
C++

Table des matières.

Le Langage C++.

1. Introduction.

Un objet est une donnée informatique regroupant les principales caractéristiques des éléments du monde réel.
Un objet comprend plusieurs notions :

- les attributs : données qui caractérisent l'objet
- les méthodes : ensemble d'actions qui caractérisent le comportement de l'objet
- l'identité : permet de distinguer un objet des autres.

La programmation orientée objet se base sur 3 principes :

1. L'encapsulation : mécanisme consistant à fusionner les données et les méthodes au sein d'une structure en cachant l'implémentation de l'objet.
2. L'héritage : mécanisme permettant de créer une nouvelle classe à partir d'une classe existante qui peut utiliser les données et les méthodes de la classe de base, mais aussi d'ajouter ses propres données et fonctions.
3. Le polymorphisme (qui peut prendre plusieurs formes) : mécanisme permettant d'avoir des fonctions de même nom avec des fonctionnalités similaires dans des classes sans aucun rapport, mécanisme permettant de redéfinir une méthode dans des classes héritant d'une classe de base.

Les avantages de la programmation orientée objet sont une plus grande modularité, un meilleur contrôle des structures, une création de nouveaux types de données avec leurs propres opérateurs, des textes sources facilement réutilisables, …

Le langage C++ est "presque" un sur-ensemble du C, seules existent quelques "incompatibilités". C++ est un langage de haut niveau puissant et polyvalent.
Ses avantages sont l'existence d'un grand nombre de fonctionnalités, les performances du C, la facilité d'utilisation des langages objets, la portabilité des fichiers sources, …

2. Spécificités du C++.

2.1. Les commentaires.

Deux façons de commenter un programme :

- en utilisant /*Commentaire*/
- en utilisant //Commentaire de fin de ligne

2.2. La déclaration des variables et des constantes.

Les déclarations peuvent être déclarées n'importe où pour autant qu'elles soient déclarées avant d'être utilisées.
Les déclarations des constantes peuvent se faire également suivant la syntaxe suivante : const type identificateur=valeur.

2.3. Les fonctions.

2.3.1. Déclaration des fonctions.

La déclaration des prototypes de fonctions est obligatoire.
C++ permet d'avoir des appels aux fonctions avec un nombre variable de paramètres en donnant des valeurs par défaut aux paramètres.
Syntaxe :
Type nom_fonction (type variable1 [=valeur1], type variable2 [=valeur2],…)
Seuls les derniers paramètres d'une fonction peuvent avoir des valeurs par défaut.
La liste des paramètres est la signature de la fonction.

2.3.2. Appel des fonctions.

Les fonctions peuvent être appelées avec un nombre de paramètres inférieur à celui de la déclaration et de la définition de la fonction, si les paramètres contiennent des valeurs par défaut lors de la déclaration.
Les valeurs par défaut sont prises automatiquement quand les paramètres ne sont pas fournis lors de l'appel.
Si un paramètre manque lors de l'appel de la fonction, tous les paramètres qui suivent doivent aussi manquer.
Seuls les derniers paramètres d'une fonction peuvent avoir des valeurs par défaut.

Exemple :

```
#include <stdio.h>

int somme3 (int a=0, int b=0, int c=0);

int main ()
{
 printf ("Somme de 1+2+3=%d\n", somme3 (1, 2,3));
 printf ("Somme de 1+2=%d\n", somme3 (1,2));
 printf ("Somme de 1=%d\n", somme3 (1));
 printf ("Somme par d,faut=%d\n", somme3 ());
 return 0;
}
int somme3 (int a, int b, int c)
{
 return a+b+c;
}
```

2.3.3. Surcharge des fonctions.

En C++, il est possible, au sein d'un même programme, que plusieurs fonctions possèdent le même nom. Le compilateur peut différencier 2 fonctions en regardant le type des paramètres (leurs signatures) qu'elle reçoit. Deux fonctions peuvent porter le même nom si elles peuvent être différenciées par leur signature.
Le compilateur dispose de règles pour déterminer la meilleure fonction étant donné le jeu de paramètres.

L'algorithme de sélection des fonctions surchargées est le suivant :

1. Utiliser la réplique exacte si elle est trouvée. Remarques : short et char sont considérés comme des répliques exactes d'un entier (int), float est considéré comme une réplique exacte d'un double.
2. Essayer des types de conversion standard et utiliser l'une ou l'autre réplique. Remarque : les conversions ne doivent pas perdre d'information.
3. Essayer les conversions définies par le programmeur.

Exemple :

```
#include <stdio.h>

float test (int i, int j);
float test (float i, float j);

int main ()
{
 printf ("test (2,3)=%f\n", test (2,3));
 printf ("test (2.5, 3)=%f\n", test (2.5, 3));
 printf ("test (2.5, 3.5)=%f\n", test (2.5F, 3.5F));
 printf ("test (2.5F, 3)=%f\n", test (2.5F, 3));
 return 0;
}

float test (int i, int j)
{
 printf ("test (int, int)\n");
 return (float) (i+j);
}

float test (float i, float j)
{
 printf ("test (float, float)\n");
 return i*j;
}
```

2.3.4. Les fonctions en ligne (inline).

Pour rendre « en ligne » une fonction membre, on peut :

- soit fournir directement la définition de la fonction dans la déclaration même de la classe, dans ce cas le qualificatif inline n'a pas à être utilisé
- soit procéder comme pour une fonction ordinaire en fournissant une définition en dehors de la déclaration de la classe, dans ce cas le qualificatif inline doit apparaître, à la fois devant la déclaration et devant l'en-tête.

Par l'introduction du mot-clé inline devant le nom de la fonction, le compilateur insère directement le code de la fonction dans le flux d'instructions du programme. Cette insertion diminue le temps nécessaire aux appels de fonctions, aux passages de paramètres et à la valeur de retour. Mais le code devient plus important car la fonction est réécrite à chaque fois qu'elle est appelée.

2.3.5. La transmission par référence.

En faisant précéder du symbole & le nom d'un argument dans la déclaration et la définition de la fonction, on réalise une transmission par référence. Cela signifie que les éventuelles modifications effectuées au sein de la fonction porteront sur l'argument effectif de l'appel et non plus sur une copie.

2.3.6 Autoréférence : le mot clé this.

Au sein d'une fonction membre, **This** représente un pointeur sur l'objet ayant appelé la dite fonction membre.

2.3.7. Fonctions membre constantes.

On peut déclarer des objets constants (**const**). Dans ce cas, seules les fonctions membre déclarées (et définies) avec ce même qualificatif peuvent recevoir (implicitement ou explicitement) en argument un objet constant.

2.4. Gestion dynamique de la mémoire.

Les fonctions malloc, free sont remplacées par les opérateurs new et delete.
Si type représente la description d'un type quelconque et n une expression de type entier, **new type[n]** alloue l'emplacement pour n éléments du type indiqué et fournit en résultat un pointeur de type type*. On obtient un pointeur nul si l'allocation a échoué. **delete adresse** libère un emplacement préalablement alloué par new à l'adresse indiquée.

