 :      ABC
 9 :       ABC
10 :        ABC
11 :         ABC
```

Si la valeur fournie à `setw` est plus courte que l'information à afficher, celle-ci ne sera pas tronquée.

8.2.2.3 Choix de la précision l'information écrite

Le nombre de chiffres significatifs est par défaut de 6. L'opérateur `setprecision` permet le choix du nombre de chiffres significatifs. L'effet du manipulateur reste valable jusqu'à sa prochaine modification.

Exemple 8.3 (Formatage avec de l'opérateur <<)

```
1   #include <iomanip>
2   #include <iostream>
3
4   using namespace std ;
5
6   int main()
7   {
8       double a = 123456 ;
9       cout << "par defaut : " << a << "\n" ;
10      for
11      (int i=1 ; i<8 ; i++)
12      cout << "Nombre de chiffres significatifs : "<< i << setprecision (i) << " : " << a << "\n" ;
13  return 0;
14  }
```

Listing 8.5 – Exemple sur le choix de la précision de l'information

le programme affiche

```
par defaut : 123456
Nombre de chiffres significatifs : 1 : 1e+005
Nombre de chiffres significatifs : 2 : 1.2e+005
Nombre de chiffres significatifs : 3 : 1.23e+005
Nombre de chiffres significatifs : 4 : 1.235e+005
Nombre de chiffres significatifs : 5 : 1.2346e+005
Nombre de chiffres significatifs : 6 : 123456
Nombre de chiffres significatifs : 7 : 123456
```

8.2.2.4 Choix de la notation flottante et exponentielle

Le choix se fait avec les manipulateurs `fixed` pour la notation flottante et `scientific` pour la notation exponentielle à un chiffre avant le point de mantisse. On peut en même temps que le choix du type de la notation faire un choix de la précision. Dans ce cas, la précision correspondra au nombre de chiffres après la virgule.

Exemple 8.4 (Formatage avec de l'opérateur <<)

```
#include <iostream>
#include <iomanip>

using namespace std ;
int main()
{
    float x = 2e5/3 ;
    double pi = 3.141926536 ;
    cout << fixed <<"choix notation flottante \n" ;
    cout << "precision " << 4 <<setprecision (4) << " :" << x << ": : " << pi << ":\n" ;
    cout <<scientific << "choix notation exponentielle \n" ;
    cout << "precision " << 4 << setprecision (4) << " :" << x << ": :" << pi << ":\n" ;
return 0;
}
```

Listing 8.6 – Exemple sur le choix de la notation flottante et exponentielle

le programme affiche

```
choix notation flottante
precision 4 :66666.6641: : 3.1419:
choix notation exponentielle
precision 4 :6.6667e+004: :3.1419e+000:
```

8.2.3 L'opérateur >>

Sa Syntaxe d'utilisation est la suivante

```
istream & operator >> (type_de_base & )
```

L'opérateur reçoit deux opérandes :

1. L'objet flot l'ayant appelé.
2. Une valeur à droite (`lvalue`) de type quelconque.

8.2.4 La fonction get

Elle permet d'extraire une variable du flot d'entrée et de le ranger dans la variable qu'on lui fournit en argument. Elle fournit en retour une référence au flot concerné. Cette fonction peut lire n'importe quel caractère.

8.3 Les fichiers

8.3.1 Connexion d'un flot de sortie à un fichier

Pour connecter un flot de sortie à un fichier, on crée un objet de la classe `ofstream`, classe dérivant de `ostream`. La création se fait avec le constructeur de la classe `ofstream`. On peut appeler le constructeur à deux arguments :

1. Le nom du fichier concerné (chaîne de caractères).
2. Le mode d'accès, défini par une constante entière. Ce sont des constantes symboliques prédéfinies de la classe `ios`.

Le fichier est crée et ouvert en écriture. La syntaxe de création d'un flot de sortie associé à un fichier est la suivante

```
1 ofstream nom_flot(nom_fichier [,mode_ouverture] ) ;
```

Le paramètre `nom_fichier` est une chaîne de caractères qui désigne le nom du fichier. Le paramètre `mode_ouverture` est optionnel. Il désigne le mode d'ouverture du fichier. Il s'agit d'un entier qui peut prendre des valeurs qui sont contenues dans des constantes symboliques dans le fichier d'entête `ios.h`.

Les différents modes d'ouverture qui sont regroupés dans le tableau 8.1

Mode	Signification
`ios ::in`	Ouverture en lecture (`ifsream`)
`ios ::out`	Ouverture en écriture (`ofstream`)
`ios ::app`	Ajout à la fin du fichier
`ios ::ate`	Se place à la fin du fichier. Possibilité d'écrire partout dans le fichier
`ios ::trunc`	Le fichier existant est tronqué (contenu perdu)
`ios ::nocreate`	Si le fichier n'existe pas, l'ouverture échoue
`ios ::noreplace`	Si le fichier existe, l'ouverture échoue

Table 8.1 – Modes d'ouverture d'un fichier

Les constantes peuvent se combiner grâce à l'opérateur |.

8.3.2 Connexion d'un flot d'entrée à un fichier

Pour connecter un flot de sortie à un fichier, on crée un objet de la classe `ifstream`. De même, la création se fait avec le constructeur de la classe `ifstream` selon la même syntaxe que la création d'un flot de sortie :

```
1 ifstream nom_flot(nom_fichier [,mode_ouverture] ) ;
```

La connexion d'un flot d'entrée à un fichier entraîne l'ouverture du fichier. Le paramètre optionnel `mode_ouverture` est optionnel. Par défaut, les fichiers sont ouvert en lecture.

8.3.3 Ouverture et fermeture d'un fichier

Un fichier doit être ouvert avant d'y effectuer une opération de lecture ou d'écriture. Aussi, un fichier ouvert doit toujours être fermé s'il n'est plus utilisé. L'ouverture effectue la liaison entre le programme et le fichier et la fermeture la termine.

En C++, le traitement des fichiers peut s'envisager de deux façons :

1. De façon conventionnelle en utilisant la bibliothèque standard du C (incluse dans C++) sans faire appel aux flot.
2. En utilisant les flot grâce aux classes `ifstream`, `ofstream` et `fstream`.

En utilisant les flots, l'ouverture et la fermeture des fichiers peut se faire *implicitement* ou *explicitement*. Dans le cas conventionnel (sans flot), elle doit se faire explicitement.

8.3.3.1 Ouverture et fermeture explicite d'un fichier

Les classes `ifstream`, `ofstream` et `fstream` contiennent des méthodes nommées `open` et `close` qui permettent respectivement d'ouvrir et de fermer un fichier de façon explicite.

La fonction `close` n'a pas de paramètres. Sa syntaxe est la suivante

