Programmation Orientée Objet

—

Mise en œuvre en C++

VIAL Christian, CHALON René.

Polycopié du cours de tronc commun de 2ème année

Département Mathématiques-Informatique
Ecole Centrale de Lyon - septembre 2003

Avertissement :

L'enseignement d'informatique de tronc commun dispensé à l'Ecole Centrale de Lyon se compose de deux parties :

- En 1ère année : étude de l'algorithmique et des structures de données, programmation en langage C (avec les extensions non-objets du C++) ; la mise en œuvre des structures de données classiques s'effectue à travers la bibliothèque normalisée du C++, STL (Standard Template Library).
- En 2ème année : étude de la programmation orientée objet, programmation en langage C++.

Ce polycopié constitue donc le support du cours de 2ème année intitulé "Programmation Orientée Objet". Il concerne uniquement la programmation par objets en C++ et non pas l'analyse et la conception par objets.

Il fait suite au polycopié de 1ère année, et ne reprend donc pas les éléments de base du langage C et la description des classes de bibliothèques, il est donc conseillé de consulter le polycopié de 1ère année intitulé "Programmation en C".

Bibliographie :

Ce polycopié n'est pas un manuel de référence exhaustif du langage C++, mais il est tout à fait suffisant pour traiter la majorité des programmes en C++. Pour ceux qui désirent aller plus loin et étudier les recoins du langage, nous leur recommandons les ouvrages ci-dessous:

DELANNOY Claude. *Programmer en langage C++*, 5ème édition, Eyrolles 2000, ISBN 2-212-09138-9, 613 pages.

STROUSTRUP Bjarne. *Le langage C++*. 3ème édition, Campus Press/MacMillan 2001, ISBN 2-7440-1089-8, 1096 pages.

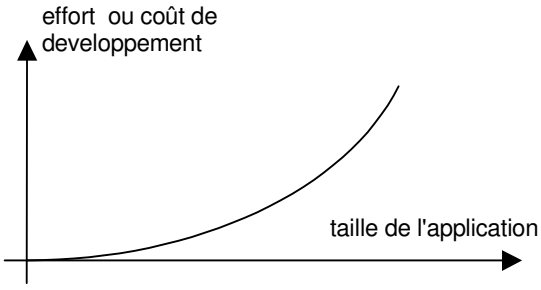
Table des Matières

Chapitre 1
Présentation générale

1.1 Emergence de la programmation par objets

1.1.1 La crise du logiciel

Le matériel est de moins en moins cher et de plus en plus puissant ce qui permet de réaliser des applications de plus en plus complexes. Les coûts informatiques sont concentrés sur la partie logicielle et le coût salarial d'un utilisateur d'ordinateur a depuis longtemps largement dépassé le coût d'acquisition, d'utilisation, d'amortissement et de maintenance de la machine.



Programmation traditionnelle

Le coût du logiciel est de plus en plus élevé et croit exponentiellement avec la complexité. Il faut donc essayer de diminuer les coûts, d'augmenter la durée de vie du logiciel en le faisant évoluer et en le réutilisant.

1.1.2 Objectifs de la programmation par objets

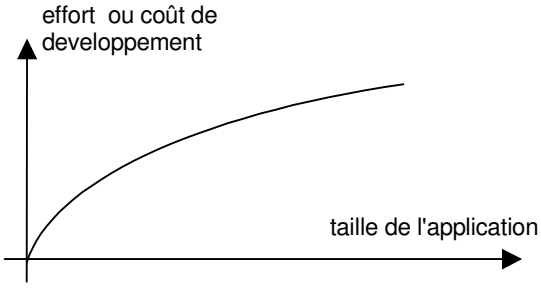
La programmation par objets cherche à modéliser directement des objets du monde réel de manière à ce qu'à un objet du monde réel corresponde une entité informatique.

Dans le monde réel on a de très nombreux **représentants d'un petit ensemble de concepts** différents, on cherche donc à exploiter cette redondance en créant des entités informatiques représentant les concepts avant de les utiliser dans une application particulière.

Dans les domaines de la construction électronique ou mécanique on dispose d'un ensemble de composants préexistants avec des interfaces standardisées (circuits électroniques, éléments d'assemblages mécaniques, etc.). On cherche à construire des **bibliothèques de composants informatiques** (objets) et de réaliser la plus grande partie d'une application par assemblage de ces composants.

Un des problèmes majeurs de la construction et de la gestion d'un assemblage complexe est la sensibilité de l'ensemble à une modification d'une partie. On cherche à réduire l'impact des modifications et extensions par **confinement** dans des petites unités (objets) qui ont peu de contact avec l'extérieur.

Pour aboutir à l'objectif précédent et faciliter la compréhension et l'évolution de l'application on produit un ensemble d'objets et on cherche à **minimiser et à expliciter les couplages entre objets**. On aboutit à des entités informatiques protégées et abstraites grâce à la **séparation entre l'interface et l'implémentation**. L'accès à un objet ne dépend pas d'une implémentation particulière mais seulement d'une interface de haut niveau ce qui permet de modifier facilement l'implémentation d'un objet sans incidence pour les autres objets qui ne connaissent que son interface.



Programmation par objets

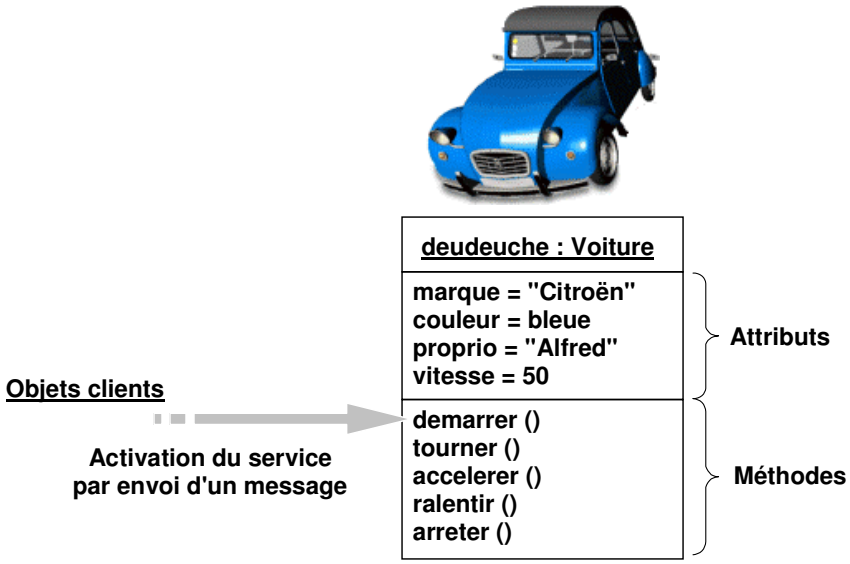
La mise en oeuvre de la programmation par objets vise à obtenir une courbe coût/complexité dont la concavité est changée. Une petite application demande un effort de développement non négligeable mais son extension demande peu d'effort car la réutilisation est maximale et la complexité est mieux maîtrisée

1.2 Le monde des objets

Le monde à modéliser correspond à une **collection d'objets informatiques**, chaque objet possède un ensemble d'**attributs** et un ensemble de **méthodes**. Les attributs sont des données dont les valeurs caractérisent l'**état de l'objet** à un instant donné. Les méthodes sont les traitements qui caractérisent la **compétence de l'objet**, l'activation d'une méthode modifie des valeurs d'attributs et change donc l'état de l'objet.

