

*La Programmation
Orientée Objets
en C++*

Sommaire

| | |
|--|----|
| I. Programmation orientée objets..... | 1 |
| II. Introduction au langage C++ | 5 |
| II.1. Structure générale d'un programme | 5 |
| II.2. Variables et constantes..... | 8 |
| II.3. Les fonctions | 9 |
| II.4. Allocation dynamique | 15 |
| III. Classes et Objets..... | 19 |
| III.1. Définition d'une classe..... | 19 |
| III.2. Utilisation d'une classe | 21 |
| III.3. Affectation entre objets..... | 22 |
| III.4. Constructeur et destructeur | 22 |
| III.5. Membres données statiques | 23 |
| III.6. Exploitation des classes | 25 |
| IV. Les propriétés des fonctions membres..... | 27 |
| IV.1. Objets transmis en argument | 28 |
| IV.2. Objets fournis en valeur de retour | 29 |
| IV.3. Le pointeur this | 29 |
| IV.4. Fonctions membres en ligne..... | 29 |
| IV.5. Fonctions membres statiques..... | 30 |
| IV.6. Les fonctions membres constantes | 31 |
| V. Construction, destruction et initialisation des objets..... | 35 |
| V.1. Classes de stockage..... | 35 |
| V.2. Déclaration et initialisation des objets | 37 |
| V.3. Constructeur par recopie..... | 41 |
| V.4. Appel des constructeurs | 42 |
| V.5. Tableaux d'objets | 45 |
| V.6. Objets d'objets | 46 |
| VI. Les fonctions et classes amies..... | 51 |
| VI.1. Les fonctions amies | 51 |
| VI.2. Classes amies | 57 |
| VII. La surcharge des opérateurs..... | 59 |
| VII.1. Règles générales pour la surcharge des opérateurs..... | 60 |
| VII.2. Les opérateurs unaires : | 62 |
| VII.3. Les opérateurs binaires..... | 64 |
| VII.4. Les opérateurs de déréférencement, d'indirection et d'accès aux membres | 67 |
| VII.5. L'opérateur d'appel de fonction | 69 |
| VII.6. L'opérateur d'indexation..... | 70 |
| VII.7. Les opérateurs d'allocation dynamique..... | 72 |
| VIII. Conversions | 75 |
| VIII.1. Conversion définie par le constructeur | 76 |
| VIII.2. Opérateur de cast | 76 |
| VIII.3. Problèmes posées par le transtypage..... | 77 |
| IX. Héritage simple | 79 |

| | | |
|---------|--|-----|
| IX.1. | Droit d'accès à la classe de base | 80 |
| IX.2. | Un exemple d'héritage simple | 81 |
| IX.3. | Redéfinition des fonctions membres..... | 82 |
| IX.4. | Constructeur et héritage..... | 84 |
| IX.5. | Conversions entre classes de base et classes dérivées..... | 88 |
| IX.6. | Typage dynamique et fonctions virtuelles..... | 90 |
| IX.7. | Fonctions virtuelles pures et classes abstraites..... | 96 |
| X. | Héritage multiple..... | 99 |
| X.1. | Un exemple d'héritage multiple | 99 |
| X.2. | Résolution des conflits entre identificateurs..... | 101 |
| X.3. | Classes virtuelles | 102 |
| XI. | Les exceptions | 105 |
| XI.1. | Lancement et récupération d'une exception..... | 106 |
| XI.2. | La logique de traitement des exceptions..... | 108 |
| XI.3. | Déclaration et classification des exceptions | 110 |
| XI.4. | Liste des exceptions autorisées dans une fonction | 110 |
| XI.5. | Exceptions et constructeurs..... | 111 |
| XII. | Namespaces | 115 |
| XIII. | Templates | 119 |
| XIII.1. | Paramètres template | 120 |
| XIII.2. | Les fonctions template..... | 121 |
| XIII.3. | Les classes template..... | 123 |
| XIII.4. | Les fonctions membres template | 127 |
| XIII.5. | Spécialisation des template | 128 |
| XIII.6. | Les template et l'amitié | 130 |

Chapitre I

Programmation orientée objets

Critères de production d'un logiciel

Un logiciel est dit de "bon qualité" s'il répond aux critères imposés par les utilisateurs et à ceux analysés par les programmeurs.

Critères imposés par l'utilisateur :

- ***L'exactitude*** : dans les conditions normales d'utilisation, un programme doit fournir exactement les résultats demandés.
- ***La robustesse*** : lorsqu'on s'écarte des conditions normales d'utilisation, un programme doit bien réagir (traitement d'erreur,...).
- ***L'extensibilité*** : Un programme doit intégrer facilement des nouvelles spécifications demandées par l'utilisateur ou imposées par un événement extérieur (Maintenance)
- ***L'efficience*** : temps d'exécution, taille mémoire, ...
- ***La portabilité*** : facilité d'exploitation dans différentes implémentations.
- La facilité de mise en œuvre et d'apprentissage.
- La qualité de la documentation.

Critères analysés par le programmeur :

- ***La réutilisabilité*** : possibilité d'utiliser certaines parties du logiciel dans un autre.
- ***La modularité*** : le programme est décomposé en des petites parties (modules) qui peuvent être compilées séparément, de telle sorte que l'ensemble des modules et les relations entre eux (système informatique) permettent de résoudre le problème initial.

Notons que seuls les critères analysés par l'utilisateur ont vraiment de l'importance. Mais pour répondre à ces derniers, il est parfois impératif de satisfaire aussi les critères perceptibles par le programmeur.

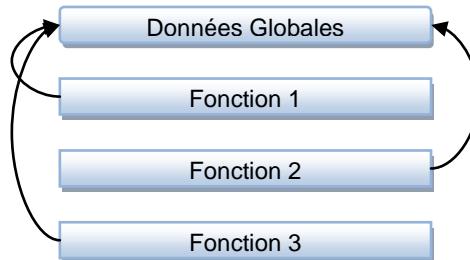
Programmation structurée :

- La construction d'un logiciel se base fondamentalement sur l'équation (de WRITH) :

$$\boxed{\text{Algorithmes} + \text{Structures de données} = \text{Programme}}$$

Par ailleurs, la conception d'un programme structuré peut être basée soit sur le traitement soit sur les données.

- La conception se basant sur le traitement est l'approche traditionnelle. Elle consiste à décomposer le programme en des tâches simples (approche descendante) ou à construire le programme en composant des fonctions disponibles (approche ascendante).



- Cette approche permet d'améliorer l'exactitude et la robustesse, mais ne permet que peu d'extensibilité : En pratique, lors de l'adaptation du système aux nouvelles spécifications (nouvelles structures de données), la dissociation entre données et fonctions conduit souvent à casser des modules.

Conceptions par objets

Dans la conception basée sur les données, une réflexion autour des données conduit à :

- ◆ Déterminer les données à manipuler.
- ◆ Réaliser, pour chaque type de données, les fonctions qui permettent de les manipuler.

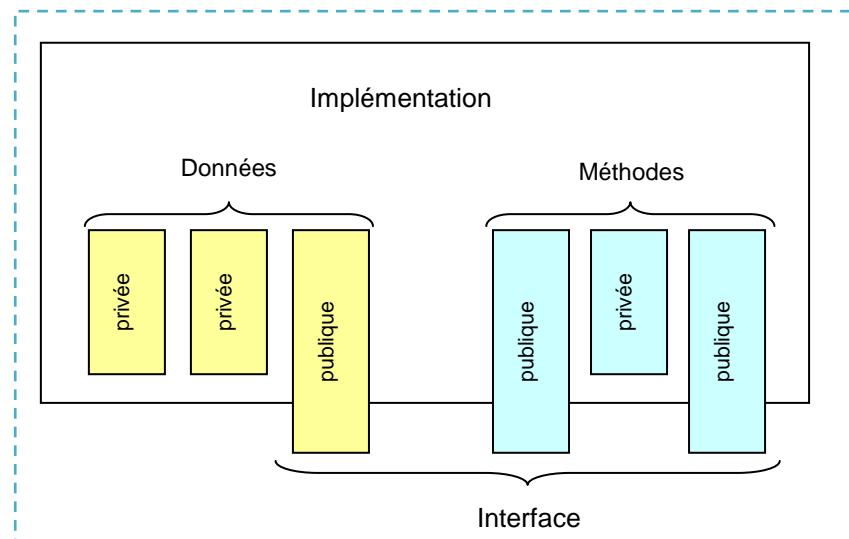
On parle alors d'**OBJETS**

Un objet est une association de données et des fonctions (méthodes) opérant sur ces données.

$$\boxed{\text{Objet} = \text{Données} + \text{Méthodes}}$$

Concepts fondamentaux des objets

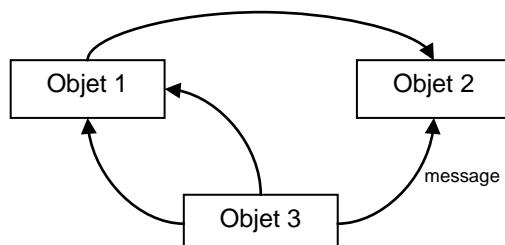
- **Encapsulation des données** : consiste à faire une distinction entre l'interface de l'objet et son implémentation.



- ◆ Interface : décrit ce que fait l'objet.
- ◆ Implémentation : définit comment réaliser l'interface.

Le principe de l'encapsulation est qu'on ne peut agir que sur les propriétés publiques d'un objet : les données sont toutes privées, leur manipulation se fait à travers les méthodes publiques.

- **Communication par messages** : Les objets communiquent entre eux par des messages (un objet demande à un autre objet un service).



- **Identité et classification** : consiste à regrouper les objets ayant le même comportement pour former un même ensemble, appelé **CLASSE** (cette notion n'est autre que la généralisation de la notion de *type*).
Un objet d'une classe s'appelle **INSTANCE** de cette classe.
- **Héritage** : consiste à définir une nouvelle classe à partir d'une classe existante, à laquelle on ajoute des nouvelles propriétés (données ou méthodes).
- **Polymorphisme** : possibilité à divers objets de classes dérivées d'une même classe de répondre au même message. Autrement dit, un même nom peut désigner des propriétés de classes différentes.
- **Généricité** : consiste à définir des classes paramétrées. Une classe générique n'est pas directement utilisable, mais permet de créer des classes dérivées qui peuvent être manipulées.

- **Modularisation** : Les modules sont construits autour des classes. Un module contiendra l'implémentation d'une classe ou d'un ensemble de classes liées.

Programmation Orientée Objets (P.O.O.)

- La programmation basée sur les objets et leurs concepts fondamentaux est dite la programmation orientée objets.
- Un programme orienté objets est formé d'un ensemble de classes autonomes qui coopèrent pour résoudre un problème donné.
- L'approche objets avec ses concepts fondamentaux répond bien aux critères de qualité de production d'un logiciel.

Chapitre II

Introduction au langage C++

Le langage C++ peut être considéré comme un perfectionnement du langage C qui offre les possibilités de la POO.

Les notions de base de la programmation en C restent valables en C++, néanmoins C et C++ diffèrent sur quelques conventions (déclaration des variables et des fonctions,...)

Ce chapitre retrace ses différences, et traite les autres outils de la programmation structurée ajouté à C++

II.1 Structure générale d'un programme

La fonction main

Comme en C, la fonction *main* est le point d'entrée de tout programme C++. Elle peut être définie de deux manières :

- pour un programme sans paramètres

```
int main( ) { ..... }
```

- pour un programme avec paramètres : programme qui accepte des arguments sur la ligne de commande

```
int main(int argc) { ..... }
int main(int argc, char* argv[ ]) { ..... }
```

♦ *argc* : nombre de paramètres

- **argv** : tableau de chaînes de caractères représentant les paramètres de la ligne de commande. Le premier élément d'indice 0 de ce tableau est le nom du programme lui-même.

Exemple 2.1 : (EXP02_01.CPP)

Le programme permet d'afficher les arguments de la ligne de commande ligne par ligne.

```
#include <stdio.h>
int main(int argc, char* argv[])
{
    int i;
    if ( argc != 1){
        printf("Le nombre des arguments de la ligne de commande : %d\n",argc);
        printf("Le nom du programme : %s\n",argv[0]);
        printf("les arguments :\n");
        for(i=1; i < argc; i++)
            printf(" ** %s\n", argv[i]);
    }
    return 0;
}
```

► Exemple d'exécution du programme

```
>exp02_01 un test
Le nombre des arguments de la ligne de commande : 3
Le nom du programme : exp02_01
Les arguments :
** un
** test
```

Les commentaires

- Commentaires sur plusieurs lignes : délimités par **/*** (début) et ***/** (fin).
- Commentaires sur une seule ligne : délimités par **//** (début) et fin de ligne (n'existe pas en C)

La bibliothèque de C++

C++ possède une bibliothèque très riche, qui comporte un très grand nombre d'outils (fonctions, types, ...) qui permettent de faciliter la programmation. Elle intègre en plus de la bibliothèque standard de C, des librairies de gestion des entrées-sorties ainsi que des outils de manipulation des chaînes de caractères, des tableaux et d'autres structures de données.

Pour utiliser, en C++, les outils qui existaient dans la bibliothèque standard de C (stdio.h, string.h, ...) ainsi que certains nouveaux outils (comme par exemple iostream.h pour la gestion des entrées-sorties) il suffit, comme en C, de spécifier avec la directive **include** le fichier entête (.h) souhaité (cf. : Exemple 2.1).

Cette méthode est devenue obsolète et elle n'est plus supportée par certains environnements de développement (surtout pour l'utilisation des nouveaux outils de la bibliothèque de C++). En effet, pour des raisons liées à la POO (généricité, modularité...) et pour éviter certains conflits qui peuvent surgir entre les différents noms des outils utilisés (prédéfinis ou définis par l'utilisateur), C++ introduit la notion de **namespace** (espace de noms), ce qui permet de définir des zones de déclaration et de définitions des différents outils (variables, fonctions, types,...).

Ainsi, chaque élément défini dans un programme ou dans une bibliothèque appartient désormais à un namespace. La plupart des outils de la bibliothèque de C++ appartiennent à un namespace nommé "**std**".

Par suite, pour pouvoir utiliser une librairie de C++, on doit spécifier le nom (sans le .h) de cette dernière avec la directive **include** et indiquer au compilateur qu'on utilise le namespace **std** avec l'instruction **using**, comme par exemple :

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

D'autre part, et pour utiliser la même notion pour les outils définis dans la bibliothèque de C, C++ redéfinie ces outils dans des fichiers qui portent les mêmes noms formés par les anciens noms précédés d'un 'c' au début (cstdio, cstring,...), exemple:

```
#include <cstdio>
using namespace std;...
```

(pour plus de détails sur l'utilisation des namespaces voir chapitre 12)

Les entrées/sorties en C++

- 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** : entrée standard (clavier par défaut)
 - **cout** : sortie standard (écran par défaut)
 - **cerr** : sortie des messages d'erreur (écran par défaut)

Syntaxes :

```
cout << exp_1 << exp_2 << ... ... << exp_n ;
```

♦ *exp_k* : expression de type de base ou chaîne de caractères

```
cin >> var_1 >> var_2 >> ... ... >> var_n ;
```

♦ *var_k* : variable de type de base ou char*

- 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.

```
int n ;

cout << " Entrez une valeur : " ;
cin >> n ;
cout << " la valeur entrée est : " << n << endl;
```

- Avantages à utiliser les nouvelles possibilités d'E/S :
 - ♦ vitesse d'exécution plus rapide.
 - ♦ il n'y plus de problème de types
 - ♦ autres avantages liés à la POO.

II.2 Variables et constantes

Les types prédefinis

- C++ conserve tous les types de base prédefinis dans C :
 - ◆ **void** : utilisé pour spécifier des variables qui n'ont pas de type.
 - ◆ **char** : type de base des caractères. Les variables de type **char** sont considérées comme des entiers (code ASCII du caractère). Ce type peut être signé ou non signé, d'où les types **signed char** et **unsigned char**. Cependant, il faut noter que la norme de C++, ne spécifie aucun signe par défaut pour le type **char** (le signe affecté à ce type dépend du compilateur utilisé), ainsi les trois types **char**, **signed char** et **unsigned char** sont considérés comme trois types différents.
 - ◆ **int** : type de base des entiers. Ce type peut être qualifié par les mots clés **short** et **long**, permettant ainsi de modifier la taille du type. Notons qu'en spécifiant ces mots clés, le mot clé **int** devient facultatif, ainsi **short int** (resp: **long int**) est équivalent à **short** (resp: **long**). D'autre part, les trois types **int**, **short** et **long** sont considérés comme des entiers signés et pour manipuler les entiers non signés on utilise versions non signés de ces types à savoir **unsigned int**, **unsigned short** et **unsigned long**.
 - ◆ **float** et **double** : types pour les réels et les réels en double précision. Le type **double** peut être qualifié du mot clé **long**: le type **long double** est utilisé pour manipuler les réels avec plus de précision que celle des variables de type **double**.
- La norme de C++ ne spécifie pas la taille mémoire occupée par ces différents types. Cette taille, qui est généralement un octet pour **char**, 2 octets pour **short**, 4 octets pour **int**, **long** et **float** et 8 octets pour **double**, change selon l'environnement (compilateur, système d'exploitation, machine,...).
- C++ introduit un nouveau type **bool**, pour manipuler les expressions booléennes. Les variables de ce type ne peuvent prendre que l'une des deux valeurs : **true** ou **false**.

Déclarations des variables

- Contrairement à C, en C++ les variables peuvent être déclarées n'importe où dans le programme: leur portée reste limitée au bloc de leur déclaration.

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

Les constantes

- Le qualificatif **const** peut être utilisé pour une expression constante :

```
const int N = 5 ;
int t[N] ; // en C il faut définir N avec #define
```

- Un symbole déclaré avec **const** à une portée limité au fichier source.

Le type **void***

- En C, le type **void*** est compatible avec tous les autres pointeurs.
- En C++, seule la conversion d'un pointeur quelconque en **void*** est implicite, par contre la conversion de **void*** vers un autre type de pointeur doit être explicite :

```
void* pt;
int * pi
```

```
pt = pi           // Acceptable en C et C++
pi = pt           // Acceptable seulement en C
// En C++ on doit écrire
pi = (int*) pt;
```

II.3 Les fonctions

Déclaration des fonctions

- L'utilisation d'une fonction sans aucune déclaration (**prototype**) ou aucune définition au préalable conduit à une erreur de compilation.
- Le prototype d'une fonction doit figurer dans tout fichier source qui utilise cette fonction et ne contient pas sa définition.
- Une fonction doit spécifier son type de retour. On utilise le type **void** pour les fonctions qui ne retourne aucune valeur.

```
fct(int,int) ;           // erreur de compilation
void fct(int,int) ;
```

- Dans la déclaration et la définition d'une fonction sans arguments, on peut fournir une liste vide.

```
int fct(void) ;
int fct()
```

Références

- Une référence d'une variable est un identificateur qui joue le rôle d'un alias (synonyme) de cette variable.

Syntaxe :

```
nom_type &ident_ref = ident_var
```

- Une variable peut être manipulée par sa référence et vice versa :

```
int n;
int &p = n; // p est une référence de n
             // n et p désignent le même emplacement mémoire.
p = 10;     // n vaut aussi 10
n = p + 1;  // après cette instruction n et p vaudront 11
```

- Une référence ne peut être vide, elle doit être toujours initialisée lors de sa déclaration.

```
int &refvar;           // erreur
```

- Il est possible de faire des références sur des valeurs numériques. Dans ce cas ils doivent être déclarés comme des constantes.

```
const int &refval = 3;      // référence à une constante
int &refval = 3;            // erreur
```

- Les références et les pointeurs sont très liés : une variable peut être manipulée à travers un pointeur ou à travers une référence, ainsi :

```
int n = 10;
int *pn = n;
*pn = *pn + 1;    // manipuler n via pn
```

et

```
int n= 10;
int &refn = n;
refn = refn + 1;           // manipuler n via refn
```

réalisent la même opération.

Transmission des arguments

- En C, il y a deux méthodes de passage des variables en paramètres à une fonction :
passage par valeur et passage par adresse
 - Passage par valeur** : la valeur de la variable (ou expression) en paramètre est copiée dans une variable temporaire. Les modifications opérées sur les arguments dans la fonction n'affectent pas la valeur de la variable passée en paramètre.

Exemple 2.2 : (EXP02_02.CPP)

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

void fct_par_valeur(int k)
{
    k = 5;
}

int main()
{
    int n = 10;
    fct_par_valeur(n);
    cout << n << endl;
    return 0;
}
```

► Sortie du programme

```
> 10
```

- Passage par adresse** : consiste à passer l'adresse d'une variable en paramètre. Toute modification du paramètre dans la fonction affecte directement la variable passée en argument correspondant, puisque la fonction accède à l'emplacement mémoire de son argument.

Exemple 2.3 : (EXP02_03.CPP)

```
void fct_par_adresse(int *k)
{
    *k = 5;
}

int main()
{
    int n = 10;
    fct_par_adresse(&n);
    cout << n << endl;
    return 0;
}
```

► Sortie du programme

```
> 5
```

- Le passage par adresse présente parfois certaines difficultés d'utilisation. Pour résoudre ces inconvénients, C++ introduit la notion de passage par référence:

- ♦ **Passage par référence** : consiste à passer une référence de la variable en argument. Ainsi, aucune variable temporaire ne sera créée par la fonction et toutes les opérations (de la fonction) seront effectuées directement sur la variable.

Exemple 2.4 : (EXP02_04.CPP)

```
void fct_par_ref(int &k)
{
    k = 5;
}

int main()
{
    int n = 10;
    fct_par_ref(n);
    cout << n << endl;
    return 0;
}
```

► Sortie du programme

```
> 5
```

- Ce mode de passage peut aussi remplacer le passage par valeur. Pour ce, on utilise le qualificatif **const** pour éviter le changement de la valeur du paramètre.

Syntaxe :

```
type fct(const nom_type & ident_arg)
```

- Il est recommandé de passer par référence tous les paramètres dont la copie peut prendre beaucoup de ressources (temps d'exécution, taille mémoire).

Valeur de retour d'une fonction

- Une fonction peut retourner une valeur, une adresse ou une référence.
- Cependant, la référence ou l'adresse retournée par une fonction ne doit pas être locale à cette fonction, vu que les variables locales à une fonction sont automatiquement détruites à la fin de la fonction.

Exemple 2.5 : (EXP02_05.CPP)

```
char* f()
{
    char temp[] = "variable locale";
    return temp;           // à éviter
}

int main()
{
    char * s = f();
    cout << s << endl;
    return 0;
}
```

Ce programme donne un résultat imprévisible, du fait que la variable locale ‘temp’ est détruite à la fin de la fonction, donc ‘s’ pointe sur un emplacement supposé libre.

- Une fonction qui retourne une référence peut être utilisée comme une l-value : elle peut être placée par exemple à gauche d'une instruction d'affectation :

Exemple 2.6 : (EXP02_06.CPP)

```
int & f(int & k)
{
    cout << ++k << endl;
    return k;
}

int main()
{
    int n = 1;
    f(n) += 5;
    cout << n << endl;
    return 0;
}
```

► Sortie du programme

```
2
7
```

Arguments par défaut

- C++ offre la possibilité de donner des valeurs par défaut aux paramètres d'une fonction.
- Une fonction peut définir des valeurs par défaut pour tous ses paramètres ou seulement pour une partie. Cependant, les paramètres ayant une valeur par défaut doivent être placés en dernier dans la liste des arguments

```
void fct(int = 15, int);      // erreur
```

- Les valeurs par défaut doivent être mentionnées soit dans la déclaration de la fonction soit dans sa définition.

Exemple 2.7 : (EXP02_07.CPP)

```
void f( int, int = 15);      // prototype

int main()
{
    int n = 10, p = 20;
    f(n,p);                // appel normal
    f(n);                  // appel avec un seul argument
    return 0;
}
void f( int a, int b )
{
    cout << "Argument 1 : " << a ;
    cout << " - Argument 2 : " << b << endl;
}
```

► Sortie du programme

```
Argument 1 : 10 - Argument 2 : 20
Argument 1 : 10 - Argument 2 : 15
```

Fonction avec un nombre variable d'arguments

La notion des fonctions avec un nombre variable d'arguments a été introduite en C, mais reste aussi valable en C++.

Syntaxe :

```
type fct(lst_param, ...)
```

L'accès et la manipulation des arguments ajoutés aux paramètres visibles ‘*lst_param*’, lors de l'appel de la fonction se font via des macros et des types définis dans la bibliothèque `<cstdarg>` :

```
void va_start(va_list ap, param_dernier);
type_p va_arg(va_list ap, type_p);
void va_end(va_list ap);
```

- `va_list` : est un type, dont l'implémentation dépend du compilateur, et qui sert comme paramètre pour les autres macros de manipulation.
- `va_start` : initialise la variable `ap`, qui servira à récupérer les paramètres à partir du `param_dernier` (le dernier argument mentionné dans la définition de la fonction). La variable `ap` est utilisée comme une liste qui contient les paramètres ajoutés.
- `va_arg` : retourne le premier paramètre de la liste `ap` non encore lu. Le type `type_p` doit correspondre au type du paramètre à récupérer.
- `va_end` : permet de libérer la variable `ap`. Toute variable initialisée par `va_start` doit être libérée par `va_end`.

Exemple 2.8 : (EXP02_08.CPP)

Exemple d'une fonction qui retourne la somme d'une suite d'entiers qui se termine par 0.