```
1 nom_flot.close() ;
```

Par contre, la fonction `open` possède deux paramètres : Un pointeur `char` qui référence le nom du fichier et un nombre entier qui peut être fournit par les mêmes constantes symboliques présentées au tableau 8.1. Ces constantes spécifient le mode d'ouverture souhaité. La syntaxe de la fonction `open` est donnée par

```
1 nom_flot.open(nom_fichier [,mode_ouverture] ) ;
```

Les classes `ifstream`, `ofstream` et `fstream` possèdent des constructeurs par défaut (sans paramètres) . Ceci permet de créer des flots sans les associer à un fichier. Ces flots pourront être utilisés par les fonctions `open` et `close`. Il est préférable d'ouvrir et de fermer les fichiers de façon implicite. Toutefois, les opérations explicites peuvent êtres utiles dans les cas suivants :

1. Lorsqu'un même fichier doit être ouvert en lecture et en écriture.
2. Lorsque le même flot doit être associé à plusieurs fichiers différents.

8.3.3.2 Ouverture et fermeture implicite d'un fichier

Comme il est déjà mentionné dans les parties 8.3.1 et 8.3.2, la création d'un flot des classes `ifstream`, `ofstream` et `fstream` entraîne l'ouverture implicite du fichier transmis en argument au constructeur et associe le flot créé à ce fichier.

Pour la fermeture explicite, elle s'effectue dès que le flot associé au fichier est détruit. Rappelons que le destructeur d'une classe est exécuté systématiquement lorsque le programme sort de la portée de l'objet. Par conséquent, les fichiers sont automatiquement fermés une fois le flot qui leur est associé est détruit.

De façon générale, tous les fichiers ouverts sont fermés à la fin d'un programme.

8.3.4 Lecture et écriture sur un fichier

La lecture et l'écriture sur des fichiers peut se faire de la même façon que les opérations d'entrée-sortie sur clavier-écran. Au lieu des flots prédéfinis `cin` et `cout`, on utilise des flots qui sont des instances des classes `ifstream`, `ofstream` et `fstream`.

Les entrées-sorties formatées se font de la même façon que celle mentionnée dans la partie 8.2.2 en utilisant les opérateurs de transfert << et >>

Grâce aux fonctions `read` et `write`, il est possible de traiter globalement un certain nombre d'octets. Les données sont dans ce cas transférés par bloc. La syntaxe de ces deux fonctions est la suivante

```
istream& istream::read(char *tampon,int max);
ostream& ostream::write(const char *tampon,int max);
```

Le paramètre `tampon` représente la chaîne de caractères à lire ou à écrire. Le paramètre max indique le nombre maximal d'octets à lire.

L'exemple suivant récapitule les manipulations présentées dans cette section avec les fichiers

Exemple 8.5 (Manipulations d'un fichier)

```
#include <cstdlib> // pour exit
#include <iostream>
#include <fstream>
#include <iomanip>
#include <conio.h>

using namespace std;

class eleve
{
    int matr;
    char nom[20],prenom[20];
public:
    void creer()
    {
        cout<<"Matricule :  ";
        cin>>matr;
        cout<<"Nom       :  ";
        cin>>nom;
        cout<<"Preno     :  ";
        cin>>prenom;
    }
    void affiche()
    {
        cout<<"\n Matricule  :"<<matr<<"\n";
        cout<<"Nom         :"<<nom<<"\n";
        cout<<"Prenom      :"<<prenom<<"\n";
    }
};
int main()
{
    char nom_fichier[20],rep;
    eleve v;
    cout<<"Nom fihier ? ";
    cin>>nom_fichier;
    ofstream flot_ecriture(nom_fichier);//creation fichier

    do
    {
```

```
40
41          v.creer();//on cée un objet
42          flot_ecriture.write((char*)&v,sizeof v );//on écrit l'objet dans le fichier
43          cout<<"Autre éléve à saisir ? o/n";
44          rep=getche();
45      }
46      while(rep=='o');
47      flot_ecriture.close();//on ferme le fichier
48  //consultation et affichage
49      ifstream flot_lecture (nom_fichier);//on ouvre le fichier
50      flot_lecture.read((char*)&v,sizeof v);//on lit un objet
51      while (flot_lecture)//pas fin de fichier
52      {
53          v.affiche();
54          flot_lecture.read((char*)&v,sizeof v);// on lit l'objet suivant
55      }
56      flot_lecture.close();
57      return 0;
58  }
```

Listing 8.7 – Lecture/Écriture sur un fichier

Chapitre 9

Polymorphisme, surcharge de fonctions

9.1 Le polymorphisme

Linguistiquement, polymorphe veut dire qui peut se présenter sous différentes formes. Le polymorphisme est un des fondements de la programmation orientée objet.

Bien que la technique de polymorphisme puisse s'utiliser avec des fonctions n'appartenant pas à des classes, elle constitue un complément fondamental à la technique d'héritage. Elle permet de redéfinir des méthodes/données héritées de la classe mère.

de façon générale, dans une classe dérivée, on distingue trois sortes de membres (méthodes/données) :

1. Les méthodes (données) qui sont propres à la nouvelle classe : On parle alors d'*extension*
   (a) Ces méthodes (données) peuvent porter un nom qui n'est pas utilisée dans la classe mère.
   (b) Ces méthodes peuvent porter un nom qui est utilisé dans la classe mère avec des paramètres en nombre et/ou en type différent : Il y a alors *surcharge* de la méthode.
2. Les méthodes/données qui sont issues de la classe mère : Il s'agit alors d'*héritage*. Lors de l'utilisation d'une méthode héritée sur un objet de la classé dérivée, ce dernier est convertit implicitement en un objet de la classe mère. Par défaut, toutes les méthodes/données de la classe mère sont héritées
3. Les méthodes/données qui redéfinissent les méthodes existantes dans la classe mère : Il s'agit de *masquage*. Ces méthodes ont le même nom et le même prototype (déclaration) que celle qu'elles redéfinissent. On dit alors que la méthode de la classe dérivée spécialise celle de la classe mère.

9.2 La surcharge des fonctions

On parle de *surcharge de fonctions*, lorsque celles-ci ont la même portée. Les fonctions membres d'une classe ont une portée limitée à cette classe et les fonctions déclarées à

l'extérieur de toutes les classes ont une portée globale. Si deux fonctions ont le même nom mais des domaines de portée différents (l'une est membre d'une classe, l'autre est membre d'une autre classe ou globale), il ne s'agit pas de surcharge de fonctions. En cas de surcharge, les méthodes ont le même nom avec des paramètres en nombre et/ou en type différents.

9.2.1 Surcharge de fonctions membre de la même classe

Exemple 9.1 (Surcharge de fonctions membres)