La communication entre objets s'effectue par envoi de **messages**, un objet envoie un message à un autre objet pour lui demander d'activer une de ses méthodes. L'objet demandeur est un client de l'autre objet qui apparaît comme un serveur. Un message contient :

- un **sélecteur,** qui permet d'identifier la méthode demandée
- des **arguments**, qui sont les paramètres de la méthode.



NB : Le schéma ci-dessus utilise la représentation UML (Unifed Modeling Language). Pour une introduction à ce formalisme on se reportera à l'Annexe A. Tous les concepts exposés par la suite sont expliqués par du texte et illustré par un schéma UML.

1.3 Principes de base de la programmation par objets

Un langage de programmation est généralement dénommé langage à objets quand il met en oeuvre les principes suivants:

1.3.1 Encapsulation

Objet = données + traitements
Donnée = attribut
Traitement = méthode

Il existe dans le langage une forme syntaxique qui permet de définir une entité insécable contenant des données et des traitements.

1.3.2 Protection des données et des traitements

Objet = interface + implémentation

Le langage permet de définir pour une entité une interface accessible de l'extérieur (attributs et méthodes) et une implémentation inaccessible directement.

1.3.3 Masquage de l'implémentation

Le langage permet de masquer complètement à l'utilisateur l'implémentation des méthodes, on peut savoir ce que sait faire un objet mais pas comment il le fait. Généralement ce masquage est mis en œuvre par une compilation des méthodes.

1.3.4 Classes et instances

Classe	**= concept**	**= type**
Instance	**= représentant du concept**	**= variable**

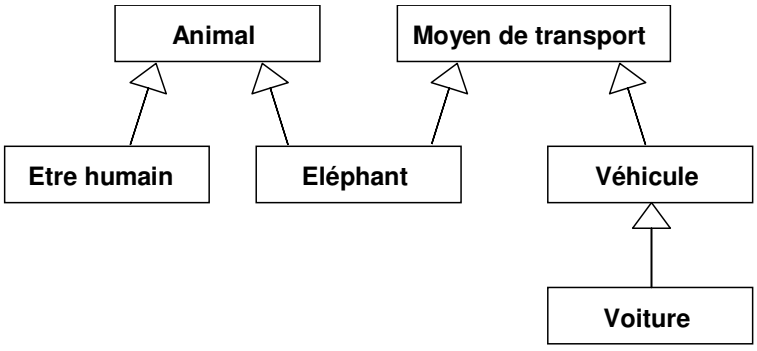
Le langage permet de définir des classes pour représenter les concepts caractérisés par des données et des méthodes. La réalisation d'une application s'effectue en manipulant des représentants des concepts (instances de classes). Dans le cas du langage C++, les classes apparaissent comme des nouveaux types définis par le programmeur et les instances comme des variables de ce type.

1.3.5 Communication par envoi de messages

La structure de contrôle principale utilisée par les objets est l'envoi de messages pour activer des traitements de l'objet destinataire. Certains langages comme smalltalk n'ont pas d'autres structures de contrôle, le langage C++ conserve les structures de contrôle de C et simule un envoi de message en appelant directement les méthodes publiques d'un objet.

1.3.6 Héritage de classes

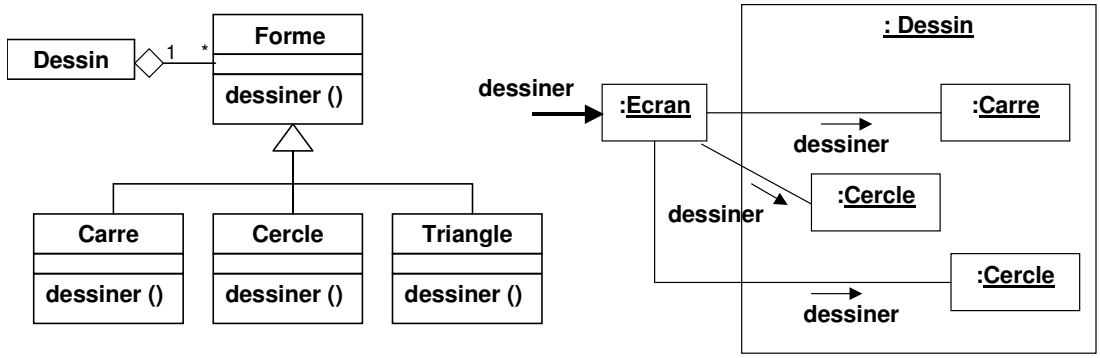
Les différentes classes définissables à l'aide d'un langage à objets peuvent être organisées en un treillis d'héritage. Une classe qui hérite d'une autre classe est considérée comme une **spécialisation** de la classe initiale, elle dispose des méthodes et attributs de la classe ancêtre auxquels elle ajoute ses propres attributs et méthodes. On obtient ainsi une factorisation des traitements mais aussi un puissant outils d'extension et de réutilisation.



A l'inverse, on appelle **généralisation** l'action de factoriser des classes en groupant des propriétés qui sont communes.

1.3.7 Polymorphisme

Un objet polymorphe est susceptible de changer de forme pendant sa vie. Les langages à objets permettent de définir des données polymorphes comme par exemple une liste ou un tableau polymorphe qui peuvent contenir à un instant donné des objets de types divers.

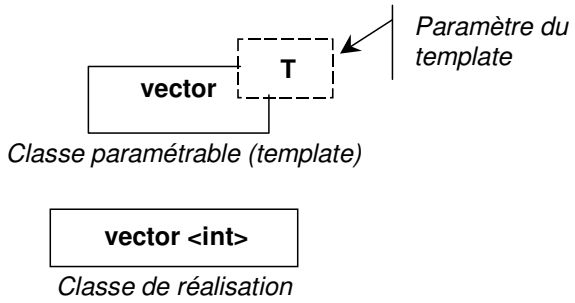


Dans l'exemple ci-dessus, le diagramme de classe spécifie qu'un Dessin est constitué d'un ensemble de Forme. Chaque Forme peut être spécialisée en Carre, Cercle, Triangle, etc. qui redéfinissent chacune la méthode dessiner() propre. Lorsqu'un objet comme Ecran veut demander à un Dessin de se redéssiner, il invoque la méthode dessiner() de chaque Forme composant ce Dessin sans qu'il ait à se préoccuper de savoir si c'est en réalité un Cercle ou un Carre qui est concerné, la méthode dessiner() *ad hoc* étant correctement appelée.

1.3.8 Généricité

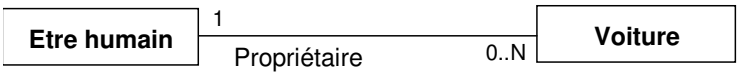
Une **fonction générique** est définie comme un modèle qui permet de créer différentes versions de cette fonction pour prendre en compte des arguments de types variables. Par exemple une fonction de tri générique est écrite une seule fois en appliquant un algorithme classique ce qui constitue un modèle pour cette fonction de tri. On peut ensuite générer automatiquement différentes versions de la fonction, à partir du modèle pour trier des entiers, des réels, des structures, etc.