2.5. Les entrées et les sorties.

Trois objets particuliers existent en C++ : cin (entrée standard, clavier), cout (sortie standard, écran et cerr (erreurs de sortie standard).
Pour utiliser ces objets, il faut inclure l'en-tête <iostream.h> ou <stdiostr.h>.
Syntaxe :
cout<<expression1<<expression2<< … <<expressionn
Affiche sur le flot cout (connecté à la sortie standard stdout) les valeurs des différentes expressions indiquées.
cin>>expression1>>expression2>> … >>expressionn
Lit sur le flot cin (connecté par défaut à l'entrée standard stdin) des informations de l'un des types char, int, long, float, double ou char *.

Exemple:

```
#include <iostream.h>
int main ()
{
 int i; float f;
 cout<<"Entrer un entier et un float"<<endl;
 cin>>i>>f;
 cout<<"i= "<<i+1<<"et f= "<<f<<"\n";
 return 1;
}
```

3. Les classes.

Une classe est une généralisation de la notion de type défini par l'utilisateur, dans lequel se trouvent associées à la fois des données (données membres) et des fonctions (fonctions membres ou méthodes). Une classe est donc une extension de la notion de structure qui ne comprend en C que des données.

3.1. Déclaration, définition et utilisation d'une classe.

La déclaration d'une classe précise quels sont les membres (données ou fonctions) publics (c'est-à-dire accessibles à l'utilisateur de la classe) et quels sont les membres privés (inaccessibles à l'utilisateur de la classe).
La visibilité des membres et des données d'une classe est établie par les propriétés suivantes :

- public : données et méthodes de la classe sont accessibles par tout utilisateur de la classe
- private : données et méthodes de la classe sont inaccessibles à l'utilisateur de la classe
- protected : identique à private avec des nuances en ce qui concerne l'héritage (voir plus loin, notion d'héritage).

Par défaut, une classe a ses données et ses méthodes de type private.

Exemple:

```
#include <stdiostr.h>
#include <string.h>

class personne
{
 char *nom;
 char *prenom;
 int age;
 public:
    void init (char *n, char *p, int a)
    {
         strcpy (nom, n);
         strcpy (prenom, p);
         age = a;
     }
    void affiche ()
    {
         cout<<nom<<" "<<prenom<<" "<<age<<endl;
    }
};

void main ()
{
 personne p;
 p.init ("Durant","Louis",20);
 p.affiche ();
}
```

La définition d'une classe consiste à fournir les définitions des fonctions membres. On indique le nom de la classe correspondante à l'aide de l'opérateur de résolution de portée (::). Au sein de la classe même, les membres sont directement accessibles sans qu'il soit nécessaire de préciser le nom de la classe.

Exemple:

```
#include <stdiostr.h>
#include <string.h>

class personne
{
 char *nom;
 char *prenom;
 int age;
 public:
    void init (char *n, char *p, int a);
    void affiche ();
};
void personne:: init(char *n, char *p, int a);
    {
        strcpy (nom, n);
        strcpy (prenom, p);
        age = a;
     }
void personne:: affiche()
    {
        cout<<nom<<" "<<prenom<<" "<<age<<endl;
    }

void main()
{
 personne p;
 p.init("Durant","Louis",20);
 p.affiche();
}
```

Pour utiliser une classe, on déclare un objet d'un type classe donné en faisant précéder son nom du nom de la classe. On accède à n'importe quel membre public d'une classe en utilisant l'opérateur. (Point).
En pratique, à l'utilisateur d'une classe, on fournit :

- un fichier en-tête contenant la déclaration de la classe : l'utilisateur utilisant la classe devra l'inclure dans tout programme faisant appel à la classe en question.
- Un module objet résultant de la compilation du fichier source contenant la définition de la classe, c'est-à-dire la définition de ses fonctions membre.

3.2. Constructeurs et destructeur.

3.2.1. Définition et rôles.

Pour simplifier l'initialisation des données membres d'une classe, le C++ offre une fonction spéciale, dite **constructeur**, que le compilateur appelle à chaque construction d'objet. Cette fonction a son pendant dans une autre fonction, dite **destructeur**, que le compilateur exécute chaque fois qu'un objet est abandonné.
L'appel au constructeur a lieu après l'allocation de l'emplacement mémoire destiné à l'objet tandis que l'appel au destructeur est appelé avant la libération de l'espace mémoire associé à l'objet.
Le rôle du constructeur est de réserver de la mémoire pour les données membres, d'initialiser les données membres, …
Le rôle du destructeur est de libérer de la mémoire pour les membres et les méthodes lorsqu'un objet est détruit.
Le constructeur a les caractéristiques suivantes :

- il ne spécifie aucun type de retour et n'est pas défini comme étant de type void
- son identificateur est le nom de la classe
- il peut être surchargé.

Sa déclaration : nom (type1 patamètre1, type2 paramètre2, …) ;

Le destructeur a les caractéristiques suivantes :

- il ne spécifie aucun type de retour
- son identificateur est le nom de la classe précédé de ~
- il est toujours sans paramètre
- il est unique.

Sa déclaration : ~nom ();

Exemple:

```
#include <stdiostr.h>
#include <string.h>

class personne
{
 char *nom;
 char *prenom;
 int age;
 public :
    personne (char *n, char *p, int a);
    void affiche ();
};

personne :: personne(char *n, char *p, int a);
{
 cout<<"Appel du constructeur\n";
 strcpy (nom, n);
 strcpy (prenom, p);
 age = a;
}

personne::~personne()
{
 cout<<"Appel du destructeur\n";
}

void personne:: affiche()
{
 cout<<nom<<" "<<prenom<<" "<<age<<endl;
}

void main()
{
 personne p("Durant","Louis",20);
 p.affiche();
 personne p1("Durant","Marcel",22à;
 p1.affiche();
}
```

3.2.2. Constructeurs par défaut.

Un constructeur par défaut est un constructeur de la classe ayant une liste d'arguments vide et ayant des valeurs par défaut dans tous ses arguments.
Si l'utilisateur ne spécifie pas de constructeur, le compilateur génère un constructeur par défaut. Il doit être utilisé quand une classe a des données membres qui sont du type d'autres classes.

3.2.3. Surcharge des constructeurs.

En C++, il est possible de surcharger des fonctions, comme les constructeurs sont des fonctions, ils supportent aussi la surcharge.
Il est donc possible de définir autant de constructeur que nécessaire. Pour les distinguer, chacun d'entre eux doit disposer d'une liste unique de paramètres (nombre et/ou type différent) c'est-à-dire une signature différente.
Pour rappel, la signature d'une fonction est formée dans l'ordre, par :

- le nom de la classe s'il s'agit d'une fonction membre ou d'un constructeur
- l'identificateur de la fonction
- la liste des types des paramètres, dans l'ordre dans lequel ils sont définis.

