Langage C++ et POO

Programme du cours

SOMMAIRE

CHAPITRE I PRINCIPE DE LA PROGRAMMATION ORIENTEE OBJET 1
I Définition 1
II Concepts fondamentaux de la POO 1
II.1 Objet et classe 1
II.2 Encapsulation des données 2
II.3 L'HERITAGE 3
II.4 Le polymorphisme 4
CHAPITRE II BASE DE LA PROGRAMMATION EN C++ 5
I Généralités sur le langage C++ 5
I.1 Schéma d'un programme C++ 5
I.2 La compilation 5
II Les commentaires 6
III Les déclarations 6
III.1 Déclaration des variables 6
III.2 Déclaration des constantes 7
IV Les instructions de bases 8
IV.1 Les opérateurs en C++ 8
IV.2 Opérateur ternaire : 8
IV.3 Les entrées/sorties 8
V Les structures de contrôle et itératives 9
V.1 La structure if (Si) 9
V.2 Les choix multiples 9
V.3 La structure while 2
V.4 Structure do..while 2
V.5 Structure for 2
VI Les pointeurs 3
VII Les tableaux 3
VIII Les fonctions 5
VIII.1 Généralités 5
VIII.2 Surcharge des fonctions 6
VIII.3 Paramètres constants 8
VIII.4 Arguments par défaut 8
VIII.5 Passage des paramètres 9
VIII.6 Fonctions inline 10
CHAPITRE III Classes 11
I Généralités sur les classes 11
I.1 Définition 11
I.2 Encapsulation 11
I.3 Déclaration/définition et utilisation d'une classe 11
I.4 Surcharge des méthodes et arguments par défaut 13
II Les constructeurs 13
II.1 Constructeurs : Méthodes d'initialisation d'un objet 13
II.2 Constructeur de recopie 14
III Tableau d'objets 14
IV Surcharge des opérateurs 15
IV.1 Surcharge des opérateurs en interne 15
IV.2 Utiliser les opérateurs >> et << 15

V Fonctions amies 16
VI Attributs et méthodes statiques 17
VI.1 Attributs statiques 17
VI.2 Méthodes statiques 18
VII Liste d'initialisation 19
Travaux Dirigés 20
CHAPITRE IV HERITAGE ET DERIVATION 23
I Définition et représentation 23
I.1 Définition 23
I.2 Représentation 23
I.3 Mise en œuvre de l'héritage simple 24
II Méthodes virtuelles 27
II.1 Généralités : fonctions virtuelle 27
II.2 Méthodes virtuelles pures, classes abstraites et interface 28
TRAVAUX DIRIGES 29
CHAPITRE V Template, Gestion des erreurs et collections d'objet 32
I La généricité : Les Templates 32
I.1 Patrons de fonctions 32
I.2 Classe template : patron de classes 33
II Gestion des collections d'objets 34
II.1 Les méthodes communes aux conteneurs 35
II.2 Les itérateurs 35
II.3 Les conteneurs séquences concrets: list , vector 36
II.4 Les conteneurs associatifs : cas du map 37
III Les espaces de nommage (namespace) 38
III.1 Introduction 38
III.2 Espaces de nommage nommés 39
III.3 Les déclarations using 39
IV Gestion des exceptions 40
IV.1 La problématique :Gérer toutes les erreurs dans un programme 40
IV.2 Principe de la gestion des exceptions 40
TRAVAUX DIRIGES 42

CHAPITRE I PRINCIPE DE LA PROGRAMMATION ORIENTEE OBJET

I Définition

La programmation Orientée Objet est un paradigme (style) de programmation consistant à assembler des briques logicielles (objets). La programmation orientée objet (POO) consiste à définir des objets logiciels et à les faire interagir entre eux. Ce style de programmation est aujourd'hui plébiscitée par un grand nombre de développeurs.

En effet, ce type de programmation possède de nombreux avantages :

- Réutilisation du code, donc gain de temps
- Code mieux structuré, maintenance plus aisée structuration du code

La programmation orientée objet est apparue avec, pour objectifs principaux :

1) de concevoir l'organisation de grands projets informatiques autour d'entités précisément structurés, mêlant données et fonctions (les objets) facilitant la modélisation de concepts sophistiqués ;
2) d'améliorer la sûreté des logiciels en proposant un mécanisme simple et flexible des données sensibles de chaque objet en ne les rendant accessibles que par le truchement de certaines fonctions associées à l'objet (encapsulation) afin que celles-ci ne soient pas accessibles à un programmeur inattentif ou malveillant.
3) de simplifier la réutilisation de code en permettant l'extensibilité des objets existants (héritage) qui peuvent alors être manipulés avec les mêmes fonctions (polymorphisme).

II Concepts fondamentaux de la POO

Les concepts fondamentaux de la POO regroupent celles de :

- Objet
- Encapsulation des données
- Classe
- Héritage
- Polymorphisme

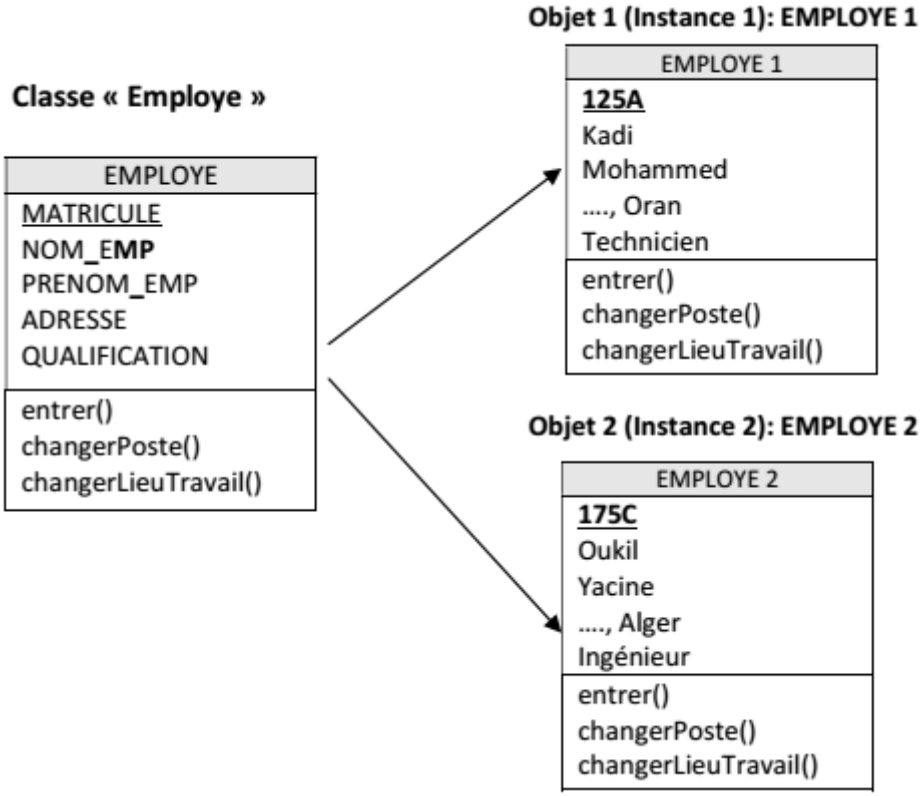
II.1 Objet et classe

Un objet est une entité cohérente rassemblant des données et du code travaillant sur ses données.

- **Son Identité :** C'est ce qui permet d'identifier un objet parmi d'autres. Dans le code, le nom de variable remplit cette fonction.
- **Ses Propriétés** (ou attributs) : Les propriétés sont les données intrinsèques de l'objet que l'on souhaite gérer. En prenant l'exemple d'un objet Rectangle, il faudra au minimum gérer sa longueur et sa largeur.
- **Ses Méthodes :** Les méthodes sont des fonctions applicables à un objet.

Une classe peut être considérée comme un moule à partir duquel on peut créer des objets. Une classe est une structure informatique regroupant les caractéristiques et les modes de fonctionnements d'une famille d'objets, appelés instances de la classe.

Exemple :



« EMPLOYE 1 » et « EMPLOYE 2 » sont caractérisés par les mêmes propriétés (matricule, nom, prénom, qualification) mais associés à des valeurs différentes. Ils ont le même comportement (entrer/ changerposte,…) mais ont des identités différentes. Et il en serait de même pour tous les employés par conséquent on a une classe Employé.

Exemple : Donnez le schema des classes ci dessous

- Un Etudiant
- Un Cercle
- Une Figure
- Une Fraction
- Un Complexe
- Une Porte
- Une voiture

II.2 Encapsulation des données

L'encapsulation est l'un des trois grands principes du paradigme objet. Il consiste à protéger la définition des objets. Ce principe, digne héritier des principes d'abstraction de données et d'abstraction procédurale prône les idées suivantes :

- **Un objet rassemble en lui même ses données** (les attributs) et le code capable d'agir dessus (les méthodes)
- **Abstraction de données :** la structure d'un objet n'est pas visible de l'extérieur, son interface est constituée de messages invocables par un utilisateur. La réception d'un message déclenche l'exécution de méthodes.
- **Abstraction procédurale :** Du point de vue de l'extérieur (c'est-à-dire en fait du client de l'objet), l'invocation d'un message est une opération atomique. L'utilisateur n'a aucun élément d'information sur la mécanique interne mise.

Dans les versions canoniques du paradigme objet, les services d'un objet ne sont invocables qu'au travers de messages, lesquels sont individuellement composés de :

- Un nom
- Une liste de paramètres en entrée
- Une liste de paramètres en sortie

L'ensemble constitué du nom de la méthode et du type de chaque paramètre d'entrée constitue sa ***signature***

La liste des messages auxquels est capable de répondre un objet constitue son interface : C'est la partie publique d'un objet. Tout ce qui concerne son implémentation doit rester caché à l'utilisateur final : c'est la partie privée de l'objet.

Il est très important de cacher les détails d'implémentation des objets à l'utilisateur. En effet, cela permet de modifier, par exemple la structure de données interne d'une classe (remplacer un tableau par une liste chaînée) sans pour autant entraîner de modifications dans le code de l'utilisateur, l'interface n'étant pas atteinte. De plus cela permet de pouvoir assurer la cohérence interne des propriétés de la classe(S'assurer que le sexe vaut M ou F)

L'encapsulation est gérer à travers le niveau de visibilité. On distingue :

- Le niveau **public** : L'élément est visible partout
- Le niveau **private** : L'élément est visible uniquement à l'intérieur de l'objet
- Le niveau **protected** : L'élément est visible par l'objet et ses descendants.
- Le niveau **package** : L'élément est visible par l'objet et par celles situées dans le même package.

II.3 L'HERITAGE

Il est chargé de traduire le principe naturel de Généralisation / Spécialisation. En effet, la plupart des systèmes réels se prêtent à merveille à une classification hiérarchique des éléments qui les composent. Il est basé sur l'idée qu'un objet spécialisé bénéficie ou hérite des caractéristiques de l'objet le plus général auquel il rajoute ses éléments propres.

En termes de concepts objets cela se traduit de la manière suivante :

- On associe une classe au concept le plus général, nous l'appellerons classe de base ou classe mère ou super - classe.
- Pour chaque concept spécialisé, on dérive une classe du concept de base. La nouvelle classe est dite classe dérivée ou classe fille ou sous-classe

On parle également de relation est-un pour traduire le principe de généralisation / spécialisation.

L'héritage peut être simple ou multiple.

Exemple : Héritage simple

Exemple : Héritage multiple

II.4 Le polymorphisme

Le polymorphisme est le troisième des trois grands principes sur lequel repose le paradigme objet. C'est assurément son aspect à la fois le plus puissant et le plus troublant. Comme son nom l'indique le polymorphisme permet à une méthode d'adopter plusieurs formes sur des classes différentes.

Exemple :

- Les classes carré et Rectangle ont probablement la méthode surface() ;
- Dans une classe Complexe, on aura probablement une méthode add(Complexe c) et une seconde add(double x)

CHAPITRE II BASE DE LA PROGRAMMATION EN C++

Il est question dans ce chapitre de présenter les principes de la programmation en C++

I Généralités sur le langage C++

Le langage C++ a été conçu par ***Bjarne Stroustrup*** au début des années 1980. Le C++ est une surcouche au langage C qui intègre les concepts de la POO. Le C++ préserve donc à de minimes exceptions la syntaxe du langage C. L'objectif de ***B. Stroustrup*** en créant ce langage était de faciliter aux programmeurs C la transition vers la programmation orientée objet.