On peut de la même manière avoir des **classes génériques**, pour lesquels on crée des versions spécialisées. Une classe générique pile dispose des méthodes empile, dépile, etc. pour traiter des éléments de type quelconque, on peut créer, à partir du modèle, une pile d'entiers, une pile de réels, etc.



1.3.9 Relations d'instance

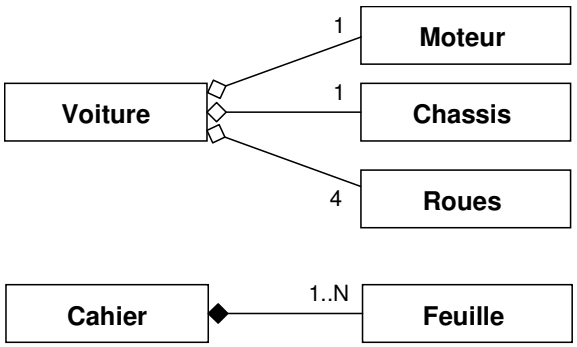
Les objets peuvent être liés entre eux par des relations. Dans le cas le plus général, ces relations sont appelées **associations**. Le sens, la nature, la cardinalité peuvent être précisés sur le diagramme UML.



Lorsque des objets sont des parties d'un objet plus grand alors on parle de **relation d'aggrégation**.

Enfin, lorsque l'agrégation est forte on parle de **relation de composition**. Dans ce cas, l'objet composant a une durée de vie identique à l'objet composé (construction et destruction simultanées).

NB : Ces relations sont représentées en UML sur le diagramme des classes bien qu'elles soient en réalité entre les instances de ces classes.



1.4 Les langages à objets

1.4.1 Emergence des langages objets

Les langages à objets sont issus de deux souches qui ont défini les fondements de la programmation par objets, ce sont le langage **Simula** apparu dans les années 60 et le langage **Smalltalk** apparu dans les années 70.

1.4.2 Langage objets, langages orientés objets

Il existe des langages dits de programmation par objets qui possèdent les éléments syntaxiques permettant de mettre en oeuvre explicitement tous les principes de la programmation par objets.

Il existe des langages dits orientés objets qui ne mettent en œuvre explicitement que quelques principes de la programmation objet. Cette subtile distinction permet à tous les langages actuels de se prévaloir de l'appellation objet

1.4.3 Langage objets natifs, langages à extension objet

Certains langages ont été défini à partir du concept d'objet et ne peuvent pas manipuler autre chose que des objets. Smalltalk est l'archétype de ces langage, sa définition est homogène et rigoureuse mais il ne permet pas de récupérer facilement tout l'acquis de programmation dans les langages antérieurs.

Une autre approche consiste à étendre certains langages existants pour pouvoir traiter des objets tout en conservant la programmation traditionnelle. C'est le cas de C++ qui est un surensemble objet de C,

un compilateur C++ traite sans problème une texte C écrit il y a plusieurs années. Actuellement on ne commercialise plus de compilateurs C puisque les compilateurs C++ peuvent les remplacer. Cette approche permet de récupérer tous les anciens programmes c'est pourquoi elle a été mise en oeuvre pour de nombreux langages.

C	**-->**	**objectiveC, C++**
Pascal	**-->**	**Turbo Pascal objet**
Lisp	**-->**	**flavors, xlisp**
Prolog	**-->**	**Emicat**

1.4.4 Le langage C++

Le langage C++ a été créé en 1980 par Bjarne Stroustrup dans les laboratoires Bell. C++ fait suite au langage C qui avait été défini et développé dans la même entreprise. En 1986 est apparu la première définition publique du langage à travers l'ouvrage de B. Stroustrup "The C++ Programming language". En 1990 le langage C++ a été normalisé dans sa version 2.0 par l'ANSI ce qui lui a donné un accès universel ; actuellement la version utilisée est celle normalisée en 1998 par l'ISO sous la référence ISO 14882. En plus du langage, cette norme définit la bibliothèque STL (Standard Template Library) qui fournit les structures de données les plus classiques (liste, vecteur, pile, file, etc...)

Chapitre 2
Extensions C++ de C hors objet

C++ a introduit un certain nombre d'extensions au langage C. ces extensions facilitent la programmation, renforcent les contrôles mais ne concernent pas les objets.

Ces extensions ont déjà été abordées dans le cours de 1ère année et ce chapitre constitue donc essentiellement un rappel.

2.1 Commentaires de fin de ligne

Un commentaire peut être introduit dans une ligne C++ à l'aide de la séquence // (double slash), le commentaire prend fin à la fin de la ligne. Les commentaires C (qui se notent **/* . . . */**) sont également utilisables.

```
x = z-3;           // calcul de la vitesse du vent
if (x == 0)        // test de vent nul
```

2.2 Nom simplifié des struct, union, enum

En C de base un type struct se définit et s'utilise de la manière suivante :

```
struct article     // definition du type struct article
{
    int numero;
    double prix;
};

struct article x;  // x variable de type struct article
```

NB: Le nom du type est **struct article**.

En C++ un type struct se définit et s'utilise de la manière suivante :

```
struct article     // definition du type article
{
    int numero;
    double prix;
};

article x;         // x variable de type article
```

NB: Le nom du type est **article**.

2.3 Passages de paramètres et retour de fonctions

2.3.1 Passage de paramètres par adresse à une fonction

Par défaut les arguments d'une fonction C sont passés par valeur, la fonction travaille sur des copies et ne peut donc pas modifier les arguments d'appel. Le fait d'indiquer comme types de paramètre des références permet de travailler directement sur les arguments d'appel et donc de les modifier dans la fonction.

```
void permute1 (int x, int y)
{
    int temp = x;
    x = y;
    y = temp;
}

int main()
{
    int a=3, b=4;
    permute1 (a,b);
    cout << "Résultat: " << a;
    cout << " " << b << endl;
    return 0;
}
```

```
void permute2 (int &x, int &y)
{
    int temp = x;
    x = y;
    y = temp;
}

int main()
{
    int a=3, b=4;
    permute 2(a,b);
    cout << "Résultat: " << a;
    cout << " " << b << endl;
    return 0;
}
```

A l'exécution :

```
Résultat : 3 4
```

```
Résultat : 4 3
```

NB1: permute1 travaille sur des copies des arguments (passage par valeur), les copies sont permutées mais pas les arguments d'appels à l'extérieur de la fonction.

NB2: permute2 travaille directement sur les arguments d'appels (passage par référence ou par adresse) les valeurs des arguments d'appels sont donc permutées.

NB3: Les corps des fonctions permute1 et permute2 sont identiques, seul l'entête indique un mode de passage de paramètres différent.

2.3.2 Passage de paramètres par référence intangible

Le passage de paramètre par référence peut aussi être utilisé pour gagner du temps lors de l'exécution. Un argument volumineux passé par valeur nécessite un temps de recopie non négligeable il est donc plus efficace de le passer par référence à une fonction.

Si un argument volumineux est passé par référence mais que l'on désire le protéger d'une modification de la part de la fonction appelée, on peut le passer comme une **référence intangible** à l'aide du mot clé **const**

```
struct bidule                    // bidule est une structure
{                                // très volumineuse
    . . . . .
};

void f(const bidule &b)  // b référence intangible de bidule
{                        // f a accès directement à l'argument
    . . . . .            // d'appel mais ne peut pas le
}                        // modifier

int main()
{
    bidule x;
    . . . . .
    f(x);                // appel de f en passant x par référence
    return 0;
}
```

2.3.3 Renvoi d'une variable par une fonction

Une fonction C renvoie une valeur qui est une copie de la valeur de l'expression située dans l'instruction return. Un appel de fonction désigne une valeur et non pas une variable, il ne peut donc jamais figurer à gauche d'un opérateur =.