Exemple : voir Labo 2

3.2.4. Liste d'initialisation.

Un constructeur peut comme toute fonction accepter des arguments qui lui sont fournis lors de la création d'un objet. L'initialisation d'un objet, via un constructeur se fait en deux temps : la liste d'initialisation est d'abord utilisée pour initialiser chacun des champs, puis le corps de la fonction constructeur est exécutée.
La liste d'initialisation permet donc d'initialiser les données membres.
Syntaxiquement, cette liste commence par le caractère : et contient le nom des champs de l'objet, chacun suivi de parenthèses dans laquelle se trouve la valeur initiale de ces champs. Un champ qui n'est pas initialisé explicitement ne figure pas dans la liste d'initialisation.
Il est préférable, dans la liste d'initialisation, de respecter l'ordre des déclarations des données membres pour les initialiser.

3.2.5. Paramètres par défaut d'un constructeur.

Il est possible d'attribuer des valeurs par défaut aux paramètres d'un constructeur. Si l'utilisateur ne précise pas la valeur de chaque paramètre, le constructeur utilise ces valeurs par défaut.

3.2.6. Les constructeurs de copie.

Le constructeur par copie est une forme spéciale de constructeur invoquée dans les situations suivantes (à chaque copie d'objet) :

- initialisation d'un objet d'une classe à partir d'un objet de la même classe
- passage par valeur d'un objet comme argument d'une méthode qui attend un objet
- retour d'un objet.

Il est unique dans une classe.
La signature d'un constructeur par copie désigne soit un membre de la classe reçu par référence, soit la version constante (seule l'une des 2 versions est nécessaire). Sa syntaxe est la suivante : type (type &) ou type (const type &) où type désigne le type de l'objet.
Par défaut, toute classe comporte 2 constructeurs (que nous les invoquions ou pas) :

- le constructeur par défaut
- le constructeur de copie par défaut (ce dernier recopie les valeurs des différents membres donnés de l'objet, comme le fait l'affectation).

Quand on appelle le constructeur par copie, il recopie l'état complet d'un objet donné dans un nouvel objet de la même classe.

Exemple de surcharge de constructeur et constructeur par copie :

```
#include <string.h>
#include <stdiostr.h>

class personne
{
    char *nom;
    char *prenom;
    int age;
    public:
        personne ();
        personne (char *n, char *p, int a);
        personne (personne& p); //Constructeur par copie.
        void affiche ();
        personne fonction (personne p);
        ~personne ();
};

personne::personne():nom("Dupont"),prenom("Charles"),age(19)
{
 cout<<"Constructeur par d,faut\n";
}
personne::personne(char *n,char *p,int a):nom(n),prenom(p),age(a)
{
 cout<<"Constructeur d'initialisation\n";
}
personne::personne(personne &p):nom(p.nom),prenom(p.prenom),age(p.age)
{
 cout<<"Constructeur par copie\n";
}
```

```
void personne::affiche()
{
 cout<<"Nom:"<<nom<<endl;
 cout<<"Prénom:"<<prenom<<endl;
 cout<<"Age:"<<age<<endl;
}
personne personne::fonction(personne p)
{
 strcpy(p.nom,"nomfn");
 strcpy(p.prenom,"prenomfn");
 p.age=20;
 return p;
}
personne::~personne()
{
 cout<<"destructeur\n";
}

void main()
{
 personne p;
 p.affiche();
 personne p1("Durant","Georges",20);
 p1.affiche();
 personne p2=p1;
 p2.affiche();
 personne p3;
 p3=p3.fonction(p);
 p3.affiche();
}
```

Construction, destruction et initialisation d'objets.

a) Les objets automatiques et statiques

Les objets **automatiques** sont créés par une déclaration soit dans une fonction, soit au sein d'un bloc. Ils sont créés au moment de l'exécution de la déclaration. Ils sont détruits lorsque l'on sort de la fonction ou du bloc.
Les objets **statiques** sont créés par une déclaration située en dehors de toute fonction ou par une déclaration précédée du mot **static**. Ils sont créés avant l'entrée dans la fonction main et détruits après la fin de son exécution.

b) Les objets temporaires

L'appel explicite, au sein d'une expression, du constructeur d'un objet provoque la création d'un objet temporaire qui pourra être automatiquement détruit dès lors qu'il ne sera plus utile.

c)Les objets dynamiques

Ils sont créés par l'opérateur new, auquel il faut fournir les valeurs des arguments destinés à un constructeur.
L'accès au membre d'un objet dynamique est réalisé comme pour les variables ordinaires, à l'aide de l'opérateur ->.
Les objets dynamiques sont détruits à la demande en utilisant l'opérateur delete.

d)Initialisation d'objets

Une initialisation d'objet a lieu dans les 3 situations suivantes :

- déclaration d'un objet avec initialiseur
- transmission d'un objet par valeur en argument d'appel d'une fonction
- transmission d'un objet par valeur en valeur de retour d'une fonction.

L'initialisation d'un objet provoque toujours l'appel d'un constructeur de copie.

e)Les tableaux d'objets

Syntaxe : type identificateur[nombre]

Permet de créer un tableau identificateur de nombre objets de type type en appelant le constructeur pour chacun d'entre eux (constructeur sans argument).
Exemple : point ptab[20] //crée un tableau de 20 objets de type point

Syntaxe : type * identificateur=new type [nombre]

Alloue l'emplacement mémoire nécessaire à nombre objets (consécutifs) de type type, en appelant le constructeur pour chacun d'entre eux, puis place l'adresse du premier dans identificateur.
La destruction d'un tableau d'objets se fait en utilisant l'opérateur delete

3.3. L'opérateur de résolution de portée (::).

L'opérateur de résolution de portée peut être utilisé dans différentes situations :

1. Lorsqu'on déclare le prototype de la fonction dans la classe et qu'on la définit à l'extérieur de la classe
2. Lorsqu'on spécifie un bloc d'instructions qui n'appartient à aucune classe (par exemple : une fonction globale) de même signature qu'une fonction de la classe. Pour accéder à la fonction globale, il faut utiliser l'opérateur de résolution de portée.
3. Lorsqu'on veut accéder à une variable globale qui a le même nom qu'une donnée membre de la classe. Pour accéder à la variable globale, il faut utiliser l'opérateur de résolution de portée.
4. Lorsqu'on veut éviter les conflits entre les noms des paramètres et ceux des données membres de la classe, il faut faire précéder le nom des données membres par le nom de la classe suivi de l'opérateur de résolution de portée.

4. Les données et fonctions membres statiques.

Elles permettent de signifier au compilateur que la donnée membre ou la fonction membre est seule et unique dans la classe et est partagée par tous les objets créés grâce à cette classe.

4.1. Les données membres statiques.

Une donnée membre statique est une donnée qui est partagée par toutes les instances de la classe. La donnée doit être déclarée **static** et être définie en dehors de la classe. Son initialisation est 0, par défaut.
Un membre statique est toujours accédé en qualifiant son nom par le nom de sa classe.

4.2. Les fonctions membres statiques.

Une fonction membre statique d'une classe n'est pas appliquée à un objet de la classe, elle peut être déclarée avec l'attribut **static**. Dans ce cas, une telle fonction peut être appelée, sans mentionner d'objet particulier, en préfixant simplement son nom du nom de la classe concernée, suivie de l'opérateur de résolution de portée.