```
#include <iostream>
#include <cstdarg>

using namespace std;

int somme(int premier,...)
{
    va_list liste;
    va_start(liste,premier);           // initialisation de liste
    int resultat = 0;
    int i = premier;
    while (i != 0){
        resultat += i;
        i = va_arg(liste,int);         // récupère le suivant
    };
    va_end(liste);                   // libère la liste
    return resultat;
}

int main()
{
    cout << somme(1,2,3,0) << endl;
    cout << somme(1,2,3,4,5,0) << endl;
    return 0;
}
```

Surcharge

- La surcharge des fonctions consiste à donner un même nom à plusieurs fonctions.

```
int max( int a, int b);           // fonction I
int max( int a, int b, int c);   // fonction II
int max( int * tab, int taille); // fonction III
```

- Pour différencier entre deux fonctions qui portent le même nom, le compilateur regarde le type et le nombre des arguments effectifs : La liste des types des arguments d'une fonction s'appelle la ***signature*** de la fonction.

```
int a,b,c;
int t[10];

max( a, b);           // appel de fonction I
max( a, b, c);       // appel de fonction II
max( t, 5);          // appel de fonction III
```

- Il faut noter que la surcharge n'est acceptable que si toutes les fonctions concernées aient des signatures différentes. D'autre part, pour que la surcharge ait un sens, les surdéfinitions doivent avoir un même but.
- Il est également possible de surcharger les opérateurs (classes).

L'ordre d'évaluation des arguments

En C++, l'ordre d'évaluation des arguments d'une fonction n'est spécifié. Ainsi l'appel :

```
f(expr1, expr2, ..., exprn)
```

sera l'appel de la fonction **f**, après l'évaluation de toutes les expressions mais dans un ordre qui ne dépend que du compilateur, ce qui peut rendre un résultat imprévisible si les expressions sont dépendantes l'une de l'autre :

```
int x = 0 ;
f(++x,++x)           // peut être f(1,2) ou f(2,1)
```

Les fonctions inline

- Une fonction "inline" est une fonction dont les instructions sont incorporées par le compilateur à chaque appel. Une telle fonction est déclarée ou définie comme suit :

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

- Les fonctions "inline" permettent de gagner au niveau de temps d'exécution, mais augmente la taille des programmes en mémoire.
- Contrairement aux macros dans C, les fonctions "inline" évitent les effets de bord (dans une macro l'argument peut être évalué plusieurs fois avant l'exécution de la macro).

Exemple 2.9 : (EXP02_09.CPP)

L'exemple suivant montre les effets de bord provoqués par les macros.

```
#include <iostream>
#include <conio.h>
using namespace std;

#define toupper(x) ((x) >= 'a' && ((x) <= 'z') ? ((x)-('a'-'A')):(x))

int main()
{
    char ch = toupper( getch() );
    cout << ch << endl;
    return 0;
}
```

Le programme est censé lire un seul caractère, le transformer en majuscule avec la macro 'toupper', puis l'afficher. Or, le programme ne s'arrête qu'après avoir lu le troisième caractère (les deux premiers caractères semblent être ignorés).

En effet, le compilateur remplace la macro par sa valeur en substituant l'argument 'x' par le paramètre 'getch()'. Donc le premier caractère est comparé avec 'a', et le deuxième avec 'z', puis le troisième est utilisé pour évaluer l'expression (suivant le test) qui va être affecté à 'ch'.

Les effets de bords disparaissent, en remplaçant la macro par la fonction :

```
inline char toupper(char a)
{
    return ((a) >= 'a' && ((a) <= 'z') ? ((a)-('a'-'A')):(a));
```

II.4 Allocation dynamique

En C, la manipulation dynamique de la mémoire se fait avec les routines ***malloc*** et ***free*** (**<stdlib.h>**). En C++, ces fonctions sont remplacées avantageusement par les opérateurs unaires ***new*** et ***delete***.

Exemple:

```
// Allocation d'une variable et d'un tableau en C
    int *pi = malloc(sizeof(int));
    int *tab = malloc(sizeof(int) * 10);
// Libération
    if ((pi != NULL) && (tab != NULL)) {
        free(pi);
        free(tab);
    }
// Allocation d'une variable et d'un tableau en C++
    int *pi = new int;
    int *tab = new int[10];
// Libération
    if ((pi != NULL) && (tab != NULL)) {
        delete pi;
        delete [] tab;
    }
```

Syntaxes:

| |
|----------------------------|
| <i>new nom_type</i> |
|----------------------------|

Permet d'allouer dynamiquement un espace mémoire nécessaire pour une variable de type ***nom_type*** (nom d'un type quelconque).

`delete adresse`

Permet de libérer l'emplacement mémoire désigné par **adresse**, où **adresse** est une expression devant avoir comme valeur un pointeur sur un emplacement alloué par **new**.

`new nom_type [n]`

Permet d'allouer dynamiquement un espace mémoire nécessaire pour un tableau de **n** éléments de type **nom_type**.

`delete [] adresse`

Permet de libérer l'emplacement mémoire désigné par **adresse** alloué préalablement par **new[]**

Si l'allocation avec l'opérateur **new** a réussi, le résultat est un pointeur sur l'emplacement correspondant, sinon le résultat est un pointeur nul (0) et l'exception **bad_alloc** est générée.

Le traitement de manque de mémoire passe alors par la gestion de cette exception. A part le traitement normal des exceptions (voir exceptions), il est possible :

- soit d'éviter la génération de cette exception, en utilisant la syntaxe suivante :

`new (nothrow) nom_type`

et dans ce cas il faut tester la valeur renournée pour s'assurer que l'allocation a bien réussi.

- soit d'utiliser la fonction **set_new_handler (<new>)** pour définir la fonction qui sera appelée lors de l'échec d'allocation.

`new_handler set_new_handler (new_handler fonc_gest)`

où **fonc_gest** est une fonction qui ne prend aucun argument et qui retourne **void** et **new_handler** est un type de pointeur sur ce type de fonctions.

Exemple 2.10 : (EXP02_10.CPP)

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

#define GIGA 1024*1024*1024

void gest_debordement () {
    cout << "Memoire insuffisante - arret du programme \n";
    exit(1);
}

int main () {
    set_new_handler(gest_debordement);
    int n;
```

```
cout << "Entrer le nombre de giga à allouer...";  
cin >> n;  
char* p = new char [n * GIGA];  
cout << "Réussi ... \n";  
delete[] p;  
return 0;  
}
```


Chapitre III

Classes et Objets

- En C++, la notion de classe est une extension de la notion de type défini par l'utilisateur dans laquelle se trouvent associées des variables et des fonctions.
- Les variables et les fonctions d'une classe sont dites les **membres** de cette classe. On dit aussi :
 - ◆ **Membres données** ou propriétés pour les variables
 - ◆ **Fonctions membres** ou méthodes pour les fonctions
- La POO pure exige l'encapsulation des données, c'est-à-dire que les membres données ne peuvent être manipulés qu'à travers les fonctions membres. En C++, on peut n'encapsuler qu'une partie des membres données.
- C++ traite les structures et les unions comme des classes. Ainsi C++ permet d'associer des fonctions membres à des structures et des unions, mais dans ce cas aucune encapsulation n'est possible (tous les membres sont publiques). D'autre part, dans les unions on ne peut définir qu'une seule fonction membre.

III.1 Définition d'une classe

- La déclaration d'une classe consiste à décrire ses membres (membres données et prototypes de ses fonctions membres) groupés en sections. Chaque section est étiquetée par l'un des mots clés : **private**, **public**, ou **protected**, qui précisent le mode d'accès aux membres contenus dans la section.
 - ◆ **private** : accès autorisé seulement dans fonction membres
 - ◆ **public** : accès libre

- ***protected*** : accès autorisé seulement dans les fonctions membres de la classe et de ses dérivées (voir héritage)
- Pour déclarer une classe, on utilise le mot clé **class**.

Considérons, par exemple, la classe *point* qui comporte deux membres données privés *x* et *y* de type *int* et trois fonctions membres publiques *initialise*, *deplace* et *affiche*.

```
class point
{
private:
    int x;
    int y;
public:
    void initialise(int,int);      // initialise un point
    void deplace(int,int);        // déplace le point
    void affiche();               // affiche les coordonnées
};
```

- La mention d'accès par défaut dans une classe est **private**:

```
class point
{
    // membres privés
    int x;
    int y;
public:           // membres publiques
    ....
};
```

- Il n'est pas nécessaire de regrouper tous les membres d'un même type d'accès dans une même section. Une déclaration de type :

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

est acceptable.

- La définition d'une classe consiste à définir les fonctions membres. Pour ce, on utilise l'opérateur de résolution de portée (**::**) :

```
type_name class_name::fct_name(arguments) {.....}
```

- Au sein de la définition d'une fonction membre, les autres membres (privés ou publiques) sont directement accessibles (il n'est pas nécessaire de préciser le nom de la classe):

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

III.2 Utilisation d'une classe

- Un objet (ou instance) *object_name* d'une classe nommée *class_name* est déclaré comme une variable de type *class_name* :

```
class_name object_name;
```

- On peut accéder à n'importe quel membre public d'une classe en utilisant l'opérateur *(.)* (point). Par exemple:

```
point a;
a.initialise(10,12); //appel de la fct membre initialise de la classe 'point'
a.affiche();
```

Exemple 3.1 : (EXP03_01.CPP)

Exemple complet de déclaration, définition et utilisation de la classe ‘point’.

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

// Déclaration de la classe point
class point {
    /* déclaration des membres privés */
    int x;
    int y;
    /* déclaration des membres publics */
public:
    void initialise(int,int);
    void deplace(int,int);
    void affiche();
};

// Définition des fonctions membres de la classe point
void point::initialise(int abs, int ord){
    x=abs; y=ord;
}

void point::deplace(int dx, int dy){
    x=x+dx; y=y+dy;
}

void point::affiche(){
    cout << "(" << x << "," << y << ")" << endl;
}

// exemple d'utilisation
int main()
{
    point a,b;
    a.initialise(5,2);
    cout << "a = "; a.affiche();
    a.deplace(-2,4);
    cout << "Apres deplacement a = "; a.affiche();
    b.initialise(1,-1);
    cout << "b = "; b.affiche();
    return 0;
}
```

► Sortie du programme

```
a = (5,2)
Apres deplacement a = (3,6)
b = (1,-1)
Appuyez sur une touche pour continuer...
```

III.3 Affectation entre objets

- C++ autorise l'affectation d'un objet d'un type donnée à un autre objet de même type. Dans ce cas il recopie tout simplement les valeurs des membres données (privés ou publiques) de l'un dans l'autre.

Exemple 3.2 : (EXP03_02.CPP)

Exemple d'affectation entre objets

```
point a,b;           // déclare deux instance de la classe point

a.initialise(2,5) // a.x = 2 et a.y = 5
b = a;             // b.x = 2 et b.y = 5
```

- Il faut noter que les pointeurs ne sont pas pris en considération dans un cas simple d'affectation : Si parmi les membres données, se trouve un pointeur, l'emplacement pointé ne sera pas recopié (voir surcharge des opérateurs)

III.4 Constructeur et destructeur

- Un objet suit les règles habituelles concernant les variables (lors de la déclaration, un emplacement mémoire lui est réservé et à la fin de sa durée de vie l'emplacement est libéré), par suite, seuls les objets statiques voient leurs données initialisées à zéro. C++ permet l'utilisation de deux fonctions membres dites **constructeur** et **destructeur** qui sont implicitement appelées respectivement lors de la création et la destruction d'un objet.
- Le constructeur est une fonction membre qui porte le même nom que sa classe. Ce constructeur est appelé après l'allocation de l'espace mémoire destiné à l'objet.

Exemple 3.3 : (EXP03_03.CPP)

Exemple d'une classe avec un constructeur

```
class point
{
    int x;
    int y;
public:
    point(int,int);           // constructeur
    ...
};

// Définition du constructeur
point::point(int abs, int ord)
{
    x = abs;
    y = ord;
}
```

- Pour déclarer une instance d'une classe ayant un constructeur, on doit spécifier les valeurs des arguments requis par le constructeur.

```
point a(2,5);
point b;           // erreur
```

- Même si le constructeur ne retourne aucune valeur, il est déclaré et défini sans le mot clé **void**
- Le destructeur est une fonction membre qui porte le même nom que sa classe, précédé du symbole (~). Le destructeur est appelé avant la libération de l'espace associé à l'objet.

```
class point
{
    int x;
    int y;
public:
    point(int,int);           // constructeur
    ~point();                 // destructeur
    ...
};
```

La définition du destructeur sera de la forme:

```
point::~point() { ... ... }
```

- Le destructeur est une fonction qui ne prend aucun argument et ne renvoie aucune valeur.
- En pratique les destructeurs sont utilisés pour libérer d'éventuels emplacements mémoire occupée par des membres données.

Exemple 3.4 : (EXP03_04.CPP)

Considérons par exemple une classe *tab_entiers*, qui permet de traiter des tableaux d'entiers dont les dimensions sont fournies en données. Le constructeur aura donc la tache d'allouer dynamiquement l'espace mémoire nécessaire pour l'instance à créer et le destructeur doit libérer cette zone.

```
class tab_entiers
{
    int nb;
    int * tab;
public:
    tab_entiers(int);           // constructeur
    ~tab_entiers();             // destructeur
    ....
};
/*----- définition du constructeur -----*/
tab_entiers::tab_entiers(int n) {
    nb = n;
    tab = new int [nb];
}
/*----- définition du destructeur -----*/
tab_entiers::~tab_entiers(){
    delete[] tab;
}
```

III.5 Membres données statiques

- Un membre donnée déclaré avec l'attribut **static** est une donnée partagée par tous les instances d'une classe.
- Un membre donnée statique est initialisé par défaut à zéro. Mais :
 - ◆ Il doit être défini à l'extérieur de la déclaration de la classe, même s'il est privé, en utilisant l'opérateur de porté (::).
 - ◆ Ne peut être initialisé à l'intérieur de la classe.

Considérons, par exemple, une classe *A* dont on veut connaître à chaque instant le nombre d'instances créées. Pour ce, on munit la classe *A* d'un membre statique *nb_obj*, qu'on incrémente dans le constructeur et qu'on décrémente dans le destructeur.

```
class A
{
    static int nb_obj;
    ...
public:
    A(...);           // constructeur
    ~A(); ;
    ...
};
// définition obligatoire du membre statique
int A::nb_obj;
// constructeur & destructeur
A::A(...) { ... nb_obj++; ... }
A::~A(...) { ... nb_obj--; ... }
```

- L'accès à un membre donnée statique d'une classe suit les mêmes règles que les autres membres. D'autre part, un membre donnée statique est une donnée qui existe même si aucun objet de cette classe n'est déclaré, dans ce cas l'accès se fait à l'aide du nom de la classe et l'opérateur de porté (::).

Exemple 3.5 : (EXP03_05.CPP)

Dans l'exemple suivant on modifie la fonction *affiche()* de la classe *point* pour qu'elle affiche, en plus des coordonnées, le nombre de points créés.

```
----- Déclaration de la classe -----
class point{
    int x;
    int y;
public:
    static int nb_points; // membre donnée static
    point(int, int);
    ~point();
    void affiche();
};

----- Définition de la classe -----
// la définition du membre donnée static est obligatoire
int point::nb_points; // Son initialisation peut être faite à ce niveau

point::point(int abs, int ord){
    x = abs; y = ord;
    nb_points++;           // un point est créé
}

point::~point(){
    nb_points--;           // un point est détruit
}

void point::affiche(){
    cout << "(" << x << "," << y << ")" << endl;
    cout << "----> nombre de points : " << nb_points << endl;
}
// ----- une fonction quelconque
void f(){
    point c (2,2);
    c.affiche();
}
```

```

//----- TEST
int main()
{
    // l'accès au membre static peut se faire même si aucun objet
    // n'est encore crée
    cout << "Nombre de points : " << point::nb_points << endl;
    // appel d'une fonction qui manipule les objets de type 'point'
    f();
    // tous les objets créés par f() sont détruits
    cout << "Nombre de points après f() : " << point::nb_points << endl;
    //-----
    point a(5,2);
    cout << "a = "; a.affiche();
    point b(-2,4);
    cout << "b = "; b.affiche();
    return 0;
}

```

► Sortie du programme

```

Nombre de points : 0
(2,2)---> nombre de points : 1
Nombre de points apres f() : 0      //l'objet local à f est détruit
a = (5,2)---> nombre de points : 1
b = (-2,4)---> nombre de points : 2

```

III.6 Exploitation des classes

- Pour bien exploiter une classe (pouvoir la réutiliser, la compiler séparément, ...), il est préférable de ranger la déclaration et la définition dans des fichiers qui porte des noms corrects. Pour une classe *class_name*, par exemple, on crée les fichiers suivants:

CLASS_NAME.H: interface de la classe, contient la déclaration de la classe.

```

/*----- CLASS_NAME.H -----*/
/*----- INTERFACE -----*/
#ifndef           CLASS_NAME_H
#define            CLASS_NAME_H
...
class class_name {
    ...
};
#endif

```

CLASS_NAME.CPP: corps de la classe, contient la définition de la classe.

```

/*----- CLASS_NAME.CPP -----*/
/*----- CORPS DE LA CLASSE -----*/
#include <class_name.h>
/*----- Définition des fonctions membres -----*/
...

```

- #ifndef, #define et #endif sont utilisés dans les fichiers entêtes (.h) pour que le fichier ne soit inclus qu'une seule fois lors d'une compilation.
- Enfin, dans tout programme utilisant la classe *class_name*, on doit inclure le fichier d'entête <*class_name.h*>. Un tel programme doit aussi pouvoir accéder au module objet résultant de la compilation du fichier source contenant la définition de la classe

Exemple 3.6 :

L'exemple suivant implémente la classe **point** de l'exemple 3.3, en définissant dans des fichiers séparés l'interface, le corps de la classe et l'exemple d'utilisation.

Interface: (POINT1.H)

```
#ifndef POINT_H
#define POINT_H
class point{
    int x;
    int y;
public:
    point(int, int);
    void deplace(int,int);
    void affiche();
};
#endif
```

Corps de la classe: (POINT1.CPP)

```
#include <iostream>           // utilisé dans affiche()
#include "point1.h"
using namespace std;

point::point(int abs, int ord){
    x = abs; y = ord;
}
void point::deplace(int dx, int dy){
    x=x+dx; y=y+dy;
}
void point::affiche(){
    cout << "(" << x << "," << y << ")" << endl;
}
```

Programme test: (EXP03_06.CPP)

```
#include "point1.h"           // fichier interface de la classe

int main(){
    point a(5,2);
    ...
    return 0;
}
```

Chapitre IV

Les propriétés des fonctions membres

Toutes les possibilités offertes par C++ pour les fonctions restent valables pour les fonctions membres (surcharge, arguments par défaut, ...)

Exemple 4.1 : (EXP04_01.CPP)

On désir munir la classe *point* d'un constructeur qui permet de créer le point :

- (0,0) si aucun argument n'est fourni
- (abs,0) si on lui fournit **abs** comme seul argument
- (abs,ord) si on lui fournit les deux arguments **abs** et **ord**

Pour cela, il suffit de définir le constructeur de la classe avec des paramètres par défaut

```
class point
{
    int x; int y;
public:
    point(int = 0, int = 0);
    ...
}

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

Exemple 4.2: (EXP04_02.CPP)

On désire maintenant que le constructeur de la classe *point* crée le point

- (abs,abs) si on lui fournit un seul argument (abs)

Dans ce cas, on ne peut pas utiliser des valeurs par défaut, il faut par suite surcharger le constructeur :

```
class point
{
    int x; int y;
public:
    point();
    point(int);
    point(int,int);
    ...
}

point::point() { x = 0; y = 0; }
point::point(int abs) {x = abs; y = abs; }
point::point(int abs, int ord){ x = abs; y = ord; }
```

IV.1 Objets transmis en argument

- Considérons une classe T et une fonction F dont l'un des paramètres est un objet de T transmis par valeur, par adresse ou par référence. Soit U une instance de T transmis en argument à F, alors:
 1. Si F est une fonction membre de T, elle aura accès à tous les membres données de U, sinon elle n'aura accès qu'aux membres publics de U
 2. Si la transmission de U se fait par valeur, il y a recopie des membres données de U dans un emplacement locale à F, ce qui entraîne certaines problèmes si la classe contient des pointeurs

Exemple 4.3: (EXP04_03.CPP)

Définir une fonction qui permet de comparer deux instances de la classe point. Cette fonction devra retourner "true" si les deux objets coïncident et "false" sinon.

La comparaison nécessite l'accès aux coordonnées des points, qui sont des données privés, par suite, la fonction doit être une fonction membre de la classe

La déclaration de la fonction dans la classe sera :

```
bool coincide(point)
```

et sa définition

```
bool point::coincide(point pt)
{
    return ( (pt.x == x) && (pt.y == y));
}
```

- On peut prévoir pour cette fonction une transmission par adresse ou par référence.

IV.2 Objets fournis en valeur de retour

- Etant donné une classe T et une fonction F qui a l'une des formes suivantes :

| | |
|-------------------|---------------------------|
| T F(arguments); | // retour par valeur |
| T * F(arguments); | // retourne l'adresse |
| T & F(arguments) | // retourne une référence |

Alors F aura accès à tous les membres de l'objet retourné si elle est une fonction membre de T, si non elle n'aura accès qu'aux membres publics de la classe.

Exemple 4.4: (EXP04_04.CPP)

On désire définir la fonction **symetrique()** qui permet de retourner le symétrique d'un point.

Cette fonction doit être une fonction membre de la classe **point**, puisqu'elle doit accéder aux coordonnées du point, qui sont des données privées. La valeur de retour sera de type **point**

La déclaration de la fonction dans la classe sera :

```
point symetrique();
```

et sa définition

```
point point::symetrique(){
    point pt;
    pt.x = -x; pt.y = -y;
    return pt;
}
```

IV.3 Le pointeur this

- Dans une fonction membre, **this** représente un pointeur sur l'instance ayant appelé cette fonction
- this** est un **pointeur constant**, c'est-à-dire qu'on ne peut le modifier.