Le fait d'indiquer comme type de valeur de retour une référence permet de renvoyer non pas une valeur mais l'accès véritable à la zone de stockage, on peut donc modifier directement le résultat d'une fonction et même utiliser l'appel de fonction à gauche d'un opérateur =

```
struct complexe                          struct complexe
{                                        {
public:                                  public:
    double r;  // partie réelle              double r; // partie réelle
    double i;  // partie imaginaire           double i; // partie imaginaire
);                                       };

double reel(complexe c)                  double & reel(complexe &c)
{                                        {
    return c.r;                              return c.r;
}                                        }
```

```
int main()                          int main()
{                                   {
    complexe a;                         complexe a;
    double z;                           double z;
    . . . . .                           . . . . .
    z = reel(a);     // OK             z = reel(a);          // OK
    reel(a) = 6.35;  // ERREUR         reel(a) = 6.35        // OK
    . . . . .                           . . . . .
}                                   }
```

NB1: Attention à ne pas renvoyer une référence à une variable locale qui est systématiquement détruite en franchissant l'accolade fermante du bloc où elle a été créée.

NB2: L'argument de la fonction *reel* de droite est passé par référence. S'il était passé par valeur, la fonction travaillerait sur une copie locale et tenterait donc de renvoyer par référence une variable locale (cf. NB1)

2.4 Paramètres par défaut dans les fonctions

Il est possible d'indiquer dans l'entête d'une fonction des valeurs par défaut pour certains paramètres ce qui permet d'appeler la fonction uniquement avec les paramètres intéressants, les autres prenant les valeurs par défaut.

```
int f(int x, int y = 2, float z = 3.14)
{    . . . . .    }
```

```
t = f(5, 6, 6.35);       // OK f(5, 6, 6.35)
t = f(5, 6);             // OK f(5, 6, 3.14)
t = f(5);                // OK f(5, 2, 3.14)
t = f();                 // ERREUR
```

NB1: Seuls les arguments de la fin de liste peuvent avoir des valeurs par défaut ce qui oblige à choisir un ordre des arguments en fonction de ceux qui ont des valeurs par défaut

NB2: La même possibilité existe pour les arguments des méthodes des classes.

2.5 Surcharge des fonctions

En C, il ne peut exister qu'une seule fonction ayant un certain nom, en C++ on peut avoir plusieurs fonctions de même nom mais avec des arguments en nombre et type différents

```
int add (int a, int b)          // fonction add à 2 arguments int
{   return a+b;   }

int add (int a, int b, int c)  // fonction add à 3 arguments int
{   return a+b+c; }

int main()
{
    int a, b, c, d, e;
    d = add(a,b);       // appel add à 2 arguments int
    e = add(a,b,c);     // appel add à 3 arguments int
}
```

Lors de l'appel, la distinction entre les deux fonctions se fait par le type et le nombre des arguments.

2.6 Opérateurs new et delete en C++

La réservation et la libération d'espace mémoire pendant l'exécution (mémoire dynamique) s'effectue en C à l'aide de fonctions de bibliothèque (malloc, calloc, free). En C++ la gestion de mémoire dynamique s'effectue à l'aide des 2 opérateurs new et delete.

```
int x,*y,*z;   // x est un int, y, et z sont des pointeurs sur int

y = new int;   // réservation dynamique de place pour un int pointé par y

z = new int[3];  // réservation d'une zone de 3 int pointée par z

delete y;        // libération de l'espace pointé par y

delete[] z;     // libération de l'espace occupé par le tableau de 3 int
```

NB. delete ne change pas la valeur de y ni celle de z, ces 2 pointeurs pointent maintenant sur une zone qui ne leur est plus réservée.

2.7 Entrées-sorties sur flot

Pour effectuer des entrées sorties en C de base on utilise une série de fonctions (printf(), scanf(), read(), write(), etc.) avec une syntaxe parfois complexe. C++ a unifié et simplifié la programmation des entrées-sorties en utilisant les flots définis sous forme d'objets avec les opérateurs '>>' et '<<'. Même si les flots reposent sur une extension objet de C++ par rapport à C, l'utilisation des entrées-sorties sur flots ne necessite pas de programmation par objets, ni même de comprehension profonde du mécanisme. L'utilisation des flots d'entrée-sortie est exposée au Chapitre 6.

```
int x;
double y;
char c;
cout << "x vaut : " << x << endl; // affichage d'1 chaine puis d'1 int
cout << "y vaut : " << y << endl; // affichage d'1 chaine puis d'1 double
cout << "c vaut : " << c << endl; // affichage d'1 chaine puis d'1 char
```

NB: endl correspond au caractère de passage à la ligne

2.8 Les chaînes de caractères (char * ou string)

Les chaînes de caractères du langage C de base (char *) posent généralement problème pour les utilisateurs débutants c'est pourquoi la classe string de la bibliothèque C++ a été introduite. La classe string encapsule tous les mécanismes de manipulation dans des opérateurs (=, +, <, >, etc.) supprimant ainsi tous les problèmes et il est donc conseillé de l'utiliser.

On présente cependant ici le fonctionnement des chaînes de caractères du langage C de base car il permet de comprendre comment a été réalisée la classe string qui s'appuie sur les chaînes de base. Par ailleurs, pour illustrer à l'aide d'exemples courts certains mécanismes de C++, il est commode d'utiliser les chaines char*.

2.8.1 Chaîne du langage C de base

Une chaîne de caractères C est une succession de caractères en mémoire, terminée par un caractère de fin de chaîne (caractère null ASCII). Une chaîne est repérée et manipulée à travers un pointeur sur son premier caractère

Création d'une chaine

Pour créer une chaîne il est nécessaire de réserver de la place en mémoire, puis il faut transférer les caractères dans la zone et placer la marque fin de chaîne .

```
char ch1[20];            // ch1 tableau statique de 20 char
strcpy(ch1,"bonjour")    // recopie d'une chaîne dans ch1
char * ch2;              // ch2 pointeur sur char
ch2 = new char[10];      // ch2 pointe sur une zone dynamique de 10 char
strcpy(ch2,"salut");     // recopie d'une chaine dans ch2
```

Affectation de chaine

Pour recopier une chaîne dans une autre il faut s'assurer que la chaîne réceptrice a une taille suffisante puis il faut utiliser la fonction de bibliothèque strcpy() pour effectuer la copie. On ne doit pas utiliser l'opérateur = qui ne recopierait rien mais ferait simplement pointer les deux variables sur la même zone mémoire.

```
strcpy(ch2,ch1);         // ch2 contient "bonjour"
```

Comparaison de chaînes

Pour comparer deux chaînes il ne fait pas utiliser les opérateurs de relation (==, !=, <, >, >=, <=) qui comparent les adresse en mémoire des zones contenant les chaînes. On doit utiliser la fonction de bibliothèque strcmp() qui rend -1 si la première chaîne est inférieure à la seconde, 0 si les chaînes sont égales, 1 si la première chaîne est supérieure à la seconde (ordre lexicographique)

```
if (strcmp(ch1,ch2))
    . . . . .
```