Exemple données et fonctions statiques:

```
#include <stdiostr.h>
class test
{
     int i;
 public :
     test(int ii);
     ~test();
   static int j;
   static int get_value();
     void affiche();
};

int test::j=0;
test::test(int ii)
{
 i=ii;
 ++j;
}
test::~test()
{
 --j;
}
int test::get_value()
{
 return j;
}
void test::affiche()
{
 cout<<"i="<<i<<endl;
 cout<<"nombre objets créés="<<j<<endl;
}
```

```
void main()
{
 cout<<test::j<<endl;
 test t1(5);
 t1.affiche();
 test t2(2);
 t2.affiche();
 t1.affiche();
 test t3(1);
 cout<<test::get_value()<<endl;
 cout<<t1.get_value()<<endl;
}
```

5. La surcharge des opérateurs.

5.1. Définition, intérêt.

Les opérateurs ne se différencient des fonctions que syntaxiquement, pas logiquement. Le compilateur traite l'appel à un opérateur comme un appel à une fonction.
Le C++ dispose d'une syntaxe qui permet de définir les opérateurs pour les classes comme des fonctions classiques.
Comme on peut surcharger des fonctions ayant des signatures différentes, et comme les opérateurs sont des fonctions, il est possible de les surcharger.

La surcharge des opérateurs permet de donner une nouvelle signification lorsqu'ils portent en partie ou en totalité sur des objets de type classe.

Pour sur définir un opérateur existant op, on définit une fonction nommée operator op (on peut placer un ou plusieurs espaces entre le mot operator et l'opérateur, mais ce n'est pas une obligation).

5.2. Deux façons de surcharger des opérateurs.

Le C++ permet de surcharger les opérateurs :
- soit par des fonctions membres (en interne)
- soit par des fonctions indépendantes généralement amies d'un ou plusieurs classes (en externe).

Règles :
- définir en priorité les opérateurs comme membres pour bénéficier de l'encapsulation assurée par la classe
- définir avec une fonction externe lorsque l'argument de gauche est de type élémentaire.

5.2.1. Surcharge des opérateurs en interne.

Les opérateurs sont considérés comme des méthodes normales de la classe sur laquelle ils s'appliquent.
L'opérande de gauche est l'objet de la classe dont la fonction est membre (pointeur *This).

Syntaxe : type operator op (paramètres)
Où

- type est le type de résultat retourné par l'opération
- paramètres = opérandes
- première opérande est l'objet auquel la fonction s'applique
- deuxième opérande = premier paramètre de la fonction opérateur

L'écriture A op B se traduisant par : A.operator op (B) pour un opérateur binaire
L'écriture op A se traduisant par A.operator () pour un opérateur unaire

5.2.2. Surcharge des opérateurs en externe.

La surcharge de l'opérateur ne se fait plus dans la classe qui l'utilise, mais en dehors de celle-ci, par surcharge d'un opérateur prédéfini. L'opérateur est redéfini sous forme d'une fonction indépendante, généralement une fonction amie.

Fonction amie permet de déclarer dans une classe, les fonctions que l'on autorise à accéder aux membres privés d'une classe (données ou fonctions). Les fonctions amies sont précédées du mot clé FRIEND et elles sont externes à la classe.

Syntaxe : type operator op (opérandes)
Où

- type est le type du résultat de l'opération (type de retour de la fonction operator)
- opérandes = listes complètes des opérandes

L'écriture A op B se traduisant par operator op (A, B) pour un opérateur binaire
L'écriture op A se traduisant par operator op (a) pour un opérateur unaire.

5.3. Limites et possibilités de surcharge des opérateurs.

1. Seuls les opérateurs prédéfinis peuvent être surchargés.
2. La pluralité (unaire, binaire) des opérateurs ne peut être changée.
3. La priorité et l'associativité d'un opérateur ne changent pas.
4. Les opérateurs sizeof, ::, ., ?: ne peuvent être surchargés
5. Les opérateurs new, delete, =, (), [],-> ne peuvent être surchargés que par des fonctions membres en non par des fonctions globales.
6. Un opérateur sur défini doit toujours posséder une opérande de type classe.

5.4. Surcharge des opérateurs arithmétiques.

5.4.1. Surcharge des opérateurs arithmétiques standard.

Les opérateurs arithmétiques sont +, -, *, /, %, &, |, ^, >>, <<.
Le C++ permet de définir des fonctions qui utilisent les symboles des opérateurs arithmétiques standard directement sur des objets.
La surcharge des opérateurs arithmétiques standard doit se faire par des fonctions amies pour pouvoir accepter des opérations du style constante opérateur objet.

5.4.2. Surcharge des opérateurs d'affectation.

Cette surcharge existe par défaut mais n'est pas suffisante pour les classes qui ont un membre alloué dynamiquement.
Cette surcharge doit se faire par une fonction membre (l'opérateur de gauche est un objet et il est modifié) et doit retourner une référence pour permettre des affectations en cascade.
Syntaxe de l'opérateur surchargé pour une classe T :

```
T& T :: operator =(const T& t1){
        //affecter chaque donnée membre de t à la donnée membre correspondante de
        //l'objet propriétaire (*This)
        return *This ;
}
```

5.4.3. Surcharge des opérateurs arithmétiques d'affectation.

Cette surcharge doit être faite par une fonction membre (l'opérateur de gauche est un objet et il est modifié).

5.4.4. Surcharge des opérateurs relationnels

Les opérateurs <, >,<=,>=,== et != peuvent être surchargés de la même façon que les opérateurs arithmétiques c'est-à-dire par des fonctions amies pour pouvoir accepter des comparaisons telles que constante opérateur objet.
Les opérateurs relationnels retournent un type entier, soit 1 (vrai) soit 0 (faux).

5.4.5. Surcharge des opérateurs de flux

Les opérateurs >> (insertion de flux) et << (extraction de flux) peuvent être surchargés pour être personnalisés, ils doivent être surchargés par des fonctions amies car l'opérande de gauche doit être de type flux.
Syntaxe pour l'opérateur de sortie (<<) pour une classe T de donnée membre d :

```
friend ostream& operator << (ostream& o, T& t1) {
return o<<t1.d;
}
```

Syntaxe pour l'opérateur d'entrée (>>) pour une classe T de donnée membre d :

```
friend istream& operator << (istream& i, T& t1) {
return i>>t1.d;
}
```

5.4.6. Surcharge des opérateurs d'incrémentation et de décrémentation

La surcharge doit se faire par une fonction membre car il y a une modification de l'objet. Ce sont des opérateurs unaires : le seul argument est l'objet incrémenté.
La pré incrémentation (pré décrémentation) est appliquée à des types entiers, ajoute (retire) 1 à la valeur de l'objet et retourne la valeur.
Syntaxe de la surcharge pour une classe T :

```
T operator ++() ;
```

T operator – () ;

Comme pour la post-incrémentation (post-décrémentation), les opérateurs portent le même nom, le c++ les distingue de la pré incrémentation (pré décrémentation) en ajoutant un paramètre fictif de type entier.
Syntaxe de la surcharge pour une classe T :
T operator ++ (int) ;
T operator – (int) ;

5.4.7. Surcharge de l'opérateur d'indexation

L'opérateur d'indexation [] possède 2 opérandes : l'objet courant et l'indice de l'élément, soit a[i] ⇔ a.operator [] (i) où a est la première opérande (*This) et i la deuxième opérande (indice de type entier).
La surcharge de l'opérateur d'indexation doit se faire par une fonction membre car l'opérande de gauche est un objet. Elle doit pouvoir apparaître à gauche ou à droite d'une assignation et donc renvoyer une référence sur une valeur et non pas la valeur elle-même.
Syntaxe de la surcharge pour une classe T :
T& operator [] (int i);

5.4.8. Surcharge des opérateurs de conversion

Pour permettre à des objets de base de se comporter comme les objets des types fondamentaux c'est-à-dire des variables ordinaires, il faut utiliser une opérateur de conversion :
Syntaxe : operator type() ; où type est le type vers lequel l'objet doit être converti.