Exemple 4.5: (EXP04_05.CPP)

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

point::point(int abs, int ord) { x = abs; y = ord; }
void point::affiche()
{
    cout << " le point : (" << this->x << "," << this->y << ")" ;
    cout << " est à l'adresse →" << this << endl;
}
```

la fonction membre **affiche()** de la classe **point** affiche les coordonnées de l'objet et de son adresse.

IV.4 Fonctions membres en ligne

- Toute fonction membre définie dans sa classe (dans la déclaration de la classe) est considérée par le compilateur comme une fonction **inline**. Le mot clé **inline** n'est plus utilisé.

Dans l'exemple suivant, le constructeur de la classe point est défini en ligne

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

IV.5 Fonctions membres statiques

- On distingue deux types de membres :
 - ◆ **Membres d'instance** : membres associés à une instance de la classe.
 - ◆ **Membres de classe** : membres associés à la classe et qui ne dépendent d'aucune instance de la classe.
- Les membres de classe, sont aussi dits **membres statiques**. Ils sont déclarés avec l'attribut **static**, et existent même si aucun objet de la classe n'est créé.
- Ainsi on peut définir un membre donnée statique comme on peut définir une fonction membre statique. L'accès à ces membres se fait avec l'opérateur de résolution de portée (::) précédé par le nom de la classe ou d'un quelconque objet de la classe

Exemple 4.6 :

Dans cet exemple, on prévoit dans la classe point un membre donnée statique privé nb_obj et une fonction membre NbObj() qui retourne le nombre d'instances créées.

Interface: (POINT2.H)

```
#ifndef POINT_H
#define POINT_H

class point{
    int x;
    int y;
    static int nb_obj;
public:
    point(int = 0, int = 0);
    ~point();
    void affiche();
    static int NbObj();
};

#endif
```

Corps de la classe: (POINT2.CPP)

```
#include "point2.h"
#include <iostream>           // utilisé dans affiche()
using namespace std;

---- définition obligatoire du membre donnée statique
int point::nb_obj;
---- constructeur
point::point(int abs, int ord){
    x = abs; y = ord;
    nb_obj++;
}
---- Destructeur
point::~point(){
```

```

        nb_obj--;
}
//--- affiche
void point::affiche(){
    cout << "(" << x << "," << y << ")" << endl;
}
//--- definition de la fonction membre statique
int point::NbObj(){
    return nb_obj;
}

```

Programme test: (EXP04_06.CPP)

```

#include "point.h"
#include <iostream>
using namespace std;

int main()
{
    / Acces à la fonction membre statique avant la création des objets
    cout << "Nombre de points : " << point::NbObj() << endl;
    // Appel de la fct membre statique en utilisant un objet
    point a;
    cout << "Nombre de points : " << a.NbObj() << endl;
    return 0;
}

```

IV.6 Les fonctions membres constantes

- Les objets, comme les autres types de C++, peuvent être déclarés constants avec l'attribut **const**. Dans ce cas, seules les fonctions membres déclarées et définies avec l'attribut **const** peuvent être appelées par des objets constants.

Plus précisément :

```

class T {
    ...
public:
    ...
    type_a F(...);           // fct membre ordinaire
    type_b G(...) const;   // fct membre constante
    type_c K(const T);       // fct avec argument constant
};

// déclarations
T u;           // instance non constante
const T v;     // instance constante
// appels
u.F(...)      // OK
v.F(...)      // erreur : instance constante
u.G(...)      // OK
v.G(...)      // OK
//
T w;
u.K(v)        // OK
u.K(w)        // OK
v.K(w)        // erreur : instance constante et fct non constante

```

v est une instance constante, donc elle ne peut être appelée ni par F ni par K, qui ne sont pas des fonctions constantes

Exemple 4.7 : (EXP04_07.CPP)

Cet exemple montre la différence entre les variables simples et les objets lors de passage en arguments à des fonctions demandant des arguments constants et la différence d'une fonction membre et une fonction non membre agissant sur des objets constants.

```
#include <iostream>
using namespace std;
//-----
class point{
    int x;
    int y;
public:
    point(int abs = 0, int ord = 0){
        x = abs; y = ord;
    }
    void affiche() const;           // fct membre constante
    bool coincide(const point);   // argument constant
    void deplace(int,int);
};

void point::affiche() const{
    cout << "(" << x << "," << y << ")" << endl;
}
bool point::coincide(const point pt){
    return ( (pt.x == x) && (pt.y == y) );
}
void point::deplace(int dx, int dy){
    x += dx; y += dy;
}
//-----
// fonction avec argument constant de type simple
void fct(const int k) { cout << k << endl; }
// fonction avec argument constant de type classe
point sosie(const point pt) { return pt; }

//----- TEST
int main()
{
    // Cas d'une fonction simple avec argument de type simple
    int i = 5;
    const int j = 10;
    fct(i);                      // OK
    fct(j);                      // OK

    // -----Cas des objets
    point a;
    const point b(5,6);

    //----Appel d'une fonction non membre
    point c;
    c = sosie(a);      // OK
    c = sosie(b);      // OK

    //----Appel d'une fonction membre non constante
    a.deplace(1,1);
    //b.deplace(1,1); // erreur : instance constante, fonction non
    //constante

    //----Appel de la fonction membre constante
    cout << "a = " ; a.affiche();
}
```

```
cout << "b = " ; b.affiche();
cout << endl;

-----Appel de la fonction membre non constante avec paramètres
constants
    // instance non constant, argument constant
    cout << ( a.coincide(b) ? "coincident" : "ne coincident pas" )<< endl;
    // instance non constante, argument non constant
    cout << ( a.coincide(a) ? "coincident" : "ne coincident pas" )<< endl;
// instance constante, fonction non constante : ERREUR
//cout << ( b.coincide(a) ? "coincident" : "ne coincident pas" )<< endl;

    return 0;
}
```


Chapitre V

Construction, destruction et initialisation des objets

V.1 Classes de stockage

- Toute variable dans un programme à :
 - ♦ **une portée** : la zone du programme où elle est accessible
 - ♦ **une durée de vie** : le temps pendant lequel elle existe
- L'utilisation d'une variable nécessite une déclaration et notamment une initialisation (ou définition)
- Une variable peut être initialisée implicitement (par le compilateur) ou explicitement lors de sa déclaration ou initialisée dans une autre instruction autre que sa déclaration. Les pointeurs ne sont pas initialisés implicitement (par défaut)
- On distingue deux types de variables : Globales et Locales

Les variables globales:

- Toute variable déclarée en dehors de tout bloc d'instructions est dite globale:
 - ♦ sa durée de vie est la durée de vie du programme.
 - ♦ sa portée commence par la ligne de sa déclaration jusqu'à la fin du fichier.
- Une variable globale peut être déclarée de deux manières :
 - ♦ Sans aucun modificateur : dans ce cas sa déclaration constitue aussi sa définition

- ♦ Avec le modificateur **extern** : dans ce cas sa définition se trouve dans un autre fichier (programmation modulaire) ou dans le même fichier (sur une ligne dans la suite du fichier)
- Si une variable globale est déclarée avec le modificateur **static**, sa portée sera restreinte au fichier contenant sa déclaration (elle ne peut être utilisée dans d'autres fichiers).
- Si une variable globale n'est pas initialisée explicitement, elle sera implicitement initialisée à 0 par le compilateur.

Exemple 5.1 :

L'exemple suivant, montre le comportement des variables globales.

► Fichier 2 : (EXP05_01b.CPP)

```
int ig2 = 5;           // variable globale
static int igs2;     // variable globale statique
// fonction normale
void fn_2(){
    cout << "Fonction Normale de Fichier 2" << endl;
}
// fonction statique
static void fs_2(){
    cout << "Fonction statique de Fichier 2" << endl;
}
```

► Fichier 1 : (EXP05_01a.CPP)

```
// Déclarations des variables
extern int ig1;      // var globale définie ultérieurement dans le même fichier
extern int ig2;      // var globale définie dans Fichier 2
extern int igs2;     // var globale définie dans Fichier 2 avec l'attribut static

// Déclaration des fonctions définies dans Fichier 2
void fn_2();
void fs_2();

int main()
{
    /** VARIABLES GLOBALES */
    cout << "ig1 = " << ++ig1 << endl;          // --> ig1 = 1
    cout << "ig2 = " << ++ig2 << endl;          // --> ig2 = 6
    /** VARIABLES GLOBALES STATIQUES*/
    cout << "igs2 = " << igs2 << endl;        // erreur : igs2 est statique
    /** FONCTIONS */
    fn_2();           // définie dans Fichier 2, déclarée dans Fichier 1
    /** FONCTIONS STATIQUES*/
    fs_2();           // erreur : fs_2() est statique

    return 0;
}
int ig1 ;           // définition de ig1
```

► Sortie du programme :

```
ig1 = 1
ig2 = 6
Fonction Normale de Fichier 2
```

ig1 est une variable globale définie à la fin du Fichier 1, sa déclaration avec le mot **extern** au début du fichier permet son utilisation. D'autre part, puisque ig1 n'est pas initialisée explicitement, elle sera initialisée par défaut à 0.

ig2 et igs2 sont des variables globales définies dans Fichier 2, leurs déclarations avec le mot **extern**, étend leur portée à Fichier 1, mais la déclaration de igs2 dans Fichier 2 avec le mot clé **static**, interdit son utilisation à l'extérieur de ce fichier.

Les fonctions **fn_2()** et **fs_2()** sont définies dans Fichier 2, leurs utilisations dans Fichier 1 nécessitent des déclaration. La définition de **fs_2()** avec le mots clé **static** interdit son utilisation à l'extérieur du Fichier2.

Les variables locales

- Toute variable déclarée dans un bloc (au sein d'une fonction) est dite locale, sa portée commence par la ligne de sa déclaration et se termine à la fin du bloc
- Sa durée de vie dépend de sa classe de stockage :
 - ◆ **auto** : classe de stockage par défaut, la durée de vie est la même que celle du bloc
 - ◆ **static** : la durée de vie commence lors du premier passage par le bloc jusqu'à la fin du programme
 - ◆ **register** : variable dont l'emplacement se trouve dans un registre
 - ◆ **extern** : variable déclarée comme variable globale dans un autre fichier. Sa durée de vie est donc la même que celle du programme
 - ◆ **volatile** : variable qui peut être modifiée par un autre processus
- Seules les variables locales statiques sont initialisées par défaut par le compilateur.

Exemple 5.2 : (EXP05_02.CPP)

L'exemple suivant, montre le comportement des variables locales statiques.

```
// Fonction contenant une variable locale statique
void f(int appel){
    static int is;
    cout << "appel = " << appel << "    is = " << ++is << endl;
}

int main()
{
    // Les variables locales statiques sont implicitement initialisées
    // et gardent leurs valeurs (durée de vie est celle du programme)
    for(int i = 0; i < 3; i++)
        f(i);

    return 0;
}
```

►Sortie du programme :

```
appel = 0    is = 1
appel = 1    is = 2
appel = 2    is = 3
```

La variable locale statique **is** est implicitement initialisée à 0 et garde sa valeur (sa durée de vie est celle du programme).

V.2 Déclaration et initialisation des objets

- Les objets suivent les mêmes règles que les variables ordinaires, nous distinguerons alors :
 - ◆ Les objets globaux : ceux qui sont déclarés en dehors de tout bloc.
 - ◆ Les objets automatiques : ceux qui sont déclarés au sein d'un bloc
 - ◆ Les objets statiques : ce sont les objets locaux statiques (définie avec le mot clé **static**)

Cas d'une classe sans constructeur

- Les données membres des objets globaux et statiques seront initialisées par défaut à 0.

- L'initialisation explicite des objets lors d'une déclaration ne peut se faire qu'avec des objets de même type, comme par exemple :

```
T a;           //Déclaration de a
T b = a;       // Déclaration et initialisation de b
```

où T est une classe.

Dans ce cas, le compilateur fera une copie simple des valeurs des membres données de a dans ceux de b.

- Notez que, l'utilisation d'une classe sans constructeur et comportant un pointeur est fortement déconseillée, du fait qu'aucune initialisation implicite ne sera parfaite et que lors de l'initialisation d'un objet par un autre, les deux objets concernés partageront la même zone mémoire pointée par le pointeur

Exemple 5.3 :

Les exemples suivants, montrent le comportement des objets globaux, statiques et automatiques d'une classe sans constructeur, lors d'une initialisation implicite.

Cas d'une classe avec des membres de type simple et un tableau: (EXP05_03a.CPP)

```
class A{
    float x;
    char c;
    int t[4];
public:
    void initialise(float xx , char cc ){
        x = xx;
        c = cc;
        for(int i = 0; i<5; i++)      t[i] = i;
    }
    void affiche(){
        cout << " Reel      : " << x << endl;
        cout << " Caractere : " << c << endl;
        cout << " Tableau   : " ;
        for(int i = 0; i<5; i++)      cout << "[" << t[i] << "] ";
        cout << endl;
    }
};
```

►Programme test :

```
A ag;          // Un Objet global
// Une fonction comportant un objet statique
void fct(){
    static A as;
    static int appel;
    cout << "--- Appel : " << ++appel << endl;
    as.affiche();
    as.initialise(0.1, 'A');
}
int main(){
    cout << "** Objets globaux" << endl;
    ag.affiche();
    cout << "\n** Objets statiques" << endl;
    fct();      // passage 1
    fct();      // passage 2
    cout << "\n** Objets automatiques" << endl;
    A a ;        //objet locale non initialisé
    a.affiche(); //--> résultat aléatoire
    return 0;
}
```

►Sortie du programme :

```
** Objets globaux
Reel      : 0
Caractere :
Tableau   : [0] [0] [0] [0] [0]

** Objets statiques
--- Appel : 1
Reel      : 0
Caractere :
Tableau   : [0] [0] [0] [0] [0]
--- Appel : 2
Reel      : 0.1
Caractere : A
Tableau   : [0] [1] [2] [3] [4]

** Objets automatiques
Reel      : 7.80085e+033
Caractere :
Tableau   : [2293624] [4247142] [4247040] [4] [0]
```

Les membres de type simple et les éléments du tableau des objets globaux et statiques sont initialisés par défaut à 0, ceux des objets automatiques non initialisés fournissent un résultat aléatoire. De plus les objets locaux statiques gardent leurs valeurs.

Cas d'une classe comportant un pointeur : (EXP05_03b.CPP)

```
class B{
    int taille;
    int *pi;
public:
    void initialise(int n = 0){
        taille = n;
        if (taille == 0) pi = NULL ;
        else {
            pi = new int[taille];
            *pi = 0;
        }
    }
    void affiche(){
        cout << " -->[" << pi << "] - " << *pi << endl;
    }
};
```

►Programme test :

```
// Un Objet global
B bg;
// Une fonction comportant un objet statique
void fct(){
    static B bs;
    bs.affiche();
}
//-----
int main()
{
    bg.affiche(); // Erreur d'exécution
    fct(); // Erreur d'exécution
    B b ;
    b.initialise(4);
    b.affiche();
    return 0;
}
```

► Sortie du programme :

```
-->[ 0x3e3dd8 ] - 0
```

Toute utilisation d'objets de cette classe, non initialisés explicitement, conduit à une erreur d'exécution du programme (même si les membres des objets globaux et statiques sont initialisés par défaut). Seul l'objet b qui est initialisé sera affiché

Cas d'une classe avec constructeur

- Pour une telle classe, toute déclaration doit comporter une initialisation. Ainsi, pour déclarer par exemple un objet u d'une classe T, on doit utiliser l'une des syntaxes suivantes :

Syntaxe 1:

```
T u = v;
```

Syntaxe 2:

```
T u(liste_des_valeurs);
```

Syntaxe 3:

```
T u = T(liste_des_valeurs);
```

Dans *syntaxe 1*, v est un objet de la classe T (défini antérieurement).

Les syntaxes 2 et 3 sont équivalentes. Dans ces deux formes, *liste_des_valeurs* est une liste de valeurs qui correspond à la liste des arguments du constructeur, en tenant compte des valeurs par défaut définies par le constructeur et des surcharges éventuelles du constructeur (autrement dit, T(*liste_des_valeurs*) doit être un appel valable de la fonction membre T). Par exemple :

```
class T{
    int i; char c;
public:
    T(int n, char cc = 'a'){ i=n; c = cc; }
.....
};

T a(5, 'A');           // ou T a = T(5,'A');
T b = T(3);           // T b(3) est équivalent à T b(3,'a');
T c = a;
```

- Par ailleurs, notons les cas particulier suivants :
 - Si tous les arguments du constructeur ont une valeur par défaut, alors :


```
T u();      et      T u;
```

 seront équivalentes
 - Si la classe ne comporte qu'un seul membre donnée de type de base, alors


```
T u(valeur);      et      T u = valeur;
```

 seront équivalentes. Dans la deuxième, il y aura une conversion implicite vers T

```
class T{
    float z;
public:
    T(float x = 0.0){ z = x; }
};

// Les déclarations suivantes sont équivalentes
T u = 1.2;           // conversion implicite float → T
T u(1.2);
```

V.3 Constructeur par recopie

- Considérons une classe T, dont l'un des membres données est un pointeur nommé adr. Cette classe doit alors comporter un constructeur qui alloue dynamiquement une zone mémoire à ce pointeur pour l'initialiser et un destructeur qui libère cette zone mémoire lors de destruction des objets. Dans ce cas, si

```
T u = v;
```

Et v est un objet de classe T, il y aura une copie simple des valeurs des membres données de v dans ceux de u. Par suite, les pointeurs u.adr et v.adr désigneront la même adresse, ce qui posera deux problèmes :

- Toute modification contenu de *adr de l'un des objets se fera aussi pour l'autre.
- La destruction de l'un des objets, entraînera la destruction du pointeur du deuxième.

Exemple 5.4: (EXP05_04.CPP)

Cet exemple, met en évidence les problèmes cités.

On considère la classe :

```
class T{
    int *adr;
public:
    T(int val = 0){
        adr = new int;
        *adr = val;
    }
    ~T(){
        delete adr;
    }
    void affiche(){
        cout << *adr << " -->[" << adr << "] " << endl;
    }
    void modifier(int k){
        *adr = k;
    }
};
```

►Programme test :

```
int main(){
    T u(1);
    cout << "u : ";u.affiche();
    // initialisation d'un objet avec un autre
    T v = u;
    cout << "v : ";v.affiche();
    // on modifie v
    cout << "\n---- Modification de v" << endl;
    v.modifier(2);
    cout << "v : ";v.affiche();
    cout << "u : ";u.affiche();
    return 0;
}
```

►Sortie du programme :

```
u : 1 -->[0x3e3d58]
v : 1 -->[0x3e3d58]
----- Modification de v
v : 2 -->[0x3e3d58]
u : 2 -->[0x3e3d58]
```

Les pointeurs de u et v désignent le même emplacement mémoire [0x3e3d58], ainsi la modification de la valeur en v entraîne un changement dans u.

- Pour résoudre ce type de problèmes, C++ offre la possibilité de définir un constructeur particulier approprié à ce genre de situation. Ce constructeur est appelé constructeur par recopie (*copy constructor*) et il est déclaré sous l'une des formes suivantes :

```
T ( T & );
T (const T &);
```

- Le code du constructeur par recopie doit contenir les instructions nécessaires pour créer un nouvel objet à partir de l'objet passé en paramètre.

Exemple 5.5: (EXP05_05.CPP)

On ajoute à la classe T de l'exemple 5.4, un constructeur par recopie défini comme suit :

```
T(const T & u){
    adr = new int;
    *adr = *(u.adr);
}
```

Avec le même programme test que dans l'exemple 5.4, et on obtient :

```
u : 1 -->[0x3e3d58]
v : 1 -->[0x3e3dd8]

----- Modification de v
v : 2 -->[0x3e3dd8]
u : 1 -->[0x3e3d58]
```

Les deux instances u et v ne partagent plus le même emplacement mémoire, ce qui entraîne que le changement de v n'affecte pas u. Les problèmes posés par l'initialisation d'un objet par un autre sont bien résolus.

V.4 Appel des constructeurs

Appel implicite

Lors de la création d'un objet et suivant le contexte, le compilateur fait appel au constructeur ou au constructeur par recopie d'une classe et à défaut, il fait appel à des constructeurs par défaut et ce de la manière suivante :

- Lors d'une simple déclaration :

```
T u(liste_des_valeurs);
T u = T(liste_des_valeurs);
```

il y a appel du constructeur de la classe et à défaut, les objets seront considérés comme des variables simples, donc leurs initialisations dépendra du type des champs de la classe.

- Lors d'une initialisation d'un objet par un autre objet de même type :

```
T u = v;
```

il y a appel du constructeur par recopie et à défaut, il y a appel d'un constructeur de recopie par défaut, qui recopie les membres données de l'objet comme le fait une affectation.

- Lors d'une transmission d'un objet par valeur en argument d'une fonction ou lors d'une transmission par valeur en valeur de retour d'une fonction : il y a appel du constructeur par recopie et à défaut il y a appel du constructeur de recopie par défaut.
- Lors de la création d'un objet dynamique avec l'opérateur `new`, il y a appel du constructeur et à défaut il y a appel du constructeur par défaut comme pour une déclaration simple, par exemple:

```
T * adr;
adr = new T(liste_des_valeurs);
```

où `liste_des_valeurs` est une liste de valeurs valable pour l'appel du constructeur

Exemple 5.6: (EXP05_06.CPP)

L'exemple suivant montre les différents cas cités de l'appel du constructeur.

On considère la classe suivante :

```
class T{
    int *pi;
public:
    T( int = 0);           // constructeur
    T( const T &);        // constructeur par recopie
    ~T();                  // destructeur
    T sosie();             // retourne l'objet en cours
    void affiche();         // affiche les informations de l'objet
};
```

On introduit des messages qui affichent les informations sur l'objet (`pi`, `*pi` et l'adresse de l'objet) dans les deux constructeurs et dans le destructeur, ce qui permettra de voir laquelle de ces fonctions sera appelée.

La fonction `sosie()` permettra de tester le retour d'un objet par valeur.

```
// ----- Définition des fonctions membres
T::T(int n){
    cout << "+++++ Constructeur : ";
    pi = new int;
    *pi = n;
    cout << "( " << *pi << " , " << pi << " )-->[ " << this << " ]" << endl;
}

T::T(const T & v){
    cout << "***** Constr. par recopie : ";
    pi = new int;
    *pi = *(v.pi);
    cout << "( " << *pi << " , " << pi << " )-->[ " << this << " ]" << endl;
}

T::~T(){
    if(pi != NULL){
        cout << "----- Destructeur : ";
        cout << "( " << *pi << " , " << pi << " )-->[ " << this << " ]" << endl;
    }
}
```

```

        delete pi;
    }
}
T T::sosie(){
    return *this;
}

void T::affiche(){
    cout << " ( " << *pi << " , " << pi << " )-->[ " << this << " ]" << endl;
}

```

On définit ensuite deux fonctions de tester les cas de transmission d'un objet par valeur et par référence :

```

---- fonction admettant un argument de type T
void fct1(T t){
    t.affiche();
}
---- fonction admettant un argument de type T &
void fct2(T &t){
    t.affiche();
}

```

Puis un programme test, dans lequel, on teste tous les cas y compris le cas de création d'un objet dynamique

```

int main(){
    cout << "\n----- Déclaration simple" << endl;
    T u(1);
    cout << "\n----- Initialisation d'un objet avec un autre" << endl;
    T v = u;
    // appel d'une fonction dont l'argument est un objet transmis par valeur
    cout << "\n----- Objet transmis par valeur" << endl;
    fct1(u);
    // appel d'une fonction dont l'argument est un objet transmis par
référence
    cout << "\n----- Objet transmis par référence" << endl;
    fct2(u);
    // appel d'une fonction dont la valeur de retour est un objet
    cout << "\n----- Objet comme valeur de retour" << endl;
    u.sosie();
    cout << "\n----- Objet dynamique" << endl;
    T * pt;
    pt = new T(2);
    delete pt;
    cout << "\n----- FIN" << endl;
    return 0;
}

```

►Sortie du programme :

```

----- Declaration simple
+++++ Constructeur : ( 1 , 0x3e3d58 )-->[ 0x22ff50 ]
----- Initialisation d'un objet avec un autre
***** Constr. par recopie : ( 1 , 0x3e3dd8 )-->[ 0x22ff40 ]
----- Objet transmis par valeur
***** Constr. par recopie : ( 1 , 0x3e3de8 )-->[ 0x22ff30 ]
( 1 , 0x3e3de8 )-->[ 0x22ff30 ]
----- Destructeur : ( 1 , 0x3e3de8 )-->[ 0x22ff30 ]
----- Objet transmis par reference
( 1 , 0x3e3d58 )-->[ 0x22ff50 ]
----- Objet comme valeur de retour
***** Constr. par recopie : ( 1 , 0x3e3de8 )-->[ 0x22ff30 ]

```

```

----- Destructeur : ( 1 , 0x3e3de8 )-->[ 0x22ff30 ]
----- Objet dynamique
+++++ Constructeur : ( 2 , 0x3e3df8 )-->[ 0x3e3de8 ]
----- Destructeur : ( 2 , 0x3e3df8 )-->[ 0x3e3de8 ]

----- FIN
----- Destructeur : ( 1 , 0x3e3dd8 )-->[ 0x22ff40 ]
----- Destructeur : ( 1 , 0x3e3d58 )-->[ 0x22ff50 ]

```

Appel explicite d'un constructeur

- On peut appeler explicitement un constructeur de la classe au sein d'une expression, dans ce cas un objet temporaire est créé. Cet objet temporaire est automatiquement détruit lorsqu'il n'est plus utile. Si, par exemple, T est une classe qui admet un constructeur et si u est objet de T, nous pourrons écrire :

```
u = T(liste_des_valeurs);
```

et si T admet un constructeur par recopie, nous pouvons écrire

```
v = T(u);
```

Affectation et initialisation

- Notez bien qu'il y a une différence entre une initialisation et une affectation. Autrement dit:
- $$T\ u = v; \quad \text{et} \quad u = v;$$
- sont différents. La première est une initialisation qui fait appel au constructeur par recopie (s'il existe) et la deuxième est une affectation, qui permet la copie des membres données sans faire appel au constructeur par recopie.
- L'affectation pose certains problèmes, dans le cas où la classe contient des pointeurs. Ces problèmes seront résolus par la surcharge de l'opérateur = (voir surcharge des opérateurs).

V.5 Tableaux d'objets

- En théorie, un tableau peut posséder des éléments de n'importe quel type. Ainsi nous pouvons déclarer un tableau de N objets d'une classe T par :

```
T tab[N];
```

Or, du fait qu'on ne peu déclarer un objet sans l'initialiser, cette déclaration ne sera possible que si la classe T admet un constructeur sans arguments (ou un constructeur dont tous les arguments ont des valeurs par défaut).

Notons que pour initialiser un tableau d'objets on procède comme pour l'initialisation des tableaux classiques :

```
T tab[] = { T(...), T(...), ... };
```

- Ces mêmes remarques s'appliquent pour les tableaux dynamiques d'objets. Une déclaration de type :

```
T * adr = new T[N];
```

nécessite aussi un constructeur sans arguments.

V.6 Objets d'objets

- Une classe peut comporter un membre donnée de type classe. Considérons alors la situation suivante, où la classe T comporte un membre donnée de type classe A :

```
class A{
.....
public:
    A(liste_arguments_a);
    ...
};

class T{
    A a;
    ...
public:
    T(liste_arguments);
    ...
};
```

Lors de la création d'un objet de type T, il y aura appel du constructeur de T puis un appel du constructeur de A, car la création d'un objet nécessite la définition de tous les membres données de l'objet (en particulier le membre a de type A).

Par suite, dans ce cas, la création d'un objet de type T ne sera possible que si A possède un constructeur sans arguments.

- Pour gérer ce genre de situation, C++ nous offre la possibilité de mentionner, dans la définition du constructeur de T, la liste des arguments à fournir au constructeur de A et ce de la manière suivante :

```
T::T (liste_arguments) : a (liste_arguments_a){ ... }
```

où liste_arguments_a est une sous liste de liste_arguments, valable pour le constructeur de A.

Exemple 5.7: (EXP05_07.CPP)

Exemple d'une classe (pointcol) dont l'un des membres est un objet de type classe (point).