```
#include <iostream>

using namespace std;

class voiture {
    int matricule;
    float puissance;
public:
    void creer (int n)
    {
        matricule=n;
    }
    void creer(float m)
    {
        puissance=m;
    }
    void creer(int n,float m)
    {
        matricule=n;
        puissance=m;
    }
    void affiche()
    {
        cout<<"Matricule : "<<matricule<<"\n";
        cout<<"Puissance : "<<puissance<<"\n";
    }
};

int main() {
    int n;
    float m;
    voiture v;
    cout<<"Matricule et puissance de la voiture ";
    cin>>n>>m;
    v.creer(n);
    v.creer(m);
    v.affiche();
    voiture v2;
    cout<<"Matricule et puissance de la voiture ";
    cin>>n>>m;
    v2.creer(n,m);
    v2.affiche();
    return 0;
}
```

Listing 9.1 – Surcharge de fonctions membre de la même classe

9.2.2 Surcharge de fonctions membre d'une classe dérivée

Exemple 9.2 (Surcharge de fonctions membres)

```
#include <iostream>

using namespace std;

class vehicule
{
protected:
    int matricule;
    float puissance;
public:
    void creer(int n,float m)
    {
        matricule=n;
        puissance=m;
    }
    void affiche()
    {
        cout<<"Matricule : "<<matricule<<"\n";
        cout<<"Puissance : "<<puissance<<"\n";
    }
};
class voiture : public vehicule
{
    int nbporte;
public:
    void creer(int n,float m,int p)
    {
        vehicule::creer(n,m);
        nbporte=p;
    }
    void affiche()
    {
        vehicule::affiche();
        cout<<"nombre portes : "<<nbporte<<"\n";
    }
};

int main()
{
    int n;
    float m;
    int p;
    voiture v;
    cout<<"Matricule et puissance de la voiture et nombre de portes ";
    cin>>n>>m>>p;
    v.creer(n,m,p);
    v.affiche();
    return 0;
}
```

Listing 9.2 – Surcharge de fonctions membre d'une classe dérivée

Remarque 9.1
Si dans le même domaine de portée, une fonction a le même nom, la même liste de paramètres et des types différents, ceci va générer une erreur de compilation, car le compilateur de saura pas à quelle fonction il est fait appel.

9.3 Fonctions virtuelles

Dans l'exemple du listing 9.2, la fonction `affiche` de la classe dérivée `voiture` masque la fonction `affiche` de la classe mère `vehicule`. Toutefois, il est possible d'appeler la fonction `affiche` de la classe mère en spécifiant son nom complet `vehicule::affiche`.

Pour la surcharge des constructeurs, voir le chapitre dédié à l'héritage.

Si un pointeur vers une classe de base est déclaré, il est possible d'instancier des objets de toutes la hiérarchie des classe dérivées de cette classe de base. Cependant, si une méthode de la classe de base est masquée dans les classes dérivées (la fonction garde exactement la même signature), l'appel de cette dernière sur un des objets de la classe dérivée invoquera toujours la méthode du même prototype de la classe de bas. Grâce aux *fonction virtuelles*, le polymorphisme permet deux objet de classes différentes de réagir différemment au même appel (fonction de même nom et signature).

9.3.1 Définition d'une fonction virtuelle

Une*fonction virtuelle* appelée également *fonction polymorphe* est une fonction membre qui est appelée en fonction du type d'objet et non du type de pointeur. La déclaration d'une fonction virtuelle dans la classe de base se fait en précédant le prototype de la fonction du mot-clé `virtual`. Ceci permet de définir des versions différentes de la fonction pour chaque classe dérivée et d'appeler la fonction correspondant au type d'objet.

Dans le cas d'une définition déportée de la fonction virtuelle, il n'est pas nécessaire de rappeler le mot-clé `virtual` dans le pavant le prototype du corps de la fonction. Il est également suffisant d'utiliser le mot-clé `virtual` devant la méthode de la classe de base sans le rappeler pour celles des classe dérivées. Toutefois, il est conseillé de rappeler ce mot-clé pour toutes les classes dérivées pour ne pas avoir à remonter toute la hiérarchie des classes à la recherche de fonctions virtuelles.

A l'appel d'une fonction virtuelle, le programme cherche la fonction dans la classe de l'objet. S'il ne la trouve pas, il remonte la hiérarchie des classes. le choix de la fonction virtuelle à exécuter se fait au moment de l'execution et non pas au moment de la compilation comme pour les autres fonctions. Cette technique s'appelle la *ligature dynamique.*

L'exemple suivant reprend les la méthode `affiche` des classes `vehicule` et `voiture` définies précédemment. La différence avec l'exemple précédent est qu'on utilise un pointeur `pv` vers la classe de base `vehicule` qui es d'abord référencé vers avec une instance de la classe `vehicule` puis une instance de la classe `voiture`. La méthode `affiche()` est appelée en utilisan le même pointeur.

Exemple 9.3 (Fonctions virtuelles)

```
1  #include <iostream>
2
3  using namespace std;
4
5  class vehicule
6  {
7  protected:
8      int matricule;
9      float puissance;
10 public:
11     vehicule(int n,float m)
```

```
12      {
13          matricule=n;
14          puissance=m;
15      }
16      virtual void affiche()
17      {
18          cout<<"Matricule : "<<matricule<<"\n";
19          cout<<"Puissance : "<<puissance<<"\n";
20      }
21  };
22  class voiture : public vehicule
23  {
24      int nbporte;
25  public:
26      voiture(int n,float m,int p)
27          :vehicule(n,m)
28      {
29
30          nbporte=p;
31      }
32      virtual void affiche()
33      {
34          vehicule::affiche();
35          cout<<"nombre portes : "<<nbporte<<"\n";
36      }
37  };
38
39  int main()
40  {
41      int n;
42      float m;
43      int p;
44      vehicule *pv;
45      cout<<"\n Matricule et puissance de la voiture et nombre de portes ";
46      cin>>n>>m>>p;
47      pv=new voiture(n,m,p);
48      pv->affiche();
49      cout<<"\n Matricule et puissance du véhicule ";
50      cin>>n>>m;
51      pv=new vehicule(n,m);
52      pv->affiche();
53      return 0;
54  }
```

Listing 9.3 – Polymorphisme et redéfinition des fonctions virtuelles

A l'exécution du programme, le texte suivant s'affiche