Libération d'une chaîne

Si une chaîne a été stockée dans une zone dynamique à l'aide de l'opérateur new, on peut libérer la place quand elle n'est plus utile à l'aide de l'opérateur delete

```
delete[] ch1;            // libération de la place pointée par ch1
```

longueur d'une chaîne

Pour connaître la longueur d'une chaîne (nombres de caractères avant la marque de fin de chaîne) on peut utiliser la fonction strlen().

```
char * ch;
ch = new char[strlen("bonjour")+1];   // réservation de place
strcpy(ch,"bonjour");                 // recopie de la chaîne
```

Concaténation de deux chaînes

Pour ajouter une chaîne à la suite d'une autre chaîne on doit d'abord s'assurer que la place est réservée pour accueillir les deux chaînes dans la chaîne réceptrice, puis on utilise la fonction strcat() pour effectuer la concaténation.

```
char * ch1;
char * ch2;
ch1 = new char[12];      // réservation zone de 12 char
ch2 = new char[4];       // réservation zone de 4 char
strcpy(ch1,"bon");       // recopie "bon" dans ch1
strcpy(ch2,"jour");      // recopie "jour" dans ch2
strcat(ch1,ch2)          // ch1 = "bonjour"
```

2.8.2 Utilisation de la classe string

La mise en œuvre de chaîne en utilisant la classe string est très facile, les chaînes se manipulent comme des objets simples on peut utiliser les opérateurs =, <, >, = =, != , + et il n'y a pas de problème de réservation et de libération de place.

```
{
    string ch1("bon");   // création de ch1
    string ch2;          // création de ch2
    string ch3;          // création de ch3
    ch2 = "jour";        // affectation de chaînes
    ch3 = ch1 + ch2;     // concaténation et affectation de chaîne
    if (ch1 < ch2)       // comparaison de chaînes
        . . . . .
}                        // libération des chaînes (franchissement
                         // de l'accolade fermante)
```

Chapitre 3
Les Classes en C++

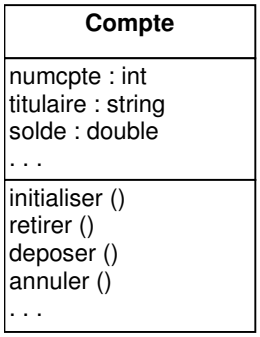
3.1 Les objets en C++

3.1.1 Encapsulation des données et des traitements en C++

Pour appliquer le principe d'encapsulation, c'est à dire regrouper en une seule entité des données et des traitements, le langage C++ a introduit la notion de **classe** qui apparaît comme une structure qui comporte :

- les données ou **attributs** de la classe
- les traitements associés ou **méthodes** (appelés également fonctions membres ou opérations)

Par exemple, un compte bancaire peut être représenté par un objet avec des attributs (numero, nom du titulaire, solde, etc.) et des méthodes (initialiser, retirer, deposer, annuler, etc.).



3.1.2 Protection des données et des traitements en C++

Pour compléter le mécanisme d'encapsulation le langage C++ propose un mécanisme de protection qui s'applique aussi bien aux attributs qu'aux méthodes.

Les classes C++ apparaissent comme des structures avec :

- des zones publiques à accès libre (introduites par le mot clé **public:**)
- des zones privées protégées. (introduites par le mot clé **private:**)

Les données et méthodes (ou fonctions membres) situées dans la partie privée sont accessibles uniquement à travers les méthodes de la partie publique.

NB1: sans indication, le contenu d'une classe est **privé par défaut.**

NB2: Il existe également une zone protégée (introduite par le mot-clé **protected**) qui sera vue dans le paragraphe sur l'héritage et qui se comporte comme la zone privée vis-à-vis des accès externes.

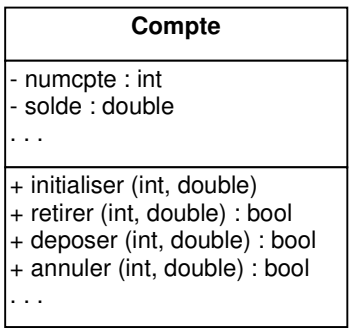
Il est alors possible de modifier la réalisation interne d'une classe (données et traitements privés) sans que les utilisateurs de la classe ne soient modifiés car les appels des méthodes publiques peuvent demeurer inchangés.

L'accès aux données privées à travers les méthodes publiques permet aussi de filtrer les valeurs aberrantes pour les attributs. Dans le problème de compte bancaire vu plus haut, il est possible d'avoir les attributs numcpte et solde en zone privée ce qui interdit leur accès direct et de vérifier dans les méthodes d'accès publiques, comme, par exemple, retirer(), que le solde du compte est suffisant pour le retrait demandé.

Une classe apparaît comme un type de donnée défini par l'utilisateur au moyen de données et de traitements et les variables du type comme les instances de la classe. Les données sont dupliquées dans les instances et les fonctions sont mise en commun dans les classes.

Dans l'exemple du compte bancaire, on souhaite qu'aucun programme ne puisse modifier directement l'attribut solde ou le numéro du compte d'une variable de type Compte, seules les opération sécurisées deposer, retirer, etc. doivent être autorisées.

Dans ce cas les attributs seront déclarés dans la section privée (introduite par le mot-clé **private**) de la classe. Ils sont ainsi inaccessibles dans les instances, seules les méthodes de la classe peuvent y accéder.



```
class Compte                              // définition d'une classe
{
private:                                  // zone privée
    int numcpte;                          // données privées
    double solde;
public:                                   // zone publique
    void initialiser (int n, double s);
    int retirer (int num, double somme);
    bool deposer (int num, double somme);
    bool annuler (int num, double somme);
    . . . . .
};

int main ()
{
    Compte cpt;
    cpte.initialiser(2555, 20000);
    cpte.solde -= 22000;                  // ERREUR COMPILATION
    . . . . .
}
```

NB1: Le mot clé **class** introduit la définition d'une classe.

NB2: Les mots clé **private:** et **public:** introduisent les zones privées et publique.

NB3: Une tentative d'accès direct à un élément de la zone privée provoque une erreur de compilation.

NB4: Attention, l'accolade fermante de déclaration d'une classe est suivie obligatoirement d'un poitn-virgule.

3.1.3 Exemple

Nous voulons construire une classe qui permette de stocker des dates sous la forme jour/mois/année.

Il faut donc prévoir 3 attributs entiers pour stocker les valeurs. Ces attributs seront déclarés privés pour interdire leur modification directe. L'accès aux données se fait donc au moyen de 3 méthodes :

- initialiser() qui permet de rentrer une date. Cette méthode vérifie que la date entrée est valide et retourne false sinon,
- lire_valeur() qui permet de relire la date.
- afficher() qui permet d'afficher directement la date à l'écran.