I.1 Schéma d'un programme C++

Un programme C++ est généralement constitué de plusieurs modules, chaque module est composé de deux fichiers sources :

- Un fichier contenant la description de l'interface du module
- Un fichier contenant l'implémentation proprement dite du module

Un suffixe est utilisé pour déterminer le type de fichier

- ***.h***, .hpp, : pour les fichiers de description d'interface (header files ou include files)
- ***.cpp***, .c, .cc, .cxx,.c++ : pour les fichiers d'implémentation

Dans un fichier source on peut trouver :

- Commentaires
- Instructions pré-processeur
- Instructions C++

Tout programme doit avoir un point d'entrée nommé main

```
int main()
{
 return 0;
}
```

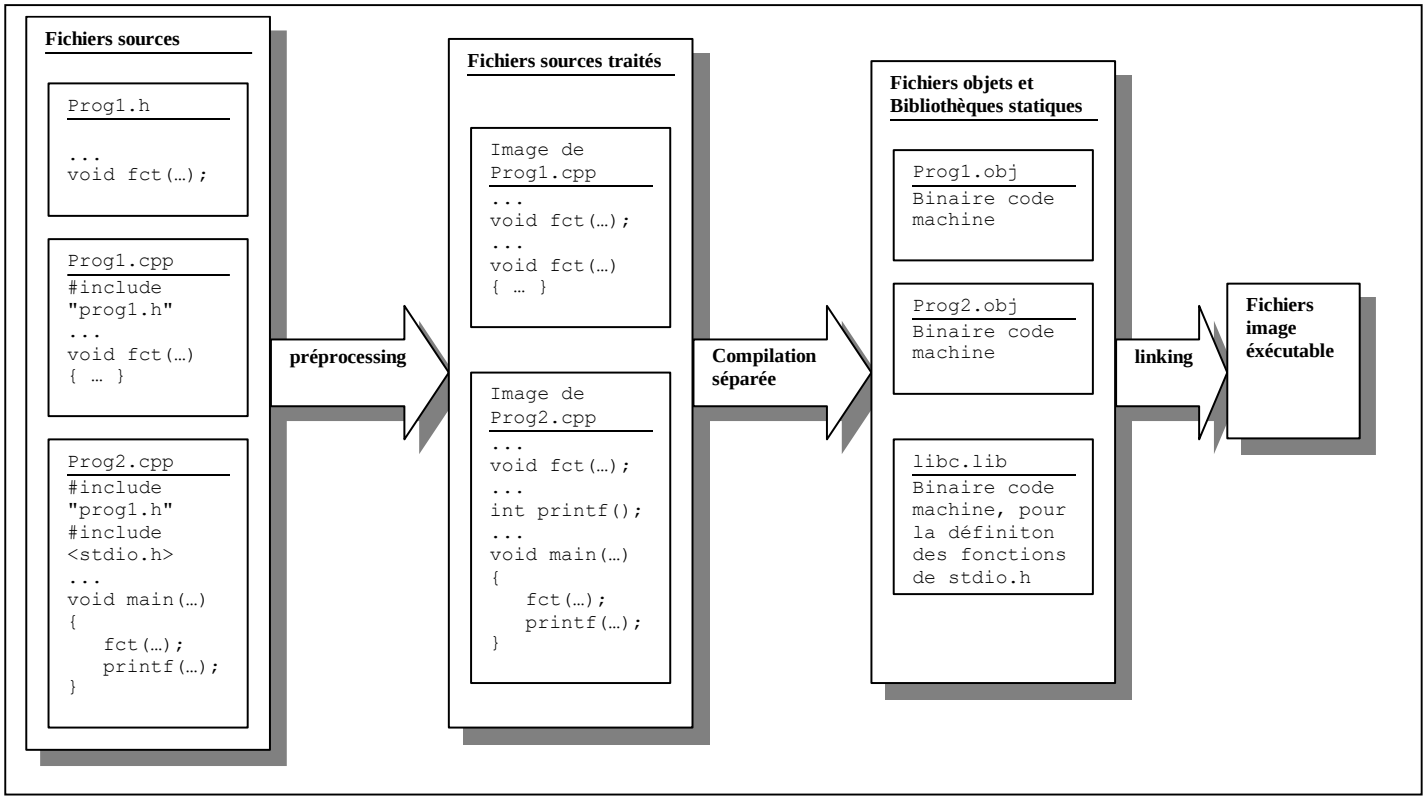
La ***fonction main*** est la fonction appelée par le système d'exploitation lors de l'exécution du programme. Il retourne un entier au système: 0 (zéro) veut dire succès

I.2 La compilation

Au départ du processus de compilation, on ne dispose que des fichiers sources C/C++.

- ✓ La première étape est le **traitement des fichiers sources avant compilation** (*préprocessing* en anglais). Le préprocesseur fabrique de nouveaux fichiers sources C/C++, avec les macros remplacées, les commentaires supprimés, les fichiers d'entête inclus…
- ✓ La deuxième étape est la **compilation séparée**. Le résultat pour chaque fichier source compilé est un *fichier objet*. Un fichier objet contient la traduction en langage machine du code du fichier source correspondant. Les bibliothèques statiques utilisées pour inclure des fonctionnalités aux programmes sont des fichiers objets.
- ✓ La dernière étape est **l'édition de lien** (linking en anglais). On y effectue le regroupement de tous le code des fichiers objets (sources et bibliothèque), et la résolution des références inter-fichier. Le produit est un *fichier image* exécutable par la machine.

En résumé, la compilation s'effectue comme suit :



II Les commentaires

On distingue plusieurs types de commentaires :

II.1.1 Les Commentaires sur plusieurs lignes : délimités par /* (début) et */ (fin).

```
/* Un commentaire en une seule ligne */
/*
  * Un commentaire sur plusieurs
  * lignes
*/
```

II.1.2 Commentaires sur une seule ligne : délimités par // (début) et fin de ligne

III Les déclarations

III.1 Déclaration des variables

En C++, toute variable doit être déclarée avant d'être utilisée. Une variable se déclare de l'une des façons suivantes:

```
type Nom_de_la_variable < (valeur) > ;
```

ou bien utiliser la même syntaxe que dans le langage C.

```
type Nom_de_la_variable < = valeur > ;
```

C++ est un langage typé . Les types de bases utilisables sont :

Type de donnée	Signification	Taille (en octets)	Plage de valeurs acceptée
char	Caractère	1	-128 à 127
unsigned char	Caractère non signé	1	0 à 255
short int	Entier court	2	-32 768 à 32 767
unsigned short int	Entier court non signé	2	0 à 65 535
int	Entier	2 (sur processeur 16 bits) 4 (sur processeur 32 bits)	-32 768 à 32 767 -2 147 483 648 à 2 147 483 647
unsigned int	Entier non signé	2 (sur processeur 16 bits) 4 (sur processeur 32 bits)	0 à 65 535 0 à 4 294 967 295
long int	Entier long	4	-2 147 483 648 à 2 147 483 647
unsigned long int	Entier long non signé	4	0 à 4 294 967 295
float	Flottant (réel)	4	$-3.4*10^{-38}$ à $3.4*10^{38}$
double	Flottant double	8	$-1.7*10^{-308}$ à $1.7*10^{308}$
long double	Flottant double long	10	$-3.4*10^{-4932}$ à $3.4*10^{4932}$
bool	Booléen	Même taille que le type int, parfois 1 sur quelques compilateurs	Prend deux valeurs: ***true*** et ***false*** mais une conversion implicite (valant 0 ou 1) est faite par le compilateur lorsque l'on affecte un entier

Notez qu'en C/C++ il n'y a pas de type de base pour les chaînes de caractères. Elles sont représenté comme un tableau dont les éléments sont des caractères à l'aide du type ***string.***

Exemple :

Int n=5,k,s(10) ;

III.2 Déclaration des constantes

Contrairement aux variables les constantes ne peuvent pas être initialisées après leur déclaration et leur valeur ne peut pas être modifiée après initialisation. Elles doivent être déclarées avec le mot clé ***const*** et obligatoirement initialisées dès sa définition.

Syntaxe:

const <type> <NomConstante> = <valeur>;

ou

const <type> <NomConstante> (<valeur>);

Exemple:

const *char c ='A'; //exemple de constante caractère*

const *int annee=2022; //exemple de constante entière*

const *double PI =3.14; //exemple de constante réelle*

const string motDePasse="secret@KMER237"; //exemple de constante chaine de caractères

IV Les instructions de bases

IV.1 Les opérateurs en C++

Un identificateur a une portée limitée au bloc dans lequel il est déclaré. Un bloc commence par une accolade ouvrante et se termine par une accolade fermante.

Les différents opérateurs sont consignés dans le tableau ci-dessous.

Syntaxe	Sémantique
x = y	affectation (x prend la même valeur que celle de y)
x = = y	(test d'égalité) vrai si x = y, faux sinon
x != y	(test de différence) faux si x = y, vrai sinon
x < y	infériorité stricte (vrai si x<y)
x > y	supériorité stricte (vrai si x>y)
x <= y	infériorité large (vrai si x<=y)
x >= y	Supériorité large (vrai si x>=y)
x + y	Addition
x-y	Soustraction
x *y	Multiplication
x/y	Division
x % y	x modulo y
++x	Pré incrémentation
- -x	Pré décrémentation
x++	Post incrémentation
x- -	Post décrémentation
x op =y	Où op est un opérateur : x= x op y
!	Négation logique
x && y	Et logique
x || y	Ou logique
(T)x	transtype x vers le type T (si possible)

IV.2 Opérateur ternaire :

Cond ?a :b
Si cond est vrai alors a sinon b.
Exemple :
x=(a>b ?a :b) permet de calculer le plus grand entre a et b.

IV.3 Les entrées/sorties

On peut utiliser les routines d'E/S da la bibliothèque standard de C (<stdio.h>). Mais C++ possède aussi ses propres possibilités d'E/S. Les nouvelles possibilités d'E/S de C++ sont réalisées par l'intermédiaire des opérateurs **<<** (sortie), **>>** (entrée) et des flots (stream) définis dans la bibliothèque <iostream>, suivants

- ✓ ***cin*** : flot d'entrée correspondant à l'entrée standard
- ✓ ***cout*** : flot de sortie correspondant à la sortie standard
- ✓ ***cerr*** : flot de sortie correspondant à la sortie standard d'erreur

Les Syntaxes :

```
cout << exp_1 << exp_2 << … … << exp_n ;
```

```
cin >> var_1 >> var_2 >> … … >> var_n ;
```

Tous les caractères de formatage comme '\t', '\n' peuvent être utilisés. Par ailleurs, l'expression endl permet le retour à la ligne et le vidage du tampon.

Exemple:

```
#include <iostream> //Bibliotheque d'E/S standard de C++
using namespace std;
int main()
{   int val1, val2;
    cout << "Entrer deux entiers: " << endl;
    cin >> val1 >> val2;
    cout << "Valeurs entrées: " << val1 << " et " << val2 << endl;
    cout << "valeur 1+valeur 2 =" << val1 + val2 << endl;
    return 0;
}
```

V Les structures de contrôle et itératives

V.1 La structure if (Si)

```
if(test){
    à exécuter si test est vrai ;
}
```

avec la variante :

```
if(test){
    à exécuter si test est vrai ;
}else{
    à exécuter si test est faux ;
}
```

V.2 Les choix multiples

Pour une variable x dont le type est byte, char, short et int, Java dispose aussi de l'instruction **switch** : Il correspond à l'instruction de sélection multiple **case** du Pascal.

La syntaxe est :

```
switch(x){
        case a1: Instructions si x=a1 ;break;
        case an: Instructions si  x = an ;break;
        default: // Optionnel mais fortement recommandé
                //partie si x n'est aucune des valeurs a1, …, an
    }
```

Notons que dans la définition précédente que chaque instruction case se termine par une instruction break, qui provoque un saut à la fin du corps de l'instruction switch. Ceci est la manière habituelle de construire une instruction switch.

Cependant break est optionnel. S'il n'est pas là, le code associé à l'instruction case suivante est exécuté, et ainsi de suite jusqu'à la rencontre d'une instruction break. Bien qu'on n'ait généralement pas besoin d'utiliser cette possibilité, elle peut se montrer très utile pour un programmeur expérimenté. La dernière instruction, qui suit default, n'a pas besoin de break car en fait l'exécution continue juste à l'endroit où une instruction break l'aurait amenée de toute façon. Il n'y a cependant aucun problème à terminer l'instruction default par une instruction break si on pense que le style de programmation est une question importante~~Les principales structures c++ sont :~~

Ex : switch(c){

```
        case 'a':
        case 'e':
        case 'i':
        case 'o':
        case 'u':
        case 'y':
                cout
        break;
        default:
                System.out.println("consonant");
}
```

V.1V.3 La structure while

```
while(expression)
        bloc_instructions
```

Tant que l'expression est vraie, on répète le traitement, sinon on passe à la suite du programme.

V.2V.4 Structure do..while

```
do
        Bloc d'instructions
while(expression);
```

Tant que l'expression n'est pas nulle (c'est-à-dire fausse), le traitement est répété. L'expression est évaluée après l'exécution de "Traitements", qui est au moins exécuté une fois.

V.3V.5 Structure forfor

```
for(instruction;expression1;expression2)
        bloc d'instructions
```

- ✓ ***instruction*** précise l'initialisation de la boucle ;
- ✓ ***expression1*** est évaluée à chaque itération et provoque la sortie de la boucle lorsqu'elle devient fausse ;
- ✓ à chaque itération, le bloc d'instructions est exécuté et expression2 est évaluée. Il est possible de supprimer l'une ou les deux de ces derniers.

Par exemple, pour le calcul de la somme des dix premiers nombres entiers :

```
int som = 0;
int i=0;
for(i = 0;i<10;i++)
        som+ = i;
```

VI Les pointeurs

Un pointeur est une variable qui contient une adresse. Rappelons qu'un pointeur représente un type.

Une variable de type **T***est destinée à recevoir l'adresse d'une variable de type **T**.

Lorsque **ptr** est une variable pointeur de type **T***, l'opérateur ***** permet de dé-référencer ce pointeur :

- ✓ ***ptr** est la variable de type **T**
- ✓ L'adresse est stockée dans la variable ***ptr***.

Par ailleurs, l'opérateur **&** permet de fournir l'adresse d'une variable. En synthèse on a :

```
int var=12; // variable de type int déclarée et initialisée
int * ptr;  // variable de type pointeur sur int non initialisée.
// ne pas exécuter *ptr=45  ici, la variable pointeur n'est pas initialisée !!!
//on peut initialiser en faisant ptr=new int ;
ptr=&var ;  // ptr initialisée avec l'adresse de la variable var.
*ptr=45  ;  // *ptr  dé-référence ptr. A cet instant, *ptr est la variable var.
```

VII Les tableaux

Une variable entière de type int ne peut contenir qu'une seule valeur. Si on veut stocker en mémoire un ensemble de valeurs, il faut utiliser une structure de données appelée tableau qui consiste à réserver des espaces de mémoire contiguë dans lequel les éléments du tableau peuvent être rangés.