```
1  Matricule et puissance de la voiture et nombre de portes 251140000 10 5
2  Matricule : 251140000 Puissance : 10 nombre portes : 5
3
4  Matricule et puissance du véhicule 251140001 8
5  Matricule : 251140001 Puissance : 8
```

On voit bien qu'avec la même syntaxe d'appel `pv->affiche()`, le programme appelle en fonction du type d'objet des méthodes différentes. Dans le premier cas, il appelle la méthode de la classe dérivée `voiture` et dans le second la méthode de la classe de base `vehicule`.

9.3.2 Rôle d'une fonction virtuelle

Une fonction virtuelle peut être définie dans une classe dérivée pour plusieurs raisons :

1. Pour modifier le traitement standard de la classe de base.
2. Pour compléter le traitement standard de la classe de base. La fonction de la classe de base est alors appelée par celle de la classe dérivée.
3. Pour annuler le traitement standard de la classe de base. Elle est dans ce cas définie avec u corps vide.

9.3.3 Fonctions virtuelles pures et classes abstraites

La déclaration d'un fonction virtuelle pure se fait en suivant son prototype de l'expression `=0`. A l'opposé des fonctions virtuelles ordinaires, toute fonction virtuelle pure doit obligatoirement être redéfinie dans les classes dérivées.

Une *classe abstraite* est une classe qui contient au moins une fonction virtuelle pure. une classe abstraite ne peut pas être instanciée : Il n'est pas possible de créer directement des objets de cette classe.

Les classes abstraites servent généralement à factoriser les membres communs aux classes dérivées et à imposer un comportement (une interface) que toutes les classes dérivées doivent implémenter de la même façon. Ceci permet de rendre les applications plus robuste : Même si un changement des traitement des classes dérivée est effectué, aucun changement ne sera nécessaire dans les programmes utilisant ces classes.

9.4 Surcharge des opérateurs

La surcharge des opérateurs du C++ permet de les utiliser comme des opérateurs normaux pour des types nouveaux, comme les classe définies par l'utilisateur par exemple.

Il est possible de définir des fonctions membres pour les nouvelles classes qui jouent le rôle des opérateurs, mais la surcharge des opérateurs permet un usage étendu et intuitif des opérateurs du C++. Ainsi, si x et y sont deux objets d'une nouvelle classe, la surcharge de l'opérateur arithmétique d'addition `+` permet d'écrire `s=x+y` au lieu de `s= somme(x,y)` où `somme` est une fonction membre qui définit la somme de deux instance de la classe.

9.4.1 Surcharge des opérateurs arithmétiques

La surcharge des opérateurs arithmétique `+`,`-`,'`*`,`/` peut se faire à l'aide d'une fonction amie pour accéder aux membres privés de la classe. Ceci est illustré dans le listing 9.4

Exemple 9.4 (Surcharge d'opérateurs)

```
1  #include <iostream>
2
3  using namespace std;
4
5  class surface
6  {
7      float L;
8      float l;
9  public:
10     surface(float n, float m)
11     {
12         L=n;
```

```
13          l=m;
14      }
15      friend surface operator +(surface&,surface&);
16      void afficher()
17      {
18          cout<<"Longueur L = "<<L<<'\n';
19          cout<<"Largeur l = "<<l<<'\n';
20      }
21  };
22  surface operator +(surface& m,surface& n)
23  {
24      surface s(m.L+n.L,m.l+n.l);
25      return s;
26  }
27  int main()
28  {
29      surface s1(1,2);
30      surface s2(2,4);
31      surface s3=s1+s2;
32      s1.afficher();
33      s2.afficher();
34      s3.afficher();
35      return 0;
36  }
```

Listing 9.4 – Surcharge des opérateurs arithmétiques

9.4.2 Surcharge d'opérateurs unaires

La surcharge d'opérateurs unaires peut se faire par :
- Des fonctions globales à un argument.
- Des fonctions membre sans argument.

9.4.3 Surcharge de l'opérateur préfixé

L'opérateur préfixé s'applique avant l'opérande.

Syntaxe de surcharge de l'opérateur préfixé

```
1  Type operator  op()
```

op : opérateur à surcharger.

operator : mot clé.

Exemple 9.5 (Surcharge d'opérateurs)

```
1  #include <iostream>
2
3  using namespace std;
4
5  class indice
6  {
7  private :
8      int n;
9  public:
10     void set_n(int m)
11     {
```

```
12          n=m;
13      }
14      int get_n()
15      {
16          return n;
17      }
18      indice operator ++();
19  };
20  indice indice::operator++()
21  {
22      ++n;
23      return *this;
24  }
25  int main()
26  {
27      indice a;
28      a.set_n(5);
29      cout<<a.get_n()<<'\n';
30      ++a;
31      cout<<a.get_n()<<'\n';
32      return 0;
33  }
```

Listing 9.5 – Surcharge de l'opérateur préfixé

9.4.4 Surcharge de l'opérateur suffixé

La surcharge de l'opérateur suffixé se fait de la même manière que l'opérateur préfixé, sauf qu'un entier est passé comme paramètre à l'opérateur suffixé.

Syntaxe de surcharge de l'opérateur suffixé

```
1  Type operator  op(int)
```

op : opérateur à surcharger.

operator : mot clé.

Exemple 9.6 (Surcharge d'opérateurs)

```
1   #include <iostream>
2
3   using namespace std;
4
5   class indice
6   {
7   private :
8       int n;
9   public:
10      void set_n(int m)
11      {
12          n=m;
13      }
14      int get_n()
15      {
16          return n;
17      }
18      indice operator ++();
19      indice operator ++(int );
20  };
21  indice indice::operator++()
```

```
{
    ++n;
    return *this;
}
indice indice::operator++(int x)
{
    n++;
    return *this;
}
int main()
{
    indice a;
    a.set_n(5);
    cout<<a.get_n()<<'\n';
    ++a;
    cout<<a.get_n()<<'\n';
    a++;
    cout<<a.get_n()<<'\n';
    return 0;
}
```

Listing 9.6 – Surcharge de l'opérateur suffixé

9.4.5 Surcharge des opérateurs d'entrée-sortie

Il est possible de surcharger les opérateurs de flux << et >> pour personnaliser les entrées-sorties pour une classe donnée.Ceci se fait en utilisant les classe `istream` et `ostream` en incluant le fichier d'entête `iostream.h`

Exemple 9.7 (Surcharge d'opérateurs)