```
---- classe 'point'
class point {
    int x, y;
public:
    point(int abs=0,int ord=0){
        x=abs; y=ord;
        cout << "++ Const. point : (" << x << "," << y << ")"
            << " - Adresse : " << this << "\n";
    }
    ~point(){
        cout << "-- Dest. point : (" << x << "," << y << ")"
            << " - Adresse : " << this << "\n";
    }
};
---- classe 'pointcol'
class pointcol {
    point p;
    int couleur;
public:
    pointcol(int,int,int);
    ~pointcol();
```

```

};

pointcol::pointcol(int abs, int ord, int coul):p(abs,ord){
    couleur = coul;
    cout << "++ Const. pointcol " << couleur << "\n";
}
pointcol::~pointcol(){
    cout << "-- Dest. pointcol " << couleur << "\n";
}
// exemple d'utilisation
int main()
{
    pointcol a(1,2,9);
    return 0;
}

```

► *Sortie du programme :*

```

++ Const. point : (1,2) - Adresse : 0x22ff60
++ Const. pointcol 9
-- Dest. pointcol 9
-- Dest. point : (1,2) - Adresse : 0x22ff60

```

Ce programme montre que lors de la construction d'un objet `pointcol`, il y a premièrement appel du constructeur de `point`, puis l'appel du constructeur de `pointcol`. Mais lors de la destruction de l'objet, il y a appel du destructeur de `pointcol` avant l'appel du destructeur de `point`.

- Si par ailleurs une classe comporte plusieurs membres données de type classe, les arguments à passer aux différents constructeurs sont séparés par des virgules, comme dans l'exemple suivant :

```

class A { ... ... };
class B { ... ... };
class T{
    A a;
    B b;
    A c;
    ...
public:
    T(liste_arguments);
    ...
};

T::T (liste_arguments) : a(liste_arguments_a), b(liste_arguments_b),
c(liste_arguments_c)
{ ... ... }

```

- L'appel du constructeur par recopie d'une classe T comportant des membres données de type classe A, n'entraîne pas automatiquement l'appel du constructeur par recopie de la classe A. Le constructeur de la classe A appelé est celui qui sera mentionné dans la définition du constructeur par recopie de T. (La classe A peut ne pas avoir un constructeur par recopie).

Considérons l'exemple suivant, où A est une classe muni d'un constructeur et d'un constructeur par recopie :

```
class A{...};
class T{
    A a;
    ...
public:
    T(liste_arguments);
    T( T & );
    ...
};
```

Le constructeur de T peut être défini par deux manières :

```
T::T( T & t): a(t.a){... ...}
```

ou

```
T::T( T & t): a(liste_arguments){... ...}
```

Nous donnerons, ici deux exemples, reflétant ces deux situations. Mais remarquons, qu'il est naturel et vivement conseillé d'utiliser la première possibilité à savoir que le constructeur par recopie de la classe T appelle le constructeur par recopie de la classe A.

Exemple 5.8: (EXP05_08.CPP)

On ajoute à la classe pointcol, de l'exemple 5.7, un constructeur par recopie qui fait appel au constructeur normal de la classe point, comme suit :

```
pointcol::pointcol(const pointcol & pc):p(0,0){
    couleur = pc.couleur;
    cout << "++ Const.Recopie pointcol " << couleur << "\n";
}
```

►Programme test :

```
int main()
{
    pointcol a(1,2,9);
    cout << "CONSTRUCTION D'UN OBJET PAR UN AUTRE\n";
    pointcol b = a;
    cout << "FIN\n";
    return 0;
}
```

►Sortie du programme :

```
++ Const. point : (1,2) - Adresse : 0x22ff50
++ Const. pointcol 9
CONSTRUCTION D'UN OBJET PAR UN AUTRE
++ Const. point : (0,0) - Adresse : 0x22ff40
++ Const.Recopie pointcol 9
FIN
-- Dest. pointcol 9
-- Dest. point : (0,0) - Adresse : 0x22ff40
-- Dest. pointcol 9
-- Dest. point : (1,2) - Adresse : 0x22ff50
```

Exemple 5.9: (EXP05_09.CPP)

Exemple d'une classe étudiant dont deux de ses membres sont de type classe chaîne. Les deux classes comportent un constructeur par recopie.

```
#include <iostream>
#include <string.h>
using namespace std;
```

```

// ----- la classe chaine
class chaine{
    char * s;
public:
    chaine(const char * = NULL);
    chaine ( chaine & );
    ~chaine();
    void affiche();
};

chaine::chaine(const char * ch){
    cout << "--- construction - chaine -" << endl;
    s = new char[strlen(ch)+1];
    strcpy(s,ch);
}

chaine::chaine( chaine & sch){
    cout << "--- construction par recopie - chaine -" << endl;
    s = new char[strlen(sch.s)+1];
    strcpy(s,sch.s);
}

chaine::~chaine(){
    if (s!=NULL) {
        cout << "--- destruction - chaine -" << endl;
        delete[] s;
    }
}

void chaine::affiche(){
    cout << s ;
}

//----- la classe etudiant
class etudiant{
    chaine _nom;
    chaine _prenom;
    unsigned int _age;
public:
    etudiant(const char * = NULL, const char * = NULL, unsigned int = 0);
    etudiant( etudiant & );
    ~etudiant();
    void affiche();
};

etudiant::etudiant(const char * nom, const char * prenom, unsigned int age):_nom(nom),_prenom(prenom)
{
    cout << "--- construction - etudiant -" << endl;
    _age = age;
}

etudiant::etudiant( etudiant & e):_nom(e._nom), _prenom(e._prenom)
{
    cout << "--- construction par recopie - etudiant -" << endl;
    _age = e._age;
}

etudiant::~etudiant(){
    cout << "--- destruction - etudiant -" << endl;
}

```

```

}

void etudiant::affiche(){
    _nom.affiche();
    cout << ", ";
    _prenom.affiche();
    cout << ", " << _age << endl;
}
//----- TEST
int main()
{
    etudiant E1( "DELUXE" , "DANIEL" , 17 );
    E1.affiche();
    etudiant E2 = E1;
    E2.affiche();
    return 0;
}

```

Sortie du programme :

```

--- construction - chaine -
--- construction - chaine -
--- construction - etudiant -
KORCHI, JAMAL, 17
--- construction par recopie - chaine -
--- construction par recopie - chaine -
--- construction par recopie - etudiant -
KORCHI, JAMAL, 17
--- destruction - etudiant -
--- destruction - chaine -
--- destruction - chaine -
--- destruction - etudiant -
--- destruction - chaine -
--- destruction - chaine -

```

Chapitre VI

Les fonctions et classes amies

- L'encapsulation des données restreint l'accès aux membres données d'une classe aux fonctions membres de cette classe. Mais parfois, il est utile d'avoir des fonctions non membres d'une classe qui ont accès aux données privées de cette dernière.
- C++ introduit alors la notion "d'amitié" entre fonctions et classes, ce qui permet de définir des fonctions, appelées fonctions amies d'une classe, qui peuvent accéder à tous les membres de cette classe.
- C++ permet aussi de déclarer une classe amie d'une autre classe.
- Une classe ou une fonction amie d'une classe est déclarée dans la classe avec le mot clé *friend*.

VI.1 Les fonctions amies

Pour une fonction amie d'une classe, on distingue trois situations :

- ◆ La fonction est indépendante (n'est membre d'aucune classe)
- ◆ La fonction est membre d'une autre classe
- ◆ La fonction est une fonction amie d'une autre classe

Fonction indépendante amie d'une classe

- Pour qu'une fonction fct de prototype

```
type_r fct(liste_arguments);
```

soit une fonction amie d'une classe T, il faut la déclarer dans la classe T avec le mot-clé **friend**, comme suit :

```
class T{
    ...
public:
    ...
    friend type_r fct(liste_arguments);
    ...
}
```

- Notons que pour justifier la déclaration d'amitié d'une fonction donnée à une classe, il faut que cette fonction possède un argument ou une valeur de retour de type de la classe.

Exemple 6.1: (EXP06_01.CPP)

Pour la classe point (manipulation des points du plan), on souhaite définir une fonction **coincide()** qui permet de comparer deux points.

Pour ce, on peut définir la fonction soit comme fonction membre (voir *Exemple 4.3*), soit comme fonction amie de la classe point.

```
----- Classe point
class point{
    int x;
    int y;
public:
    point(int abs = 0, int ord = 0){
        x = abs; y = ord;
    }
    friend bool coincide(point,point);
    void point::affiche(){
        cout << "(" << x << ", " << y << ")" << endl;
    }
};

----- TEST
int main()
{
    point a,c;
    point b(5,6);
    cout << ( coincide(a,c) ? "-> a = c" : "-> a <> c" ) << endl;
    cout << ( coincide(a,b) ? "-> a = b" : "-> a <> b" ) << endl;
    return 0;
}
----- définition de la fonction amie coincide
bool coincide(point u, point v){
    return ( (u.x == v.x) && (u.y == v.y) );
}
```

Notez qu'il est inutile de déclarer la fonction `coincide()` à l'extérieur de la classe, parce qu'elle l'est déjà à l'intérieur de la classe.

Fonction membre d'une classe amie d'une autre classe

- Pour déclarer une fonction `fct` membre d'une classe A, comme fonction amie d'une classe T, il faut que la fonction soit déclarée avec le nom de sa classe et l'opérateur de résolution de portée, comme suit :

```
class T{
    ...
public:
    ...
    friend type_r A::fct(liste_arguments);
    ...
}
```

- Pour compiler convenablement la classe T, il faut que le compilateur accède à la déclaration complète de A. Pour ce, il faut que la déclaration de A précède celle de T. D'autre part, puisque `fct` a un argument ou une valeur de retour de type T, il faut donc signaler au compilateur que T est une classe, avant la déclaration de la classe A. Ainsi, si les deux déclarations se trouvent dans le même fichier, on doit avoir le schéma suivant:

```
class T; // signale au compilateur que T est une classe
//----- déclaration de A
class A{.....};
//----- déclaration de T
class T{.....};
//--- définition de la fonction membre de A amie de T
...
```

Exemple 6.2: (EXP06_02)

L'exemple suivant montre l'utilisation d'une fonction membre d'une classe et amie d'une autre classe en programmation modulaire.

On considère une classe nommée `matrice` qui permet de manipuler les matrice 2x2, et une classe nommé `vecteur` pour manipuler les vecteurs à 2 composantes. Puis, on souhaite définir une fonction nommée `colonne()` qui permet d'extraire d'une matrice ses vecteurs colonnes.

L'exemple offre une solution, en définissant la fonction `colonne` comme membre de la classe `matrice` et amie de la classe `vecteur`.

►Interface de la classe matrice (MATRICE1.H)

```
#ifndef _MATRICE
#define _MATRICE

class vecteur;           // nécessaire pour la compilation

class matrice{
```

```

        double mat[2][2];
public:
    matrice (double [2][2]);
vecteur colonne(int);
    void affiche();
};

#endif

```

►Interface de la classe vecteur (VECTEUR1.H)

```

#ifndef _VECTEUR
#define _VECTEUR

#include "matrice1.h"           // la déclaration complète de 'matrice'

class vecteur{
    double vect[2];
public:
    vecteur (double = 0, double = 0);
    friend vecteur matrice::colonne(int);
    void affiche();
};

#endif

```

►Corps de la classe matrice (MATRICE1.CPP)

```

#include "matrice1.h"
#include "vecteur1.h"           // nécessaire pour la fonction colonne(...)
#include <iostream>
using namespace std;

matrice::matrice (double t[2][2]){
    int i,j;
    for(i = 0; i < 2; i++)
        for(j = 0; j < 2; j++)
            mat[i][j] = t[i][j];
}
void matrice::affiche(){
    cout << mat[0][0] << "\t" << mat[0][1] << endl;
    cout << mat[1][0] << "\t" << mat[1][1] << endl;
}

*****
Fonction : colonne() amie de 'vecteur'
Retourne colonne k si k = 1 ou 2, sinon elle retourne le vecteur nul
*****
vecteur matrice::colonne(int k){
    vecteur v;
    if((k == 1) || (k == 2))
        for(int i = 0; i < 2; i++)
            v.vect[i] = mat[i][k-1];
    return v;
}

```

►Corps de la classe vecteur (VECTEUR1.CPP)

```

#include "vecteur1.h"
#include <iostream>
using namespace std;

vecteur::vecteur(double v1, double v2){
    vect[0]= v1;
}

```

```

        vect[1]= v2;
    }

void vecteur::affiche(){
    for(int i=0; i<2; i++)
        cout << vect[i] << " ";
    cout << endl;
}

```

► *Programme test* (EXP06_02.CPP)

```

#include "matrice1.h"
#include "vecteur1.h"
#include <iostream>
using namespace std;

int main(){
    double tb[2][2] = {{1,2},{3,4}};
    matrice a = tb;
    a.affiche();
    vecteur v1 = a.colonne(1);
    cout << "colonne 1 : " ; v1.affiche();
    vecteur v2 = a.colonne(2);
    cout << "colonne 2 : " ; v2.affiche();
    return 0;
}

```

Fonction amie de plusieurs classes

- Pour qu'une fonction soit amie des classes A et B, il suffit de déclarer cette amitié pour les deux classes concernées. Cependant, il faut noter que pour une compilation séparée de A il faut signaler au compilateur que B est une classe et vice-versa.

Exemple 6.3: (EXP06_03)

On reprend les mêmes classes décrites dans l'*exemple 6.2*, et on définit cette fois ci le produit d'une matrice par un vecteur.

La fonction `produit()` sera déclarée comme amie des deux classes et sera compilée séparément.

► *Interface de la classe matrice* (MATRICE2.H)

```

#ifndef _MATRICE
#define _MATRICE

class vecteur; // nécessaire pour la compilation

class matrice{
    double mat[2][2];
public:
    matrice (double [2][2]);
    friend vecteur produit(matrice,vecteur);
    void affiche();
};

#endif

```

► *Interface de la classe vecteur* (VECTEUR2.H)

```

#ifndef _VECTEUR
#define _VECTEUR

```

```

class matrice; // nécessaire pour la compilation

class vecteur{
    double vect[2];
public:
    vecteur (double = 0, double = 0);
    friend vecteur produit(matrice,vecteur);
    void affiche();
};

#endif

```

►Corps de la classe matrice (MATRICE2.CPP)

```

#include "matrice2.h"
#include <iostream>

matrice::matrice (double t[2][2]){
    int i,j;
    for(i = 0; i < 2; i++)
        for(j = 0; j < 2; j++)
            mat[i][j] = t[i][j];
}

void matrice::affiche(){
    std::cout << mat[0][0] << "\t" << mat[0][1] << std::endl;
    std::cout << mat[1][0] << "\t" << mat[1][1] << std::endl;
}

```

►Corps de la classe vecteur (VECTEUR2.CPP)

```

#include "vecteur2.h"
#include <iostream>

vecteur::vecteur(double v1, double v2){
    vect[0]= v1;
    vect[1]= v2;
}

void vecteur::affiche(){
    std::cout << "[" << vect[0] << ", "
        << vect[1] << "]" << std::endl;
}

```

►Implémentation de la fonction produit (PRODUIT.CPP)

```

#include "matrice2.h"
#include "vecteur2.h"

vecteur produit(matrice m, vecteur u){
    int i, j;
    double som;
    vecteur res;
    for(i=0; i<2; i++){
        for(j=0,som=0; j<2; j++)
            som += m.mat[i][j] * u.vect[j];
        res.vect[i] = som;
    }
    return res;
}

```

►Programme test (EXP06_03.CPP)

```

#include "matrice2.h"
#include "vecteur2.h"

```

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

int main()
{
    double tb[2][2] = {{1,2},{3,4}};
    matrice a = tb;
    a.affiche();
    vecteur u(4,5);
    u.affiche();
    vecteur v;
    v = produit(a,u);
    cout << "produit = " ; v.affiche();
    return 0;
}
```

VI.2 Classes amies

- Pour rendre toutes les fonctions membres d'une classe B, amies d'une classe A, on déclare la classe B comme amie de la classe A à l'intérieur de la déclaration de A avec le mot clé **friend**, comme suit:

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

- Cependant, il faut noter que l'amitié n'est pas transitive, autrement dit, si une classe B est amie d'une classe A et si une classe C est amie de B, la classe C n'est pas automatiquement amie de A.

Chapitre VII

La surcharge des opérateurs

- En C++, les opérateurs définis pour les types de base sont traités par le compilateur comme des fonctions, ce qui permet donc, avec la technique de la surcharge, d'utiliser la plus part d'entre eux avec les types ‘class’

Les opérateurs redéfinissables sont :

| | | | | | | | | | |
|-----|----|-----|-----|----|-----|--------|----|----|---|
| + | - | * | / | % | ^ | & | | ~ | ! |
| += | -= | *= | /= | %= | ^= | &= | = | | |
| ++ | -- | = | == | != | < | > | <= | >= | |
| << | >> | <<= | >>= | && | | | | | |
| ->* | , | -> | [] | () | new | delete | | | |

Les opérateurs qui ne peuvent être surchargés

| | | | | | |
|---|----|----|----|---|----|
| . | .* | :: | ?: | # | ## |
|---|----|----|----|---|----|

- Pour surcharger un opérateur `x`, on définit la fonction `operatorx` (par exemple `operator+` pour l'addition).
- L'implémentation de la surcharge d'un opérateur peut se faire soit comme une fonction classique (amie d'une classe), soit comme une fonction membre d'une classe.
- La surcharge d'un opérateur est appelée implicitement par le compilateur chaque fois que l'opérateur est rencontré dans le code, mais elle peut aussi être appelée explicitement.

*Exemple 7.1 : (EXP07_01.CPP)**Exemple de surcharge de l'opérateur +*

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

----- Classe point -----
class point{
    int x;
    int y;
public:
    point(int abs = 0, int ord = 0){
        x = abs; y = ord;
    }
    ----- surcharge de l'opérateur +
    friend point operator+ (const point &, const point &);

    void point::affiche(){
        cout << "(" << x << ", " << y << ")" << endl;
    }
};

----- définition de la fonction operator+
point operator+(const point &u, const point &v){
    point res;
    res.x = u.x + v.x;
    res.y = u.y + v.y;
    return res;
}

----- TEST
int main()
{
    point a(2,4);
    point b(5,6);

    cout << "a = " ; a.affiche();
    cout << "b = " ; b.affiche();
    cout << "a+b = " ; (a+b).affiche();

    cout << endl;
    return 0;
}
```

VII.1 Règles générales pour la surcharge des opérateurs

Les règles suivantes doivent être respectées pour surcharger un opérateur. Elles sont applicables pour tous les opérateurs sauf pour *new* et *delete*.

1. La surcharge d'un opérateur doit être

- a. soit une fonction membre.
- b. soit une fonction indépendante dont un des ses arguments est de type classe ou de type énumération.

```
int operator+ (int, int)      //erreur : cette surcharge ne possède aucun
                                // argument de type formel(classe,énumération)
// T est une classe
int operator+(T,int)          // acceptable
T operator+(int,int)          // erreur
```

2. La surcharge d'un opérateur doit respecter le nombre d'arguments:

- a. Un opérateur unaire déclaré comme fonction membre ne prend aucun argument, et s'il est déclaré comme fonction indépendante, il prend un seul argument.
- b. Un opérateur binaire déclaré comme fonction membre prend un seul argument et prend deux arguments s'il déclaré comme fonction indépendante

```
class point{
    int x; int y;
public:
    bool operator!= (const point &); //binaire
    friend point operator+ (const point &, const point &); //binaire
};
```

Les opérateurs ainsi déclarés gardent leur priorité et leur associativité.

- 3. La surcharge d'un opérateur ne peut avoir des arguments par défaut.
- 4. Le premier argument d'un opérateur, surchargé comme fonction membre, détermine l'objet qui invoque la fonction.