Comme toujours en C++, une variable est composée d'un nom et d'un type. Comme les tableaux sont des variables, cette règle reste valable. Il faut juste ajouter une propriété supplémentaire, la taille du tableau.

La Syntaxe :

```
type nomVar [taille];
```

- ***type*** : définit le type des éléments que contient le tableau.
- ***nomVar***: le nom que l'on décide de donner au tableau.
- ***taille***: nombre entier qui détermine le nombre de cases que le tableau doit comporter.
- La taille du tableau ne peut être fixée dynamiquement
- ***nomVar[i]*** représente le ième élément du tableau. Le premier se trouve à l'indice 0

Exemple:

```
int  meilleurScore  [5];
char    voyelles    [6];
double notes [20];
```

Il est aussi possible d'initialiser le tableau à la déclaration en plaçant entre accolades les valeurs, séparées par des virgules. Exemple :

```
int meilleurScore [5] = {100, 432, 873, 645, 314};
```

- Le nombre de valeurs entre accolades ne doit pas être supérieur au nombre d'éléments du tableau.
- Les valeurs entre accolades doivent être des constantes, l'utilisation de variables provoquera une erreur du compilateur.
- Si le nombre de valeurs entre accolades est inférieur au nombre d'éléments du tableau, les derniers éléments sont initialisés à 0.
- Si le nombre de valeur entre accolades est nul, alors tous les éléments du tableau s'initialisent à zéro.

En interne, les variables du tableau sont stockées les une à la suite des autres. Ce qui implique qu'un tableau est un pointeur sur le premier élément :

```
long  TableauDentier[13] ;
long  PremierEntier = TableauDentier[0];
```

Équivalent à :

```
long  PremierEntier = *TableauDentier;
```

Il existe une arithmétique des pointeurs :

p + i = Adresse contenue dans p + i * taille en octets d'un élément de p, d'où :

long DernierEntier = *(TableauDentier+12);

Notez que la fonction C **sizeof()** retourne la taille d'un type ou d'une variable en octets :

```
long Entier = 0;
int taille1 = sizeof(Entier);  // taille 1 = 4
int taille2 = sizeof(long);    // taille 2 = 4
```

Commentaires

Sous Java, il existe deux types de commentaires. Les commentaires servant à expliquer le code ainsi que les commentaires utilisés dans le cadre de la documentation du code (JAVADOC).

Les commentaires servant à expliquer le code sont placés après « // » si le commentaire tient sur une ligne, et entre « /* */ » si celui-ci tient sur plusieurs lignes.

Ex :

// ceci est un commentaire sur une ligne

/* Ceci est le début du commentaire qui commence sur cette ligne

Continue sur celle-ci

Et se termine ici */

Pour la génération de la JavaDoc, le commentaire est placé entre « /** … */ ».

Ex:

/** ici se trouve le code de la JavaDoc*/

VIII Les fonctions

VIII.1 Généralités

Une fonction est une opération définie par le programmeur caractérisé par :

- Son nom,
- Le type de valeur qu'elle renvoie (**void** si la fonction ne renvoie rien)
- Les paramètres qu'elle reçoit pour faire son travail,
- L'instruction-bloc qui effectue le travail (corps de la fonction).

Toute fonction doit être déclarée avant d'être appelée pour la première fois. La définition d'une fonction peut faire office de déclaration. Le rôle des déclarations est de signaler l'existence des fonctions aux compilateurs afin de les utiliser, tout en reportant la définition de ces fonctions plus loin ou dans un autre fichier.

Syntaxe :

```
type nomDeLafonction (paramètres) ;
```

Exemple :

```
double Moyenne(double x, double y);   // Fonction avec paramètres et type retour
char LireCaractere();   // Fonction avec type de retour et sans paramètres
```

```
void VoirValeurs(int nb, double val); //Fonction avec paramètres et sans type de retour
```

Toute fonction doit être définie avant d'être utilisée.

Syntaxe:

```
type nomDeLafonction (paramètres)
{
   Instructions ;
}
```

Exemple:

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

double moyenne(double a, double b);  // déclaration de la fonction somme :

int main()
{    double x, y, res;
      cout << "Tapez la valeur de x : ";
      cin >> x;
      cout << "Tapez la valeur de y : ";
      cin >> y;
      res = moyenne(x, y);  //appel de notre fonction somme
      cout <<" Moyenne entre "<< x <<" et " << y << " est : " << res<<endl;
      return 0;
 }
// définition de la fonction moyenne :
 double moyenne(double a, double b)
 {    double moy;
      moy = (a + b)/2;
      return moy;
 }
```

- ✓ Dans ce programme, on a créé une fonction nommée moyenne qui reçoit deux nombres réels a et b en paramètre et qui, une fois qu'elle a terminé, renvoie un autre nombre moy réel qui représente la moyenne de a et b. l'appel de cette fonction se fait au niveau du programme principale (main).
- ✓ Ici, le prototype est nécessaire, car la fonction est définie après la fonction main () qui l'utilise. Si le prototype est omis, le compilateur signale une erreur.
- ✓ Le prototype peut être encore écrit de la manière suivante :

double moyenne (double , double)

VIII.2 Surcharge des fonctions

En C++, Plusieurs fonctions peuvent porter le même nom si leurs signatures diffèrent***. La signature d'une fonction correspond aux caractéristiques de ses paramètres***

- ✓ leur nombre

- ✓ le type respectif de chacun d'eux

Le compilateur choisira la fonction à utiliser selon les paramètres effectifs par rapport aux paramètres formels des fonctions candidates.

Exemple:

```
#include <iostream>
using namespace std;
// définition de la fonction somme qui calcul la somme de deux réels
double somme(double a, double b)
 {    double r;
      r = a + b;
      return r;
 }
// définition de la fonction somme qui calcul la somme de deux entiers
 double somme(int a, int b)
 {    double r;
      r = a + b;
      return r;
 }
 // définition de la fonction somme qui calcul la somme de trois réels
 double somme(double a, double b, double c)
 {    double r;
      r = a + b + c;
      return r;
 }
int main()
{     double x, y,z, rep;
      cout << "Tapez la valeur de x : ";
      cin >> x;
      cout << "Tapez la valeur de y : ";
      cin >> y;
      cout << "Tapez la valeur de z : "; cin >> z;
      //appel de notre fonction somme double,double)
      rep = somme(x, y);
      cout <<  x << " + " << y << " = " << rep << endl;
      //appel de notre fonction somme (int,int)
      int a= static_cast<int>(x);
      int b= static_cast<int>(y);
       rep = somme(a,b);
      cout <<a <<" + "<< b << " = " << rep << endl;
      //appel de notre fonction somme (double, double, double)
      rep = somme(a, b,z);
      cout<<  a << " + " << b << " + " << z<< " = " << rep << endl;
      return 0;
}
```

VIII.3 Paramètres constants

Considérons le code :

```
float Division(int num,int den)
{
     if(den=0) return num;
     return (float)num/den ;
}
```

La fonction précédente contient une erreur de programmation que le compilateur ne peut détecter.

En effet, **den=0** est une affectation et non un test d'égalité. Le paramètre den prend donc systématiquement la valeur 0. Comment prévenir de telles erreurs? On peut utiliser le modificateur **const** pour les paramètres d'entrée (paramètres qui ne sont pas censés évoluer dans la fonction).

```
float Division(const int num, const int den)
{
     if(den=0) return num;// error C2166: l -value specifies const object
     else return (float)num/den ;
}
```

Dans ce cas, le compilateur indique qu'une variable (ou un objet) constante est à gauche d'une affectation (l-value). Le compilateur détecte ainsi très facilement ce type d'erreur. Le programmeur qui prend l'habitude de déclarer comme constants les paramètres d'entrée prévient ce type d'erreur.

VIII.4 Arguments par défaut

On peut, lors de la déclaration d'une fonction, donner des valeurs par défaut à certains paramètres des fonctions. Ainsi, lorsqu'on appelle une fonction, on ne sera pas obligé d'indiquer à chaque fois tous les paramètres

Exemple :

```
#include <iostream>
using namespace std;
void afficheLigne(const char c, const int n=5)
{  for(int i=0 ; i<n ;  ++i)
     cout<<c ;
}

int main()
{    afficheLigne('+') ;
     cout<<endl ;
     afficheLigne('*',8) ;
     return 0 ;
}
```

Il y a quelques règles que vous devez retenir pour les valeurs par défaut :

- ✓ Seul le prototype doit contenir les valeurs par défaut.
- ✓ Les valeurs par défaut doivent se trouver à la fin de la liste des paramètres.
- ✓ Vous pouvez rendre tous les paramètres de votre fonction facultatifs.

VIII.5 Passage des paramètres

Par défaut le passage de paramètre est un passage par valeurs. Il existe 2 possibilités pour transmettre des paramètres par variables :

- ✓ Les références
- ✓ Les pointeurs

VIII.5.1 Passage par références

C'est la méthode la plus simple. Elle consiste à utiliser le mécanisme de référence « & »(On précède chaque paramètre passé par adresse avec le symbole &).

Exemple :

```
#include <iostream>
using namespace std;
void sp (int  &);   // ajout de la référence au niveau du prototype
int main() {
      int n = 3;
      cout << "n=" << n << endl;
      sp (n); //appel de la fonction
      cout << "n=" << n<<endl ;;
      return 0 ;
}
void sp (int &  nbre)        //Ajout de la référence dans la fonction
{
      cout << "----------"<<endl ;
      cout << "nbre=" << nbre << endl;
       nbre = nbre + 2;
      cout << "nbre=" << nbre<<endl ;
       cout << "----------"<<endl ;
}
```

VIII.5.2 Passage par adresses (pointeurs)

Consiste à passer l'adresse d'une variable en paramètre au moment de l'appel et d'utiliser un pointeur au moment de la déclaration. Le pointeur indique au compilateur que ce n'est pas la valeur qui est transmise, mais une adresse (un pointeur).

Exemple :

```
#include <iostream>
using namespace std;
void sp (int  *);   // ajout de l'étoile au niveau du prototype

int main() {
      int n = 3;
      cout << "n=" << n << endl;
      sp (&n); //appel de la fonction
      cout << "n=" << n<<endl ;
      return 0 ;
}
void sp (int *  nbre)       //Ajout de la référence dans la fonction
{   cout << "----------"<<endl ;
    cout << "nbre=" << *nbre << endl;
     *nbre = *nbre + 2;
    cout << "nbre=" <<* nbre<<endl ;
    cout << "----------"<<endl ;
}
```

VIII.6 Fonctions inline

Parfois le temps d'exécution d'une fonction est petit comparé au temps nécessaire pour appeler la fonction. Le mot clé **`inline`** informe le compilateur qu'un appel à cette fonction peut être remplacé par le corps de la fonction.

```
Syntaxe : inline type fonct(liste_des_arguments){...}
```

Exemple

```
inline int max(int a, int b)
{
   return (a < b) ? b : a;
}
void main()
{     int m = max(134, 876);
      cout<< "m=" << m<<endl ;
      return 0 ;
}
```

Les fonctions "***inline***" permettent de gagner au niveau de temps d'exécution, mais augmente la taille des programmes en mémoire.

CHAPITRE III Classes

I Généralités sur les classes

I.1 Définition

La notion de classe généralise la notion de type. La classe comporte des champs (données) et des fonctions appelées méthodes. L'accès à ces attributs est limité par un certain nombre de mot-clé dont le principal est la visibilité(***encapsulation***). En programmation orientée objet pure, les données sont encapsulées et leur accès ne peut se faire que par le biais des méthodes.

Une classe est un type dont une valeur est appelée ***objet***.

Pour créer un objet, on utilise une méthode particulière appelée ***constructeur*** (on peut y avoir plusieurs dans une seule classe).

Une méthode est automatiquement appelée lors de la destruction d'un objet. C'est le ***destructeur***. Par exemple il désalloue des ressources. Contrairement au constructeur il ne peut y avoir qu'un seul destructeur par classe.

I.2 Encapsulation

L'encapsulation est un puissant concept qui permet de maîtriser la visibilité qu'on a des membres d'une classe vue de l'extérieur de la classe. L'utilisateur d'une classe, (c'est à dire un de ceux qui écrit du code qui instancie des objets de la classe), n'a donc accès qu'à un nombre restreint d'attributs et de méthodes. Cela réduit d'autant la complexité d'utilisation des objets.

Chaque attribut et méthode a un niveaux de visibilité parmi :

- Le niveau de visibilité **publique** : L'attribut (resp. méthode) peut être lu et écrit (resp. appelée) partout dans le code, à partir d'une instance de la classe.
- Le niveau de visibilité **privé** : L'attribut (resp. méthode) peut être lu et écrit (resp. appelée) seulement dans le code des méthode de la classe.
- Le niveau de visibilité **protégé** : L'attribut (resp. méthode) peut être lu et écrit (resp. appelée) seulement dans le code des méthode de la classe et des méthodes des classes dérivées.

Notez :

- par défaut les membres d'une classe déclarée avec le mot clé ***`class`*** sont privés.
- on peut déclarer une classe avec le mot clé ***`struct`***. Dans ce cas les membres sont par défaut publiques.

I.3 Déclaration/définition et utilisation d'une classe

En C++, la programmation d'une classe se fait en trois phases : déclaration, définition, utilisation. Il est recommandé de mettre :

- la première lettre du nom de la classe en majuscule
- la liste des membres publics en premier
- les noms des méthodes en minuscules
- le caractère _ comme premier caractère du nom d'une donnée membre

I.3.1 Déclaration

C'est la partie interface de la classe. Elle se fait dans un fichier dont le nom se termine par ***.h*** Ce fichier se présente de la façon suivante :