Exemple 1 : Surcharge d'opérateurs :

```
#include <stdiostr.h>
#include <conio.h>
class complexe
{
    int reel, imag;
    public:
    complexe (int r=0, int i=0);
    void affiche ();
    //complexe operator +(complexe c);
    friend complexe operator + (complexe c1, complexe c2);
    complexe operator= (complexe c);
    complexe operator += (complexe c);
    friend int operator == (complexe c1, complexe c2);
    friend istream& operator>>(istream& i, complexe& c);
    friend ostream& operator<<(ostream& o, complexe& c);
    int& operator [] (int i);
    complexe operator++ (); //préfixé
    complexe operator++ (int); //postfixé
    operator int ();
};

complexe::complexe(int r,int i)
{
 reel=r;
 imag=i;
}
void complexe::affiche()
{
 cout<<reel<<" + i "<<imag<<endl;
}
/*complexe complexe::operator +(complexe c)
{
 complexe res;
 res.reel=reel+c.reel;
 res.imag=imag+c.imag;
 return res;
}*/
/*complexe complexe::operator +(complexe c)
{
 reel=reel+c.reel;
 imag=imag+c.imag;
 return *This;
}*/
```

```
complexe operator +(complexe c1,complexe c2)
{
 complexe res;
 res.reel=c1.reel+c2.reel;
 res.imag=c1.imag+c2.imag;
 return res;
}
complexe complexe::operator=(complexe c)
{
 if(This==&c) return *This;
 reel=c.reel;
 imag=c.imag;
 return *This;
}
complexe complexe::operator +=(complexe c)
{
 reel=reel+c.reel;
 imag=imag+c.imag;
 return *This;
}
int operator==(complexe c1, complexe c2)
{
 if(c1.reel==c2.reel && c1.imag==c2.imag) return 1;
 else return 0;
}
istream& operator>>(istream& i, complexe& c)
{
 cout<<"Entrez la partie r,elle\n";
 i>>c.reel;
 cout<<"Entrez la partie imaginaire\n";
 i>>c.imag;
 return i;
}
ostream& operator <<(ostream& o, complexe& c)
{
 o<<c.reel<<" + i "<<c.imag<<endl;
 return o;
}
int& complexe::operator[](int i)
{
 if(i==1) return reel;
 if(i==2) return imag;
}
complexe complexe::operator++()
{
 ++reel;
 ++imag;
 return *This;
}
```

```
complexe complexe::operator ++(int)
{
 complexe tmp=*This;
 reel++;
 imag++;
 return tmp;
}
complexe::operator int()
{
 return reel;
}

void main()
{
 int re;
 clrscr();
 complexe c1(1,2),c2(3,4),res;
 complexe c3;
 cin>>c3;
 cout<<c3;
 cout<<++c3;
 cout<<c3++;
 cout<<c3;
 re=(int)c3;
 cout<<"re="<<re<<endl;
 if(c1==c2) cout<<"Les complexes sont identiques\n";
 else cout<<"Les complexes sont différents\n";
 res=c1+c2;
 cout<<"R,el:"<<c1[1]<<endl;
 cout<<"Imag:"<<c1[2]<<endl;
 c1[1]=5;
 c1[2]=6;
 cout<<c1;
 cout<< "somme de ";
 c1.affiche();
 cout<<" et de ";
 c2.affiche();
 cout<<" vaut ";
 res.affiche();
 res+=c1;
 res.affiche();
 //res=15+c1;
 res.affiche();
 res=c1=c2;
 res.affiche();
}
```

6. L'héritage.

6.1. Définition

L'héritage permet de dériver de nouvelles classes à partir des classes de base. Il permet d'exprimer la spécificité.

L'héritage permet à une classe dérivée de reprendre les caractéristiques d'une classe de base préexistante et d'ajouter des caractéristiques propres à elle.

Si une relation d'héritage existe entre une classe de base A et une classe dérivée B, alors B se compose de 2 parties :

- la partie héritée ou dérivée
- la partie incrémentale : ensemble de champs et méthodes définis spécialement pour la classe B.

6.2. Héritage simple

6.2.1. Syntaxe de déclaration

La syntaxe pour dériver une classe B d'une classe A est :

```
Class B :public A                //ou private A  ou protected A
{
        //définitions des membres supplémentaires pour B (données ou fonctions)

        //redéfinition des membres existants dans A (données ou fonctions)

};
```

Exemple :

```
class base
{
   void pri( );
  protected :
   void pro( );
  public:
   void publ( );
};

class derivee: public base
{
  public:
   void traitement( );
};

void base::pri( )
{
  cout<<"pri( )"<<endl;
}
void base::pro( )
{
  cout<<"pro( )"<<endl;
}
void base::publ( )
{
  cout<<"publ( )"<<endl;
}

void derivee::traitement( )
{
  pri( );        //Pas bon car pri( )n'est accessible uniquement que par la
classe base.
  pro( );        //pro( ) est accessible par la classe derivee.
  publ( );
}

void main( )
{
  derivee d;
  d.traitement( );
  base b;
  b.pri( );      //inaccessible car private
  b.pro( );      /*Les données protégées ne sont accessible que par la classe
                 elle-même pas par ses objets*/
  b.publ( );
}
```

6.2.2. Appel des constructeurs et des destructeurs

Si B est une classe dérivée de la classe de base A.
Dès que B possède au moins un constructeur, la création d'un objet de type B entraîne l'appel du constructeur de A puis de B
Le destructeur de la classe B appelle le destructeur de la classe B puis celui de la classe A.