```
#include <iostream>

using namespace std;

class surface
{
    float L;
    float l;
public:
    surface(float n, float m)
    {
        L=n;
        l=m;
    }
    friend ostream& operator <<(ostream&, const surface&);
    void afficher()
    {
        cout<<"Longueur L = "<<L<<'\n';
        cout<<"Largeur l = "<<l<<'\n';
    }
};
ostream& operator <<(ostream& ostr, const
                     surface& r)
{
    return ostr<<(r.L)*(r.l)<<" m2";
}
int main()
{
    surface s1(1,2);
    surface s2(2,4);
    cout<<"surface de s1="<<s1<<'\n';
    cout<<"surface de s2="<<s2<<'\n';
    return 0;
```

```
34 }
```

Listing 9.7 – Surcharge des opérateurs d'entrée-sortie

Deuxième partie

Travaux pratiques

TP n° 1

Éléments de Base

1.1 Objectifs du TP

- Savoir utiliser l'environnement de programmation.
- Effectuer des entrée-sorties élémentaires.
- Savoir utiliser les opérateurs de base, déclaration et manipulation de variables.

1.2 Partie 1 : Entrées et sorties avec les flots `cin` et `cout`

1. Écrire un programme C++ qui déclare deux entiers `a` et `b`, récupère leurs valeurs à partir du clavier, puis affiche leur somme.
2. Écrire un programme C++ qui déclare deux entiers `a` et `b`, récupère leurs valeurs à partir du clavier, puis affiche leur produit.
3. Écrire un programme C++ qui affiche la lettre W avec des `*` comme dans la grille suivante :

```
1  *                   *
2   *                 *
3    *               *
4     *      *      *
5      *    * *    *
6       *  *   *  *
7        *       *
```

1.3 Partie 2 : Manipulation des opérateurs

Soit `a` une variable entière initialisée à `10`.

1. Écrire un programme C++ qui rajoute `1` à `a` de quatres façons différentes. Afficher les résultats pour les vérifier.
2. Sans utiliser l'opérateur d'incrémentation, écrire un programme C++ qui a le même effet que le code suivant :

```
int a,b=1;
a=5 + b++;
```

3. Sans utiliser l'opérateur d'incrémentation, écrire un programme C++ qui a le même effet que le code suivant

```
int a,b=1;
a=5 + ++b;
```

4. Soit `x,y` et `z` trois variables entières initialisées à `1`. Écrire une seule ligne de code en C++ qui rajoute à `z` la somme de `x` et de `y` puis incrémente `y`.
5. Écrire un programme C++ qui affiche le code ASCII d'une lettre de l'alphabet latin.

TP n° 2

Structures de Contrôle

2.1 Objectifs du TP

- Utiliser les structures de contrôle,
- Choisir la meilleure structure pour un problème donné.

2.2 Partie 1 : boucles `while`, `do-while`

1. En utilisant une boucle `while`, écrire un programme C++ qui affiche l'alphabet latin en minuscules.
2. En utilisant une boucle `do-while`, écrire un programme C++ qui affiche le code ASCII des caractères de l'alphabet latin en majuscules.

2.3 Partie 2 : Calcul du factoriel

1. En utilisant une boucle `for`, écrire un programme C++ qui calcule le factoriel $n!$ d'un entier n.

2.4 Partie 3 : Algorithme d'Euclide

L'algorithme d'Euclide permet de calculer le PGCD de deux entiers a et b (on suppose que $a > b$). Il procède en remplaçant la paire (a, b) par la paire (b, r) où r est le reste de la division euclidienne de a par b. L'algorithme s'arrête quand le reste est nul. Le PGCD est alors le dernier reste r non nul.

1. Écrire un programme C++ qui calcule le PGCD de deux entiers positifs quelconques a et b.

TP n° 3

Les Tableaux

3.1 Objectifs du TP

- Manipuler des tableaux unidimensionnels.
- Lire et afficher des tableaux.

3.2 Partie 1 : Affichage inverse d'un tableau

1. Écrire un programme C++ qui :
   (a) Lit un tableau unidimensionnel de réels `tab1` de 10 éléments.
   (b) Range les éléments de `tab1` en ordre inverse dans un autre tableau `tab2` (du dernier élément au premier).
   (c) Affiche les deux tableaux.

3.3 Partie 2 : Lecture et affichage d'un tableau-Somme de deux tableaux

1. Écrire un programme C++ qui :
   (a) Lit deux tableaux bidimensionnels d'entiers `A` et `B` de `3x3` éléments.
   (b) Calcule et affiche leur somme `S`.
   (c) Définit un tableau `MAX` qui contient pour l'élément `C[i][j]` le plus grand élément entre `A[i][j]` et `B[i][j]`.

TP n° 4

Les pointeurs, les structures

4.1 Objectifs du TP

- Manipuler des pointeurs.
- Manipuler des structures.
- Faire le lien entre tableaux et pointeurs.

4.2 Partie 1 : Manipulation d'un tableau par un pointeur

4.2.1 Exercice 1

Écrire un programme C++ qui :

1. Déclare deux variables entières `a` et `b` et les initialise respectivement à `1` et `2`.
2. Crée deux pointeurs `pa` et `pb` respectivement vers les variables `a` et `b`.
3. Affiche les valeurs de `a` et `b` et les adresses de `a` et `b`.

4.2.2 Exercice 2

Écrire un programme C++ qui déclare un tableau dynamique de deux dimensions en procédant comme suit :

1. Déclarer un pointeur vers un pointeur vers un entier `pp`.
2. Lire le nombre de lignes `n` et le nombre de colonnes `m` du tableau dynamique.
3. Réalise une allocation dynamique avec `new` d'abord pour le nombre de lignes puis pour le nombre de colonnes.
4. Lit les valeurs des éléments du tableau dynamique à partir du clavier.
5. Affiche le tableau dynamique.

4.3 Partie 2 : Manipulation d'une structure

Écrire un programme C++ qui :

1. Déclare une structure `date` qui comporte trois champs de type entier : `jour`, `mois`, `annee`.
2. Déclare une structure `etudiant` comportant les champs suivants
   - `char nom[30]`
   - `char prenom[20]`
   - `date date_naissance`
3. Crée deux variables de cette structure `personne1` et `personne2`.
4. Initialise la variable `personne1`.
5. Initialise la variable `personne2` avec la variable `personne1`.
6. Modifie les champs de `personne2`.
7. Affiche les champs des deux variables.

TP n° 5

Les fonctions

5.1 Objectifs du TP

- Déclarer et définir des fonctions.
- Utiliser différentes techniques de passage d'arguments.
- Manipuler des pointeurs sur des structures.

5.2 Partie 1 : Déclaration d'une structure

Écrire un programme en C++ qui :

1. Déclare une structure `etudiant`