```
class Maclasse
{     public:
         //déclarations des données et fonctions-membres publiques
      protected :
         //déclarations des données et fonctions-membres protégées
      private:
        //déclarations des données et fonctions-membres privées
};
```

I.3.2 Définition et utilisation

C'est la partie implémentation de la classe. Elle se fait dans un fichier dont le nom se termine par ***.cpp*** Ce fichier contient les définitions des fonctions-membres de la classe, c'est-à-dire le code complet de chaque fonction.

L'utilisation quant à elle se fait dans un fichier d'extension ***.cpp***.

Remarque :

- *Le code des méthodes peut être directement écrit à l'intérieur de la déclaration de la classe, ou à l'extérieur.*
- *On appelle l'opérateur :: l'opérateur de résolution de portée. S'il n'y a pas de nom de classe devant le nom de cet opérateur, on considère que c'est la portée globale, sinon elle se limite à celle de la classe.*
- *Lorsqu'un attribut est privé, on peut ajouter sur celui-ci un get ou un set*
  - *Le **set**... modifie la valeur de l'attribut*
  - *Le **get**... récupère la valeur de l'attribut. On utilise **is**... pour les booléens*
- *Le mot cle **this** est un pointeur, du type de la classe, qui pointe vers l'objet courant*

```
class Point
{
   public:
     double distance(const Point & p);
     double getX();
     double getY();
     void setX(double x);
     void setY(double y);
   private:
        double _x;
        double _y;
};
```

Point.h

```
#include "Point.h"
#include <cmath>
double Point::distance(const Point & p){
    double dx=p._x-this->_x;
    double dy=p._y-this->_y;
    return sqrt(dx*dx + dy*dy);
}
double Point::getX(){return _x;}
double Point::getY(){return _y;};
void Point::setX(double x){
    this->_x=x;
}
void Point::setY(double y){
    this->_y=y;
}
```

Point.cpp

I.4 Surcharge des méthodes et arguments par défaut

La surcharge est la possibilité de donner le même nom à des fonctions ou des opérateurs différents. Elle est le plus souvent utilisée dans le but d'améliorer la lisibilité des programmes. Le principe est celui de la section [CHAPITRE II VIII.2 ci-dessus] et [CHAPITRE II VIII.4 ci-dessus].

En effet, l'identification d'une fonction en C++ se fait par son nom, le nombre de ses arguments et le type de ses arguments d'entrée. Par exemple, ***int somme(int,int);*** et ***int somme(int,int,int)***; représentent deux fonctions distinctes.

II Les constructeurs

II.1 Constructeurs : Méthodes d'initialisation d'un objet

Un constructeur est une fonction membre qui porte comme nom le nom de la classe et qui ne retourne rien. Un constructeur peut avoir zéro ou plusieurs arguments, éventuellement avec valeurs par défaut. Il peut y avoir plusieurs constructeurs dans la même classe (surcharge) dans la mesure où ils ont des signatures différentes.

On peut ainsi ajouter les constructeurs a notre classe Point.

```
class Point
{
  public:
    Point();
    Point(double);
    Point(double, double );
    double distance(const Point & p);
    double getX() const ;
    double getY() ;
    void setX(const double x);
    void setY(const double y);
   private:
        double _x;
        double _y;
};
```

Point.h

```
#include "Point.h"
#include <cmath>
Point::Point()
{
   _x=0;
   _y=0;
}
Point::Point(double x,double y)
{
   _x=x;
   _y=y;
}
Point::Point(double x)
{
   _x=x;
   _y=0;
}
//suite initiale
```

Point.cpp

Les objets peuvent être crées de manière statiques ou dynamiques.
De manière statique un objet est crée comme suit :

- ***Type nomObjet(listePar) ;***
- ***Type nomObjet=Type(listePar) ;***

Exemple : Point p, p1(2,3), p2(5) ,p6=Point(4,5);

Les objets dynamiques sont créés à travers les pointeurs. On utilise new pour créer l'instance et delete pour supprimer l'instance. On utilise la notation fléchée pour acceder aux membres d'un pointeur.

Exemple :

```
point *p ,*p2;
p=new point(7,2);
p2=new Point(5,3) ;
double d=p->distance(*p2); //ou (*p).distance(*p2);
delete p;
```

II.2 Constructeur de recopie

L'initialiseur d'un objet peut être d'un type quelconque, en particulier, il peut s'agir du type de l'objet lui-même.

Exemple :

Point p1(12); // création d'un Point d'abscisse 12

Il est possible de déclarer un nouvel objet 'p3' tel que :

Point p3=p1; // écriture équivalente à : Point p3(p1);

Cette situation est traitée par C++, selon qu'il existe un constructeur ou il n'existe pas de constructeur correspondant à ce cas.

II.2.1 Il n'existe pas de constructeur approprié

Cela signifie que, dans la classe 'Point', il n'existe aucun constructeur à un seul argument de type 'Point'. Dans ce cas, C++ considère que l'on souhaite initialiser l'objet 'p3' après sa déclaration avec les valeurs de l'objet 'p1'. Le compilateur va donc mettre en place une recopie de ces valeurs.

II.2.2 Il existe un constructeur approprié

Cela signifie qu'il doit exister un constructeur de la forme : '***Point (Point &);***'. Dans ce cas, ce constructeur est appelé de manière habituelle, après la création de l'objet, sans aucune recopie.

Remarque

C++ impose au constructeur en question que son unique argument soit transmis par référence. La forme 'Point (Point) ;' serait rejetée par le compilateur.

III Tableau d'objets

De la même manière que C peut définir des tableaux de variables de même type, C++ autorise l'utilisation de tableaux d'objets de même type.

Un tableau d'objet est déclaré sous la forme : ***Type nomObjet[taille]***

Exemple

```
Point sommet[3];
Sommet[0]=Point(2,3) ;
```

Contrairement au C, on peut aussi créer dynamiquement des tableaux d'objets. La taille du tableau peut être spécifiée dynamiquement.

Ceci est possible grâce à l'opérateur new[]. De même les tableaux dynamiques sont désalloués avec l'opérateur delete[]. Ces opérateurs appellent les constructeurs et destructeurs de la même façon que lors de la création des tableaux d'objets non dynamiques.

Exemple

```
Point * sommet;
sommet=new Point[8];
*sommet= Point(2,3);
*(sommet+1)=Point(5,8);
*(sommet+2)=Point(9,2);
for(int i=0;i<=2;i++){
   Point s=*(sommet+i);
   cout<<"\n Point au Sommet"<<i<<": x=" <<s.getX()<<"\t y="<<s.getY();
}
```

IV Surcharge des opérateurs

IV.1 Surcharge des opérateurs en interne

Le langage C++ étend la notion de surcharge aux opérateurs du langage. Concrètement on peut par exemple définir une classe Fraction ou Complexe, et surcharger les opérateurs +,-,*,\ afin d'étendre ces opérations au type concerné.

A titre d'exemple si nous souhaitons surcharger l'opérateur +, nous déclarons la fonction-membre publique suivante :

Complexe operator + (Complexe u);

Si on considère que la soustraction de deux points consiste à soustraire les abscisses et ordonnées, de même pour l'addition, on écrira le code.

```
Point Point::operator-(Point u)
{
    return Point(_x - u._x, _y - u._y);
}
Point Point::operator+(Point u)
{
    return Point(_x + u._x, _y + u._y);
}
```

Par la suite, si z1 et z2 sont deux variables de type Complexe, on pourra écrire tout simplement l'expression z1+z2 pour désigner le nombre complexe obtenu en additionnant z2 et z1, sachant que cette expression est équivalente à l'expression ***z1.operator+(z2).***

IV.2 Utiliser les opérateurs >> et <<

Il est possible de redéfinir les opérateurs de lecture et saisie. A titre d'exemple :

- Pour pouvoir saisir un Point au clavier, on pourrait écrire tout simplement cin >> A; où A est une instance de la classe Point.
- Pour écrire un Point à l'écran, on peut écrire tout simplement cout << B.

Pour ce faire, on redéfinit les opérateurs operator >> et operator <<.

```
class Point
{
  public:

    void operator>>(ostream &out);
    void operator<<(istream &in);


  private:
        double _x;
        double _y;

};
```

```
#include "Point.h"
#include <cmath>
void Point::operator>>(ostream &out)
{
    out << "( " <<_x << "," <<_y<< ") " <<endl;
}
void Point::operator<<(istream &in)
{
    cout << "Tapez l'abscisse : "; in >>_x;
    cout << "Tapez l'ordonnée : "; in >>_y;
}
```

On peut directement saisir les coordonnées d'un point par ***A<<cin*** ou l'afficher à l'écran par ***A>>cout***.

Certains préfèrent toutefois écrire de manière plus usuelle : ***cout<<A***. Ils argumentent parfois en disant qu'en plus cela permet d'enchaîner les affichages en écrivant : ***cout<<A<<endl<<B;***

Il faudrait donc normalement définir une méthode operator<<(const Point &A) sur la classe ostream. Or la classe iostream est déjà écrite !

La solution consiste à utiliser les fonctions amies

V Fonctions amies

En principe, l'encapsulation interdit à une fonction membre d'une classe ou toute fonction d'accéder à des données privées d'une autre classe.

Mais grâce à la notion d'amitié entre fonction et classe, il est possible, lors de la définition de cette classe d'y déclarer une ou plusieurs fonctions (extérieurs de la classe) amies de cette classe.

Une telle déclaration d'amitié les autorise alors à accéder aux données privées, au même titre que n'importe quelle fonction membre.

Il existe plusieurs situations d'amitiés :

- Fonction indépendante, amie d'une classe.
- Fonction membre d'une classe, amie d'une autre classe.
- Fonction amie de plusieurs classes.
- Toutes les fonctions membres d'une classe, amies d'une autre classe.

Pour déclarer une fonction amie d'une classe, il suffit de la déclarer dans cette classe en la précédent par le mot clé '***friend***'.

Les fonctions amies sont utilisées pour la surcharge externe des opérateurs ***a op b*** (op est un opérateur) est représentée par la fonction amie :

```
friend typeRetour operator op (const T1 a, const T2 b);
```

Nous allons utiliser ce principe pour l'affichage et la saisie des données

```
class Point
{
      int x,y;
      public :
      Point(int a=0,int b=0) {x=a; y=b;}
      friend   int  coincide(Point,Point);
      friend ostream & operator<<(ostream &out, const Point &P)
      friend istream & operator>>(istream &in,Point &P)
};
//-------------------------------------
int  coincide(Point  p1,Point p2)
{
      if(p1.x==p2.x && p1.y==p2.y) return 1; else return 0;
}
ostream & operator<<(ostream &out, const Point &P)
{
    out << "( " <<P._x << ", " <<P._y<< ") " <<endl;
    return out;
}
istream & operator>>(istream &in,Point &P)
{
    cout << "Tapez l'abscisse : "; in >>P._x;
    cout << "Tapez l'ordonnée : "; in >>P._y;
    return in;
}
//----------------------
main()
{
      Point pt1 ;
      cin>>pt1;
      cout<<pt1;
      Point o1(15,2), o2(15,2), o3(13,25);
      if(coincide(o1,o2)) cout<<"les objets coïncident\n";
      else cout<<"les objets sont différents\n";
      if(coincide(o1,o3)) cout<<"les objets coïncident\n";
      else cout<<"les objets sont différents\n";
}
```

VI Attributs et méthodes statiques

VI.1 Attributs statiques

De manière générale chaque attribut d'une classe possède sa propre instance (attributs de la classe), cependant certains attributs d'une classe gardent une valeur unique quelque soit l'instance. Ces attributs sont des attributs de classe ou static.

Un membre donné déclaré avec l'attribut ***static*** :

- Est partagé par tous les objets de la même classe. Il existe même lorsque aucun objet de cette classe n'a été créé. Il existe par conséquent en un seul exemplaire.

- Doit être initialisé ***explicitement***, à l'extérieur de la classe (généralement dans le fichier .cpp), en utilisant l'opérateur de résolution de portée (::) pour spécifier sa classe.
- Est accessible de l'extérieur de la classe en utilisant l'opérateur de résolution de portée (::) pour spécifier sa classe.

Exemple : Nous allons ajouter à la classe Point un attribut nbInstance qui compte le nombre d'instances de la classe crée.

```
class Point
{
  public:
    Point();
    Point(double);
    Point(double, double );


   private:
        double _x;
        double _y;
        static int nbInstances ;
};
```

```
#include "Point.h"
#include <cmath>
int Point::nbInstances=0; //initialisation
Point::Point()
{  _x=0;
   _y=0;
    nbInstances++;
}
Point::Point(double x,double y)
{  _x=x;
   _y=y;
   nbInstances++;
}
Point::Point(double x)
{  _x=x;
   _y=0;
   nbInstances++;
}
//suite initiale
```

VI.2 Méthodes statiques

Lorsqu'une fonction membre a une action indépendante d'un quelconque objet de sa classe, on peut la déclarer avec l'attribut static.

Dans ce cas, une telle fonction peut être appelée, sans mentionner d'objet particulier, en préfixant simplement son nom du nom de la classe concernée, suivi de l'opérateur de résolution de portée (::).

Les fonctions membre statiques :

- Ne peuvent pas accéder aux attributs de la classe car il est possible qu'aucun objet de cette classe n'ait été créé.
- Peuvent accéder aux membres données statiques car ceux-ci existent même lorsque aucun objet de cette classe n'a été créé.

On va utiliser une fonction membre statique ***decompte()*** pour connaître le nombre d'objets Point existants à un instant donné.