```
// x est un opérateur
class A{
    ...
public:
    type-ret operatorx (B)
};

class B{
    ...
public:
    type-ret operatorx (A)
};

A u;
B v;
u x v;           // appel de u.operatorx(v)
v x u;           // appel de v.operatorx(u)
```

5. les surcharges de tous les opérateurs, excepté `operator=`, sont héritées par les classes dérivées.

Enfin, notons que :

- ♦ Un opérateur peut avoir plusieurs implémentations :

```
class point{
    int x; int y;
public:
    friend point operator+ (point &, int);
    friend point operator+ (int, point &);
};
```

- ♦ La signification d'un opérateur peut être complètement changée par une surcharge (ce qui est fortement déconseillé).
- ♦ Aucune hypothèse ne doit être faite sur la signification a priori d'un opérateur, par exemple la signification de `+=` ne doit être déduite de la signification de `+` et `=`

VII.2 Les opérateurs unaires :

- Liste des opérateurs unaires :

| Opérateur | Nom |
|-----------|-----------------|
| ! | NON logique |
| ~ | Complément à un |
| + | Plus unaire |
| ++ | Incrémantation |
| - | Moins unaire |
| -- | Décrémantation |

- Syntaxe de déclaration d'un opérateur unaire comme fonction membre :

```
type_ret operatorop();
```

Où type_ret est le type de retour et op est l'opérateur unaire

- Syntaxe de déclaration d'un opérateur unaire comme fonction indépendante :

```
type_ret operatorop(arg);
```

Où type_ret est le type de retour, op est l'opérateur unaire et arg est un argument de type de la classe pour laquelle l'opérateur est surchargé

Incrémantation et décrémantation:

- Les deux opérateurs d'incrémantation et de décrémantation sont spéciales, du fait qu'il peuvent être placés avant (pré-incrémantation, pré-décrémantation) ou après (post-incrémantation, post-décrémantation) l'opérande. Chacun d'entre eux définit alors deux opérateurs. La surcharge de ces opérateurs doit donc être spéciale, du fait que le post ou le pré utilisent le même symbole et sont tous les deux des opérateurs unaires.
- C++ offre la technique suivante pour faire la différence entre le post et le pré :
 - Les préfixes sont déclarés normalement, mais doivent retourner une référence :

```
T & operator++();
T & operator--();
```

- Le suffixe est déclaré avec un paramètre fictif de type entier

```
T operator++(int);
T operator--(int)
```

Exemple 7.2 : (EXP07_02.CPP)

L'exemple suivant montre une implémentation des opérateurs d'incrémantation et décrémantation pour la classe point.

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

----- Classe point
class point{
    int x;
    int y;
public:
    point(int abs = 0, int ord = 0){
```

```

        x = abs; y = ord;
    }
    //----- surcharge de l'opérateur ++
    point &operator++();           // pré
    point operator++(int);        // post
    //----- surcharge de l'opérateur --
    point &operator--();           // pré
    point operator--(int);        // post

    void point::affiche(){
        cout << "(" << x << "," << y << ")" << endl;
    }
};

//----- définition des surcharges
point & point::operator++(){           // pre-incrementation
    ++x; ++y;
    return *this;
}

point point::operator++(int) {          // post-incrementation
    point temp = *this;
    x++; y++;
    return temp;
}
point & point::operator--(){           // pre-incrementation
    --x; --y;
    return *this;
}

point point::operator--(int) {          // post-incrementation
    point temp= *this;
    x--; y--;
    return temp;
}
//----- TEST
int main()
{
    point a(2,4);
    point b(5,6);

    cout << "a = " ; (--a).affiche();
    cout << "b = " ; (b--).affiche();
    cout << "b = " ; b.affiche();
    cout << endl;
    return 0;
}

```

VII.3 Les opérateurs binaires

- Liste des opérateurs binaires :

| <i>Opérateur</i> | <i>Nom</i> |
|-------------------|-------------------------------|
| , | Virgule |
| != | Inégalité |
| % | Modulo |
| %= | Modulo/affectation |
| & | ET binaire |
| && | ET logique |
| &= | ET binaire/affectation |
| * | Multiplication |
| *= | Multiplication/affectation |
| + | Addition |
| += | Addition/affectation |
| - | Soustraction |
| -= | Soustraction/affectation |
| / | Division |
| /= | Division/affectation |
| < | Plus petit |
| << | Décalage à gauche |
| <<= | Décalage à gauche/affectation |
| <= | Plus petit ou égal |
| = | Affectation |
| == | Egalité |
| > | Plus grand |
| >= | Plus grand ou égal |
| >> | Décalage à droite |
| >>= | Décalage à droite/affectation |
| ^ | OU exclusif |
| ^= | OU exclusif/affectation |
| | OU binaire |
| = | OU binaire/affectation |
| | OU logique |

- Syntaxe de déclaration d'un opérateur binaire comme fonction membre d'une classe T:

```
type_ret operatorop(arg);
```

Où `type_ret` est le type de retour et `op` est l'opérateur binaire, et `arg` est un argument de type classe T.

- Syntaxe de déclaration d'un opérateur binaire comme fonction indépendante :

```
type_ret operatorop(arg1, arg2);
```

Où `type_ret` est le type de retour, `op` est l'opérateur binaire et `arg1` et `arg2` sont deux arguments dont au moins un est de type de la classe pour laquelle l'opérateur est surchargé.

Affectation

- L'opérateur d'affectation peut être surchargé comme n'importe quel autre opérateur binaire, mais présente certaines exceptions :
 1. Il doit être une fonction membre
 2. Il n'est pas hérité par les classes dérivées
 3. Il est généré par défaut par le compilateur

Exemple 7.3 : (EXP07_03.CPP)

Implémentation de l'opérateur d'affectation pour la classe point.

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

----- Classe point
class point{
    int x;
    int y;
public:
    point(int abs = 0, int ord = 0){
        x = abs; y = ord;
    }
    ----- surcharge de l'opérateur =
    point & operator= (const point &);

    void point::affiche(){
        cout << "(" << x << "," << y << ")" << endl;
    }
};

----- définition de la fonction operator=
point & point::operator= (const point & source){
    if( &source != this){
        x = source.x;
        y = source.y;
    }
    return *this;
}

----- TEST
int main()
{
    point a(1,2);
    point b,c;
    c = b = a;
    cout << "a = " ; a.affiche();
    cout << "b = " ; b.affiche();
    cout << "c = " ; c.affiche();
    cout << endl;
    return 0;
}
```

Notez que l'opérateur d'affectation retourne une référence pour pouvoir utiliser des affectations de type : $U = V = W;$

Exemple 7.4 : (EXP07_04.CPP)

Implémentation de l'opérateur d'affectation pour une classe contenant des pointeurs.

```
----- Déclaration de la classe
class T{
    int i;
    int *pi;
```

```

public:
    T( int = 0, int = 0 );
    T( const T & );
    ~T();
    //----- opérateur d'affectation
    T & operator= (const T &);
    void affiche();
    void modifier(int,int);
};

// ----- Définition des fonctions membres
T::T(int n, int p){
    i = n;
    pi = new int;
    *pi = p;
}
// ---- Constructeur par recopie
T::T( const T & v){
    i = v.i;
    pi = new int;
    *pi = *(v.pi);
}

T::~T(){
    if(pi != NULL)
        delete pi;
}
//----- opérateur d'affectation
T & T::operator= (const T & source){
    i = source.i;
    *pi = *(source.pi);
    return *this;
}

void T::affiche(){
    cout << "(" << i << "," << *pi << ")" --> " << pi << endl;
}
void T::modifier(int n, int p){
    i = n ; *pi = p;
}
// ----- Test
int main()
{
    T u(3,4);
    cout << "u : ";u.affiche();
    // affectation
    T v;
    v = u;
    cout << "v : ";v.affiche();
    // on modifie v
    cout << "\n-----Modification" << endl;
    v.modifier(2,2);
    cout << "v : ";v.affiche();
    cout << "u : ";u.affiche();
    return 0;
}

```

Des entrées sorties simplifiées

- Les opérateurs << et >> peuvent être surchargés de manière à lire le contenu d'une classe donnée T sur un flux d'entrée (*istream*) ou écrire son contenu dans un flux de sortie (*ostream*), pour ce, on utilise des fonctions amies :

```
istream & operator >> (istream &, T &)
ostream & operator << (ostream &, const T &);
```

Exemple 7.5 : (EXP07_05.CPP)

Surcharge des opérateurs << et >> pour des E/S simplifiées

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

----- Classe point
class point{
    int x;
    int y;
public:
    point(int abs = 0, int ord = 0){
        x = abs; y = ord;
    }
    //----- surcharge des operateurs << et >>
    friend ostream & operator<< (ostream &, const point &);
    friend istream & operator>> (istream &, point &);
};

ostream & operator<< (ostream & out, const point & pt){
    out << "(" << pt.x << "," << pt.y << ")";
    return out;
}

istream & operator>> (istream & in, point &pt){
    cout << "abscisse : "; in >> pt.x;
    cout << "ordonnee : "; in >> pt.y;
    return in;
}

----- TEST
int main()
{
    point a;
    cin >> a;
    cout << "a = " << a << endl;
    return 0;
}
```

VII.4 Les opérateurs de déréférencement, d'indirection et d'accès aux membres

- Opérateur * : opérateur d'indirection, est un opérateur unaire qui permet l'écriture des classes dont les objets peuvent être utilisés dans des expressions manipulant des pointeurs.

Exemple 7.6 : (EXP07_06.CPP)

Cet exemple montre une utilisation classique de l'opérateur * :

```
#include <iostream>
using namespace std;
----- Déclaration de la classe
class T{
    int i;
    int *pi;
public:
```

```

T( int = 0, int = 0);
T( const T &);
~T();
//----- opérateur d'indirection
int & operator* ();
void affiche();

};

// ----- Définition des fonctions membres
T::T(int n, int p){
    i = n;
    pi = new int;
    *pi = p;
}

T::T( const T & v){
    i = v.i;
    pi = new int;
    *pi = *(v.pi);
}

T::~T(){
    if(pi != NULL)
        delete pi;
}

//----- opérateur d'indirection
int & T::operator* (){
    return *pi;
}

void T::affiche(){
    cout << "(" << i << "," << *pi << ")" --> " << pi << endl;
}

// ----- Test
int main()
{
    T u(3,4);
    cout << "u : ";u.affiche();
    *u = 5;
    cout << "u : ";u.affiche();
    cout << &(*u) << endl;
    return 0;
}

```

- Opérateur & : opérateur de déréférencement (adresse de) est un opérateur unaire qui a une définition par défaut pour les objets. Il est généralement surchargé pour renvoyer une adresse autre que celle de l'objet sur lequel il s'applique.
- Opérateur -> : opérateur de sélection d'un membre, il permet de réaliser des classes qui encapsulent d'autres classes. Cet opérateur est considéré comme un opérateur unaire, il doit être surchargé comme fonction membre et il doit retourner un pointeur sur un type classe (class, struct ou union), comme suit :

*type_classe *operator->();*

Exemple 7.7 : (EXP07_07.CPP)

```

#include <iostream>
#include <string.h>
using namespace std;

```

```

//----- Définition d'une structure
struct etudiant{
    char * nom;
    unsigned int age;
};

//----- Declaration de la classe
class T{
    etudiant e;
    int numero;
public:
    T(const char *, unsigned int , unsigned int);
    T( const T & );
    ~T();
    //
    etudiant *operator-> ();
};

// ----- Définition des fonctions membres
T::T(const char * nom, unsigned int age, unsigned int num){
    e.nom = new char[strlen(nom)+1];
    strcpy(e.nom,nom);
    e.age = age;
    numero = num;
}

T::T( const T & v){
    int l = strlen(v.e.nom);
    e.nom = new char[l+1];
    strcpy(e.nom,v.e.nom);
    e.age = v.e.age;
    numero = v.numero;
}

T::~T(){
    if(e.nom != NULL)
        delete e.nom;
}

//-----
etudiant * T::operator-> (){
    return &e;
}

// ----- Test
int main()
{
    T a ("JALIL",23,445);
    cout << a->nom << endl;
    a->age = 21;
    cout << a->age << endl;
    return 0;
}

```

VII.5 L'opérateur d'appel de fonction

- L'opérateur d'appel de fonction (), appelé aussi opérateur fonctionnel, permet de réaliser des objets qui se comportent comme des fonctions.
- Cet opérateur peut avoir plusieurs opérandes (il n'est ni unaire, ni binaire), et il ne peut être surchargé que par une fonction membre.

Exemple 7.8 : (EXP07_08.CPP)

Cet exemple montre la surcharge de l'opérateur fonctionnel pour la classe des matrices 2x2, pour pouvoir accéder à chacun des éléments de la matrice.

```
#include <iostream>
using namespace std;
// -----
class matrice{
    double m[2][2];
public:
    matrice (double [2][2]);
    matrice (double = 0, double = 0, double = 0, double = 0);
    void affiche();
    // surcharge de l'opérateur fonctionnel
    double & operator()( unsigned int, unsigned int);
};

// -----
matrice::matrice(double t[2][2]){
    int i,j;
    for( i = 0; i < 2; i++)
        for( j = 0; j < 2; j++)
            m[i][j] = t[i][j];
}

matrice::matrice(double m11, double m12, double m21, double m22){
    m[0][0] = m11; m[0][1] = m12;
    m[1][0] = m21; m[1][1] = m22;
}

void matrice::affiche(){
    cout << endl;
    cout << m[0][0] << "\t" << m[0][1] << endl;
    cout << m[1][0] << "\t" << m[1][1] << endl;
}

double & matrice::operator ()(unsigned int i, unsigned int j){
    int i1 = ((i == 1) || (i == 2)) ? i-1 : 0;
    int j1 = ((j == 1) || (j == 2)) ? j-1 : 0;
    return (m[i1][j1]);
}

//---- TEST
int main()
{
    matrice mt(1,2,3,4);
    cout << "ligne 2, colonne 1 : " << mt(2,1) << endl;
    mt(2,2) = 5;
    mt.affiche();
    return 0;
}
```

VII.6 L'opérateur d'indexation

- L'opérateur d'indexation [] est considéré comme un opérateur binaire. Il doit être surchargé comme fonction membre avec un seul argument.

Exemple 7.9 : (EXP07_09.CPP)

L'exemple montre l'utilisation de l'opérateur d'indexation avec une classe de gestion des tableaux dynamique d'entiers.

```

#include <iostream>
using namespace std;

----- Déclaration de la classe
class TabInt{
    int nbelem;
    int *tab;
public:
    TabInt( int );
    TabInt( const TabInt & );
    ~TabInt();
    ----- opérateur d'indexation
    int & operator[] (int);
    void affiche();
};

// ----- Définition des fonctions membres
TabInt::TabInt(int n){
    nbelem = n;
    tab = new int[n];
}
// ---- Constructeur par recopie
TabInt::TabInt( const TabInt & v){
    nbelem = v.nbelem;
    tab = new int[nbelem];
    for(int i; i < nbelem; i++)
        tab[i] = v.tab[i];
}

TabInt::~TabInt(){
    if(tab != NULL)
        delete [] tab;
}
----- opérateur d'indexation
int & TabInt::operator[] (int n){
    if( (n>=0) && (n < nbelem) )
        return tab[n];
    else
        return tab[0];
}

void TabInt::affiche(){
    for(int i=0; i < nbelem; i++)
        cout << "(" << tab[i] << ")";
    cout << endl;
}
// ----- Test
int main()
{
    TabInt a(15);
    for(int i=0; i < 15; i++)
        a[i] = i;
    a.affiche();
    return 0;
}

```

VII.7 Les opérateurs d'allocation dynamique

- La surcharge de l'opérateur ***new*** doit se faire par une fonction membre, de la forme :

```
void * operator new(size_t, lst_param);
```

le premier argument doit être de type ***size_t*** (stddef.h ou stdlib.h), il correspond à la taille en octets de l'objet à allouer. La valeur de retour est un pointeur de type ***void***. ***lst_param*** est une liste de paramètres facultatifs.

L'appel d'une telle surcharge pour une classe donnée **T** se fait de la manière suivante:

```
T* p;
p = new (lst_param) T(...);
```

- La surcharge de l'opérateur ***delete*** se fait aussi par une fonction membre, qui a la forme suivante:

```
void operator delete(void *, size_t);
```

le pointeur en paramètre représente l'adresse de l'objet à libérer. Le paramètre de type ***size_t*** est facultatif et il représente la taille de la zone mémoire à libérer.

- Il faut cependant, noter que :
 - les opérateurs ***new*** et ***delete*** sont prédéfinis pour tous les types classes. Lorsqu'ils sont surchargés, on peut accéder aux opérateurs globaux prédéfinis en utilisant l'opérateur de résolution de portée.
 - Les surcharges des opérateurs ***new*** et ***delete*** sont considérées comme des fonctions statiques, ce qui interdit l'utilisation du pointeur ***this*** dans ses fonctions.
 - L'appel de l'opérateur ***new*** est toujours suivi par l'appel du constructeur et celle de ***delete*** par le destructeur
- Les surcharges des opérateurs ***new[]*** et ***delete[]*** suivent les mêmes règles que celles de ***new*** et ***delete***

Exemple 7.10 : (EXP07_10.CPP)

Exemple de surcharge des opérateurs ***new*** et ***delete***

```
#include <iostream>
#include <stddef.h>
using namespace std;

/////////// Déclaration de la classe
class T{
    int i;
public:
    T( int = 0);
    //----
    void * operator new (size_t);
    void operator delete(void *);
    void affiche();
};

// ----- Définition des fonctions membres
T::T(int n)
{
    i = n;
}
```

```
//----- opérateur new
void * T::operator new (size_t taille)
{
    cout << "operateur new" << endl;
    return ::new char[taille];
}
//----- opérateur delete
void T::operator delete(void * pi)
{
    ::delete [] pi;
}

void T::affiche()
{
    cout << "(" << i << ")" << endl;
}
// ----- Test
int main()
{
    T a = 5;
    a.affiche();
    T * pa;
    pa = new T(10);
    (*pa).affiche();
    delete pa;
    return 0;
}
```


Chapitre VIII

Conversions

- En plus des conversions implicites ou explicites (cast) définies dans C, C++ offre la possibilité de définir des conversions d'un type classe vers un autre type classe ou un type de base.
- De telles conversions, dites *conversions définies par l'utilisateur*, peuvent être définies soit par les constructeurs à un seul argument, soit par la surcharge de l'opérateur de cast.
- Les conversions, ainsi définies, peuvent être appelées explicitement ou implicitement par le compilateur. Le compilateur fait appel à l'une des conversions définies par l'utilisateur, s'il ne peut appliquer une conversion standard.
- Notez que le compilateur cherche à appliquer implicitement une conversion chaque fois qu'il trouve (dans une initialisation, l'argument d'une fonction, le paramètre de retour d'une fonction, lors de l'évaluation d'une expression, etc....) un objet dont le type ne correspond pas au type attendu.
- D'autre part, le compilateur ne cherche pas des conversions intermédiaires pour réaliser une autre. Si par exemple une conversion est définie d'un type A vers un type B, et une autre est définie de ce type B vers un type C, le compilateur ne peut pas utiliser implicitement ces deux conversions pour réaliser une conversion de A vers C.

VIII.1 Conversion définie par le constructeur

- Un constructeur avec un seul argument réalise une conversion du type de cet argument vers le type de sa classe, et ce quelque soit le type de cet argument.
- Notez que ce type de conversion ne peut être utilisé pour définir les conversions de type classe vers un type de base.

Exemple 8.1 : (EXP08_01.CPP)

L'exemple suivant montre comment le constructeur de la classe point à un seul argument de type int définit une conversion de int vers point

```
class point{
    int x;
    int y;
public:
    point(): x(0), y(0){}
    point(int abs): x(abs), y(abs){}
    point(int abs, int ord): x(abs), y(ord){}
    void affiche(){
        cout << "(" << x << "," << y << ")" << endl;
    }
};
// fonction avec argument de type point
void fct(point pt){
    pt.affiche();
}
//----- TEST
int main()
{
    point a;
    // conversion implicite de int -> point lors d'une affectation
    a = 6;
    a.affiche();
    // conversion implicite de int -> point lors de l'appel d'une fonction
    fct(7);
    // conversion explicite
    ((point)5).affiche();
    return 0;
}
```

VIII.2 Opérateur de cast

- Pour définir l'opérateur de conversion d'une classe T vers un type A, on utilise la fonction membre :

`operator A () ;`

- Notez que :
 - Cette fonction membre doit retourner une valeur de type A.
 - le type de retour ne doit pas figurer dans l'entête de la fonction.
 - la fonction ne prend aucun argument.

Exemple 8.2 : (EXP08_02.CPP)

On considère dans cet exemple une conversion de point vers un entier

```
class point{
    int x;
    int y;
public:
    point(int abs = 0, int ord = 0):x(abs), y(ord){}
    operator int(){
        return x;
    }
};

//----- TEST
int main()
{
    point a(5,6);
    int i,j;
    // conversion implicite de point -> int
    i = a;
    cout << i << endl;           // i = 5
    // conversion lors d'une expression
    j = a + 4;                   // j = 9
    cout << j << endl;
    return 0;
}
```

VIII.3 Problèmes posées par le transtypage

- Les conversions définies par l'utilisateur doivent être sans ambiguïtés, autrement dit, on ne peut avoir plusieurs chaînes de conversions qui conduisent vers un même type :

Plusieurs problèmes peuvent être posés, lorsqu'on définit dans une classe des conversions. On peut citer par exemple, les cas suivants :

1. Une classe qui définit deux conversions vers des types de bases :

Exemple 8.3 : (EXP08_03.CPP)

On définit pour une classe T, deux conversions, une vers int et l'autre vers double.

```
class T{
    int n;
public:
    T(int k = 0): n(k){}
    operator int(){return n;}
    operator double(){ return (double) n;}
};

int main()
{
    T a;
    double r;
    r = 2.5 + a;           //ambiguïté
    return 0;
}
```

Ici, deux opérateurs prédéfinis peuvent être appliqués `operator+(double, double)` et `operator+(double, int)`. Par suite, le compilateur ne peut choisir entre l'une des deux.

2. Une classe qui définit une conversion vers un type de base, et surcharge certains opérateurs :

Exemple 8.4 : (EXP08_04.CPP)

On définit pour la classe point, une conversion vers int et on surcharge l'opérateur +.

```
class point{
    int x;
    int y;
public:
    point(): x(0), y(0){}
    point(int abs): x(abs), y(abs){}
    point(int abs, int ord): x(abs), y(ord){}
    operator int() {return x;}
    friend point operator+(point,point);
};

point operator+(point u, point v){
    point temp;
    temp.x = u.x + v.x;
    temp.y = u.y + v.y;
    return temp;
}
//----- TEST
int main()
{
    point a(5,6);
    point b;
    b = a + 4;           //ambiguïté
    return 0;
}
```

Ici, le compilateur ne peut choisir entre les deux opérateurs candidats operator+(point,point) et operator+(int,int).

Chapitre IX

Héritage simple

- La notion de l'héritage consiste à créer une nouvelle classe incorporant les caractéristiques d'une ou de plusieurs autres classes :
- La nouvelle classe, dite *classe dérivée* (ou classe fille), hérite de toutes les propriétés (membres données et méthodes) des classes d'origines dites *classes de base* (classes mères). Mais elle peut posséder ses propres membres données et méthodes, comme elle peut masquer certaines méthodes de sa classe de base.
- Pour déclarer une classe T dérivée des classes A, B,..., on utilise la syntaxe suivante :

```
class T : mode_a A, mode_b B, ...
{ ... };
```

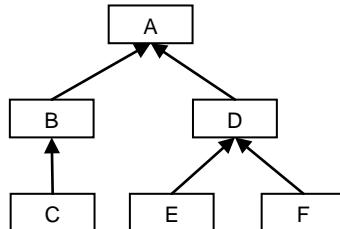
où *mode_a* (*mode_b*, ...) est l'une des mots clés suivants :

public, *private* ou *protected*

qui précise le mode d'héritage choisi (publique, privée ou protégée). Le mode d'héritage détermine le droit d'accès dans la classe dérivée des membres hérités de la classe de base.

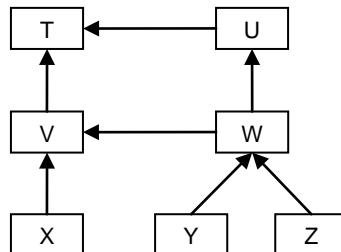
- Une déclaration complète de la classe de base doit précéder toute déclaration d'une de ses classes dérivées.

- Une classe dérivée peut servir comme classe de base d'une ou de plusieurs autres classes. La hiérarchie de l'héritage est généralement représentée par une arborescence, comme par exemple :



Héritage simple

- Ainsi une classe dérivée peut hériter directement ou indirectement d'une autre classe. Les classes de base dont une classe dérivée hérite directement sont appelées classes de base directe. Dans l'exemple précédent, la classe C admet deux classes de base A et B et une seule classe de base directe B.
- Dans l'exemple précédent, toutes les classes dérivées n'ont qu'une classe de base directe, on dit alors que c'est un *héritage simple*. Mais une classe peut avoir plusieurs classes de base directe. Dans ce cas on dit que *l'héritage est multiple*, comme par exemple:



Héritage multiple : La classe W a deux classes de base directe U et V

Dans ce chapitre, nous n'étudierons que le cas d'un héritage simple, avec une classe et ses descendantes directes, mais la plupart des résultats sont applicables dans n'importe quelle situation d'héritage.