Exemple :

```
#include <iostream.h>
#include <string.h>
class personne
{
    char nom[20];
    char prenom[20];
public:
    personne(char *n,char *p);
    void affiche();
    ~personne();
};

class etudiant:public personne
{
    int annee;
public:
     etudiant(int a,char *n,char* p);
     void affiche();
     ~etudiant();
};

personne::personne(char *n,char *p)
{
strcpy(nom,n);
strcpy(prenom,p);
cout<<"Construsteur de personne\n";
}
void personne::affiche()
{
cout<<"Nom:"<<nom<<endl;
cout<<"Pr,nom:"<<prenom<<endl;
}
personne::~personne()
{
cout<<"Destructeur de personne\n";
}
```

```
etudiant::etudiant(int a,char *n,char *p):personne(n,p)
{
annee=a;
cout<<"Constructeur de etudiant\n";
}
etudiant::~etudiant()
{
cout<<"Destructeur etudiant\n";
}
void etudiant::affiche()
{
personne::affiche();
cout<<"Annee:"<<annee<<endl;
}

void main()
{
etudiant e(2,"Durant","Marcel");
e.affiche();
}
```

Apres compilation ce programme donne :

```
Construsteur de personne
Constructeur de etudiant
Nom:Durant
Prénom:Marcel
Annee:2
```

6.2.3. Définition des constructeurs dérivés

Si B est une classe dérivée de la classe de base A.
Si le constructeur de la classe de base possède des paramètres, il faut les mettre au niveau de la classe dérivée lorsqu'on définit le constructeur dérivé.

Exemple :

```
class personne
{
   char nom[20];
   char prenom[20];
  public :
   personne (char *n, char *p);
   void affiche( );
   ~personne( );
};

class etudiant:public personne
{
   int annee;
  public :
   etudiant (int a, char *n, char *p);
   void affiche_etudiant( );              //void affiche( );
   ~etudiant( );
};

personne::personne(char *n, char *p)
{
  strcpy (nom,n);
  strcpy (prenom,p);
  cout<<"constructeur de personne\n";
}
void personne::affiche( )
{
  cout<<"Nom : "<<nom<<endl;
  cout<<"Prenom : "<<prenom<<endl;
}
personne::~personne( )
{
  cout<<"Destructeur de personne\n";
}

etudiant::etudiant(int a,char *n,char *p):personne(n,p)
{
  annee=a;
  cout<<"Constructeur de etudiant\n";
}
etudiant::~etudiant( )
{
  cout<<"Destructeur de etudiant\n";
}
void etudiant::affiche_etudiant( )          //void etudiant::affiche( )
{                                           //{
  affiche( );                               //personne::affiche( );
  cout<<"Annee : "<<annee<<endl;            //cout<<"Annee : "<<annee<<endl;
}                                           //}
void main ()
{
  etudiant e (2,"Durant","Marcel");
  e.affiche_etudiant( );                    //e.affiche( );
}
```

6.2.4. Membres protégés (protected)

Pour rendre accessible une donnée membre de la classe de base par les méthodes de la classe dérivée, il faut lui donner le type d'accès protected.
Accès protected est une sorte d'accès privé étendu aux membres de la famille c'est-à-dire aux classes dérivées.
Un membre protégé se comporte comme un membre privé pour un utilisateur quelconque de la classe ou de la classe dérivée mais comme un membre public pour la classe dérivée.
Il est accessible à l'intérieur de la classe elle-même, dans ses classes amies, dans ses classes dérivées et dans les classes amies des classes dérivées.

Tableau indiquant suivant le modifieur comment les données sont accessibles :

Modifieur	Classe de base	Classe dérivée
Public	Public	Public
	Private	Pas accessible
	Protected	Protected
Protected	Public	Protected
	Private	Pas accessible
	Protected	Protected
Private	Public	Private
	Private	Pas accessible
	Protected	Private

Exercice sur les héritages réalisé par V. Ricq.

```
class personnel
{
        char *nom;
        char *position;
        int salaire;
    public :
        personnel (char *n, char *p, int s);
        void affiche_personnel ( );
};

class patron:public personnel
{
        char *marque;
        int bonus_salaire;
   public:
        patron (int bs, char *m, char *n, char *p, int s);
        void affiche_patron ( );
};

personnel::personnel (char *n, char *p, int s)
{
        strcpy (nom, n);
        strcpy (position, p);
        salaire=s;
}
void personnel::affiche_personnel ( )
{
        cout<<"Nom :"<<nom<<endl;
        cout<<"Position : "<<position<<endl;
        cout<<"Salaire : "<<salaire<<endl;
}

patron::patron (int bs, char *m, char*n, char *p, int s):personnel(n,p,s)
{
        bonus_salaire=bs;
        strcpy(marque, m);
}
void patron::affiche_patron ( )
{
        affiche_personnel ( );
        cout<<"Bonus Salaire : "<<bonus_salaire<<endl;
        cout<<"Marque : "<<marque<<endl;
}

void main ( )
{
patron pat (500,"OPEL", "Durant", "Employe", 1000);
pat.affiche_patron ( );
}
```

6.2.5. Substitution et dominance des membres hérités

Lorsqu'un membre (donnée ou fonction) est redéfini dans une classe dérivée, il est toujours possible d'accéder aux membres de même nom de la classe de base, moyennant l'utilisation de l'opérateur de résolution de portée, sous réserve qu'un accès soit autorisé.

6.2.6. Cas particulier du constructeur par recopie

Si la classe dérivée ne possède pas de constructeur par recopie, il y aura appel du constructeur par recopie par défaut de la classe dérivée, qui procédera comme suit :

- appel au constructeur de recopie de base (par défaut ou celui défini)
- initialisation des membres de classe dérivée non hérités de la classe de base.

Un problème se pose lorsque la classe dérivée définit explicitement un constructeur par recopie. A savoir que l'appel à ce constructeur par recopie entraînera l'appel :

- du constructeur de la classe de base mentionné dans son en-tête
- d'un constructeur sans argument si aucun constructeur de la classe de base n'est précisé dans l'en-tête => classe de base doit avoir un constructeur sans argument pour éviter une erreur de compilation.

6.2.7. Conversion implicite des enfants en parents

- Même si l'enfant "est un" parent, ce sont 2 types différents.
- Les objets enfants peuvent être traités comme des objets parents. C'est logique puisque l'enfant a toutes les propriétés du parent (attributs et méthodes).
- Par contre, l'inverse n'est pas vrai. C'est-à-dire que l'on ne peut traiter un parent comme un enfant, car il n'en possède pas nécessairement tous les attributs et méthodes.
- Un pointeur ou référence vers la classe de base peut pointer vers n'importe quel héritier.

6.2.8. Les pointeurs sur des objets dans les héritages

Les règles de compatibilité entre une classe de base et une classe dérivée permettent d'affecter à un pointeur sur une classe de base la valeur d'un pointeur sur une classe dérivée. Le type des objets pointés est défini lors de la compilation .
Un pointeur ou référence vers la classe de base peut pointer vers n'importe quel héritier.
Conversion d'un pointeur sur une classe dérivée en un pointeur sur une classe de base est légale, l'inverse non
Un pointeur d'une classe de base peut contenir l'adresse d'un objet d'une classe dérivée.

6.2.9. Les fonctions virtuelles et polymorphisme

Problème :
Si une classe de base comprend une méthode avec un nom identique à la méthode de la classe dérivée, tout appel à cette méthode par un pointeur de la classe de base fait référence à la méthode de la classe de base.