```
struct etudiant
{
char * nom, * prenom;
int  rang, annee_naissance, moyenne;
};
```

2. Déclare une variable de type `etudiant`.
3. Initialise la variable à travers le clavier et l'affiche.

5.3 Partie 2 : Manipulation par pointeur

Reprendre la structure `etudiant` de l'exercice précédent. Écrire un programme en C++ qui :

1. Déclare un pointeur sur une variable de type `etudiant`.
2. Initialise la variable à travers le clavier et l'affiche en utilisant le pointeur.

5.4 Partie 3 : Utilisation des fonctions

Utiliser deux fonctions `init_etud` et `affiche_etud` pour initialiser et afficher une variable de type `etudiant` à travers un passage par valeur puis par adresse. Mettre la

définition de la structure et les fonctions dans un fichier d'entête séparé et l'inclure dans l'entête.

TP n° 6

Les classes d'objets

6.1 Objectifs du TP

- Mettre en pratique les notions de classe et d'objet.
- Déclarer et initialiser des objets.
- Manipuler des variables membres et fonctions membres.

6.2 Exercice : Classe `etudiant`

Écrire un programme C++ qui :

1. Déclare une classe `etudiant` comportant les variables membres suivantes :
   - `char nom[15], prenom[15];`
   - `int rang, annee_naissance, moyenne;`

   comme variables **privées**.
2. Déclare des fonctions membres pour accéder et modifier toutes les variables membres.
3. Déclare une fonction `init_etud` qui permet d'initialiser les objets au moment de leur création par des entrées du clavier.
4. Déclare une fonction `affich_etud` qui permet d'afficher les attributs d'un objet.
5. Crée un tableau d'objets `etudiant` de dimension 10.
6. Remplit ce tableau au clavier, puis affiche la liste des étudiants et de leurs attributs.

TP n° 7

Classes et fonctions spéciales

7.1 Objectifs du TP

- Mise en pratique des données et fonctions membres statiques.
- Mise en pratique de la notion de fonction amie.
- Mise en pratique de la notion de classe amie.

7.2 Partie 1 : Classe modélisant un point

On souhaite écrire une classe qui modélise des points du plan. Écrire un programme en C++ qui :

1. Déclare une classe `point` qui comporte :
   (a) Trois variables membres de type `float x,y` de protection `private`.
   (b) Une variable statique `nb_pt` qui contient le nombre de points à un instant donné.
   (c) Un constructeur par défaut qui initialise le point à l'origine, incrémente `nb_pt` et affiche *nouveau point créé.*
   (d) Un constructeur qui admet les coordonnées du point comme arguments, incrémente et affiche *nouveau point créé.*
   (e) Une fonction membre `affich_coor()` qui affiche les coordonnées d'un point.
   (f) Une fonction membre statique `affich_nbpt()` qui affiche le nombre de points.
   (g) Une méthode `symetrique` qui renvoie le symétrique d'un point par rapport à l'origine sans changer ce dernier.
   (h) Une méthode `symetrie` qui change les coordonnées d'un point pour devenir celle de son symétrique par rapport à l'origine.
   (i) Une méthode `distance` qui renvoie la distance entre le point et l'origine.
2. Définir une fonction `distance` amie de la classe `point` qui calcule la distance entre deux points.
3. Créer deux points `a(1,1)` et `b(-1,2)`. Calculer leur distance par rapport à l'origine et la distance ab.

4. Calculer leurs symétriques et les associer à deux nouveaux points `a1` et `b1`,
5. Calculer la distance par rapport à l'origine de `a1` et `b1` et la distance `a1b1`.
6. Afficher le nombre de points créés.

7.3 Partie 2 : Classe modélisant une droite

1. Modifier le programme de l'exercice 1 en créant une classe `droite` qui modélise une droite passant par les points `a` et `b` ayant pour équation

$$\texttt{y=pente*x + y0} \tag{7.3.1}$$

   du plan et possédant les attributs suivants :
   - `point a,b;`
   - `float pente, y0;`
2. Déclarer la classe `droite` comme une classe amie de la classe `point`.
3. Déclarer deux fonctions membres privées `calculpente()` et `calculx0()` qui calculent la pente et `y0`.
4. Déclarer une fonction `affich_droite()` qui affiche la pente et `y0`.
5. Tester la classe `droite` avec un exemple de votre choix.

TP n° 8

Héritage

Objectifs du TP

- Mettre en pratique les notions d'héritage.
- Manipuler les classes dérivées.

8.1 Partie 1 : Classe modélisant un mammifère

On souhaite d'abord déclarer une classe `classemammifere` qui a les membres suivants :

1. `nom`, `couleuryeux` : variables membres de type tableau de 10 caractères.
2. `age`, `poid` : variables membres de type int.
3. `classemammifere()` : constructeur sans arguments qui initialise les variables membres avec les valeurs suivantes :
   (a) `nom= " Le nom "`,
   (b) `couleuryeux= " marron "`,
   (c) `age=0`,
   (d) `poid=0.`
4. `afficher()` : qui affiche les attributs de l'objet.

Toutes les variables membres possèdent l'étiquette `protected`.

Déclarer la classe `classemamifere` et définir ses méthodes.

8.2 Partie 2 : Classes dérivées modélisant un homme et un chien

On dérive maintenant deux classes `classehomme` et `classechien` de la classe `classemammifere`. La classe `classehomme` possède en plus des membres qu'elle hérite les membres suivants :

ismaried : de type booléen.

classehomme() : constructeur qui initialise l'objet crée avec les valeurs suivantes :

```
nom = "nom homme";
couleuryeux = "Bleu";
age = 40;
poid=70
ismaried= true;
```

afficher() : C'est une redéfinition de la méthode de la classe de base.

La classe `classechien` possède en plus des membres qu'elle hérite les nouveaux membres suivants :

hasqueue : varaible membre de type booléen.

afficher() : C'est une redéfinition de la méthode de la classe de base pour afficher les objet de la classe `classechien`.

classechein() : constructeur qui initialise l'objet crée avec les valeurs suivantes :