```
class Point
{
  public:
    //contenu initial de la classe

      static int decompte()
   private:
        static int nbInstances ;

};
```

```
#include "Point.h"
#include <cmath>
int Point::nbInstances=0; //initialisation
int Point::decompte()
{
   return Point::nbInstances;
}
//suite initiale
```

VII Liste d'initialisation

Un meilleur moyen d'affecter des valeurs aux données membres de la classe lors de la construction est la liste d'initialisation. On va utiliser cette technique pour définir le constructeur de la classe Point : Point::Point(double x/*=0.*/, double y/*=0.*/) : x(x), y(y) { }

La liste d'initialisation permet d'utiliser le constructeur de chaque donnée membre, et ainsi d'éviter une affectation après coup.

La liste d'initialisation doit être utilisée pour certains objets qui ne peuvent pas être construits par défaut : c'est le cas des références et des objets constants.

```
Point::Point():_x(0),_y(0)
{
}
Point::Point(double x, double y):_x(x),_y(y)
{
}
Point::Point(double x):_x(x),_y(0)
{

}
```

Travaux Dirigés

Exercice 1 :

Écrire un programme utilisant une classe Rectangle dont le constructeur prend deux paramètres, largeur et hauteur et qui offre les fonctions suivantes avec cpp

- calcul du périmètre
- calcul de la surface
- affichage

Ainsi que les accesseurs et mutateurs triviaux (lecture et modification de la largeur et de la hauteur).

Exercice 2 :

On souhaite implémenter une classe représentant un compteur entier. Un tel objet se caractérise par :

- Une valeur entière, positive ou nulle, nulle à l'origine.
- Le fait qu'il ne peut varier que par pas de 1 (incrémentation ou décrémentation). On convient qu'une décrémentation d'un compteur nul est sans effet.

Il s'agit de créer une classe Compteur pour rendre le service demandé. On écrira en outre un petit programme de Test qui :

- créera un compteur et affichera sa valeur;
- l'incrémentera 10 fois, puis affichera à nouveau sa valeur;
- le décrémentera 20 fois, puis affichera une troisième fois sa valeur

La sortie de ce programme doit donner (quelque chose comme) "0 10 0"

Exercice 3 :

Une pile est un ensemble dynamique d'éléments où le retrait se fait d'une façon particulière. En effet, lorsque l'on désire enlever un élément de l'ensemble, ce sera toujours le dernier inséré qui sera retiré. Un objet pile doit ré pondre aux fonctions suivantes :

- ✓ Initialiser une pile
- ✓ Empiler un élément sur la pile (push)
- ✓ Dépiler un élément de la pile (pop)
- ✓ La redéfinition de l'opérateur d'affichage

Nous allons supposer que les éléments à empiler sont de type int.

Le programme main comprend la définition d'une classe Pile et un programme de test qui crée deux piles p1 et p2, empile dessus des valeurs entières et les dépiler pour vérifier les opérations push et pop.

Exrcice 4 :

Réaliser **une classe Fraction** permettant de manipuler les Fractions . On prévoira :

- Une Fraction est définit par son numérateur et son dénominateur (des membres privés) ;
- Le numérateur doit être non nul
- Des constructeur (implémenter les trois types de constructeur). Le dénominateur et le numérateur ont la valeur 0 par défaut.
- Une méthode de simplification ;
- La redéfinition des opérateurs(+,-,*,/,==) avec les types (int, double, Fraction);

on écrira séparément:

- un ficher source constituant la déclaration et la définition de chaque classe.
- un petit programme d'essai (main) effectuant des tests..

Exercice 5 :

Réaliser **une classe Point** permettant de manipuler un point d'un plan. On prévoira :

- un point est définit par ses coordonnées x et y (des membres privés)
- un constructeur (vous pouvez également implémenter les trois types de constructeur)
- une fonction membre déplace effectuant une translation définie par ses deux arguments dx et dy (double)
- La redéfinition des fonctions de saisie et d'affichage.
- une fonction membre distance effectuant calculant la distance entre deux point.
- une fonction membre milieu donnant le milieu d'un segment.

Réaliser **une classe Segment** permettant de manipuler les segments de droite. On prévoira :

- un Segment est définit par son orignine(O) et sa destination(B) (des membres privés de type Point)
- un constructeur (vous pouvez également implémenter les trois types de constructeur)
- une fonction membre distance effectuant calculant la distance entre deux point.
- une fonction membre milieu donnant le milieu d'un segment.
- Une fonction membre appartient qui indique si un Point p appartient au segment de droite.

on écrira séparément:

- un ficher source constituant la déclaration et la définition de chaque classe.
- un petit programme d'essai (main) gérant la classe Point et Segment de droite.

Exercice 6 :

L'objectif de cet exercice est de gérer les notes des étudiants d'une institution à l'aide d'une classe C++ **Etudiant** définie par :

- Les attributs suivants :

  - ✓ **matricule**: l'identifiant de l'étudiant **(auto incrémenté)**
  - ✓ **nom**: nom d'un étudiant
  - ✓ **nbrNotes**: le nombre de notes de l'étudiant
  - ✓ ***tabNotes**: tableau contenant les notes d'un étudiant (**allocation dynamique**).

- Les méthodes suivantes :

  - ✓ Un constructeur d'initialisation
  - ✓ Un constructeur avec arguments
  - ✓ Un destructeur ~**Etudiant** ()
  - ✓ Un constructeur de recopie **Etudiant** (**const Etudiant &**)
  - ✓ Les **getters** et **setters**
  - ✓ **void saisie** () : permettant la saisie des note d'un étudiant
  - ✓ **void affichage** () : permettant l'affichage des informations d'un étudiant
  - ✓ **float moyenne** () : retourne comme résultat la moyenne des notes de l'étudiant.

- ✓ **bool admis** () : retourne comme résultat la valeur ***true***, si un étudiant est admis et la valeur ***false***, sinon. Un étudiant est considéré comme étant admis lorsque la moyenne de ses notes est supérieure ou égale à 10.
- ✓ **bool comparer():** qui compare la moyenne des deux étudiants, retourne comme résultat la valeur true, si deux étudiants ont la même moyenne et la valeur false, sinon.

Exercice 7 :

Un Article est caractérisé par les attributs : référence, désignation, stock, PrixHT, tauxTva.
Ces attributs doivent seulement être accessibles par le biais des accesseurs (get / set). Le taux de TVA est le même pour tous les articles.

1. Créer la classe Article munies des méthodes ci-dessous :
   - Les constructeurs suivants :
     - ✓ Un constructeur par défaut
     - ✓ Un constructeur initialisant tous les attributs.
     - ✓ Un Constructeur qui permet de renseigner la référence et la désignation lors de l'instanciation
     - ✓ Un constructeur de recopie
   - Les accesseurs
   - Le stock est initialisé a zero lors de la création d'un article. Sa valeur est accessible uniquement en lecture.
   - Une méthode stocker(int qte, double prix) qui permet de mettre à jour le stock. La mise à jour du stock se fait en CMUP. Le prix est le prix HT.
   - Une méthode destocker(int qte) qui permet de déstocker la quantité ;
   - La méthode prixTTC() qui calcule le prix TTC d'un article qui équivaut à : PrixHT + (PrixHT*TauxTVA/100) ;
   - La methode valeurStock qui retournera la montant total du stock.
2. Créer un programme de Test où il faut créer des objets (en utilisant les différents constructeurs) et leur calculer le prix TTC.

CHAPITRE IV HERITAGE ET DERIVATION

Il est question dans ce chapitre de présenter les concepts de généralisation/Spécification (héritage, méthode virtuelle, polymorphisme, classes abstraites) et leurs mises en œuvre en C++.

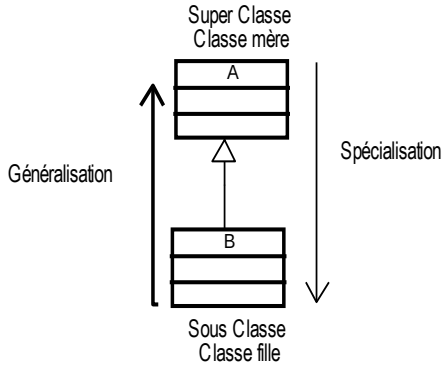
I Définition et représentation

I.1 Définition

L'héritage est un concept fondamental de la programmation orientée objet. Elle se nomme ainsi car le principe est en quelque sorte le même que celui d'un arbre généalogique. Ce principe est fondé sur des classes « filles » qui héritent des caractéristiques des classes « mères ».

L'héritage permet d'ajouter des propriétés à une classe existante pour en obtenir une nouvelle plus précise. Il permet donc la spécialisation ou la dérivation de types.

La modélisation de l'héritage est :



- ✓ On dit que B hérite de A ou encore B est un A ;
- ✓ Toute instance de B est une instance de A
- ✓ Tous les attributs de A sont des attributs de B ;
- ✓ Toutes les méthodes de A sont des méthodes de B ;
- ✓ L'héritage est transitif ;
- ✓ L'héritage peut être simple ou multiple ;
- ✓ En c++ une classe peut hériter de plusieurs classes.

I.2 Représentation

Pour indiquer qu'une classe B hérite de la classe A on utilise

```
class B :<mode>A {


};
```

<mode> optionnel permet d'indiquer la nature de l'héritage: Si aucun mode n'est indiqué alors l'héritage est privé. Il est possible de :

- ✓ D'ajouter des caractéristiques (attributs) et/ou des comportements(méthodes) propres ;
- ✓ D'utiliser les caractéristiques héritées ;
- ✓ De redéfinir les comportements (méthodes héritées).

- Il ne faut pas confondre la redéfinition et surcharge :
  - ✓ Une ***surcharge*** permet d'utiliser plusieurs méthodes qui portent le même nom au sein d'une même classe avec des signatures différentes.
  - ✓ Une ***redéfinition*** permet de fournir une nouvelle définition d'une méthode d'une classe ascendante pour la remplacer. Elle doit avoir une signature rigoureusement identique à la méthode parente.
- Les constructeurs ne s'héritent pas. Le constructeur d'une sous classe peut s'appuyer sur le constructeur de la classe de base, en y faisant appel . L'appel au constructeur de la classe mère doit être le premier dans la liste d'initialisation :

- ✓ B::B() : A() pour l'appel au constructeur de la classe mère A sans paramètre.
- ✓ B::B(int a, int b, int c) : A(a,b) pour l'appel au constructeur de la classe mère A avec 2 paramètres.
- ✓ Si aucun appel n'est explicité, le compilateur ajoute un appel implicite au constructeur par défaut s'il existe.

- Une ***classe dérivée B***, peut appeler la ***version f(…)*** d'une méthode de la classe de base A , en utilisant l'instruction ***A ::f(…)***
- Il existe 3 types(modes) d'héritages ou dérivations :

Mode de dérivation	Statut dans la classe de base	Statut dans la classe dérivée
Public	public	Public
	protected	Protected
	private	inaccessible
Protected	public	Protected
	protected	Protected
	private	inaccessible
Private	public	Private
	protected	Private
	private	inaccessible

On peut par exemple écrire

```
class B :protected A {



};
```

Les éléments publics dans A seront protégés dans B

I.3 Mise en œuvre de l'héritage simple

I.3.1 Les classes Point et Pixel

On reprend la classe Point précédente

```
class Point
{
    public:
      Point();
      Point(double);
      Point(double x, double y );
      string toString();
      virtual ~Point();
      double distance(const Point &);
      double getX() const ;
      double getY() ;
      void setX(const double x);
      void setY(const double y);
    private:
      double _x;
      double _y;
};
```

Point.h

```
#include "Point.h"
#include <cmath>
#include <sstream>
Point::Point():_x(0),_y(0){}
Point::Point(double x,double y):_x(x),_y(y){}
Point::Point(double x):_x(x),_y(0){}
string Point::toString()
{    stringstream oss;
    oss << "( " <<  _x << ", " <<_y<< ") ";
    return oss.str();// retourne (x,y)
}
Point::~Point(){}
double Point::distance(const Point & p){
    double dx=p.x-this->_x;
    double dy=p.y-this->_y;
    return sqrt(dx*dx + dy*dy);
}
double Point::getX(){return _x;}
double Point::getY(){return _y;};
void Point::setX(double x){ this->_x=x;}
void Point::setY(double y){ this->_y=y;}
```

Point.cpp

Un Pixel est un Point munie d'une couleur.