Pourquoi ?
Quand le compilateur compile le programme, par défaut, il fige les différents appels aux méthodes en fonction du type de pointeur (typage statique des objets).

Solution :
L'emploi des fonctions virtuelles permet d'éviter des problèmes inhérents au typage statique.
Lorsqu'une fonction est déclarée virtuelle (mot clé virtual) dans une classe, les appels à une telle fonction ou à n'importe laquelle de ses redéfinitions dans des classes dérivées sont résolues au moment de l'exécution, en fonction du type concerné (typage dynamique des objets).

Règles :
1. le mot clé virtual ne s'emploie qu'une fois pour une fonction donnée (uniquement dans la classe de base, pas dans les classes dérivées)
2. une méthode déclarée virtuelle dans une classe de base peut ne pas être redéfinies dans ses classes dérivées
3. une fonction virtuelle peut être surdéfinie
4. un constructeur ne peut être virtuel, un destructeur peut l'être
5. une méthode statique ne peut être déclarée comme virtuelle
6. lors de l'héritage, le statut de l'accessibilité de la méthode virtuelle (public, protected, private) est conservé dans toutes les classes dérivées, même si elles sont redéfinies avec un statut différent. Le statut de la classe de base prime.

Polymorphisme :
Le polymorphisme rend possible l'utilisation d'une même instruction pour appeler dynamiquement des méthodes différentes dans la hiérarchie de classe. En effet, lorsque le compilateur rencontre une méthode virtuelle, il sait qu'il faut attendre l'exécution pour déterminer la bonne méthode à appeler.

6.2.10. Les fonctions virtuelles pures et classes abstraites

Une fonction virtuelle pure se déclare avec une initialisation à 0.
Une fonction virtuelle pure dans une classe de base doit obligatoirement être redéfinie dans une classe dérivée ou déclarée à niveau virtuelle pure.
Lorsqu'une classe comporte au moins une fonction virtuelle pure, elle est considérée comme « abstraite » c'est-à-dire il n'est plus possible de créer des objets de son type.
On ne peut utiliser une classe abstraite qu'à partir d'un pointeur ou d'une référence.

6.2.11. Les destructeurs virtuels

Un constructeur ne peut être déclaré comme virtuel (pas possible de construire un objet sans connaître son type).
Par contre, il est possible de détruire un objet sans pour autant connaître exactement son type.
Chaque fois que la classe utilise des liaisons dynamiques, il faut déclarer le destructeur comme virtuel.

6.3. L'héritage multiple

Il permet de créer des classes dérivées à partir de plusieurs classes de base. Pour chaque classe de base, on peut définir le mode d'héritage.

Exemple :

```
#include<iostream.h>
#include<stdiostr.h>
#include<string.h>
#define MAX 20

class personne
{
protected :                // pour que accessible dans affiche independant
        char nom[MAX];
        char prenom[MAX];
public:
        personne(char n[MAX],char p[MAX]);
        void affiche();
        ~personne();
};

class compagnie
{
protected :
        char nom[MAX];
public:
        compagnie(char n[MAX]);
        void affiche();
        ~compagnie();
};

class independant:public personne, public compagnie
{
float numtva;
public:
        independant(float nt, char n[MAX], char p[MAX], char no[MAX]);
        void affiche();
        ~independant();
};

personne::personne(char n[MAX], char p[MAX])
{
strcpy(nom,n);
strcpy(prenom,p);
cout<<"Constructeur de personne\n";
}
```

```
void personne::affiche()
{
cout<<"NOM : "<<nom<<endl;
cout<<"PRENOM : "<<prenom<<endl;
}
personne::~personne()
{
cout<<"Destructeur de personne\n";
}

compagnie::compagnie(char n[MAX])
{
strcpy(nom,n);
cout<<"Constructeur de compagnie"<<endl;
}
void compagnie::affiche()
{
cout<<"Nom : "<<nom<<endl;
}
compagnie::~compagnie()
{
cout<<"Destructeur de compagnie\n";
}

independant::independant(float nt, char n[MAX], char p[MAX], char
no[MAX]):personne(n,p),compagnie(no)
{
numtva=nt;
cout<<"constructeur de independant\n";
}
void independant::affiche()
{
cout<<"Nom : "<<personne::nom<<endl;
cout<<"Prenom : "<<personne::prenom<<endl;
cout<<"Nom compagnie : "<<compagnie::nom<<endl;
personne::affiche();
compagnie::affiche();
cout<<"Numero de tva : "<<numtva<<endl;
}
independant::~independant()
{
cout<<"Destructeur de independant\n";
}

void main()
{
independant i(456,"DURANT","MARCEL","CFA");
i.affiche();
}
```

6.3.1. Appel des constructeurs et des destructeurs

En cas d'héritage multiple, les constructeurs des classes de base sont appelés dans l'ordre de la déclaration d'héritage, puis celui de la classe dérivée.
Les destructeurs sont appelés dans l'ordre inverse.
Le constructeur de la classe dérivée peut mentionner dans son en-tête des informations à retransmettre à chacun des constructeurs des classes de base.

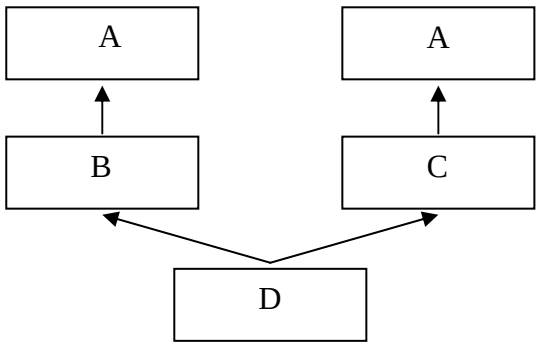
Exemple :

```
#include<stdiostr.h>
#include<conio.h>
#include<string.h>
#define MAX 20

class etat_civil
{
char nom[MAX];
public:
        etat_civil(char n[MAX]);
        void affiche();
        ~etat_civil();
};

class revenus:public etat_civil
{
int salaire;
public:
        salaire(int s,char n[MAX]);
        void affiche();
        ~revenus();
};

class abattements:public etat_civil
{
int hypotheque;
public:
        abattements(int h,char n[MAX]);
        void affiche();
        ~abattements();
};

class declaration_simplifiee:public revenus,public abattements
{
float impot;
public:
        declaration_simplifiee(float i,int s,int h,char n[MAX],);
        void affiche();
        ~declaration_simplifiee();
};
```

```
etat_civil::etat_civil(char n[MAX])
{
strcpy(nom,n);
cout<<"COnstructeur etat_Civil\n";
}
void etat_civil::affiche()
{
cout<<"nom : "<<nom<<endl;
}
etat_civil::~etat_civil()
{
cout<<"Destructeur etat civil"<<endl;
}

revenus::revenus(ints, char n[MAX]):etat_civil(n)
{
salaire=s;
cout<<"Constructeur revenus"<<endl;
}
void revenus::affiche()
{
cout<<"Salaire: "<<salaire<<endl;
}
revenus::~revenus()
{
~cout<<"destructeur de revenus"<<endl;
}

abattements::abattements(int h,char n[MAX]):etat_civil(n)
{
hypotheque=h;
cout<<"Constructeurs de abattements"<<endl;
}
void abattements::affiche()
{
cout<<"Hypotheque:"<<hypotheque<<endl;
}
abattements::~abattements()
{
cout<<"Destrcuteur de abattements"<<endl;
}

declaration_simplifiee::declaration_simplifiee(float i, int s, int h, char
n[MAX]):revenus(s,n),abattemennts(h,n)
{
impot=i;
cout<<"Constructeur de declaration simplifee"<<endl;
}
```

```
void declaration_simplifiee::affiche()
{
cout<<"Impot: "<<impot<<endl;
}
declaration_simplifiee::~declaration_simplifiee()
{
cout<<"destructeur de declaration simplifiee"<<endl;
}

void main()
{
declaration_simplifiee ds(1000,50,2000,250,"DUPONT");
ds.affiche();
}
```