Date
- jour : int - mois : int - annee : int
+ initialiser (j : int, m : int, a : int) : bool + lire_valeur (out j : int, out m : int, out a : int) + afficher ()

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

class Date
{
    int jour, mois, annee;      // attributs privés par défaut
public:
    bool initialiser (int j, int m, int a);
    void lire_valeur (int & j, int & m, int & a);
    void afficher () const;
};

bool Date::initialiser (int j, int m, int a)
{
    if (j < 1 && j > 31)  return false;
    jour = j;
    if ( m < 1 && m > 12) return false;
    mois = m;
    if (a < 0)  return false;
    annne = a;
    return true;
}
```

```
void Date::lire_valeur (int& j, int& m, int& a)
{
    j = jour; m = mois; a = annee;
}

void Date::afficher ()
{
    cout << jour << "/" << mois << "/" << annee;
}
```

```
int main ()
{
    Date hier ;                          // variable Date
    Date *ptjour = new Date;             // pointeur sur Date
    if (hier.initialiser (4, 11, 92))
        hier.afficher();                 // appel direct d'une méthode
    if(ptjour->initialiser (2,10,95))
        ptjour->afficher();              // appel d'une méthode à travers
}                                        // un pointeur
```

NB: Dans l'expression **void afficher() const** le mot clé const placé à cet endroit indique que cette méthode ne modifie pas les données de la classe. Si afficher tente de modifier jour, mois ou annee, l'indication const permet au compilateur de générer un message d'erreur.

3.2 Définitions des méthodes

Dans les exemples proposés ci-dessus, les méthodes (ou fonctions membres) sont déclarées dans la classe mais définies à l'extérieur de la classe.

Dans la définition de la méthode, le nom de la méthode est précédé par la nom de la classe et de **::** qui est appelé **opérateur d'accès**. Le nom d'une fonction membre préfixé par le nom de la classe à l'aide de l'opérateur :: est appelé **nom qualifié** (qualified name) de la fonction.

Il est également possible, de définir les méthodes à l'intérieur de la classe simultanément à la déclaration, de telles méthodes sont appelées **inline**.

Par exemple, en définissant inline la méthode afficher() de la classe Date cela donnerait :

```
class Date2 {
    int jour, mois, annee;
public:
    bool initialiser (int j, int m, int a);
    void lire_valeur (int & j, int & m, int & a);
    void afficher () const                    // methode inline
        { cout << jour << "/" << mois << "/" << annee; }
};
```

Les fonctions membres inline sont souvent utilisées par souci de concision des programmes mais il faut savoir qu'il existe des critères de choix pour savoir si une fonction membre doit ou non être inline.

Pratiquement les fonctions inline sont **dupliquées dans les instances** alors que les fonctions membre non inline sont uniques dans la classe et accessibles par les instances via des pointeurs. L'utilisation de fonctions inline réduit la lourdeur de l'appel ce qui améliore le temps d'exécution du programme mais pénalise l'encombrement en mémoire. Seules les petites fonctions, utilisées très souvent doivent être mises inline car on doit chercher un compromis entre place mémoire et temps d'exécution

Il existe des restrictions dus à certains compilateurs sur les fonctions inline, une fonction qui contient une boucle for, while, do while est souvent refusée comme inline.

L'utilisation de fonctions inline a aussi un impact sur la protection des algorithmes mis en jeu. En effet une fonction inline a une code source toujours visible alors qu'il est possible de compiler les fonctions non inline.

3.3 Masquage de l'implémentation

Afin de ne pas révéler à l'utilisateur d'une méthode d'un objet l'algorithme ou la méthode utilisé il est possible de fournir la classe sous la forme de deux fichiers : un fichier interface en clair qui permet de

savoir comment utiliser les instances et un fichier implémentation qui contient le résultat de compilation des méthodes.

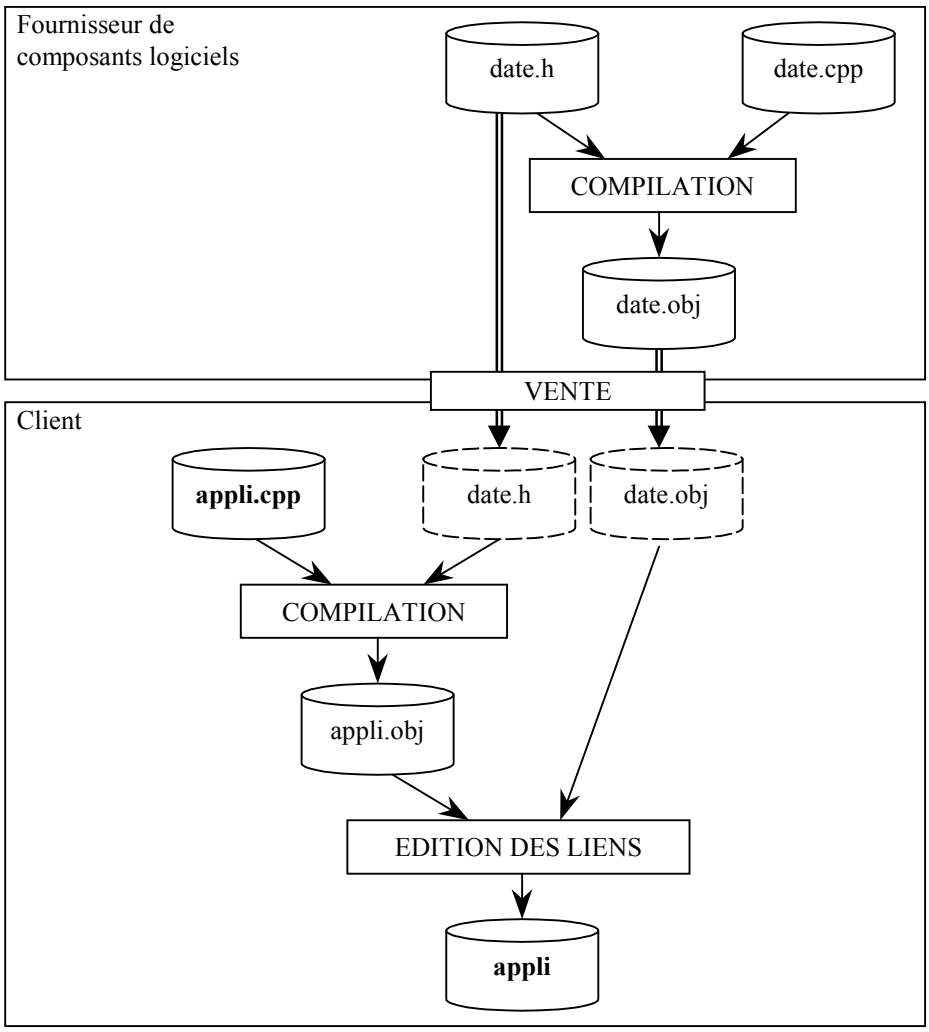
Si l'on reprend l'exemple de la classe Date vu plus haut on a alors deux fichiers.

- Le fichier **date.h** contient la partie déclaration de la classe date (interface de la classe) c'est à dire les données et les entêtes des méthodes.
- Le fichier **date.cpp** contient le code des fonctions membres de la classe date (implémentation de la classe)

Le fichier date.h est accessible à un utilisateur qui a ainsi une vue externe de la classe. Le fichier date.cpp est compilé et transmis en binaire à l'utilisateur qui n'a pas accès aux mécanismes de réalisation.

Pour utiliser des objets instances de la classe date on doit écrire un fichier appli.cpp qui inclue le fichier date.h pour que le type date soit connu, crée des variables de type date, les utilise en appelant les fonctions membres. La fichier appli.cpp doit être compilé puis être relié (linked en anglais) avec le fichier date.obj résultant de la compilation de date.cpp afin d'avoir accès au code des méthodes de la classe Date.

Schéma de production de programmes



Fichier date.h :