```
#include "Point.h"
class Pixel: public Point
{
  public:
    Pixel();
    Pixel(double,double,double);
    Pixel(double, double);
    Pixel(double);
    virtual ~Pixel();
    string toString();
    int getCouleur();
    void setCouleur(const int);
  private:
    int _couleur;
};
```

Pixel.h

```
#include "Pixel.h"
#include <sstream>
Pixel::Pixel():Point(0,0),_couleur(0){}
Pixel::Pixel(double x, double y, double
color):Point(x,y),_couleur(color){ }
Pixel::Pixel(double x, double y) : Point(x,y),
_couleur(0){ }
Pixel::Pixel(double x):Point(x,0),_couleur(0){}
int Pixel::getCouleur(){ return _couleur;}
void Pixel::setCouleur(const int couleur)
{  _couleur=couleur;}
string Pixel::toString()
{  stringstream oss;
   oss <<Point::toString();//version parent
   oss<< "Couleur : " <<  _couleur;
   return oss.str();
}
Pixel::~Pixel(){  }
```

Pixel.cpp

Commentaires :

- ***class Pixel: Point*** : Indique que la classe Pixel hérite de la classe Point. Le mode de dérivation est public ;
- Dans le constructeur ***Pixel::Pixel():Point(0,0),_couleur(0){}*** l'instruction Point(0,0) fait appel au constructeur de la classe Point(parent) ;
- L'instruction ***Point::toString()*** fait appel à la méthode toString de la classe Point(c'est un peu l'équivalent du super.toString()).

La classe de Test

```
#include <iostream>
#include <Pixel.h>
using namespace std;
int main()
{
    Point pt(3,7);
    Pixel px(3,2,8);
    cout<<px.toString()<<"\n";
    double d=pt.distance(px);
    cout<<"Distance :"<< d <<"\n";
    Point ps=Pixel(3,5,6);
    pt=px ;
    cout<<ps.toString()<<"\n";
    return 0;
}
```

Résultat :

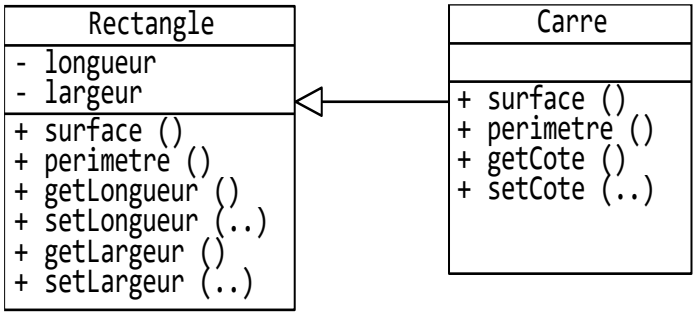
```
( 3, 2) Couleur : 8
Distance :5
( 3, 5)
```

- *Le* ***upcast***(changer un type vers son type de base) implicite est permis. Ceci explique l'affectation pt=px(un Pixel est un Point) ou encore *Point ps=Pixel(3,5,6);*
- *La conversion d'un pointeur sur un objet d'une classe générale vers un objet d'une classe spécialisée (downcast)se fait en forcant la conversion en utilisant les opérateurs de transtypage.*

Exercice : Expliquez le résultat obtenu.

I.3.2 Mise en œuvre de modes de dérivation

On considère la hiérarchie suivante :



Réalisons cette hiérarchie en utilisant le c++.

Commençons par la classe Rectangle.

```
class Rectangle
{
  public:
    Rectangle();
    Rectangle(double, double);
    Rectangle(double lg);
    double surface();
    double perimetre();
    void setLongueur(double);
    void setLargeur(double);
    double getLongueur();
    double getLargeur();
    virtual ~Rectangle();
   private:
    double longueur;
    double largeur;
};
```

Rectangle.h

```
#include "Rectangle.h"
Rectangle::Rectangle():longueur(0),largeur(0){  }
Rectangle::Rectangle(double    lg,    double    larg):
longueur(lg),largeur(larg){    }
Rectangle::Rectangle(double l):longueur(l),largeur(l)
{ }
Rectangle::~Rectangle(){    //dtor}
double Rectangle::perimetre()
{   return (longueur+largeur)*2; }
double Rectangle::surface()
{    return (longueur*largeur);   }
void Rectangle::setLongueur(double l){longueur=l;}
double Rectangle::getLongueur(){return longueur;}
void Rectangle::setLargeur(double l){largeur=l;}
double Rectangle::getLargeur(){return largeur;}
```

Rectangle.cpp

```
#include <Rectangle.h>
class Carre:private Rectangle
{
    public:
        Carre(double cote);
        Carre();
        virtual ~Carre();
        void setCote(double);
        double getCote();
        double surface();
        double perimetre();
};
```

Carre.h

```
#include "Carre.h"
Carre::Carre():Rectangle(){   }
Carre::Carre(double cote):Rectangle(cote,cote){   }
Carre::~Carre(){   }
void Carre::setCote(double cote)
{  setLongueur(cote) ;
   setLargeur(cote) ;
}
double Carre::surface(){return Rectangle::surface();}
double                      Carre::perimetre(){return
Rectangle::perimetre();}
double Carre::getCote(){return getLongueur();}
```

Carre.cpp

Commentaires :

Le mode de dérivation privée implique :

- Toutes les méthodes/attributs public et protected de la classe Rectangle sont privées dans Rectangle et donc inaccessibles hors de la classe. Les méthodes ***setLongueur, setLargeur, getLongueur, getLargeur, surface() , perimetre()*** sont donc inutilisable hors de la classe Carre. D'où la nécessité de redéfinir les méthodes surface() et perimetre() ;
- Toutes les méthodes/attributs private de la classe Rectangle sont inaccessible dans la classe Rectangle

II Méthodes virtuelles

II.1 Généralités : fonctions virtuelle

Reprenons le cas des classes Point et Pixel précédent, le bout de code ci-dessous et le résultat de l'exécution

```
Point p(3,7);
Pixel px(3,2,8);
Point px1=Pixel(7,5,9);
cout<<"p:" << p.toString()<<"\n";
cout<<"px:" <<px.toString()<<"\n";
cout<<"px1 :" <<px1.toString();
```

Résultat :

```
p :( 3, 7)
px:( 3, 2) Couleur : 8
px1:( 7, 5)
```

Constat :

- Bien que px1 soit initialisé avec le constructeur de Pixel, c'est la méthode ***toString()*** de la classe Point(qui est le type statique) qui est appelée ;
- L'affichage px1 : (7,5) Couleur :9 aurait été indiqué pour px1.

Par défaut les méthodes en c++ ne sont pas polymorphes.

Une fonction polymorphe est une méthode qui est appelée en fonction du type d'objet et non en fonction du type de pointeur utilisé.

Pour obtenir un comportement polymorphe, on doit :

- Déclarer la fonction virtuelle(virtual)
- Utiliser la méthode à partir d'une variable dynamique.

A titre d'exemple, pour la classe Point, on effectue les modifications suivantes :

- Dans la classe Point, on modifie la methode ***`string toString()`*** en ***`virtual string toString()`***
- On déclare px1 comme une variable dynamique.

```
Point *px1=new Pixel(7,5,9);
cout<<"px1 :" <<(*px1).toString();
```

Résultat :

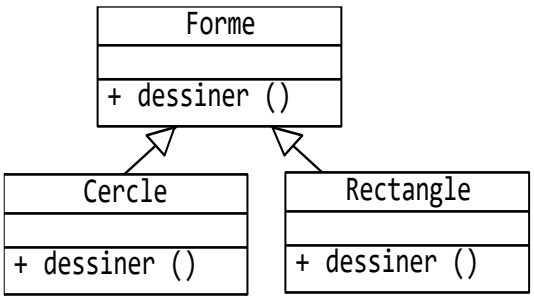
```
px1:( 7, 5) Couleur : 9
```

Cette fois c'est bien la méthode toString() de la classe Pixel(type dynamique de px1) qui a été appelée.

II.2 Méthodes virtuelles pures, classes abstraites et interface

Une méthode dont on connait la déclaration(signature) mais dont on ne peut écrire le code est une ***méthode abstraite.***

A titre d'exemple, dans la hiérarchie des Classes ci-contre, le code de la méthode dessiner ne peut être défini dans la classe Forme.



En C++une méthode abstraite est appelée ***fonction virtuelle pure***.

Une fonction virtuelle pure se déclare en remplaçant le corps de la fonction par les symboles = 0, comme suit :

```
virtual TypeRetout nomFonction (liste des arguments)=0; …
```

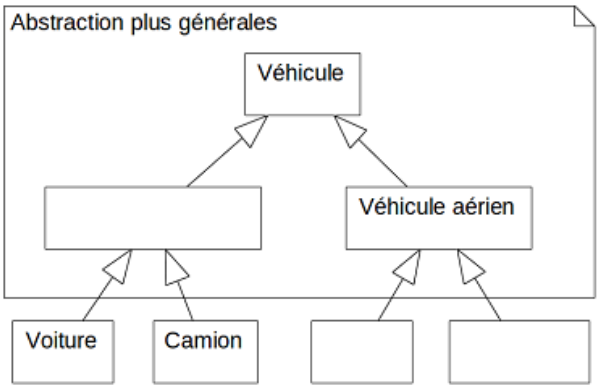
Remarque:

- Une classe comportant au moins une fonction virtuelle pure est considérée comme abstraite ***et il n'est plus possible de déclarer des objets de son type***.
- ***Une fonction déclarée virtuelle pure dans une classe de base doit obligatoirement être redéfinie dans une classe dérivée ou déclarée à nouveau virtuelle pure*** ; dans ce dernier cas, la classe dérivée est aussi abstraite.

TRAVAUX DIRIGES

Question de cours :

1. Qu'est-ce que l'héritage multiple ? Est-ce qu'il est possible en C++ ?
2. Qu'est-ce qu'une surcharge (overloading) ?
3. Qu'est-ce qu'une redéfinition (overriding) ?
4. Lorsqu'on hérite d'une classe, où doit-on appeler le constructeur de la classe parente ?
5. Comment implémente-t-on le polymorphisme en C++ ?
6. Dans quel cas une classe est dite abstraite en C++ ?
7. Compléter avec des noms de classe le diagramme ci-dessous ?



8. Qu'est-ce que le transtypage « ascendant » (upcast) ? Pose-t-il un problème particulier ?
9. Qu'est-ce que le transtypage « descendant » (downcast) ? Quel opérateur doit-on utiliser pour réaliser ce transtypage ?

Exercice 1 :

On prend les classes suivantes : Etudiant, Personne, EtudiantTravailleur, Enseignant, EtudiantSportif et Travailleur .

1. dessinez une arborescence cohérente pour ces classes en la justifiant,
2. où se situeront les champs suivants : salaire , emploiDuTemps, anneDEtude, nom, age et sportPratique

Exercice 2 :

Ecrivez une classe Bâtiment avec les attributs suivants:

- adresse

La classe Bâtiment doit disposer des constructeurs suivants:

- Batiment(),
- Batiment (adresse).

La classe Bâtiment doit contenir des accesseurs et mutateurs (ou propriétés) pour les différents attributs.
La classe Bâtiment doit contenir une méthode toString () donnant une représentation du Bâtiment.
Ecrivez une classe Maison héritant de Bâtiment avec les attributs suivants:

- NbPieces: Le nombre de pièces de la maison.

La classe Maison doit disposer des constructeurs suivants:

- Maison(),
- Maison(adresse, nbPieces).

La classe Maison doit contenir des accesseurs et mutateurs (ou des propriétés) pour les différents attributs. La classe Maison doit contenir une méthode toString () donnant une représentation de la Maison.

Ecrivez aussi un programme afin de tester ces deux classes

~~*Cours de Programmation Orientée Ob*~~

Exercice 3

- Un compte bancaire possède à tout moment une donnée : son solde. Ce solde peut être positif (compte créditeur) ou négatif (compte débiteur).
- Chaque compte est caractérisé par un code incrémenté automatiquement.
- le code et le solde d'un compte sont accessibles en lecture seulement.
- A sa création, un compte bancaire a un solde nul et un code incrémenté.
- Il est aussi possible de créer un compte en précisant son solde initial.
- Utiliser son compte consiste à pouvoir y faire des dépôts et des retraits et des virements. Pour les deux premières opérations, il faut connaître le montant de l'opération, et pour la dernière il faut connaitre en plus le compte de destination.
- L'utilisateur peut aussi consulter les informations de son compte par la méthode toString().
- Un compte Epargne est un compte bancaire qui possède en plus un champ « Taux Intérêt=6 » et une méthode calculIntérêt() qui permet de mettre à jour le solde en tenant compte des intérêts.
- Un ComptePayant est un compte bancaire pour lequel chaque opération de retrait et de versement est payante et vaut 500.

1. Définir la classe Compte.
2. Définir la classe CompteEpargne.
3. Définir la classe ComptePayant.
4. Créer un programme permettant de tester ces classes avec les actions suivantes:
   - Créer une instance de la classe Compte , une autre de la classe CompteEpargne et une instance de la classe ComptePayant
   - Faire appel à la méthode déposer() de chaque instance pour déposer une somme quelconque dans ces comptes.
   - Faire appel à la méthode retirer() de chaque instance pour retirer une somme quelconque de ces comptes.
   - Faire appel à la méthode calculInterêt() du compte Epargne.
   - Afficher le solde des 3 comptes.

Exercice 4 :

Soit la classe abstraite Employé caractérisée par attributs(privées) suivants :

- Matricule
- Nom
- Prénom
- année de naissance

La classe Employé doit disposer des méthodes suivantes :

- un constructeur d'initialisation
- des propriétés pour les différents attributs
- la méthode Tostring
- une méthode abstraite GetSalaire.

Un ouvrier est **un employé** qui se caractérise par son année d'entrée à la société.

- Tous les ouvriers ont une valeur commune appelée SMIG=25000F
- L'ouvrier a un salaire mensuel qui est : Salaire = SMIG + (Ancienneté en année)*100.

Un cadre est **un employé** qui se caractérise par un indice. Le cadre a un salaire qui dépend de son indice:

- ✓ 1 : salaire Mensuel = 100 000
- ✓ 2 : salaire Mensuel = 130 000
- ✓ 3 : salaire Mensuel = 140 000
- ✓ 4 : salaire Mensuel = 170 000

Un patron est **un cadre** qui se caractérise par un chiffre d'affaire et un pourcentage(tx).

- Le chiffre d'affaire est commun entre les patrons.
- Pour calculer le salaire du patron, on calcule le salaire du cadre et on ajoute px% du chiffre d'affaire : Salaire = salaire cadre +CA*pourcentage/100

Travail à faire :

1. Proposer un diagramme de classe
2. Créer la classe abstraite Employé.
3. Créer la classe Ouvrier, la classe Cadre et la classe Patron qui héritent de la classe Employé, et prévoir les Constructeurs et la méthode toString de chacune des 3 classes.
4. Implémenter la méthode getSalaire() qui permet de calculer le salaire pour chacune des classes.

Exercice 5 :

Une Figure est une interface caractéérisée par deux méthodes :

- Surface
- Perimetre

1) Donner le corps de la classe Figure
2) En utilisant la classe Figure ci-dessus donnez le corps des classes ci-dessous qui hérite de la classe Figure :
    a. Carre
    b. Cercle
    c. Rectangle

On considère cette fois la classe récipient. Un récipient est caractérisé par la base(une Figure) et la hauteur(un réel). Un récipient contient une méthode pour calculer le volume(surface de base*hauteur)

3) Donner le corps de la classe Récipient
4) Définissez un programme Test qui crée 03 récipient dont la hauteur est 9 :
    a. Crée un récipient dont la base est un cercle de rayon 5
    b. Crée un récipient dont la base est un Carre de cote 8
    c. Crée un récipient dont la base est un Rectangle de longueur 10 et largeur 7
    d. Affiche le volume de chaque Récipient

CHAPITRE V Template, Gestion des erreurs et collections d'objet

I La généricité : Les Templates

La généricité permet d'écrire du code générique en paramétrant des fonctions et des classes par un type de données. Un module générique n'est alors pas directement utilisable : c'est plutôt un modèle, patron (template) de module qui sera «instancié » par les types de paramètres qu'il accepte. C++ permet, grâce à la notion de patron de fonctions, de définir une famille de fonctions paramétrées par un ou plusieurs types, et éventuellement des expressions. D'une manière comparable, C++ permet de définir des "patrons de classes".

I.1 Patrons de fonctions

Pour illustrer les patrons de fonctions, prenons un exemple concret : une fonction min qui accepte deux paramètres et qui renvoie la plus petite des deux valeurs qui lui est fournie. On désire bénéficier de cette fonction pour certains types simples disponibles en C++ (int, char, float). Les notions que nous avons vu en C++ jusqu'à maintenant ne nous permettent de résoudre ce problème qu'avec une seule solution. Cette solution est d'utiliser la surcharge et de définir trois fonctions min, une pour chacun des types considérés comme ci dessous.

Exemple :