IX.1 Droit d'accès à la classe de base

- Rappelons que tout membre d'une classe est défini avec un mode d'accès (*public*, *private* ou *protected*) et que:
 - l'accès aux membres privés d'une classe est limité aux fonctions membres et aux fonctions amies de la classe.
 - l'accès aux membres protégés est limité aux fonctions membres et aux fonctions amies de la classe et de ses dérivées.
 - L'accès aux membres publics n'est pas limité.
 - L'utilisateur d'une classe ne peut invoquer (appeler) que les membres publics pour une instance de cette classe

- Par ailleurs, la mention d'accès d'un membre hérité dans la classe dérivée, dépend de sa mention d'accès dans la classe de base et du mode d'héritage. Le tableau suivant donne la valeur déduite de l'attribut d'accès dans la classe dérivée dans les différents cas:

| | | Mention d'accès dans la classe de base | | |
|-----------------|-----------|--|-----------|--------------|
| | | public | protected | Private |
| Mode d'héritage | public | public | protected | Inaccessible |
| | protected | protected | protected | Inaccessible |
| | private | private | private | Inaccessible |

Ainsi, si on considère l'exemple suivant:

```
class T{
public :
    int i1;
    int fct1();
protected:
    int i2;
    int fct2();
private:
    int i3;
    int fct3();
};

class A: public T{... ...};
class B: protected T{... ...};
class C: private T{... ...};
```

Alors, i3 et fct3 sont inaccessibles pour les trois classes A, B et C, d'autre part :

- pour la classe A :
 - i1 et fct1 sont des membres publics
 - i2 et fct2 sont des membres protégés
- pour la classe B :
 - i1 et fct1 sont des membres protégés
 - i2 et fct2 sont des membres protégés
- pour la classe C :
 - i1 et fct1 sont des membres privés
 - i2 et fct2 sont des membres privés

IX.2 Un exemple d'héritage simple

Dans l'exemple suivant nous définissons une classe dérivée pointcol de la classe point, munie de deux membres supplémentaires couleur et setcolor()

Exemple 9.1 : (EXP09_01.CPP)

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

// ----- Classe de base
class point{
    int x;
    int y;
public:
    void initialise(int = 0, int = 0);
    void affiche();
};

void point::initialise(int abs, int ord){
```

```

        x = abs; y = ord;
    }

void point::affiche(){
    cout << "(" << x << "," << y << ")" << endl;
}

// ----- Classe dérivée
class pointcol: public point{
    int couleur;
public:
    void setcolor(int col = 1){
        couleur = col;
    }
};

//----- TEST
int main()
{
    pointcol a;
    a.initialise(1,2);           //membre hérité
    a.setcolor(3);               //membre hérité
    a.affiche();                 //membre hérité
    return 0;
}

```

Remarquez que l'objet a de type pointcol invoque les méthodes publiques de la classe point comme si c'était un objet de type point.

IX.3 Redéfinition des fonctions membres

- Une classe dérivée peut redéfinir une méthode ou une donnée de sa classe de base. Si par exemple :

```

class B{
public:
    int i;
    int fct();
};
class T:public B{
    int i;
public:
    int fct();
};

```

Alors la classe T hérite de la classe B les deux membres i et fct(), mais ces deux membres sont masqués par les membres i et fct() définis dans T. Ainsi si u est une instance de T:

```
u.fct();           //invoque la méthode fct définie dans T
```

Pour accéder aux membres masqués dans une classe dérivée, on utilise l'opérateur de résolution de portée ::

```
u.B::fct();       // invoque la méthode fct définie dans B
u.B::i            // fait référence au membre i défini dans B
```

Exemple 9.2 : (EXP09_02.CPP)

On reprend ici le même exemple précédent (*Exemple 9.1*), et on définit dans la classe `pointcol`, qui dérive de la classe `point`, une méthode `affiche()` qui masque celle définie dans la classe `point` et qui utilise cette dernière.

```
// ----- Classe dérivée
class pointcol: public point{
    int couleur;
public:
    void affiche();
    void setcolor(int col = 1){ couleur = col;}
};

void pointcol::affiche(){
    point::affiche();           // appel de affiche() définie dans point
    cout << " - couleur = " << couleur << endl;
}
```

- D'une manière générale, pour accéder à un membre (redéfini ou non) dans l'arborescence de l'héritage, il suffit de spécifier son nom complet dans la hiérarchie. Ainsi, si une classe U dérive d'une classe T, et que T dérive d'une classe B, pour accéder à un membre x défini dans B avec une instance u de U, il suffit d'utiliser la syntaxe :

```
u.T::B::x;
```

Notez que cette syntaxe devient obligatoire si le membre x est redéfini dans T et dans U.

Exemple 9.3 : (EXP09_03.CPP)

Cet exemple montre comment accéder à un membre dans l'arborescence de l'héritage.

```
// ----- Classe de base
class B{
protected:
    char c;
public:
    void f(){cout << "appel de f() de B" << endl;}
};

class T:public B{
protected:
    char c;
public:
    void f(){cout << "appel de f() de T" << endl;}
};

class U:public T{
protected:
    char c;
public:
    void f(){cout << "appel de f() de U" << endl;}
};

//----- TEST
int main()
{
    U u;
    u.f();                      // f() de U
    u.T::f();                   // f() de T
    u.T::B::f();                // f() de B
    return 0;
}
```

► Sortie du programme :

```
appel de f() de U
appel de f() de T
appel de f() de B
```

- Notez que si une classe T, dérivée de la classe B, redéfinit une fonction membre, alors la fonction définie dans T masque celle définie dans B ainsi que toutes les surcharges de cette fonction.

```
class B{
public:
    void fct();
    void fct(int);
};

class T:public B{
    ...
public:
    int fct();
};

T t;
t.fct();           // appel de T::fct();
t.fct(2);         // erreur, aucune méthode ne correspond à cet appel
```

IX.4 Constructeur et héritage

- Etant donné une classe T qui hérite directement d'une classe B. Pour créer un objet de type T, il faut premièrement créer un objet de type B (puisque toute instance de T est considérée comme une instance de B). Autrement dit, tout appel du constructeur de T (constructeur défini ou par défaut) conduit premièrement à un appel du constructeur de B.

Par suite, si B possède un constructeur qui nécessite des paramètres, la classe T doit aussi avoir un constructeur, à travers lequel les paramètres nécessaires au constructeur de B, seront fournis. Cette transmission de paramètres se fait suivant la syntaxe suivante:

```
type_r T::T( liste_arg_T ) :B( liste_arg_B ){ ... }
```

où *liste_arg_B* est une sous liste de *liste_arg_T* valable pour un appel du constructeur de B

- De même, l'appel du destructeur de T conduit à l'appel du destructeur de B, sauf que dans ce cas le destructeur de T est exécuté en premier.

Exemple 9.4 : (EXP09_04.CPP)

Dans cet exemple on considère la classe *pointcol* dérivée de la classe *point*. On munit les deux classes de constructeurs et on déclare les membres *x* et *y* de la classe *point* avec la mention *protected*, pour qu'ils soient accessibles dans la classe *pointcol*.

```
// ----- Classe de base
class point{
protected:
    int x;
    int y;
```

```

public:
    point(int, int);
    ~point();
    void affiche();
};

point::point(int abs, int ord){
    x = abs; y = ord;
    cout << "--- constr classe de base" << endl;
}
point::~point(){
    cout << "--- destr classe de base" << endl;
}
void point::affiche(){
    cout << "(" << x << ", " << y << ")" << endl;
}
// -----
class pointcol: public point{
    int couleur;
public:
    pointcol(int, int, int);
    ~pointcol();
    void affiche();
};
pointcol::pointcol(int abs, int ord, int col):point(abs,ord){
    couleur = col;
    cout << "--- constr classe derivee" << endl;
}
pointcol::~pointcol(){
    cout << "--- destr classe derivee" << endl;
}
void pointcol::affiche(){
    cout << "(" << x << ", " << y << ")" - " << couleur << endl;
}
// -----
TEST
int main()
{
    pointcol a(1,2,3);
    a.affiche();
    return 0;
}

```

► Sortie du programme :

```

--- constr classe de base
--- constr classe derivee
(1,2) - 3
--- destr classe derivee
--- destr classe de base

```

Constructeur par recopie et l'héritage

- Le constructeur par recopie suit les mêmes règles que les autres constructeurs, mais dans ce cas, l'appel du constructeur par recopie de la classe dérivée n'entraîne pas automatiquement l'appel du constructeur par recopie de la classe de base.
- Considérons les différentes situations d'une classe T dérivée d'une classe B :
 - Si T ne possède pas de constructeur par recopie :
 - ◆ lors de l'appel du constructeur par recopie par défaut de T, le compilateur appelle le constructeur par recopie de la classe de base B (défini ou par défaut), puis initialise les membres de T qui ne sont pas hérités de B.

- Si la classe T a défini un constructeur par recopie :
 - ◆ l'appel de ce constructeur entraîne l'appel du constructeur de B mentionné dans l'entête du constructeur par recopie de T (ce dernier peut être le constructeur de recopie de B ou n'importe quel autre constructeur), comme par exemple:

```
T::T( T& t ):B(t) // appel du constructeur par recopie de B
// (conversion T → B)
```

ou

```
T::T(T &t ):B(liste_arg) // appel du constructeur de B qui accepte liste_arg
```

- ◆ Si aucun constructeur n'est mentionné dans l'entête du constructeur par recopie de T, le compilateur appelle un constructeur sans argument de la classe B, s'il en existe un. Dans le cas contraire, une erreur de compilation est générée

Exemple 9.5 :

Dans l'exemple suivant nous testerons les différentes implémentations de la classe pointcol dérivée de la classe point, décrivant les différents cas traités de l'appel du constructeur par recopie.

On définit dans la classe point un constructeur et un constructeur par recopie et on considère les cas suivants

Cas 1 : (EXP09_05a.CPP)

pointcol ne définit pas de constructeur par recopie.

```
// ----- Classe de base : point
class point{
protected:
    int x;
    int y;
public:
    point(int abs, int ord): x(abs), y(ord){
        cout << "++ constr classe de base" << endl;
    }
    ~point(){cout << "--- destr classe de base" << endl;}
    point(point & pt){
        x=pt.x; y = pt.y;
        cout << "*** constr recopie classe de base" << endl;
    }
    friend ostream & operator << (ostream &, const point & );
};

ostream & operator << (ostream & out, const point & pt){
    out << "(" << pt.x << "," << pt.y << ")";
    return out;
}

// ----- Classe dérivée : pointcol
class pointcol: public point{
    int couleur;
public:
    pointcol(int abs, int ord, int col):point(abs,ord){
        couleur = col;
        cout << "++ constr classe derivee" << endl;
    }
    ~pointcol(){cout << "--- destr classe derivee" << endl;}
    friend ostream & operator << (ostream &, const pointcol & );
};

ostream & operator << (ostream & out, const pointcol & pt){
```

```

        out << "(" << pt.x << "," << pt.y << ")" << " - " << pt.couleur;
        return out;
    }

//----- Programme test
int main()
{
    pointcol a(1,2,3);
    cout << "a = " << a << endl;
    cout << "-----initialisation d'un objet par un autre" << endl;
    pointcol b = a;
    cout << "b = " << b << endl;
    cout << "-----FIN PROGRAMME" << endl;
    return 0;
}

```

► Sortie du programme

```

+++ constr classe de base
+++ constr classe derivee
a = (1,2) - 3
-----initialisation d'un objet par un autre
*** constr recopie classe de base
b = (1,2) - 3
-----FIN PROGRAMME
--- destr classe derivee
--- destr classe de base
--- destr classe derivee
--- destr classe de base

```

Dans ce cas, lors de la création de l'instance b, il y a appel du constructeur par recopie de point.

Cas 2 : (EXP09_05b.CPP)

On définit un constructeur par recopie de pointcol, et on mentionne dans l'entête de ce dernier le constructeur par recopie de la classe point, comme suit :

```

...
pointcol::pointcol(pointcol & pt):point(pt)
{
    couleur = pt.couleur;
    cout << "*** constr recopie classe derivee" << endl;
}
...

```

► Sortie du programme

```

+++ constr classe de base
+++ constr classe derivee
a = (1,2) - 3
-----initialisation d'un objet par un autre
*** constr recopie classe de base
*** constr recopie classe derivee
b = (1,2) - 3
-----FIN PROGRAMME
--- destr classe derivee
--- destr classe de base
--- destr classe derivee
--- destr classe de base

```

Avec le même programme test, la création de l'instance b de pointcol fait appel au constructeur par recopie de point.

Cas 3 : (EXP09_05c.CPP)

On mentionne cette fois-ci dans l'entête du constructeur par recopie de pointcol le constructeur de point avec deux arguments, comme suit :

```

...
----- constructeur par recopie de la classe pointcol
pointcol::pointcol(pointcol & pt):point(pt.x,pt.y)
{
    couleur = pt.couleur;
    cout << "*** constr recopie classe derivee" << endl;
}
...

```

► Sortie du programme

```

+++ constr classe de base
+++ constr classe derivee
a = (1,2) - 3
-----initialisation d'un objet par un autre
+++ constr classe de base
*** constr recopie classe derivee
b = (1,2) - 3
-----FIN PROGRAMME
--- destr classe derivee
--- destr classe de base
--- destr classe derivee
--- destr classe de base

```

Ici, il y a appel du constructeur de point avec les arguments `pt.x` et `pt.y`. On peut mentionner n'importe quels autres arguments (de type `int`), la création de l'instance `b` de `pointcol` fait appel au constructeur de `point`, avec les paramètres mentionnés dans l'entête du constructeur par recopie de `pointcol`.

IX.5 Conversions entre classes de base et classes dérivées

- La conversion d'un objet d'une classe dérivée vers un type de la classe de base, se fait implicitement, en ignorant tout simplement les membres qui ne sont pas hérités.

Si par exemple, une classe `T` hérite d'une classe `B`, avec les instructions suivantes :

```

T u;
B v;
v = u;

```

il y aura conversion implicite de `u` de `T` vers `B`, autrement dit, `u` devient un objet de type `B` (tous les membres que `T` n'a pas hérité de `B` sont ignorés), puis `u` sera affecté à `v` et ceci en appelant l'opérateur d'affectation de `B` s'il est surchargé, sinon en utilisant l'affectation par défaut.

Par contre, l'instruction suivante :

```
u = v;
```

sera rejetée.

- C++ définit aussi une conversion implicite d'un pointeur (ou une référence) sur un objet d'une classe dérivée vers un pointeur (ou une référence) de type de la classe de base. Dans ce cas aussi, lors de la conversion, seuls les membres définis dans la classe de base sont accessibles.
- Notez que, la conversion d'un pointeur de la classe de base vers la classe dérivée est possible en utilisant l'opérateur de `cast`, mais cette manipulation est fortement déconseillée

Exemple 9.6 : (EXP09_06.CPP)

L'exemple suivant démontre ce concept. Avec la classe `point` et sa classe dérivée `pointcol`, cet exemple montre :

- La conversion d'un objet de la classe dérivée vers un objet de la classe de base, lors d'une affectation.
- La conversion d'une référence à un objet dérivé vers une référence de type de la classe de base, lors de l'appel d'une fonction.
- La conversion d'un pointeur d'une instance dérivée vers un pointeur de type de base, lors d'une affectation entre pointeurs.
- La conversion explicite de type de base vers le type dérivé

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

// ----- Classe de base
class point{
protected:
    int x;
    int y;
public:
    point(int abs = 0, int ord = 0){
        x = abs, y = ord;
    }
    void affiche();
    friend point inverse(point &);
};

void point::affiche(){
    cout << "(" << x << "," << y << ")" << endl;
}

point inverse(point & pt){
    int z = pt.x;
    pt.x = pt.y;
    pt.y = z;
    return pt;
}

// ----- Classe dérivée
class pointcol: public point{
    int couleur;
public:
    pointcol(int abs = 0, int ord = 0, int col = 1):point(abs,ord){
        couleur = col;
    }
    void affiche();
};

void pointcol::affiche(){
    cout << "(" << x << "," << y << ")" << " - " << couleur << endl;
}

//----- TEST
int main()
{
    pointcol a(1,2,3);
    cout << "a = " ; a.affiche();
    //--- conversion implicite de pointcol -> point
    point u;
    u = a;
    cout << "conversion dans affectation : " ; u.affiche();
    //--- conversion d'une référence
    cout << "conversion de la référence : " ; (inverse(a)).affiche();
}
```

```

//--- conversion d'un pointeur de pointcol vers un pointeur point
pointcol *pa = &a;
point * pu;
pu = pa;
cout << "conversion du pointeur de derivee vers base : " ; pu->affiche();
//----conversion d'un pointeur de point vers un pointeur pointcol
point v(5,6);
pointcol *pb ;
pb = (pointcol *)&v; // conversion avec cast
cout << "conversion du pointeur de base vers derivee : " ; pb->affiche();
return 0;
}

```

►Sortie du programme

```

a = (1,2) - 3
conversion dans affectation : (1,2)
conversion de la reference : (2,1)
conversion du pointeur de derivee vers base : (2,1)
conversion du pointeur de base vers derivee : (5,6) - -858993460

```

Remarquez que lors de la conversion d'un objet de la classe point vers un objet de type pointcol, via un pointeur, le champ couleur est resté vide (aucune valeur ne lui est affectée), ce qui montre que ce type de conversion est déconseillée.

IX.6 Typage dynamique et fonctions virtuelles

- Nous savons qu'un pointeur de la classe de base peut recevoir l'adresse de n'importe quel objet d'une des classes dérivées (après conversion). Mais avec ce pointeur, on ne peut appeler que les méthodes de la classe de base, même si le pointeur désigne un objet d'une des classes dérivées. Ce mécanisme utilisé dans C++, s'appelle le *typage statique* des pointeurs: ce qui veut dire que le type des pointeurs est déterminé lors de la compilation.

En effet, si on considère une classe T qui hérite d'une classe B, et que si dans la classe T, on définit une méthode fct() qui masque une méthode qui porte le même nom dans la classe B :

```

class B{
    ...
    type_r fct(liste_arg);
    ...
};

class T:public B{
    ...
    type_r fct(liste_arg);
    ...
};

T u;
B *pb = &u;           //pb un pointeur de type B, pointe sur un objet de type T.
pb → fct(...);       //appel de B::fct()

```

L'instruction pb → fct() appelle la méthode fct() définie dans B, même si bp pointe sur un objet de type T : typage statique.

- Pour que l'on puisse appeler les méthodes de l'objet de la classe dérivée, il faut que le type du pointeur soit désigné lors de l'exécution; on parle alors de *typage dynamique*.

Le typage dynamique est réalisé en C++ à l'aide des fonctions dites virtuelles :

- Pour définir une fonction membre d'une classe comme virtuelle, on précède son par le mot clé **virtual**. Et lorsqu'une telle méthode est invoquée par un pointeur (ou une référence), elle fait appel à la méthode correspondante à l'objet pointé par le pointeur (ou la référence).

```

class B{
    ...
    type_r virtual fct(liste_arg);
    ...
};

class T:public B{
    ...
    type_r fct(liste_arg);
    ...
};

... ...

T u;
B v;
B *pb = &u;           //pb pointe sur un objet de type T.
pb → fct(...);       //appel de T::fct()
pb = &v;             //pb pointe sur un objet de type B.
pb → fct(...);       //appel de B::fct()

```

- Notez que :
 - Si une méthode virtuelle est invoquée par un pointeur (ou une référence), le choix de la fonction à exécuter est reporté au moment de l'exécution et à ce moment là la décision est prise suivant le type de l'objet désigné par l'objet pointé.
 - Pour rendre une fonction virtuelle, il faut la déclarer virtuelle seulement dans la classe de base. Une classe dérivée peut ne pas redéfinir une fonction virtuelle.
 - Le constructeur ne peut être une méthode virtuelle, par contre le destructeur peut être déclaré virtuel.
 - Le processus réalisé par les fonctions virtuelles n'est utilisé que si la méthode est appellée par un pointeur ou par une référence.
 - La redéfinition d'une fonction virtuelle ou non masque toutes les surcharges de cette fonction dans la classe de base.
 - Les fonctions statiques ne peuvent pas être virtuelles

Exemple 9.7 : (EXP09_07.CPP)

L'exemple suivant met en œuvre le typage dynamique et le typage statique. L'exemple reprend les classes `pointcol` et `point` de l'exemple précédent en définissant une méthode virtuelle `affiche()` et une méthode non virtuelle `identifier()`.

```

// ----- Classe de base
class point{
protected:
    int x;
    int y;

public:
    point(int abs = 0, int ord = 0):x(abs),y(ord){}

    void virtual affiche(){
        cout << "">>>> affiche de Base : (" << x << ", " << y << ")\n";
    }
}

```

```

void identifier(){
    cout << "**** identifier de Base\n";
}
};

// ----- Classe derivée
class pointcol: public point{
    int couleur;
public:
    pointcol(int abs = 0, int ord = 0, int col = 1):point(abs,ord){
        couleur = col;
    }

    void affiche(){
        cout << ">>> affiche de derivee : (" << x << "," << y << ")" <<
" - " << couleur << endl;
    }

    void identifier(){
        cout << "**** identifier de derivee " << endl;
    }
};

//----- TEST
int main()
{
    pointcol a(1,2,3);
    point u(5,6);
    point *pb;
    cout << "--- pointeur de base designe un objet derive ---\n";
    pb = &a;
    pb->affiche();
    pb->identifier();
    cout << "--- le meme pointeur designe maintenant un objet de base\n";
    pb = &u;
    pb->affiche();
    pb->identifier();
    return 0;
}

```

► Sortie du programme :

```

--- pointeur de base designe un objet derive ---
>>> affiche de derivee : (1,2) - 3
**** identifier de Base
--- le meme pointeur designe maintenant un objet de base
>>> affiche de Base : (5,6)
**** identifier de Base

```

Remarquez que l'instruction `pb->identifier()` fait toujours référence à `point::identifier()`, même si l'objet pointé est de type `pointcol`, contrairement à `pb->affiche()`. qui fait appel à la fonction correspondante à l'objet pointé par `pb`.

Destructeur virtuel

La notion des fonctions virtuelles, nous offre donc une utilisation efficace d'un pointeur de type de base pour manipuler les objets des classes dérivées. Mais la destruction d'un tel pointeur devra faire appel au destructeur de l'objet pointé et non de la classe de base :

Exemple 9.8 : (EXP09_08.CPP)

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

```

// ----- Classe de base
class point{
protected:
    int x;
    int y;
public:
    point(int abs = 0, int ord = 0):x(abs),y(ord){}
    ~point(){
        cout << "--- Destructeur base" << endl;
    }
    void virtual affiche(){
        cout << "(" << x << "," << y << ")" << endl;
    }
};

// ----- Classe derivée
class pointcol: public point{
    int couleur;
public:
    pointcol(int abs = 0, int ord = 0, int col = 1):point(abs,ord){
        couleur = col;
    }
    ~pointcol(){
        cout << "--- Destructeur derivee" << endl;
    }
    void affiche(){
        cout << "(" << x << "," << y << ")" << " - " << couleur << endl;
    }
};

//----- TEST
int main()
{
    cout << "---destruction d'un pointeur qui designe un objet derive\n";
    point *pb1 = new pointcol;
    pb1->affiche();
    delete pb1;
    cout << "\n----- FIN PROGRAMME -----";
    return 0;
}

```

►Sortie du programme :

```

---destruction d'un pointeur qui designe un objet derive
(0,0) - 1
--- Destructeur base
----- FIN PROGRAMME -----

```

Remarquez que, la destruction du pointeur pb1 de type point, qui pointe sur un objet de type pointcol, ne fait appel qu'au destructeur de la classe de base.

Pour résoudre ce problème, on définit le destructeur de la classe de base comme virtuel, ce qui assure la destruction de l'objet réellement pointé, c'est-à-dire que si l'objet pointé est un objet dérivé il faut exécuter le destructeur de la classe dérivée puis celui de la classe de base.

En déclarant le destructeur de la classe de base point comme méthode virtuelle :

```

...
// ----- Classe de base

```

```

class point{
...
    virtual ~point(){
        cout << "--- Destructeur base" << endl;
    }
...
} ;
...

```

► La sortie du programme devient :

```

---destruction d'un pointeur qui designe un objet derive
(0,0) - 1
--- Destructeur derivee
--- Destructeur base
----- FIN PROGRAMME -----

```

Ce qui permet de libérer tout l'espace mémoire réservé lors de la création de pb.

Polymorphisme

Lorsque plusieurs classes dérivées d'une classe de base, redéfinissent une même fonction membre virtuelle de la classe de base, lors de l'appel de celle-ci par un pointeur (ou une référence), la fonction à exécuter est choisie lors de l'exécution du programme et suivant le type de l'objet pointé: ce processus s'appelle *polymorphisme*.