```
class Date
{
    int jour, mois, annee;
public:
    bool initialiser (int j, int m, int a);
    void lire_valeur (int & j, int & m, int & a);
    void afficher () const;
};
```

Fichier date.cpp :

```
#include "date.h"        // importation de la déclaration

bool Date::initialiser (int j, int m, int a)
{
    if (j < 1 && j > 31)  return false;
    jour = j;
    if ( m < 1 && m > 12) return false;
    mois = m;
    if (a < 0)  return false;
    annne = a;
    return true;
}

void Date::lire_valeur (int& j, int& m, int& a)
{
    j = jour; m = mois; a = annee;
}

void Date::afficher ()
{
    cout << jour << "/" << mois << "/" << annee;
}
```

Fichier appli.cpp

```
#include "date.h"              // importation de la déclaration

int main ()
{
    Date hier ;                        // variable Date
    Date *ptjour = new Date;           // pointeur sur Date
    if (hier.initialiser (4, 11, 92))
        hier.afficher();
    if(ptjour->initialiser (2,10,95))
        ptjour->afficher();
}
```

NB: il n'y a pas d'extension standard pour les noms des fichiers sources. L'extension ".cpp" est la plus répandue, mais on rencontre également ".C" ou ".cc", comme par exemple sur les machines Unix du CRI. De la même manière, selon les systèmes d'exploitation, les extensions pour les noms des fichiers objets peuvent être ".obj" (cas de Windows) ou ".o" (cas d'Unix).

3.4 Constructeurs et destructeurs d'un objet

3.4.1 Constructeur

Une instance d'une classe étant une variable comme une autre on désire pouvoir l'initialiser dans sa définition comme on le fait pour les types de base en C. Il est donc nécessaire d'avoir un mécanisme appelé à la création de l'instance pour réaliser les initialisations envisagées. Les initialisations ne concernent pas seulement les valeurs des attributs mais aussi tout traitement nécessaire à la création de l'instance. Si l'on a, par exemple, une classe communication représentant une liaison entre ordinateurs il peut être utile d'envoyer un bloc de protocole à l'ordinateur éloigné au moment de la création de la liaison.

Les fonctions d'initialisation sont réalisées par une méthode spéciale appelée **constructeur** et invoquée implicitement ou explicitement à la définition d'une instance.

3.4.1.1 Règles syntaxiques

Un constructeur est déclaré et défini de la même manière qu'une méthode ayant comme nom le nom de la classe et ayant un nombre quelconque d'arguments. Par contre, un constructeur n'a pas de type de retour (même pas void !). On peut avoir plusieurs constructeurs pour une classe (§ 3.4.3).

Un constructeur est appelé au moment de la définition d'une instance de la classe ou d'une allocation dynamique d'une instance de la classe. C'est le type et le nombre des paramètres qui permettent de sélectionner le constructeur ad hoc. Il y a deux syntaxes possibles (voir exemple ci-dessous)

3.4.1.2 Exemple d'utilisation de constructeur

On reprend la classe Date vue ci-avant en remplaçant la méthode initialiser() par un constructeur. On remarquera que dans cet exemple, la validité de la date entrée n'a pas été testée pour ne pas surcharger l'écriture.

Fichier date.h :

```
class Date
{
    int jour, mois, annee;
public:
    Date (int j, int m, int a);
    void lire_valeur (int & j, int & m, int & a);
    void afficher () const;
};
```

Fichier date.cpp :

```
#include "date.h"        // importation de la déclaration

Date::Date (int j, int m, int a)
{
    jour = j;
    mois = m;
    annne = a;
}

void Date::lire_valeur (int& j, int& m, int& a)
{
    j = jour; m = mois; a = annee;
}

void Date::afficher ()
{
    cout << jour << "/" << mois << "/" << annee;
}
```

Fichier appli.cpp :

```
#include "date.h"              // importation de la déclaration

int main ()
{
    // définitions avec appel du constructeur
    Date noel (25, 12, 2003);               // syntaxe 1
    Date aujourdhui = Date(10, 09, 2002);   // syntaxe 2

    // à rapprocher de l'initialisation des variables de type de base
    int x = 5;     // syntaxe classique
    int y (3);     // syntaxe similaire à l'appel d'un constructeur

    Date *hier = new Date (10, 09, 2002);   // pointeur sur Date

    Date demain;   // ERREUR DE COMPILATION
    . . . . .
}
```

NB: On initialise une variable de type Date à sa création aussi facilement qu'un int.

3.4.1.3 Constructeur par défaut

Il y a systématiquement appel d'un constructeur à la création d'une instance. Si on ne déclare **aucun** constructeur dans la classe, le compilateur en fournit un sans argument (**constructeur par défaut**) qui ne fait rien, c'est pourquoi le premier exemple de classe date sans constructeur est correct (§ 3.1.3)

Dès que l'on fournit un constructeur dans la classe, le compilateur **ne fournit plus** un constructeur par défaut sans argument c'est pourquoi on obtient une erreur de compilation dans l'exemple ci-dessus avec la définition de la variable demain, le compilateur n'ayant pas de constructeur sans argument à appeler.

Comme on l'a vu au paragraphe 2.4, il est possible d'indiquer des valeurs par défaut pour les paramètres d'une fonction, cette possibilité s'applique aussi aux méthodes et en particulier aux constructeurs. On peut donc facilement "retrouver" un constructeur sans paramètres en prévoyant une valeur par défaut pour tous les arguments.

3.4.1.4 Conctructeur utilisés comme des convertisseurs de type

Tout constructeur d'une classe C à un seul argument de type T peut être utilisé implicitement par C++ comme un convertisseur de T vers C. Ce mécanisme est mis en œuvre lors de l'appel d'une fonction avec comme argument effectif de type T alors que l'argument formel devait être une instance de C. Voir au paragraphe 5.3.6 un exemple d'utilisation.

3.4.2 Destructeur

En C et en C++, une variable locale a une durée de vie limitée à l'exécution du bloc dans lequel elle est définie. Que la variable soit d'un type de base ou instance d'une classe, elle disparaît au moment du franchissement de l'accolade fermante.