```
int min(int a,int b)
{
   if(a < b){
      return a;
   } else return b;
  //return ((a < b)? a:b);
}
```

```
double min(double a,double b)
{
   if(a < b){
      return a;
   } else return b;
  //return ((a < b)? a:b);
}
```

```
char min(char a,char b)
{
   if(a < b){
      return a;
   } else return b;
  //return ((a < b)? a:b);
}
```

Une fonction template permet de définir une fonction qui réalisera le même traitement pour des types de données différents (type de donnée paramétré qui sera connu au moment de l'appel de la fonction). Sa syntaxe est de la forme :

Syntaxe	Application
template<class typeName> type fonction(arguments) { …… }	***template*** **<class** T> // T est le paramètre de modèle T min(T a,T b) { return ((a < b)? a:b); }

- typeName : représentent les types paramétrés (par convention on prend la lettre T, U …).
- La fonction déclarée peut utiliser ensuite le ou les types fictifs.

Remarque : il n'est donc plus nécessaire de définir une implantation de la fonction min par type de données. On définit donc bien plus qu'une fonction, on définit une méthode permettant d'obtenir une certaine abstraction en s'affranchissant des problèmes de type.

Remarques :

- Il est possible de définir des fonctions template acceptant plusieurs types de données en paramètre. Chaque paramètre désignant une classe est alors précédé du mot-clé class, comme dans l'exemple : template <class T, class U> ....
- Chaque type de données paramètre d'une fonction template doit être utilisé dans la définition de cette fonction.
- Pour que cette fonctionnalité soit disponible, les fonctions génériques doivent être définies au début du programme ou dans des fichiers d'interface (fichiers .h).

I.2 Classe template : patron de classes

Il est possible, comme pour les fonctions, de définir des classes template, c'est-à-dire paramétrées par un type de données. Cette technique évite ainsi de définir plusieurs classes similaires pour décrire un même concept appliqué à plusieurs types de données différents. Elle est largement utilisée pour définir tous les types de containers (comme les listes, les tables, les piles, etc.), mais aussi des algorithmes génériques par exemple.

La syntaxe permettant de définir une classe template est similaire à celle qui permet de définir des fonctions template.

Exemple :

```
class Point{
   public :
      point (int abs=0, int ord=0) ;
      void affiche () ;
   private :
      int x, y;
};
```

Lorsque nous procédons ainsi, nous imposons que les coordonnées d'un point soient de valeurs de type int. Si nous souhaitons disposer de points à coordonnées d'un autre type (float, double, long ...), nous devons définir une autre classe en remplaçant simplement, dans la classe précédente, le mot clé int par le nom de type voulu.

Ici encore, nous pouvons simplifier considérablement les choses en définissant un seul patron de classe de cette façon:

```
template <class T>
class Point {
   public :
      point (T abs=0, T ord=0) ;
      void affiche () ;
   private :
        T x , y ;
};
```

Comme dans le cas des patrons de fonctions, la mention template <class T> précise que l'on a affaire à un patron (template) dans lequel apparaît un paramètre de type nommé T ; La définition de notre patron de classes n'est pas encore complète puisqu'il y manque la définition des fonctions membres, à savoir le constructeur point et la fonction affiche().

Pour ce faire, la démarche va légèrement différer selon que la fonction concernée est en ligne ou non. Voici par exemple comment pourrait être défini notre constructeur en ligne:

Point (T abs=0, T ord=0) { x = abs ; y = ord ; }

En revanche, lorsque la fonction est définie en dehors de la définition de la classe, il est nécessaire de rappeler au compilateur :

- que, dans la définition de cette fonction, vont apparaître des paramètres de type ; pour ce faire, on fournira à nouveau la liste de paramètre sous la forme: ***template <class T>***
- le nom du patron concerné. Par exemple, si nous définissons ainsi la fonction affiche, son nom sera: ***point<T>::affiche ()***
- Tout le code est contenu dans un seul fichier.

En définitive, voici comment se présenterait l'en-tête de la fonction affiche si nous le définissions ainsi en dehors de la classe :

template <class T> void Point<T>::affiche ()

Voici ce que pourrait être finalement la définition de notre patron de classe point :

```
template <class T>
 class Point    // Création d'un patron de classe
{
     public :
         Point (T abs=0, T ord=0) { x = abs ; y = ord ; }
         void affiche () ;
    private :
        T  x , y ;
};
template <class T> void point<T>::affiche () {
            cout << "Paire : " << x << " " << y << "\n" ;
 }
```

L'instanciation de tels patrons est effectuée automatiquement par le compilateur selon les déclarations rencontrées. Après avoir créé ce patron, une déclaration telle que:

Exemple :

Point <int> a; // Point dont les coordonnées sont des int

Point <float> ad ; // Point dont les coordonnées sont des float

Point <float> p(5.2,8.6) ; //

II Gestion des collections d'objets

Lorsqu'on souhaite regrouper différentes valeurs au sein d'un même agrégat, plusieurs choix sont possibles. Les tableaux sont, bien entendu, les plus fréquemment utilisés (Sauf qu'elle n'est pas indiqué). La STL(Standard Template Library) fournit un certain nombre de conteneurs pour gérer des collections d'objets : les tableaux, vector , les listes (list), les ensembles (set), les piles (stack), et beaucoup d'autres ….

Un conteneur (container) est un objet qui contient d'autres objets. Il fournit un moyen de gérer les objets contenus (au minimum ajout, suppression, parfois insertion, tri, recherche, ...) ainsi qu'un accès à ces objets.

II.1 Les méthodes communes aux conteneurs

La plupart des conteneurs implémentent les méthodes suivantes :

- bool **empty**() const; renvoie true si le conteneur n'a pas d'élément.
- size_t **size**() const; renvoie le nombre d'élément dans le conteneur.
- void **resize**(size_t nouvelle_taille, T valeur); Changement de la taille du conteneur. Dans le cas d'un agrandissement, la valeur indiquée (par défaut : la valeur par défaut du type T) est utilisée pour garnir les nouveaux emplacements.
- size_t **capacity**() const; nombre d'éléments que le conteneur peut maintenir sans nécessiter une réallocation.
- void **reserve**(size_t n); modifie la taille de l'espace réservé pour le conteneur (celui dont la taille est donnée par capacity).
- size_t **max_size**() const; nombre maximum d'éléments que le conteneur peut contenir.
- T &**operator[]**(clé);const T &operator[](clé) const;
  T &**at**(clé);const T &at(clé) const;
  Accès indexé (dans le cas des tables associatives (map), clé est du type précisé lors de l'instanciation du modèle ; dans les autres cas, clé d'un type entier).
- T &**front**(); const T &**front**() const;Accès à l'élément qui est en tête.
- T &**back**();const T &**back**() const;Accès à l'élément qui est en queue.
- void **push_front**(const T &); void **push_back**(const T &);
  Ajout d'un élément en tête [resp. en queue].
- void **pop_front**(); void **pop_back**(); Suppression de l'élément en tête [resp. en queue].

II.2 Les itérateurs

Les itérateurs sont des objets spécialement conçus pour faciliter l'accès et les opérations sur les éléments d'un conteneur.

Voici les méthodes courantes implémentées par pratiquement tous les conteneurs pour obtenir des itérateurs :

- itérateur **insert**(itérateur , const T& x);
  void **insert**(itérateur , size_t n, const T& x);
  Insertion d'une valeur **x** [resp. de **n** copies d'une valeur **x**].
- iterator **erase**(itérateur); void **clear**();
  Suppression d'une valeur [resp. de toutes les valeurs].
- itérateur **begin**(); const itérateur **end**() const;
  Itérateur positionné sur le premier [resp. le dernier] élément.
- itérateur **rbegin**(); const itérateur **rbegin**() const;
  itérateur **rend**(); const itérateur **rend**() const;
  Itérateur inverse positionné sur le premier [resp. le dernier] élément.
- itérateur void **swap**(vector<T> &v);
  Echange des valeurs des vecteurs ***this** et **v**.

✓ Lorsqu'un itérateur est valide, c'est à dire qu'il est placé sur un élément d'un conteneur, on peut accéder à l'élément avec les opérateurs * et ->.

✓ L'opérateur ++ préfixé (++i) et postfixé (i++) est défini pour les itérateurs, et place l'itérateur sur l'élément suivant.

✓ Les opérateurs d'inégalité != et d'égalité == sont définis, et permettent de vérifier ou non si deux itérateurs sont placés sur le même élément d'un même conteneur.

II.3 Les conteneurs séquences concrets: list , vector

II.3.1 Présentation

Voici un rapide comparatif de ces 2 types :

	vector	list
Accès direct aux éléments (a[78]…)	Oui en temps constant	non
Insertion/suppression au début	Temps linéaire	Temps constant
Insertion/suppression à la fin	Temps constant	Temps constant
Insertion/suppression entre 2 éléments	Temps linéaire	Temps constant

- ✓ On voit que les listes doivent être utilisées si il y a de nombreuses insertions/suppressions partout dans la séquence.
- ✓ On voit que les vecteurs doivent être utilisés lorsque l'on à besoin d'accès direct aux éléments par leurs positions dans la séquence (vect[54]…), aussi appelé accès aléatoire.

II.3.2 Les vecteurs

Un vector est un conteneur séquentiel qui encapsule les tableaux de taille dynamique. Les éléments sont stockés de façon contigüe.

Voici un exemple classique de parcours d'un conteneur :

```
#include <list>
using namespace std;
void main()
{
	vector<int> v;
	for (int i = 1; i < 10; i++) v.push_back(i);
	int sum =0;
	vector<int>::iterator it;
	for( it = v.begin() ; it != v.end() ; it++ ) sum+= *it;
	// sum = 1+2+...+9 = 45
}
```

- ✓ On aurait pu calculer la même somme en utilisant :

```
for (int i = 0; i < 4; i++) sum+=v[i];
```

II.3.3 Les listes

La classe list fournit une structure générique de listes doublement chaînées (c'est-à-dire que l'on peut parcourir dans les deux sens) pouvant éventuellement contenir des doublons.

Précisons que les listes possèdent des méthodes très spécifiques :

- `void remove(const T& valeur);` pour supprimer une valeur
- `void reverse();`Inversion des éléments d'une liste.