Exemple 9.9 : (EXP09_09.CPP)

L'exemple suivant met en œuvre ce processus

```

#include <iostream>
#include <string.h>
using namespace std;

// ----- Classe de base
class forme{
protected:
    char * _nom;
public:
    forme(const char * nom){
        _nom = new char [strlen(nom)+1];
        strcpy(_nom,nom);
    }

    virtual ~forme(){
        if (_nom != NULL)
            delete _nom;
    }

    void virtual affiche(){}
};

//----- classes dérivées
//-----rectangle
class rectangle : public forme{
    double _largeur;
    double _longueur;
public:
    rectangle (const char* nom,
               double longueur, double largeur): forme(nom)
    {
        _longueur = longueur; _largeur = largeur;
    }
    void affiche(){
        cout << _nom << " :\t\tRectangle [ "
    }
}

```

```

        << _longueur << ", " << _largeur << " ]\n";
    }
};

//-----ovale
class ovale : public forme{
    double rayon_1;
    double rayon_2;
public:
    ovale (const char* nom, double rayon1, double rayon2): forme(nom){
        rayon_1 = rayon1; rayon_2 = rayon2;
    }
    void affiche(){
        cout << _nom << " :\t\tOvale [ "
            << rayon_1 << ", " << rayon_2 << " ]\n";
    }
};

//-----rond
class rond : public forme{
    double _rayon;
public:
    rond (const char* nom, double rayon): forme(nom){
        _rayon = rayon;
    }
    void affiche(){
        cout << _nom << " :\t\tRond [ " << _rayon << " ]\n";
    }
};

//-----carre
class carre : public forme{
    double _cote;
public:
    carre (const char* nom, double cote): forme(nom){
        _cote = cote;
    }
    void affiche(){
        cout << _nom << " :\t\tCarre [ " << _cote << " ]\n";
    }
};

//----- TEST
int main()
{
    ovale          salon ("Salon", 3.5, 4.0);
    carre         sdb ("SDB", 3.0);
    rectangle     salle("Salle", 5.2, 4.5),
                  garage("Garage", 6.5, 4.0);
    rond          cour("Cour", 3.2);

    forme * tb[5];
    tb[0] = &salon;
    tb[1] = &sdb;
    tb[2] = &salle;
    tb[3] = &cour;
    tb[4] = &garage;

    for(int i = 0; i < 5; i++)
        tb[i]->affiche();

    cout << "\n----- FIN PROGRAMME -----" << endl;
}

```

```
    return 0;
}
```

► Sortie du programme :

```
Salon:          Oval [3.5,4]
SDB:           Carré [3]
Salle:          Rectangle [5.2,4.5]
Cour:           Rond [3.2]
Garage:         Rectangle [6.5,4]

----- FIN PROGRAMME -----
```

Remarquons que dans cette exemple, la fonction `affiche()` de la classe de base, n'a aucun code. Elle ne figure que pour permettre l'utilisation de ses redéfinitions définies dans les classes dérivées.

IX.7 Fonctions virtuelles pures et classes abstraites

- Une *fonction virtuelle pure* est une fonction membre virtuelle qui est déclarée mais non définie dans la classe de base. Pour déclarer une telle fonction on utilise la syntaxe suivante :

```
type_r virtual nom_fct(liste_arg) = 0
```

" = 0 " signifie que la fonction n'est pas définie.

- Lorsqu'une classe contient une méthode virtuelle pure, on ne peut déclarer aucun objet de cette classe : une telle classe est appelée *classe abstraite*.
- Lorsqu'une classe dérive d'une classe abstraite, elle doit définir toutes les méthodes virtuelles pures héritées. Sinon, elle devient à son tour une classe abstraite.
- Le mécanisme des classes abstraites, permet de définir des classes de bases contenant les caractéristiques d'un ensemble de classes dérivées, pour pouvoir les manipuler avec un type unique de pointeur.

Exemple 9.10 : (EXP09_08.CPP)

Cet exemple reprend l'exemple précédent, en déclarent la méthode `affiche()` comme virtuelle pure.

```
...
// ----- Classe de base abstraite
class forme{
protected:
    char * _nom;
public:
    forme(const char * nom){
        _nom = new char [strlen(nom)+1];
        strcpy(_nom,nom);
    }
    ~forme(){
        if (_nom != NULL)
            delete _nom;
    }

    void virtual affiche() = 0;           // fct virtuelle pure
};
...
```

La sortie du programme est exactement la même que celle de l'exemple précédent.

- Notez, enfin, qu'on peut déclarer et utiliser un pointeur d'une classe abstraite, mais qu'on ne peut pas utiliser une instance d'une telle classe.

Chapitre X

Héritage multiple

- On parle d'héritage multiple lorsqu'une classe dérivée admet plusieurs classes de base directes.
- L'héritage multiple est une généralisation de l'héritage simple. La plupart des résultats et les notions mentionnés dans l'héritage simple restent valables pour l'héritage multiple (En effet si T dérive de deux classes A et B, alors T est une classe dérivée de A (resp de B) dans le sens de l'héritage simple). Cependant, quelques problèmes qui n'existaient pas dans l'héritage simple apparaissent :
 - ◆ Conflits entre identificateurs des membres hérités
 - ◆ Duplication des membres
 - ◆ Conflits d'initialisations des constructeurs des différentes classes de base

X.1 Un exemple d'héritage multiple

- Notons premièrement que si une classe T hérite de plusieurs classes A, B, ...

```
class T : mode_a A, mode_b B, ...
{ ... };
```

où mode_a (mode_b, ...) désigne le mode d'héritage (public, protégé ou public)

le constructeur de T doit fournir les paramètres nécessaires à tous les constructeurs de ses classes de base. Pour cela on utilise la syntaxe suivante:

```
T :: T( ... ) : A( ... ), B( ... ), ...
```

De plus, avant tout appel du constructeur de T, il y aura appel des constructeurs de classes de base suivant l'ordre de la liste des classes de base qui figure dans l'entête de T. L'appel des destructeurs se fait dans l'ordre inverse.

Exemple 10.1 : (EXP10_01.CPP)

L'exemple suivant montre le cas d'un héritage multiple. On y définit deux classes point et couleur, et une classe pointcol qui dérive de ces deux classes :

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

// ----- Classes de base
class point{
protected:
    int x;
    int y;
public:
    point(int, int);
    ~point();
    void affiche_point();
};

point::point(int abs, int ord){
    x = abs; y = ord;
    cout << " --- constr de point" << endl;
}
point::~point(){
    cout << " --- destr de point " << endl;
}
void point::affiche_point(){
    cout << "(" << x << ", " << y << ")";
}

class couleur{
protected:
    unsigned int _couleur;
public:
    couleur(unsigned int);
    ~couleur();
    void affiche_couleur();
};

couleur::couleur(unsigned int c){
    _couleur = c;
    cout << " --- constr de couleur" << endl;
}
couleur::~couleur(){
    cout << " --- destr de couleur " << endl;
}
void couleur::affiche_couleur(){
    cout << "(" << _couleur << ")";
}

// ----- Classe dérivée
class pointcol: public point, public couleur
{
public:
    pointcol(int, int, unsigned int);
}
```

```

~pointcol();

};

pointcol::pointcol(int abs, int ord, unsigned int c):point(abs,ord),
couleur(c)
{
    cout << "--- constr de pointcol" << endl;
}
pointcol::~pointcol(){
    cout << "--- destr de pointcol" << endl;
}

//----- TEST
int main()
{
    pointcol a(1,2,3);
    a.affiche_point();
    cout << " - ";
    a.affiche_couleur();
    cout << endl;
    return 0;
}

```

► Sortie du programme

```

--- constr de point
--- constr de couleur
--- constr de pointcol
(1,2) - (3)
--- destr de pointcol
--- destr de couleur
--- destr de point

```

Notez l'ordre d'appel des constructeurs et des destructeurs.

X.2 Résolution des conflits entre identificateurs

- Etant donné une classe T qui hérite de deux classes A et B. Si un membre de B possède le même identificateur que celui d'un membre de A, alors T possèdera deux membres différents qui portent le même nom et tout accès à l'un de ces deux membres sera ambiguë. Pour résoudre cette ambiguïté il faut utiliser l'opérateur de résolution de portée :

```

class A { ... float x; ...}
class B { ... int x; ...}
class T: public A, public B { ... ...}

T u;
u.A::x;           // fait référence au réel x défini dans A
u.B::x;           // fait référence à l'entier x défini dans B
u.x              // rejeté, ambiguïté

```

Exemple 10.2 : (EXP10_02.CPP)

Cet exemple reprend l'exemple précédent dans lequel on renomme les fonctions membres `affiche_point()` et `affiche_couleur()` en `affiche()`. Dans ce cas le programme test devient :

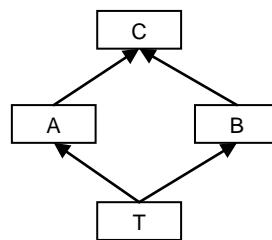
```

int main()
{
    pointcol a(1,2,3);
    a.point::affiche();
    cout << " - ";
    a.couleur::affiche();
    cout << endl;
    return 0;
}

```

X.3 Classes virtuelles

- Supposons qu'une classe T hérite de deux classes A et B et que ces deux classes héritent d'une autre classe C :



T hérite donc deux fois de C par l'intermédiaire de A et de B. Autrement dit, si A et B héritent de C un membre nommé x, la classe T aura deux membres nommés x, (un hérité de A et l'autre hérité de B) qui désignent la même donnée. Dans ce cas pour accéder à ce membre dans T, on doit spécifier le chemin complet suivant l'arborescence de l'héritage :

```

T u;
u.A::C::x;           // fait référence à x hérité de C via A
u.B::C::x;           // fait référence à x hérité de C via B

```

Or ceci n'est pratique ni efficace, en plus généralement une seule copie du membre x est suffisante.

- Pour résoudre ce problème, on déclare *virtuelle* la classe de base commune C dans la spécification de l'héritage de A et B comme suit :

```

class C {... ...};
class A : public virtual C
{... ...};
class B : public virtual C
{... ...};
class T: public A, public B {... ...}

```

dans ce cas, le compilateur n'incorpore qu'une seule copie des membres hérités de C dans T et l'accès à ces membres se fera son l'utilisation de l'opérateur de résolution de portée.

- Cependant, il faut noter que :

- Le constructeur d'une classe qui hérite directement ou indirectement d'une classe virtuelle doit fournir les paramètres nécessaires au constructeur de cette classe virtuelle. Dans le cas de notre exemple le constructeur de T doit fournir les paramètres nécessaires au constructeur de C comme suit :

T::T(...) : C(...), A(...), B(...)

- ♦ Si un membre de la classe de base virtuelle commune est redéfini dans une des classes dérivées, c'est la définition de la classe dérivée qui prédomine. Si par exemple `fct()` est une fonction membre de `C` qui est redéfinie dans `B`, alors tout appel de cette fonction dans `T` ou avec une instance de `T` invoquera `B::fct()`.

Chapitre XI

Les exceptions

- Les méthodes traditionnelles de gestion d'erreurs d'exécution consistent à :
 - traiter localement l'erreur : la fonction doit prévoir tous les cas possibles qui peuvent provoquer l'erreur et les traiter localement.
 - retourner un code d'erreur : la fonction qui rencontre une erreur retourne un code et laisse la gestion de l'erreur à la fonction appelante.
 - Arrêter l'exécution du programme (*abort*,...)
 - Etc. ...
- C++ introduit la notion d'exception:
 - Une exception est l'interruption de l'exécution d'un programme à la suite d'un événement particulier.
 - Une fonction qui rencontre un problème *lance une exception*, l'exécution du programme s'arrête et le contrôle est passé à un gestionnaire (*handler*) d'exceptions, qui traitera cette erreur, on dit qu'il *intercepte l'exception*.
- Ainsi, le code d'une fonction s'écrit normalement sans tenir compte des cas particuliers puisque ces derniers seront traités par le gestionnaire des exceptions.
- La notion d'exception repose donc sur l'indépendance de la détection de l'erreur et de son traitement. Elle permet de séparer le code de gestion de l'erreur et du code où se produit l'erreur ce qui donne une lisibilité et une modularité de traitement d'erreurs.

XI.1 Lancement et récupération d'une exception

- Les exceptions sont levées dans un bloc **try** :

```
try
{
    // Un code qui peut lever une ou plusieurs exceptions
    // Les fonctions appelées ici peuvent aussi lever une exception
}
```

- Pour lancer une exception on utilise le mot clé **throw** suivit d'un paramètre caractérisant l'exception :

```
throw e;           // e est un parametre de n'importe quel type
```

Si le type de **e** est **X**, on dit que l'exception levée est de type **X**.

- Une exception levée est interceptée par le gestionnaire d'exceptions correspondant qui vient juste après le bloc **try** et qui est formé par un bloc **catch**.

```
catch( type_exception & id_e)
{
    // Code de gestion des exceptions levée par un paramètre de type
    // type_exception

    // l'argument id_e est facultatif, il représente le paramètre avec
    // lequel l'exception a été levée

    // le passage de ce paramètre peut être par valeur ou par référence
}
```

- Plusieurs gestionnaires peuvent être placés après le bloc **try**. Ces gestionnaires diffèrent par le type de leur argument. Le type de l'argument d'un gestionnaire précise le type de l'exception à traiter.
- Chaque exception levée dans le bloc **try**, est interceptée par le premier gestionnaire correspondant (dont le type d'argument correspond à celui de l'exception levée) qui vient juste après le bloc **try**.
- C++ permet l'utilisation d'un gestionnaire sans argument, dit aussi *Le gestionnaire universel* :

```
catch( ... )
{
// Code de gestion de l'exception levée par un paramètre de type quelconque
}
```

qui permet d'intercepter toutes les exceptions. Ce gestionnaire est normalement placé à la fin pour intercepter les gestions qui n'ont pas de bloc **catch** correspondant.

Exemple 11.1 : (EXP11_01.CPP)

L'exemple suivant montre une utilisation simple des exceptions en C++.

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

----- une classe d'exceptions
class erreur
{
    const char * nom_erreur;
public:
```

```

        erreur ( const char * s):nom_erreur(s){}
        const char * raison(){ return nom_erreur; }
    };

//----- une classe test
class T{
    int _x;
public:
    T(int x = 0):_x(x){ cout << "\n +++ Constructeur\n"; }
    ~T(){ cout << "\n --- Destructeur\n"; }
    void Afficher(){ cout << "\nAffichage : " << _x << endl; }
};

//----- une fonction qui lève une exception de type 'erreur'
void fct_leve_exception()
{
    cout << "\n-----entree fonction \n";
    throw erreur("Exception de type 'erreur'");
    cout << "\n-----sortie fonction \n";
}

int main()
{
    int i;
    cout << "entrer 0 pour lever une exception de type classe\n";
    cout << "           1 pour lever une exception de type entier\n";
    cout << "           2 pour lever une exception de type char\n";
    cout << "           3 pour ne lever aucune exception \n";
    cout << "           ou une autre valeur pour lever une exception de type
quelconque \n";
    try
    {
        T t(4);
        cout << "---> "; cin >> i;
        switch(i){
            case 0:
                fct_leve_exception(); // lance une exception de type
                                      // 'erreur'
                break;
            case 1:
            {
                int j = 1;
                throw j; // lance une exception de type int
            }
            break;
            case 2:
            {
                char c = ' ';
                throw c; // lance une exception de type char
            }
            break;
            case 3:
            {
                break;
            }
            default:
            {
                float r = 0.0;
                throw r; // lance une exception de type float
            }
            break;
        }
    }
}

```

```

}

***** aucune instruction ne doit figurer ici *****

// Gestionnaires des exceptions
catch(erreur & e)
{
    cout << "\nException : " << e.raison() << endl;
}
catch(int & e)
{
    cout << "\nException : type entier " << e << endl;
}
catch(char & e)
{
    cout << "\nException : type char " << e << endl;
}
catch(...)
{
    cout << "\nException quelconque" << endl;
}

cout << "\n***** FIN PROGRAMME *****\n";
return 0;

}

```

- Notez que les objets automatiques définis dans un bloc `try` sont automatiquement détruits lorsqu'une exception est levée, il en est de même pour les objets utilisés comme paramètres de `throw`.

XI.2 La logique de traitement des exceptions

- Si aucune exception n'est levée dans un bloc `try`, les blocs `catch` qui suivent seront ignorés et le programme continue sur l'instruction après les blocs `catch`.
- Lorsqu'une exception est levée
 - ◆ Si l'instruction ne se trouve dans aucun bloc `try`, il y a appel de la fonction prédefinie `terminate()` : cette fonction provoque l'arrêt immédiat du programme (aucun destructeur n'est appelé, ni aucun code de nettoyage). Notez qu'il est possible de modifier le fonctionnement de `terminate()` en utilisant `set_terminate()`.
 - ◆ Si l'instruction se trouve dans un bloc `try`, le programme saute directement vers les gestionnaires d'exceptions qui suivent le bloc `try`. Ces gestionnaires sont examinés séquentiellement dans l'ordre du code
 - Si l'un des gestionnaires correspond, il est exécuté. S'il ne provoque pas lui-même une exception ou n'arrête pas le programme, le programme continue normalement après les blocs `catch`.
 - Si aucun gestionnaire ne correspond à l'exception levée, celle-ci est propagée au niveau supérieur de traitements d'exceptions (le bloc `try` de niveau supérieur en cas de blocs `try` imbriqués) jusqu'à arriver au programme principale qui lui appellera la fonction `terminate()`, ce qui provoquera un arrêt anormal du programme.

- Normalement un gestionnaire doit effectuer tous les traitements d'erreurs qui ne sont pas faits automatiquement par C++ (fermeture des fichiers, connexions réseaux, rétablissement des l'état des données,...), il peut ensuite arrêter le programme ou lui permettre de continuer. Mais, il peut aussi relancer l'exception qu'il vient d'intercepter ou relancer une autre exception, ce qui arrive parfois si le gestionnaire ne peut traiter l'exception et estime que l'erreur ne justifie pas l'arrêt du programme. Pour relancer l'exception interceptée, on utilise le mot `throw` sans argument.

Exemple d'utilisation de `set_terminate()`

- On a vu qu'il est possible de contrôler l'arrêt d'un programme en utilisant la fonction `set_terminate()`, en cas d'arrêt du programme suite à une exception qui ne trouve pas de gestionnaire. Cette fonction reçoit comme argument l'adresse d'une fonction qui sera appelée dans ce cas :

```
set_terminate(& fct_traitement_erreur);
```

où `fct_traitement_erreur()` est une fonction sans arguments et qui ne retourne rien.

Exemple 11.2 : (EXP11_02.CPP)

L'exemple suivant montre comment utiliser `set_terminate()`.

```
#include <iostream> // bibliothèque e/s
#include <exception> // bibliothèque exception
using namespace std;

//-----
// fonction qui sera appeler en cas de lancement d'une exception
// n'admettant pas de gestionnaire
//-----
void fct_gestionnaire()
{
    cout << "\nException non gérée.....\n";
    cout << "\n----- FIN DU PROGRAMME ----- \n";
    exit(-1);
}

int main()
{
    set_terminate(&fct_gestionnaire);
    try
    {
        cout << "\n***** LANCEMENT D'UNE EXCEPTION\n";
        throw 2.1; // exception de type double
    }

    catch(int & i)
    {
        cout << "\nException : type entier " << i << endl;
    }

    cout << "\n***** FIN NORMALE DU PROGRAMME *****\n";
    return 0;
}
```

► Sortie du programme

```
***** LANCEMENT D'UNE EXCEPTION
Exception non gérée.....
----- FIN DU PROGRAMME -----
```

- Notez que `set_terminate()` est définie dans la bibliothèque `<exception>` et que pour pouvoir l'utiliser on doit utiliser le namespace (l'espace de noms) `std` (voir chapitre 12)

XI.3 Déclaration et classification des exceptions

- Les exceptions peuvent être de n'importe quel type, mais pour bien gérer ces exceptions, il est préférable d'utiliser les types classes. Ce qui permettra entre autres d'utiliser des exceptions qui contiennent des informations (données et fonctions membres). En plus la technique de l'héritage permet d'organiser et de regrouper les exceptions
- On peut par exemple définir une classe de base

```
class Erreur {...};
```

et construire en suite les autres exceptions de type classe autour de cette classe de base, comme par exemple :

```
// erreurs de calcul
class MathErr: public Erreur {...};
class DepassErr: public MathErr {...};
class InvalidOpErr: public MathErr{...};
// erreurs d'E/S sur fichiers
class FichierErr:public Erreur{...};
class FinFichier:public FichierErr{...};
class NomInvalide:public FichierErr{...};
...
```

Ainsi une erreur de calcul qui n'a pas une classe correspondante sera traitée par la classe `MathErr`, et une erreur qui n'est ni de calcul ni d'opérations sur les fichiers sera traitée par la classe de base `Erreur`.

XI.4 Liste des exceptions autorisées dans une fonction

- Il est possible de spécifier les exceptions qui peuvent être lancées par une fonction :

```
type_r fct(liste_arg)
{
    throw(liste_types_exceptions);
```

où `liste_types_exceptions` est une liste de type des exceptions, séparés par des virgules, qui peuvent être lancer par la fonction `fct()`.

- Si dans une telle fonction, on lance une exception qui ne figure pas dans la liste des exceptions permises, la fonction `unexpected()` sera appelée et qui de sa part appelle la fonction `terminate()`.
- Notez qu'on peut changer le comportement de la fonction `unexpected()` en utilisant la fonction `set_unexpected()`. Ces deux fonctions sont définies dans la bibliothèque `<exception>`.

Exemple 11.3 : (EXP11_03.CPP)

Exemple d'utilisation de `set_unexpected()`.

```
#include <iostream> // bibliothèque e/s
#include <exception> // bibliothèque exception

using namespace std;

//-----
// fonction qui sera appeler en cas de lancement d'une exception
// non déclarée dans une fonction
//-----
void fct_gestionnaire()
{
    cout << "\nException non autorisée par la fonction.....\n";
    cout << "\nRelance une autre exception\n";
    // relance une autre exception
    throw 2;
}
//-----
// fonction qui spécifie les exceptions autorisées
//-----
int fct() throw (int)
{
    char c = 'a';
    // lance une exception de type non autorisé
    throw c;
}

int main()
{
    set_unexpected(& fct_gestionnaire);
    try
    {
        fct();
    }
    catch(int i)
    {
        cout << "Exception de type int : " << i << endl;
    }
    catch(double)
    {
        cout << "Exception de type double\n";
    }

    cout << "\n***** FIN NORMALE DU PROGRAMME *****\n";
    return 0;
}
```

► Sortie du programme

```
Exception non autorisée par la fonction.....  

Relance une autre exception  

Exception de type int : 2  

***** FIN NORMALE DU PROGRAMME *****
```

XI.5 Exceptions et constructeurs

- Il est parfaitement légal de lancer une exception dans un constructeur, ce qui permettra de signaler une erreur lors de la construction d'un objet.

- Lorsqu'une exception est lancée à partir d'un constructeur, la construction de l'objet échoue. Par suite, le destructeur pour cet objet ne sera pas appelé, ce qui pose certains problèmes si l'objet est partiellement initialisé.

Exemple 11.4 : (EXP11_04.CPP)

L'exemple suivant montre comment traiter le cas des objets partiellement initialisés, lorsque la création de ces objets échoue.

Considérons une classe "etudiant" qui permet de manipuler les notes d'un étudiant.

Les notes de chaque étudiant seront stockées dans des tableaux d'entiers. On suppose que le nombre de notes est variable et qu'il ne doit pas dépasser une certaine valeur MAXNBNOTE.

On prévoit aussi dans cette classe une surcharge de l'opérateur [] qui permet l'accès en lecture et en écriture à un note donnée

Pour gérer les différentes erreurs, on définit une classe de base "Erreur" et deux autres classes DepMaxNbNote et InvalideIndice pour gérer respectivement le dépassement de nombre de notes autorisé et le dépassement d'indice.

```
#include <iostream>
#include <string>

#define MAXNBNOTE 4

#define MAXSIZE 255

using namespace std;
//----- classes exception
class erreur{
protected:
    const char * raison;
public:
    erreur():raison(0){}
    erreur(const char* s):raison(s){}
    virtual void affiche(){
        if (raison == NULL)
            cout << "Erreur inconnu..." << endl;
        else
            cout << raison << endl;
    }
};

class DepMaxNbNote:public erreur{
public:
    DepMaxNbNote():erreur(){}
    DepMaxNbNote(const char *s):erreur(s){}
    void affiche(){
        erreur::affiche();
        cout << "Nombre max de notes est " << MAXNBNOTE << endl;
    }
};

class InvalideIndice:public erreur{
public:
    InvalideIndice():erreur(){}
    InvalideIndice(const char *s):erreur(s){}
    void affiche(int i){
        erreur::affiche();
        cout << "Indice doit etre entre 0 et " << i << endl;
    }
};
//----- classe etudiant
```

```

class etudiant{
    char * _nom;                                // nom
    int _nbnotes;                               // nombre de notes
    int * tabnotes;                            // tableau de notes
public:
    etudiant(char*,int);
    ~etudiant();
    int GetNbNotes(){ return _nbnotes; }
    int & operator[](int);
    void affiche();

};

//----- constructeur
etudiant::etudiant(char* nom, int nbnotes)
{
    try
    {
        _nom = new char[strlen(nom)+1];
        strcpy(_nom,nom);
        if( (nbnotes < 0) || (nbnotes > MAXNBNOTE) ){
            DepMaxNbNote err("Erreur dans Constructeur");
            throw err;
        }
        _nbnotes = nbnotes;
        tabnotes = new int[nbnotes];
    }
    catch(DepMaxNbNote & err)
    {
        delete _nom;
        err.affiche();
        exit(-1);
    }
}

//----- destructeur
etudiant::~etudiant()
{
    delete tabnotes;
    delete _nom;
}

//----- operateur []
int & etudiant::operator [](int i)
{
    if( (i < 0) || (i >= _nbnotes) )
        throw InvalideIndice("Indice invalide...");
    return tabnotes[i];
}

//----- affiche les notes
void etudiant::affiche()
{
    cout << "Notes de " << _nom << endl;
    for (int i = 0; i < _nbnotes; i++)
        cout << " [ " << tabnotes[i] << " ] ";
    cout << endl;
}

//----- programme principal
int main()
{
    char nom[MAXSIZE];
    int n;

```

```

cout << "NOM : ";
cin >> nom;
cout << "Entrer le nombre de notes : ";
cin >> n;
etudiant e(nom,n);
for(int i = 0; i < n; i++){
    cout << "Entrer la note [" << i << "] : ";
    cin >> e[i];
}
cout << endl;
e.affiche();
cout << "Entrer le numero de la note a modifier : ";
cin >> n;
try
{
    cout << "la note est : " << e[n] << endl;
    cout << "Entrer la nouvelle note : ";
    cin >> e[n];
}
catch(InvalidIndice & err)
{
    err.affiche(e.GetNbNotes()-1);
}
e.affiche();
cout << "\n***** FIN PROGRAMME *****\n";
return 0;
}

```

Le programme demande le nom de l'étudiant, le nombre de notes, les notes puis permet de modifier une note en saisissant son indice.