6.3.2. Classes virtuelles



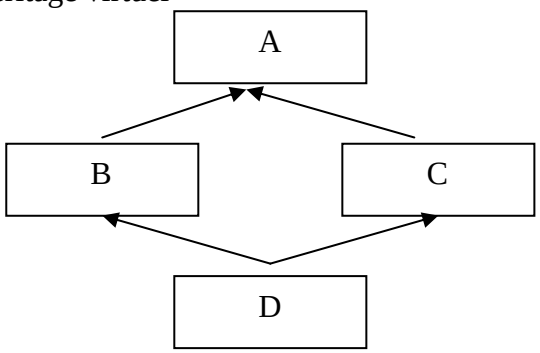
classe de base A

classes dérivées de 1er niveau

classe dérivée de 2ème niveau

Par le biais de dérivations successives, il est possible qu'une classe (classe D) hérite « 2 fois » d'une même classe (classe A).
Dans ce cas, les membres donnée de la classe en question (A)apparaissent 2 fois dans la classe dérivée de deuxième niveau (D).
Il est nécessaire de faire appel à l'opérateur de résolution de portée pour lever l'ambiguïté.
Si l'on souhaite que de tels membres n'apparaissent qu'une seule fois dans la classe dérivée de deuxième niveau, il faut dans les déclarations des dérivées de premier niveau (classe B et C) déclarer avec l'attribut **virtual** la classe dont on veut éviter la duplication.
Lorsqu'on a déclaré une classe virtuelle, il est nécessaire que les constructeurs d'éventuelles classes dérivées puissent préciser les informations à transmettre au constructeur de cette classe virtuelle.

Avec héritage virtuel



classe de base A

classes héritées virtuellement de 1er niveau

classe dérivée de 2ème niveau

Exemple :

```
#include<stdiostr.h>
#include<conio.h>
#include<string.h>
#define MAX 20

class etat_civil
{
char nom[MAX];
public:
        etat_civil(char n[MAX]);
        void affiche();
        ~etat_civil();
};

class revenus:virtual public etat_civil
{
int salaire;
public:
        revenus(int s,char n[MAX]);
        void affiche();
        ~revenus();
};

class abattements:public etat_civil
{
int hypotheque;
public:
        abattements(int h,char n[MAX]);
        void affiche();
        ~abattements();
};

class declaration_simplifiee:public revenus,public abatements
{
float impot;
public:
        declaration_simplifiee(float i,int s,int h,char n[MAX]);
        void affiche();
        ~declaration_simplifiee();
};
```

```
etat_civil::etat_civil(char n[MAX])
{
strcpy(nom,n);
cout<<"COnstructeur etat_Civil\n";
}
void etat_civil::affiche()
{
cout<<"nom : "<<nom<<endl;
}
etat_civil::~etat_civil()
{
cout<<"Destructeur etat civil"<<endl;
}

revenus::revenus(int s,char n[MAX]):etat_civil(n)
{
salaire=s;
cout<<"Constructeur revenus"<<endl;
}
void revenus::affiche()
{
cout<<"Salaire: "<<salaire<<endl;
}
revenus::~revenus()
{
cout<<"destructeur de revenus"<<endl;
}

abattements::abattements(int h,char n[MAX]):etat_civil(n)
{
hypotheque=h;
cout<<"Constructeurs de abattements"<<endl;
}
void abattements::affiche()
{
cout<<"Hypotheque:"<<hypotheque<<endl;
}
abattements::~abattements()
{
cout<<"Destrcuteur de abattements"<<endl;
}
```

```
declaration_simplifiee::declaration_simplifiee(float i, int s, int h, char
n[MAX]):revenus(s,n),abattements(h,n),etat_civil(n)
{
impot=i;
cout<<"Constructeur de declaration simplifee"<<endl;
}
void declaration_simplifiee::affiche()
{
cout<<"Impot: "<<impot<<endl;
}
declaration_simplifiee::~declaration_simplifiee()
{
cout<<"destructeur de declaration simplifiee"<<endl;
}

void main()
{
declaration_simplifiee ds(1000.50,2000,250,"DUPONT");
ds.affiche();
}
```

Exercice sur les heritages multiples

```
/*      Exercice 1 :
Créer une classe personnel comprenant 3 données membres (nom, position, salaire),
un constructeur initialisant les données membres et une function affiche_personnel
qui affiche les informations sur le personnel. Créer une classe patron dérivant
publiquement de la classe personnel et ayant 2 données membres (bonus_annuel,
marque de voiture de fonction) , un constructeur initialisant les données membres et
une fonction affiche_patron qui affiche les informations sur le patron (entant que
membre du personnel et patron).
Créer un programme permettant de tester ces classes. */

// By Nightsdarkangel

#include<stdiostr.h>
#include<iostream.h>
#include<conio.h>
#include<string.h>
#define MAX 20

class personnel
{
char nom[MAX];
char position[MAX];
int salaire;
public:
        personnel(char n[MAX],char p[MAX],int s);
        void affiche_personnel();
        ~personnel();
};

class patron:public personnel
{
int bonus_annuel;
char marque[MAX];
public:
        patron(int b,char m[MAX],char n[MAX],char p[MAX],int s);
        void affiche_patron();
        ~patron();
};

personnel::personnel(char n[MAX],char p[MAX],int s)
{
strcpy(nom,n);
strcpy(position,p);
salaire=s;
}
```

```
void personnel::affiche_personnel()
{
cout<<"Nom : "<<nom<<endl;
cout<<"Position : "<<position<<endl;
cout<<"Salaire : "<<salaire<<endl;
}

personnel::~personnel(){}

patron::patron(int b,char m[MAX],char n[MAX],char p[MAX],int s):personnel(n,p,s)
{
bonus_annuel=b;
strcpy(marque,m);
}

void patron::affiche_patron()
{
cout<<"Bonus annuel : "<<bonus_annuel<<endl;
cout<<"Marque : "<<marque<<endl;
personnel::affiche_personnel();
}

patron::~patron(){}

void main()
{
personnel p1("PINGU","pdg",1000);
p1.affiche_personnel();
patron pun(1000,"bmw","PINGU","pdg",1000);
pun.affiche_patron();
}
```