```
nom = "Snoopy"
age = 2
poid=10
hasqueue = true
```

1. Déclarer la classe `classemamifere` et définir ses méthodes.
2. Déclarer la classe `classechien` et définir ses méthodes.
3. Créer un objet de type `classemamifer` et afficher le.
4. Créer un objet de chaque classe dérivée et affichez-les.
5. Ajouter un constructeur avec arguments qui permet d'initialiser les objets des deux classes dérivées.
6. Ajouter deux méthodes qui permettent d'initialiser à partir du clavier les objets crée des classes dérivées. Faites cela avec le minimum de lignes de code possibles.
7. Tester les nouvelles méthodes en définissant de nouveaux objets puis en les affichants.

TP n° 9

Polymorphisme

Objectifs du TP

- Mettre en pratique les principaux concepts du polymorphisme.
- Voir la liaison du polymorphisme avec l'héritage.

9.1 Partie 1 : Classe modélisant un compte bancaire

```
class compte{
protected:
char proprietaire[10];
int numero;
double solde;
public:
compte(char* nom,int num, double soldinit){
strcpy(proprietaire,nom);
numero=num;
solde=soldinit;
}
compte(){
}
void depot(double somme){
solde=solde+somme;
}

void retrait(double somme){
if (solde<somme) {
cout<<"\n solde insuffisant pour le retrait sur le compte n" <<numero<<"\n";
return;
}
else
solde=solde-somme;
}
void virement(compte& c,double somme){
if (c.solde>=somme){
c.retrait(somme);
depot(somme);}
else
cout<<"\n solde insuffisant pour le virement à partir du compte n" <<numero<<"\n";
}
void affiche (){
cout<<'\n';
cout<<"Nom       :"<< proprietaire<<endl;
cout<<"Numéro    :"<<numero<<endl;
cout<<"Solde     :"<<solde<<endl;
```