Lorsque l'utilisateur entre un nombre de notes supérieur à MAXNBNOTE ou lorsqu'il entre un indice d'une note qui n'existe pas, le programme s'arrête

Chapitre XII

Namespaces

- Un *namespace* (espace de nommage ou espace de noms) est une zone qui permet de délimiter la recherche des noms des identificateurs par le compilateur. Il sert essentiellement à regrouper les identificateurs pour éviter les conflits de noms entre plusieurs parties d'un projet
- C++ offre un seul namespace de portée globale, dans lequel il ne doit avoir aucun conflit de nom. Avec les namespaces non globaux, on peut définir des objets globaux, sans qu'ils aient une portée globale

Définition

- On définit un espace de noms avec le mot clé `namespace` comme suit :

```
namespace nom
{
    // Déclarations et définitions
}
```

Où `nom` est le nom de l'espace de noms

- La définition d'un namespace peut être découpée en plusieurs parties, comme par exemple:

```
namespace A
{
    int n;
}
namespace B
{
    int n;
    float a;
}
namespace A
{
    float b;
}
```

- Un namespace peut contenir d'autres namespaces. D'autre part, on peut définir un namespace sans nom, les espaces de noms qui n'ont pas de noms sont dits *anonymes*. L'espace de nom de portée globale définit dans C++ est un espace de nom anonyme.
- Il est possible de donner un autre nom (alias) à un espace de noms, pour ce, on utilise la syntaxe suivante :

```
namespace nom_alias = nom;
```

Accès et définition des membres d'un namespace

- Pour accéder à un objet ou une fonction définie dans un namespace à l'extérieur de celui-ci, on utilise l'opérateur de résolution de portée :

```
int n;           // globale
namespace A
{
    int n;
    void f1();
    int f2() {... ...}
}
void A::f1() { ... ...}
...
void main()
{
    A::n = 0;    // n définit dans A
    n = 1;        // n globale
}
```

- Si, dans un espace de noms, un identificateur est déclaré avec le même nom qu'un autre identificateur déclaré dans un espace de noms plus global, l'identificateur global est masqué.
- Pour accéder à un identificateur en dehors de son espace de noms, il faut utiliser un nom complètement qualifié à l'aide de l'opérateur de résolution de portée. Toutefois, si l'espace de noms dans lequel il est défini est un espace de noms anonyme, cet identificateur ne pourra pas être référencé, puisqu'on ne peut pas préciser le nom des espaces de noms anonymes.

Déclaration using

- La déclaration **using** permet d'utiliser un identificateur d'un espace de noms de manière simplifiée, sans avoir à spécifier son nom complet :

```
using identificateur;
```

Ainsi, on a par exemple :

```
namespace A
{
    int n;
}
...
void main()
{
    using A::n;
    n = 1;           // fait référence à A::n;
}
```

- Une déclaration **using** peut être utilisée dans la définition d'une classe. Dans ce cas, elle doit se rapporter à une classe de base de la classe dans laquelle elle est utilisée. De plus, l'identificateur donné à la déclaration **using** doit être accessible dans la classe de base.

```
namespace A
{
    float f;
}
class B
{
    int i;
public:
    int j;
};
class T : public B
{
    using A::f; // Illégal : f n'est pas dans une classe de base.
    using B::i; // Interdit : B::i est inaccessible
public:
    using B::j; // Légal.
};
```

- La déclaration **using** sert au fait à rétablir certains droits d'accès, modifiés par un héritage : Pour cela on doit déclarer le membre avec **using** dans une zone de déclaration de même type.

```
class B
{
public:
    int i;
    int j;
};
class T : private B
{
public:
    using B::i;           // Rétablit l'accessibilité sur B::i.
protected:
    using B::j;           // Interdit : zone de déclaration différente
};
```

Directive using

La directive `using` permet d'utiliser tous les identificateurs d'un espace de noms, sans spécification de cet espace de nom

```
using namespace nom;
```

où `nom` est le nom de l'espace de noms dont les identificateurs seront utilisés sans qualification complète.

```
namespace A
{
    int n;
    void f1();
}

...
void fct()
{
    using namespace A;
    n = 0;           // référence à A::n
    f1();           // appel de A::f1()
}
```

Chapitre XIII

Template

- Template signifie modèle ou patron en anglais.
- C++ nous permet de définir des fonctions et des classes modèles, dites aussi fonctions template et classes template ou encore fonctions génériques et classes génériques.
- Les template sont au fait des classes et des fonctions définies avec des paramètres qui sont soit des objets de type générique (quelconque) ou des constantes de type intégral. Lors de l'utilisation d'une telle template, le compilateur génère un code en remplaçant les paramètres génériques par les vrais types ou les vraies valeurs : on parle alors d'*instanciation* de la template.
- Pour déclarer les template, on utilise le mot clé `template` suivi d'une liste de paramètres puis la définition d'une fonction ou d'une classe :

```
template <liste_param_template> définition
```

XIII.1 Paramètres template

- La liste des paramètres template est une liste d'un ou plusieurs paramètres séparés par des virgules et qui peuvent être :
 - ◆ Soit des types génériques précédés par l'un des mots clés **class** ou **typename**.
 - ◆ Soit des constantes dont le type est assimilable à un type intégral.

Paramètres : Types génériques

- L'emploi des types génériques se fait par exemple, comme suit :

```
template <class T, typename U, class V> ... ...
```

T, U et V sont des types génériques qui peuvent être remplacés par n'importe quel type prédéfini ou précédemment défini par l'utilisateur.

- Les mots clés **class** et **typename** peuvent être utilisés indifféremment, ils ont exactement la signification de "type"
- Les types génériques peuvent avoir des valeurs par défaut, comme par exemple

```
template <class T, typename U, class V = int> ... ...
```

Les paramètres qui ont une valeur par défaut doivent figurer à la fin de la liste des paramètres template. (Pour certains compilateurs, cette fonctionnalité est ignorée pour les fonctions templates)

- Le mot clé **typename** est utilisé pour désigner un type avec un identificateur, comme par exemple, la déclaration :

```
typename X;
```

permet d'utiliser dans la suite, X comme un type. Ce mot clé ne peut être utilisé que dans les template.

- Il est possible d'utiliser une classe template comme type générique, comme par exemple :

```
template <template <class T> class A, ... ...>
```

Le type T est le type générique utilisé dans la définition de la classe template A.

Paramètres : Constantes

- Pour déclarer des constantes parmi les paramètres template, on procède comme suit:

```
template <type_param id_param, ... ...>
```

type_param est le type du paramètre constant et **id_param** est le nom de ce paramètre.

- Les constantes template peuvent être :
 - ◆ De type intégral (int, long, short, char et wchar_t signés et non signés) ou énuméré.
 - ◆ Pointeur ou référence d'objet.
 - ◆ Pointeur ou référence de fonction
 - ◆ Pointeur sur un membre.

Comme par exemple :

```
template <class T, void (*fct)(int), int N>
```

déclare une template qui comprend un type générique T, un pointeur sur une fonction qui à un paramètre de type int et qui ne retourne rien et une constante N de type int.

- Les constantes template peuvent avoir des valeurs par défaut dans une classe template. Les fonctions template n'acceptent pas des valeurs par défaut pour ses constantes template (pour certains compilateurs ces valeurs sont acceptables mais ignorées). Rappelons que les paramètres qui ont une valeur par défaut doivent être placés à la fin de la liste.

XIII.2 Les fonctions template

- La déclaration et la définition des fonctions template se fait comme suit :

```
template <liste_param_template>
type_r nom_fonction(lst_param);
```

Pour définir par exemple une fonction générique maximum qui permet de retourner le maximum de deux objets, on procède comme suit :

```
template <class T>
T maximum (T u, T v)
{
    return ( u > v ) ? u : v;
}
```

- Pour utiliser une fonction template il faut l'instancier. Cette instanciation peut être implicite ou explicite, comme par exemple :

```
int i = maximum (2,3);           // instanciation implicite (T<-int)
int i = maximum <int>(2,3)      // instanciation explicite
```

- Chaque fois que le compilateur rencontre pour la première fois la fonction maximum avec un type donné X, il engendre une fonction avec le même nom en remplaçant le type générique T par X.

```
int i,j;
int k = maximum (i,j); // génération de maximum (int,int)

float r,t;
maximum (r,t);          // génération de maximum (float,float)
maximum (k,i);          // maximum (int,int) est déjà générée
```

- L'utilisation des fonctions template doit être sans ambiguïté :

```
float t = max(2.5,r);          // T = float ou double?
```

sera rejeté, puisque lors de la génération de la fonction, le compilateur ne saura pas par quoi remplacer le type générique T. Pour résoudre ce problème, on peut utiliser l'instanciation explicite:

```
float t = max<float>(2.5,r);
```

- Un type générique qui figure parmi les paramètres d'une fonction template doit être le type d'au moins un argument de la fonction

```
template <class T> T fct(); // erreur T doit être le type d'un
                           // argument de fct

template <class T> void fct(T); // exact
```

Exemple 13.1 : (EXP13_01.CPP)

L'exemple suivant montre une utilisation des fonctions template.

```
#include <iostream>
using namespace std;
----- fonction template avec un type générique
template < class T >
T maximum(T u, T v)
{
    return (u > v) ? u : v;
}
----- fonction template avec un type générique et une constante
template <class T, int N>
void fct(T i, int j = N)
{
    cout << i << " - " << j << endl;
}
-----
int main()
{
    int i = maximum(5,6);
    cout << maximum(i,8) << endl;

    float r = maximum(2.5f,4.5f);
    float t = 3.6f;
    cout << maximum(r,t) << endl;

    -----Ambiguité
    //cout << maximum(r,2.1)<< endl; // ambigu : T peut etre float ou double
    //cout << maximum('A',i);           // ambigu : T est int ou char?

    -----instanciation explicite
    cout << maximum<int>('A',i) << endl;
    cout << maximum("Jalal","Ahmed") << endl;

    const int n = 5;
    fct<int,4>(i);
    fct<char,n>('A');
    return 0;
}
```

XIII.3 Les classes template

- Les classes template nous permettent de créer plusieurs classes à partir d'une seule classe : on parle alors de *généricité*. Ces classes sont pratiques pour implémenter des classes conteneurs : collection d'objets (listes, piles,...)
- La déclaration des classes template se fait comme pour les fonctions template :

```
template <class T, int N >
class vect{
    int dim;
    T * v;
public:
    vect(){dim = N; v = new T[dim];}
    ~vect(){ delete [] v;}
};
```

définit une classe template paramétrée par un type générique T et une constante N de type int. Cette classe template nous permet donc d'implémenter des classes de vecteurs (ou tableaux) de n'importe quel type. Ainsi :

```
typedef vect<int,3> VECTINT3;           // définit la classe VECTINT3, des
                                         // vecteurs d'entiers à 3 composantes

vect<float,5> u;                      // génère une classe de vecteurs de réels avec
                                         // 5 composantes et déclare une instance u de
                                         // cette classe
```

- Pour accéder (définition, déclaration,...) aux membres d'une classe template à l'extérieur de la classe, on doit premièrement déclarer les paramètres template (avec le mot clé template) :

```
template <class T>
class A{

public :
    static int i;
    A(T);
    A(A&);                  // ou A( A<T> &)
    void fmembre1( A &);     // ou fmembre1( A<T> &)
    A& fmembre2();          // ou A<T>& fmembre2()
};

// ----- Définitions des membres
template <class T>
int A<T>::i = 1;

// ----- Définition des fonctions membres

// Pour les constructeurs et les méthodes qui ne retourne pas un objet
template <class T>
A<T>::A(T t){...}

template <class T>
A<T>::A(A& a){...}

template <class T>
void A<T>::fmembre1(A& a){...}
```

```
// Si le type de retour est de type classe, il faut mentionner les
// paramètres template de la classe pour la désigner
template <class T>
    A<T>& A<T>::fmembre2(A& a){...}
```

La désignation la classe template doit être sans ambiguïté, dans une zone où A n'est pas reconnue comme une classe, il faut spécifier les paramètres template (à l'intérieur de la classe A est équivalent à A<T>).

Exemple 13.2 : (EXP13_02.CPP)

L'exemple suivant montre comment définir une classe générique Pile, qui permet de créer des piles de n'importe quel type. On testera cette classe avec une pile de chaînes de caractères et une pile de points

```
#include <iostream>
using namespace std;
//----- la classe template 'Pile'
template <class T>
class Pile {
    T *items;
    int top_item;
    int _taille;
public:
    Pile(int = N);
    ~Pile() { delete [] items; }
    void push(T);
    T pop();
    bool EstVide();
    bool EstPleine();
};

//----- Définition des fonctions membres de 'Pile'
template <class T>
Pile<T>::Pile(int taille) : top_item(0), _taille(taille), items(new T[taille])
{};

template <class T>
void Pile<T>::push(T val)
{
    items[top_item++] = val;
}

template <class T>
T Pile<T>::pop()
{
    return items[--top_item];
}

template <class T>
bool Pile<T>::EstVide(void)
{
    return (top_item == 0);
}

template <class T>
bool Pile<T>::EstPleine()
{
    return (top_item >= _taille);
}
```

```

//----- Classe 'point'
class point{
    int x,y;
public:
    point(int abs = 0, int ord = 0): x(abs), y(ord) {}
    point(const int t[2]): x(t[0]), y(t[1]) {}
    friend ostream& operator<< (ostream& , const point& );
};
ostream& operator<<(ostream& out, const point& pt)
{
    out << "(" << pt.x << "," << pt.y << ")";
    return out;
}

//----- Définition des types
typedef Pile<char*> PileChar;
typedef Pile<point> PilePoints;
//----- TEST
int main()
{
    cout << "\n----- Pile de chaines caracteres ----- \n";
    PileChar pc(3);
    pc.push("de caracteres");
    pc.push("pile de chaines ");
    pc.push("Exemple de ");
    while(!(pc.EstVide()))
        cout << pc.pop();
    cout << endl;

    cout << "\n----- Pile de points ----- \n";
    PilePoints pp(2);
    int t[2][2] = {{1,2},{3,4}};
    for(int i = 0; i < 2; i++)
        pp.push(point(t[i]));

    cout << "*** la liste des points\n";
    while(!(pp.EstVide()))
        cout << pp.pop() << endl;
    cout << endl;

    return 0;
}

```

- Les paramètres template dans une classe template peuvent avoir des valeurs par défaut, par exemple :

```

template <class T = int, int N = 3 >
class vect{
    const int dim = N;
    T * v
public:
    vect(){ v = new T[dim]}
    ~vect(){ delete [] v}
};

```

définit une classe `vect`, qui à défaut de paramètres, sera une classe de vecteurs d'entiers à 3 composantes.

- L'instanciation des classes template peut être explicite ou implicite, sauf que l'instanciation implicite ne peut se faire que si les paramètres template ont une valeur par défaut :

```
// exemples d'instanciation explicite

vect<double,5> v;           // génère une classe de vecteurs de réels de 5
                             // composantes

// exemples d'instanciation implicite

vect<> u;                  // génère une classe de vecteurs d'entiers de
                             // 3 composantes

vect<double> v;             // génère une classe de vecteurs de réels de
                             // 3 composantes
```

Exemple 13.3 : (EXP13_03.CPP)

Cet exemple reprend l'exemple précédent (13.2), en spécifiant des valeurs par défaut aux paramètres template, pour qu'à défaut de paramètres la classe Pile génère une pile d'entiers de 5 éléments.

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

-----
template <class T = int, int N = 5>
class Pile {
    // memes membres que dans l'exemple 13.1
};

----- Définition des fonctions membres
template <class T,int N>
Pile<T,N>::Pile(int taille) : top_item(0),
                               _taille(taille),items(new T[taille]) {}
...

----- Définition des types
typedef Pile<> PileInt;          // par défaut pile d'entiers
typedef Pile<char*> PileChar;     // pile de char*
```

```
int main()
{
    cout << "\n----- Pile d'entiers -----\\n";
    PileInt p;                 // pile de 3 entiers
    // ----- remplissage de la pile
    int i = 0;
    while(!(p.EstPleine())){
        p.push(i++);
    }
    while(!(p.EstVide()))
        cout << "Item retire : " << p.pop() << endl;

    cout << "\n----- Pile de chaines caracteres -----\\n";
    PileChar c;                // pile de 3 char*
    c.push("de caracteres");
    c.push("pile de chaines ");
    c.push("Exemple de ");
    while(!(c.EstVide()))
        cout << c.pop();
    cout << endl;
```

```

    return 0;
}

```

XIII.4 Les fonctions membres template

- Les fonctions membres, mis à part les destructeurs et les fonctions virtuelles, d'une classe template ou non peuvent être des fonctions template. (Notez que cette fonctionnalité n'est pas supportée par certains compilateurs)
- Supposons par exemple que A est une classe et que `fct()` est fonction membre template de A, on distingue:
 - Exemple où A est une classe normale :

```

class A {
    ...
    template <class T2> void fct(T2);
};

// definition de la fonction...
template <class T2>
void A::fct(T2 t){ ... }

```

- Exemple où A est une classe template :

```

template < class T1 > class A {
    ...
    template <class T2> void fct(T2);
};

// definition de la fonction...
template <class T1> <class T2>
void A<T1>::fct(T2 t){ ... }

```

- Il faut noter que les fonctions template ne peuvent redéfinir une fonction virtuelle héritée, par exemple :

```

class B
{
    virtual void f(int);
};
class A : public B
{
    template <class T>
        void f(T);           // ne redéfinit pas B::f(int).

    void f(int i)           // redéfinition de B::f(int).
    {
        f<>(i);          // appelle de la fonction template.
        ...
    }
};

```

XIII.5 Spécialisation des template

- Si on veut utiliser la fonction template maximum (exemple 3.1) avec les chaînes de caractères pour obtenir la plus grande chaîne de caractères parmi deux chaînes dans le sens de l'ordre alphabétique, on obtient :

Exemple 13.4 : (EXP13_04.CPP)

```
#include <iostream>
#include <string>
using namespace std;
----- fonction template
template < class T >
T maximum(T u, T v)
{
    return ((u > v) ? u : v);
}
----- test
int main()
{
    cout << maximum<char*>("Jalal", "Ahmed") << endl;
    return 0;
}
```

► Sortie du programme

Ahmed

Le résultat retourné par la fonction maximum est différent de ce qu'on espère. En effet, dans ce cas le compilateur génère la fonction maximum (`char*`, `char*`) et vu le code de la fonction, la comparaison des arguments se fera au niveau des adresses et non pas au niveau des contenus comme on le souhaite.

Pour résoudre ce type de problème, C++ nous offre la possibilité de pouvoir donner une autre définition de la fonction maximum qui sera spécialement dédiée pour le type `char*`

```
...
----- spécialisation de la fonction maximum pour char*
template<>
char* maximum<char*> (char* u, char* v)
{
    return (strcmp(u, v) > 0 ? u : v) ;
}
...
```

- Ainsi, lorsqu'une fonction (ou une classe) template est définie, il est possible de la spécialiser pour un certain jeu de paramètres template, on parle alors de la *spécialisation* de la template.
- Il existe deux types de spécialisations : les *spécialisations totales* et les *spécialisations partielles*.

Spécialisation totale :

- Dans une spécialisation totale, on fournit les valeurs des paramètres template (entre les signes `<` et `>`) juste après le nom de la classe ou de la fonction template et on précède la définition de la template par :

`template <>`

ce qui permet de signaler que la liste des paramètres template est vide pour cette spécialisation, comme par exemple :

```
//----- Spécialisation totale d'une fonction template

template < class T >
    void f(T t, ...) {...}

//-- Spécialisation de f pour le type X
template <>
    void f<X> (X x,...){...}

//-----Spécialisation totale d'une classe template
template < class T1, classe T2, int N >
    class A {...};

// -- Spécialisation de la classe A
template<>
    class A<int,char,10> {...}
```

Spécialisation partielle :

- Dans une spécialisation partielle, on ne fournit les valeurs que d'une partie des paramètres template, en plus les valeurs fournies peuvent être des variantes d'un type générique (un pointeur ou une référence).
- La syntaxe d'une spécialisation partielle suit les mêmes règles qu'une spécialisation totale, à savoir le mot clé *template* suivi d'une liste de paramètres template qui seront utilisés dans la spécialisation, puis le nom de la fonction (ou classe) template suivi des valeurs des paramètres template :

```
//----- Spécialisation partielle d'une fonction template
template < class T , class U>
    void f(T t, U u) {...}
// Exemples de spécialisations de f()
template <class T>
    void f<T,int> (T t, int i){...}

template <class T>
    void f<T,T*> (T t, T* pt){...}

//-----Spécialisation partielle d'une classe template
template < class T1, classe T2, int N >
    class X {...};

// Exemples de spécialisation partielle de la classe X
template <calss T,int N>
    class X<T,T*,N>  {... ...}

// une autre spécialisation partielle
template <calss T>
    class X<char*,T*,10> {...}
```

Spécialisation d'une fonction membre

- Parfois il est utile de ne spécialiser qu'une méthode d'une classe template, dans ce cas on procède comme si la méthode de cette classe était une fonction template, comme dans l'exemple suivant :

```
// définition d'une classe template

template < class T>
class X {
    ...
    void f(T);           // une seule déclaration
    ...
};

//----- définition normale
template <class T>
void X<T>::f(T t) {....}

//----- définition d'une spécialisation de f pour le type char*
template <>
void X<char *>::f(char* c) {....}
```

XIII.6 Les template et amitié

- Une classe template peut posséder des fonctions amies ou des classes amies. L'amitié ainsi déclarée dépend du contexte et des paramètres template déclarés dans la fonction ou la classe. On peut distinguer par exemple les situations suivantes :

- *Situation 1:* une fonction template amie d'une classe normale

```
//----- Déclaration de la classe
class A {
    ...
    template <class T>
        friend void f(A, T);
    ...
};

//----- Définition de la fonction
template <class T>
void f(A a, T t) {...}
```

Toutes les fonctions générées par `f()` seront des fonctions amies de la classe A.

- *Situation 2:* une fonction normale amie d'une classe template

```
//----- Déclaration de la classe
template <class T>
class X {
    ...
    template <class T>
        friend void f(X<T>, ...);
    ...
};

//----- Définition de la fonction
template <class T>
void f(X<T>, ...) {...}
```

`f()` sera une fonction amie de toutes les classes générées par la classe template X

- *Situation 3:* une fonction template amie d'une classe template

```
template <class T>
class A {
    ...
    void f();
}
```

```
template < class T >
class X {
    ...
    friend void A<T>::f(X<T>);
}
```

A<T>::f(X<T>) sera une fonction amie de la classe X<T>

- *Situation 4:* une classe non template amie d'une classe template.

```
template < class T > class X {
    ...
    friend class A;
}
```

A sera une classe amie de toutes les classes générées par la classe template X

- *Situation 5:* une classe template amie d'une classe template

```
template < class T > class X {
    ...
    friend class A<T>;
}
```

A<T> sera une classe amie de la classe X<T>