```
#include <iostream>
#include <list>
using namespace std;
int main()
{
   list<int> l; // une liste vide
   l.push_back( 5 );
   l.push_back( 6 );
   l.push_front(4);//insertion en tête
   l.push_back( 1 );
   l.push_back( 10 );
   l.push_back( 4 );
   l.push_back( 5 );
   l.pop_back(); // enleve le dernier élément et supprime l'entier 5
   cout << "La liste l contient " << l.size() << " entiers : \n";
   // utilisation d'un itérateur pour parcourir la liste l
   for (list<int>::iterator it = l.begin(); it != l.end(); ++it)
   cout << "   " << *it;
   cout << "\n";
   // afficher le premier élément
   cout << "Premier element : " << l.front() << "\n";
   // afficher le dernier élément
   cout << "Dernier element : " << l.back() << "\n";
   // parcours avec un itérateur en inverse
   for ( list<int>::reverse_iterator rit = l.rbegin(); rit != l.rend(); ++rit )
   {
      cout << "   " << *rit;
   }
   cout << "\n";return 0;
}
```

II.4 Les conteneurs associatifs : cas du map

Les conteneurs associatifs, collectionnent des objets de même type, chacun associé à une clé permettant d'y avoir accès. Il y a deux types de conteneurs associatifs : ceux où la clé et l'objet ne font qu'un, les **ensembles** (set) et les **multi-ensembles** (multiset). Au contraire, dans les **tables associatives** (map) et les **tables associatives multiples** (multimap) la clé et la valeur associée sont séparées et peuvent être de types différents.

Une table associative map permet d'associer une clé à une donnée.

La map prend donc au moins deux paramètres :

- le type de la clé (dans l'exemple ci-dessous, une chaîne de caractères string)

- le type de la donnée (dans l'exemple ci-dessous, un entier non signé unsigned int)

```
#include <iostream>
#include <iomanip>
#include <map>
#include <string>
using namespace std;
int main()
{
    map<string,unsigned int> nbJoursMois;
    nbJoursMois["janvier"] = 31;
    nbJoursMois["février"] = 28;
    nbJoursMois["mars"] = 31;
    nbJoursMois["avril"] = 30;
    //...
    cout << "La map contient " << nbJoursMois.size() << " elements : \n";
    for (map<string,unsigned int>::iterator it=nbJoursMois.begin(); it!=nbJoursMois.end(); ++
    it)
    {
    cout << it->first << " -> \t" << it->second << endl;
    }
    cout << "Nombre de jours du mois de janvier : " << nbJoursMois.find("janvier")->second <<'\n';
    // affichage du mois de janvier
    cout << "Janvier : \n" ;
    for (int i=1; i <= nbJoursMois["janvier"]; i++)
    {
    cout << setw(3) << i;
    if(i%7 == 0)
    cout << endl;
    }
    cout << endl;
    return 0;
}
```

III Les espaces de nommage (namespace)

III.1 Introduction

Un problème survient souvent dans les grosses architectures utilisant de nombreuses librairies et fonctions développées séparément : **Il y a souvent collision entre les identifiants.**

Peut-être par manque d'imagination, surtout par respect de conventions, les programmeurs ont tendance à donner des noms identiques à leurs classes et fonctions.

Par exemple un logiciel de cartographie anglophone, définira la classe ***map*** comme un point d'accès à une carte alors que la librairie standard définie la classe ***map*** comme une classe d'association entre objets (c'est du vécu…).

Heureusement les espaces de nommage résolvent totalement ce problème.

III.2 Espaces de nommage nommés

Concrètement, on nomme une région déclarative (une portion du code) en utilisant le mot clé namespace :

```
#include <map> // ce fichier de la librairie standard définit
               // une classe 'map', à l'intérieur d'un espace de
               // nommage nommé std
namespace MesMaps
{
  class map
  {
    // moi aussi j'ai le droit de faire une classe map!!
    ...
    void Methode();
  };
  void fct(map m,double);
}

void MesMaps::fct(MesMap::map m, double d){...}
void MesMaps::map::Methode(){...}

MesMaps::map m1,m2;     // instancie ma classe
std::map<int,double> m; // instancie la classe map de la librairie
                        // standard
```

- ✓ Cette exemple montre bien comment cela se passe. A l'intérieur de notre espace de nommage **MesMaps** on donne les noms que l'on veut à nos classes et nos fonctions.
- ✓ A l'extérieur de l'espace de nommage on accède à nos classes et nos fonctions en les préfixant par ***MesMaps::.***
- ✓ On peut imbriquer les espaces de noms. Si C dans B dans A et la fonction fct() est dans C on y accède avec ***A::B::C::fct().***

III.3 Les déclarations using

La déclaration **using** permet l'utilisation d'un identificateur déclaré dans un espace de nommage sans avoir à utiliser l'opérateur de résolution de portée :

```
namespace A{  int i; int fct(int j);  }
using A::i;
using A::fct;
...
i=fct(67);   // utilise i et fct de l'espace de nommage A
```

Bien évidemment cela peut générer des collisions d'identificateurs.

On peut aussi utiliser la déclaration using dans un autre espace de nommage mais c'est à éviter :

```
namespace A{int i;}
namespace B{using A::i;}
```

```
B::i = 6;   //c'est bien le i déclaré dans A
```

- ✓ La directive **`using`** `namespace` permet d'utiliser tous les identificateurs d'un espace de nommage.

```
namespace A{  int i; int fct(int j);  }
using namespace A;
...
i=fct(67);   // utilise i et fct de l'espace de nommage A
```

- ✓ Il semblerait que cette pratique enlève le bénéfice de l'utilisation des espaces de nommage, mais dans le cas où il n'y a pas de collisions d'identificateurs, c'est beaucoup moins fastidieux que d'utiliser la déclaration **`using`** pour tous les identificateurs.

IV Gestion des exceptions

IV.1 La problématique :Gérer toutes les erreurs dans un programme

- ✓ Lors des appels systèmes ou hors du programme courant, les programmes doivent faire face à des situations exceptionnelles indépendantes du programmeur :
  - Accès à un fichier qui n'existe pas ou plus.
  - Demande d'allocation de mémoire alors qu'il n'y en a plus.
  - Accès à un serveur qui est crashé.

  …
- ✓ Ces situations, qui ne sont pas des bugs mais que l'on peut appeler erreurs, engendrent un arrêt du programme si elles ne sont pas traitées.
- ✓ Pour les traiter on peut tester les codes d'erreurs retournés par les fonctions critiques, mais ceci présente deux inconvénients :
  - Le code devient lourd, puisque chaque appel à une fonction critique est suivie de nombreux tests.
  - Le programmeur doit prévoir toutes les situations possibles dès la conception du programme, ainsi que définir les réactions du programme et les traitements à effectuer
- ✓ En fait ces inconvénients sont majeurs, et il a fallu trouver d'autres solutions à ce problème : c'est la gestion des exceptions.

IV.2 Principe de la gestion des exceptions

- ✓ Voici les étapes dans la gestion d'une exception :
  - Une erreur est trouvée.
  - On construit un objet qui contient, éventuellement, des paramètres descriptifs de l'erreur.
  - Une exception est lancée, paramétrée par l'objet.
  - Deux possibilités peuvent survenir à ce moment :
    - Un gestionnaire d'exception rattrape l'exception, l'analyse, et à la possibilité d'exécuter du code, par exemple pour sauver des données.
    - Il n'y a pas de gestionnaire d'exception prévu pour ce type d'exception. Le programme se termine.
- ✓ Voici un exemple :

```
struct CErrorAccesServeur
{
  unsigned long  m_AddressIP;
  unsigned short m_Port;
  CErrorAccesServeur(unsigned long AddressIP,unsigned short Port)
    : m_AddressIP(AddressIP), m_Port(Port) {}
};
void main(void)
{
  try
  {
     ...
     if( ! PingServeur( AddressIP , Port ) )
        throw CErrorAccesServeur(AddressIP , Port);
     ...
  }
   catch(CErrorAccesServeur &e) // gestionnaire d'erreur de ce type
   {
      // traitement de l'erreur, par exemple affichage de
      // l'adresse et du port
      cout << "Acces refusé : AdressIP :" << e.m_AddressIP <<
                                          "Port :" << e.m_Port ;
      // autre traitement, arrêt du programme, ou on réessaye...
   }
   catch(...)  // gestionnaire de toutes les erreurs non rattrapées
   {
      // ce gestionnaire rattrape toutes les erreurs, non déjà
      // rattrapées par un gestionnaire d'erreur précédent
      // Il est très pratique, par exemple pour assurer qu'un
      // serveur, ne plante jamais, mais on ne peut récupérer
      // aucune information sur l'erreur rattrapée
   }
}
```

- ✓ La syntaxe de traitement des exceptions ne fait donc appel qu'à trois mots clés : try, throw, catch.
- ✓ Cette syntaxe est la même que celle d'autres langages comme ADA ou Java.
- ✓ Notez que l'on peut imbriquer les blocs try/catch.
- ✓ Notez que un gestionnaire d'exception peut lever une exception avec le mot clé throw.

TRAVAUX DIRIGES

Exercice 1 :

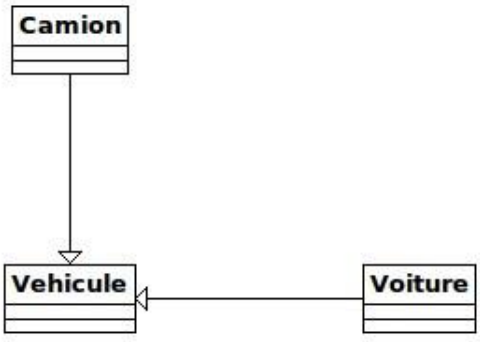
```
#include <iostream.h>
#include <list.h>
using namespace std;
typedef list<int> LISTINT;
void main(void)
{
      LISTINT liste;
      LISTINT::iterator i;
      //ajout de données dans la liste
      liste.push_front (2);
      liste.push_front (1);
      liste.push_back (3);
      liste.push_back (10);
      liste.push_back (12);
      for(i=liste.begin(); i!=liste.end(); ++i)
      cout<<*i<<" ";
      cout<<endl;
      for(i=liste.end(); i!=liste.begin(); --i)
      cout<<*i<<" ";
      cout<<endl;

}
```

1) Quelle est la sortie du programme ?
2) Exécuter ce programme comparer les résultats avec ceux de la question 1
3) Corriger le programme pour afficher les données de la liste dans l'ordre inverse

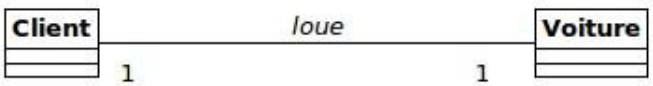
Question 2

Donner la déclaration des classes Vehicule, Camion et Voiture pour les relations du diagramme ci-dessous.



Question 3

Donner la déclaration des classes Client et Voiture pour la relation du diagramme ci-dessous.

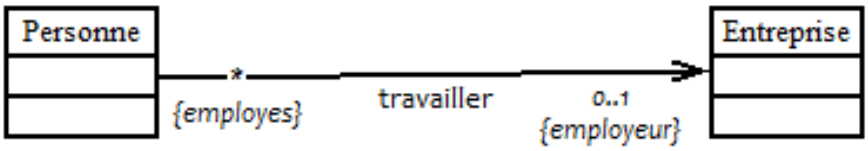


Question 4

Donner la déclaration des classes Equipe et Joueur pour la relation du diagramme ci dessous.

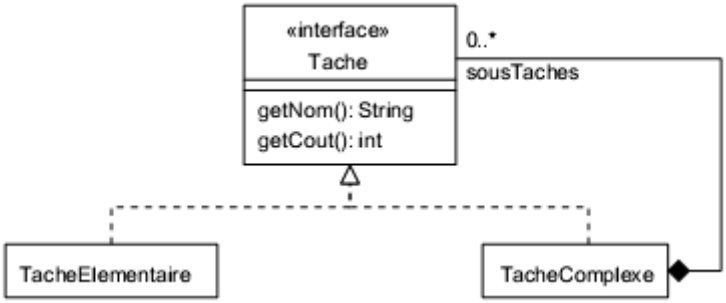
Question 5 :

Donner le code C++ correspondant au schema UML ci-dessous



Exercice 6 :

On considère le diagramme de classe ci-dessous.



L'architecture des tâches est donnée ci-dessus où le détail des classes TacheElementaire et TacheComplexe n'est pas donné. Une tâche est caractérisée par un nom et un coût. Une tâche est soit une tâche élémentaire, soit une tâche complexe qui est alors composée de sous-tâches. Il est ainsi possible d'ajouter une sous-tâche à une tâche complexe, ajouter(Tache) ou de supprimer une sous-tâche, supprimer(Tache). Le coût d'une tâche complexe est la somme des coûts des tâches qui la composent. Une tache élémentaire possède un cout de base.

Questions : 1+2+3+1 pts

1) Donnez le code de l'interface Tache ;
2) Donnez le code de la classe TacheElementaire ;
3) Donnez le codede la classe TacheComplexe. Pensez à toutes les méthodes ;
4) Donnez la définition de la classe GestionTache Cette classe contient une méthode main permettant de créer une tache complexe composée de trois taches et d'afficher le cout de cette tache.

Exercice 7 : Un répertoire téléphonique

Toutes les classes, qui seront implémentées dans cet exercice, appartiennent au même naespace nommé «organisateur».

1) complétez la classe « Contact » représentant les coordonnées d'une personne avec les champs : nom, prénom, téléphone, adresse. Munissez-la d'un constructeur des accesseurs, mutateurs et d'une méthode ***toString***().

```
class Contact
{ private:
```

```
        string nom ;
        string telephone ;
        string adresse ;
        //…à compléter
    }
```

2) Compléter ensuite la classe «Calepin» représentant un ensemble de Contacts (Repertoire téléphonique par exemple). Elle contiendra les attributs privés suivants :

- Une liste de Contacts nommé « mesContacts ». Les contacts seront gérés à travers un List
- Un constructeur pour définir un carnet vide. Le nombre maximum de contact dans le carnet est fixée à 100
- void ajouter(Contact c): pour ajouter un nouveau contact dans « mesContacts». On levera une exception si on a atteint déjà 100 contacts.
- void rechercher(String nom)pour retrouver et afficher les coordonnées de la première (téléphone, adresse) dans le carnet à partir de son nom. Si cette méthode ne trouve pas les coordonnées d'une personne dans le carnet elle affiche que cette personne n'existe pas dans le carnet.
- void affiche(): pour afficher toutes les coordonnées se trouvant dans le carnet.

```
class Calepin {
    // a completer
}

    void  main(){
        Calepin *c=new Calepin();
        c->ajouterContact(new Contact("TOTO","60458912", "Yaoundé"));
        c->ajouterContact (new Contact("SOSSO Joseph","600128912", "Douala"));
        c->ajouterContact (new Contact("OMOKOLO Joseph","58458912", "Yaoundé"));
        c->affiche();
        c->chercher("OMOKOLO");
    }
```