```
38 }
39 };
```

Écrire un programme de test qui crée deux compte, fait un dépôt sur le premier, un retrait sur le deuxième puis un virement du premier vers le deuxième.

9.2 Partie 2 : Classe modélisant un compte bancaire avec découvert

1. Étendre la classe `compte` pour créer une classe `comptedecouvert` en rajoutant les attributs suivants :
   - Une variable membre `maxdecouvert` de type `double` qui fixe le découvert maximal.
   - Un constructeur qui admet en plus des arguments de la classe mère un argument `maxdecouvert` qui fixe le découvert maximal.
2. Redéfinir alors la méthode `retrait` en tenant compte de la possibilité de découvert.
3. Redéfinir la méthode `affiche` pour afficher le découvert maximum.
4. Tester la classe `comptedecouvert` en définissant un pointeur vers la classe de base.

9.3 Partie 3 : Sauvegarde des comptes dans un fichier

En vous inspirant de l'exemple du cours, écrire un programme qui permet de saisir les compte crée dans un fichier puis d'ouvrir le fichier et de les afficher. Il sera nécessaire de rajouter une méthode `creer()` à la classe compte.

Troisième partie

Mini-projets

Mini-projet n° 1

2010-2011

1.1 Présentation du Mini-Projet

On souhaite écrire une petite application qui gère le système de réservation d'un aéroport.

Le menu principal devra afficher les options suivantes :

1. Afficher la liste des vols disponibles.
2. Ajouter/supprimer un vol.

Pour le choix Afficher la liste des vols disponibles, on dispose du sous menu.

1. Créer une nouvelle liste de passagers pour un vol disponible.
2. Afficher une liste de passagers pour un vol donné.

Pour le choix Afficher une liste de passagers pour un vol donné. on dispose du sous menu

1. Ajouter un passager à une liste donnée.
2. Supprimer un passager d'une liste donnée
3. Afficher un passager d'une liste donnée.

Les objets vol, passager sont caractérisés par les attributs suivants :

Passager : `identifiant_pass, nom, prenom, date_naissance, numero_passeport, adresse.`

Vol : `identifiant_vol, datedepart, heuredepart, heurarrivee, destination, capacite.`

Vous avez la liberté de définir les méthodes que vous jugerez nécessaires pour ces classes (affichage, saisie, ...). Vous pouvez notamment vous inspirer des TP.

Vous pouvez également, si vous le souhaiter, utiliser les éléments de la bibliothéque standard.

Mini-projet n° 2

2011-2012

2.1 Présentation du Mini-Projet

On souhaite réaliser une application en mode console (sans interface graphique) pour la gestion d'une librairie. La fonction essentielle de cette application est d'aider le libraire à gérer les ouvrages dont il dispose, à gérer ses clients et leurs commandes de livres.

2.2 Fonctions de l'application

Les fonctions principales de l'application "Gestion librairie" sont les suivantes :

1. Gérer la liste des livres disponibles.
2. Gérer la liste des clients.
3. Gérer la liste des commandes.

2.2.1 Gestion de la liste des livres disponibles

La gestion de la liste des livres disponibles devra assurer les fonctionnalités suivantes :

1. Ajouter un nouveau livre.
2. Rechercher un livre et afficher ses informations.
3. Mettre à jour (modifier) les informations concernant un livre.
4. Supprimer un livre de la liste.

La liste des livres est enregistrée dans un fichier appelé "liste_livres.poc". Toutes les opérations citées plus haut doivent êtres enregistrées sur ce fichier.

2.2.2 Gestion de la liste des clients

La gestion de la liste des clients devra assurer les fonctionnalités suivantes :

1. Ajouter un nouveau client.
2. Rechercher un client et afficher ses informations.
3. Mettre à jour (modifier) les informations concernant un client.

4. Supprimer un client de la liste.

La liste des clients est enregistrée dans un fichier appelé "liste_clients.poc". Toutes les opérations citées plus haut doivent êtres enregistrées sur ce fichier.

2.2.3 Gestion de la liste des commandes

La gestion de la liste des commandes devra assurer les fonctionnalités suivantes :

1. Créer une nouvelle commande.
2. Rechercher une commande et afficher ces informations.
3. Mettre à jour (modifier) les informations concernant une commande.
4. Supprimer une commande.

La liste des commandes est enregistrée dans un fichier appelé "liste_commandes.poc". Toutes les opérations citées plus haut doivent êtres enregistrées sur ce fichier.

2.3 Caractérisations des objets de l'application

Un livre devra posséder les caractéristiques suivantes :

Caractéristique	Signification
ISBN	Code du livre.
Titre	Titre du livre.
Auteur	Premier auteur du livre.
Editeur	Editeur du livre.
Annee	Année d'édition du livre.
Prix	Prix du livre en DA.

Table 2.1 – Caractéristiques d'un livre

Un client devra posséder les caractéristiques suivantes :

Caractéristique	Signification
IDC	Identifiant du client.
Nom	Nom ou raison sociale.
Adresse	Adresse du client.
Tel	Numéro de téléphone du client.
Anciente	Date d'inscription du client.

Table 2.2 – Caractéristiques d'un client

Une commande devra posséder les caractéristiques suivantes :
De plus, on devra avoir la possibilité d'afficher plus d'information concernant une commande à propos des objets commandés et du client.

Caractéristique	Signification
IDCM	Identifiant de la commande.
IDCCM	Identifiant du client ayant passé la commande.
Date	Date de la commande.
Total	Prix total de la commande.

Table 2.3 – Caractéristiques d'une commande

2.4 Travail demandé

Écrire une application en C++ qui réalise les fonctionnalités citées plus haut. Le libre choix est donné pour l'organisation du code et le choix des outils utilisés ou de la démarche à suivre. Vous pourrez définir les méthodes jugées nécessaires pour la réalisation des objectifs.

2.4.1 Important !

Il est impératif de respecter les points suivants :
- Écrire un code lisible, structuré et commenté.
- Réaliser une bonne analyse et organiser votre démarche avant de commencer à programmer.
- Utiliser des menus à choix multiples pour plus de convivialité. Par exemple, l'écran d'accueil devra afficher plusieurs choix :
    1. Gestion des livres.
    2. Gestion des clients.
    3. Gestion des commandes.
    4. Quitter.

pour le choix 1, il sera possible de faire les choix suivants :

1. Afficher la liste des livres.
2. Ajouter un livre.
3. Supprimer un livre.
4. Modifier un livre.
5. Rechercher un livre.
6. Revenir au menu précédent.

2.4.2 Travail à remettre

Il est demander de remettre :

1. Un programme executable (.exe) sur lequel les tests de la notation seront effectués.
2. Un dossier contenant les fichiers sources.
3. Un rapport écrit où la démarche suivie sera détaillée.

Mini-projet n° 3

2012-2013

3.1 Présentation du Mini-Projet

Le but du mini-projet est de réaliser une application qui gére les informations relatives à plusieurs championnats de football. Ceci permettera aux utilisateurs de l'application de s'informer sur les résultats, classements et programmes de chaque championnat.

Un point important est de produire une application qui peut être réutilisée pour différents championnats, différentes saisons et différentes divisions.

3.2 Fonctionnalités de l'application

Les principales fonctions de l'application "championnat" sont les suivantes :

1. Gestion des championnats.
2. Gérer les équipes d'un championnat.
3. Gérer le calendrier d'un championnat.
4. Gérer les résultats pour chaque journée.
5. Gérer le classement d'un championnat.
6. Gérer les résultats pour chaque équipe.

3.2.1 Gestion des championnat

Plusieurs championnats peuvent être gérés en même temps (de différents pays, de différentes divisions et différentes saisons). L'utilisateur peut crée un nouveau championnat, le supprimer ou modifier ces attribus.

3.2.2 Gestion des équipes

Pour un championnat donné, l'utilisateur peut crée, modifier et supprimer des équipes.

3.2.3 Gestion du calendrier

Initialement, le programmeur crée un calendrier pour un championnat donné. Il peut toutefois le modifier au cours de la saison. Les fonctionnalités principales de cette partie sont :

1. Ajouter une nouvelle journée.
2. Mettre à jour une journée.
3. Supprimer une journée.

3.2.4 Gestion des résultats

La gestion des résultats devra permettre de :

1. Saisir les résultats d'une journée.
2. Afficher les résultats d'une journée.
3. Mettre à jour les réultats d'une journée.

3.2.5 Gestion du classement

La gestion du classement permettera d'afficher un tableau qui contiendra les nombre de points, le classements et les autres informations (matchs gagnés, nuls, perdus, écart de buts) pour chaque équipe.

3.3 Travail demandé

Écrire une application en C++, en mode console (ans interface graphique) qui réalise les fonctionnalités citées plus haut. Le libre choix est donné pour l'organisation du code et le choix des outils utilisés ou de la démarche à suivre. Vous pourrez définir les méthodes jugées nécessaires pour la réalisation des objectifs.

3.3.1 Important !

Il est impératif de respecter les points suivants :
- Écrire un code lisible, structuré et commenté.
- Réaliser une bonne analyse et organiser votre démarche avant de commencer à programmer.
- Utiliser des menus à choix multiples pour plus de convivialité. Par exemple, l'écran d'accueil devra afficher plusieurs choix :
    1. Consulter les championnats.
    2. Modifier les championnats.
    3. Ajouter un nouveau championnat.
    4. Quitter.

par exemeple, pour le choix 1, l'application affichera la liste des championnat disponibles, après elle proposera les choix suivants :

1. Sélectionner un championnat.
2. Revenir au menu principal.
3. Quitter.

3.3.2 Travail à remettre

Il est demandé de remettre :

1. Un programme executable (.exe) sur lequel les tests de la notation seront effectués.
2. Un dossier contenant les fichiers sources.
3. Un rapport écrit où la démarche suivie sera détaillée.

Bibliographie

[1] Y. Gerometta and J. L. Corre, *C++ Le Guide Complet*. Micro Applications, 2008.

[2] G. Willms, *C++*. PC Poche, Micro Applications, 2000.

[3] C. Delannoy, *Programmer en C++*. Eyrolles, 2006.

[4] C. Delannoy, *Apprendre le C++*. Eyrolles, 2008.

[5] S. Dupin, *Le Langage C++*. Le tout en poche, Campus Press, 2005.

[6] M. Nebra and M. Shcaller, *Programmez avec le Langage C++*. Le Livre du Zéro, 2009.

[7] N. Benabadji and N. Bechari, *Guide Pratique de Programmation en C++*. Houma INFO, Editions Houma, 2009.

[8] S. Graine, *Le Langage C++*. L'Abeille, 2004.

[9] C. Delannoy, *Exercices en Langage C++*. Chihab-Eyrolles, 1993.

[10] B. Stroustrup, *The C++ Programming Language*. Addison-Wesley, 3rd ed., 1997.

[11] J. R. Hubbard, *Theory and Problems of Programming with C++*. Schaum's Outline Series, McGraw-Hill, 2nd ed., 2000.