```
void f()
{
    int x;      // la variable x est créée à cet endroit
    Date d;     // la variable d est créée à cet endroit
    . . . . .
}               // d et x sont détruites à cet endroit
```

Au moment où une variable disparaît il peut être utile d'effectuer un traitement spécifique associé à la mort de la variable. Dans le cas des types de base le seul travail effectué est la restitution de la place mémoire, par contre dans le cas d'une donnée structurée le travail peut être complexe et spécifique (libération d'espace mémoire acquis dynamiquement par l'objet, fermeture des fichiers, avertissement à un ordinateur éloigné de la fermeture d'une liaison, etc.)

Dans le cas d'une instance d'une classe on dispose d'un mécanisme permettant d'effectuer automatiquement ce travail au moment de la disparition de l'instance : c'est le rôle d'une méthode spéciale appelée **destructeur**.

3.4.2.1 Règles syntaxiques

Le destructeur d'une classe a pour nom le nom de la classe précédé du caractère tilde (~). Un destructeur est unique pour une classe, et n'a aucun paramètre ni de type de retour.

3.4.2.2 Exemple d'utilisation de destructeur

Pour donner un exemple d'utilisation de destructeur on considère une nouvelle version de la classe Date dans laquelle on a deux attributs entiers (jour et an) et un attribut chaîne de caractères (mois).

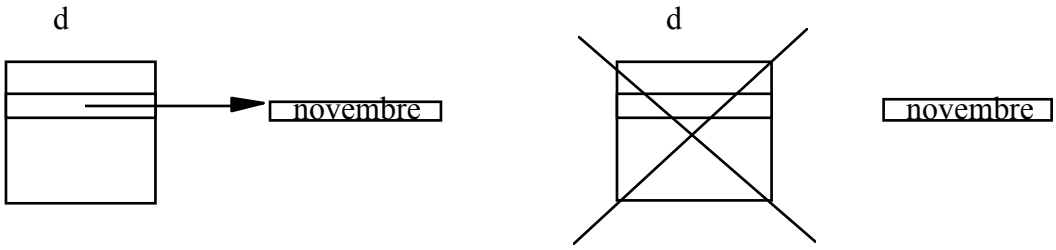
Remarque importante : cet exemple utilise les chaînes de caractères "classiques" du langage C, c-à-d des tableaux de caractères alloués dynamiquement (char *). Bien sûr, on pourrait utiliser la classe "string" de la STL qui a l'avantage d'encapsuler ce mécanisme d'allocation/libération de mémoire.

```
#include <string.h>

class date              // nouvelle version de la classe date
{
    int jour, an;
    char * mois;        // mois sous forme d'une chaîne de caractères
public:
    Date (int, char *, int);
    . . . . .
};
```

```
Date::Date (int j, char *m, int a)
{
    jour = j;
    an = a;
    mois = new char[strlen(m)+1];
    strcpy (mois, m);
}

int main()
{
    Date d (1, "novembre", 2020);
    . . . . .
}           // d disparaît ici
```

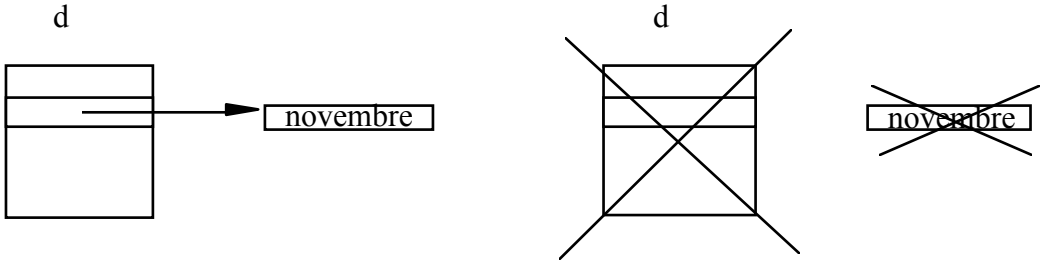
Dans le constructeur on affecte une valeur aux attributs entiers jour et an, on réserve dynamiquement une zone de mémoire pour stocker la chaîne m, puis on recopie la chaîne m dans la zone pointée par l'attribut mois.

A la disparition de la variable d la place de cette variable est bien libérée comme une structure classique mais la mémoire acquise dynamiquement reste réservée



Pour remédier à cette situation on fournit un destructeur qui libère la place acquise dynamiquement pendant la vie de l'instance.

```
#include <string.h>

class Date              // classe date avec destructeur
{
    int jour, an;
    char * mois;
public:
    Date (int, char*, int);
    ~Date ();
};

Date::Date (int j, char *m, int a)
{
    jour = j;
    an = a;
    mois = new char[strlen(m)+1];
    strcpy(mois,m);
}

Date::~Date ()          // destructeur
{
    delete [] mois;     // libère la zone pointée par mois
}
```



NB: un destructeur est nécessaire si l'objet a acquis de la mémoire dynamique (ici dans son constructeur).

Il est à noter que oublier un destructeur ne compromet pas directement le fonctionnement du programme à court terme. Cependant, la mémoire dynamique n'étant pas libérée, après un certain temps de fonctionnement, le programme n'aura plus de mémoire pour travailler. Ce genre de problème de "fuite de mémoire" est un des grands problèmes de la programmation et rares sont les programmes commerciaux qui en sont totalement exempts ! Pour les programmes à temps d'execution court, le problème ne se pose pas car le système d'exploitation libère la mémoire dynamique à chaque fin de programme.

3.4.3 Surcharge des constructeurs

Un constructeur, comme toute fonction, peut être surchargé pour permettre à un utilisateur de la classe d'initialiser une instance de plusieurs façons. Dans l'exemple ci-dessous une date peut être initialisée à partir de 2 entiers et d'une chaîne de caractères, à partir d'une entier et d'une chaîne de caractères, à partir d'un entier ou à partir de rien. Le choix d'un constructeur par le compilateur s'effectue en fonctions des paramètres figurant dans la définition de l'instance.

```
class Date
{
    int jour, an;
    char * mois;
public:
    Date (int, char *, int);    // jour, mois, an
    Date (int, char *);         // jour, mois de l'année en cours
    Date (int);                 // jour du mois et de l'année en cours
    Date ();                    // date courante
};
. . . . .
```

```
int main ()
{
    Date noel (25, "decembre", 2003);
    Date bientot (14, "decembre");
    Date debutmois (1);
    Date aujourdhui;
}
```

NB1: noel est initialisé par le premier constructeur Date (int, char *, int)

NB2: bientot est initialisé par le deuxième constructeur Date(int, char *)

NB3: debutmois est initialisé par le troisième constructeur Date (int)

NB4: aujourdhui est initialisé par le quatrième constructeur qui peut récupérer la date système.

3.4.4 Construction par copie d'objet

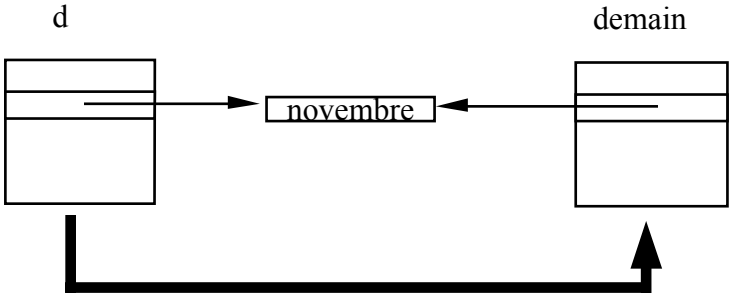
Il est possible de définir une instance d'une classe en l'initialisant à partir d'une autre instance de la même classe.

```
classe Date
{
    Date (int, char *, int);
    . . . . .
};
. . . . .
Date d (1, "novembre", 2003);
Date demain = d;          // appel du constructeur par copie
Date un_jour(d);          // appel du constructeur par copie
. . . . .
```

Ici on définit d, instance de date initialisée à partir de 2 entiers et d'une chaîne, par appel du constructeur date(int, char *, int). On définit ensuite l'instance demain identique à d, puis l'instance un_jour identique à d. Les deux dernières notations produisent le même effet qui consiste à appeler un constructeur appelé **constructeur par copie**, ayant comme argument une Date. Ce constructeur Date(Date) peut être soit fourni par le programmeur de la classe date, soit fourni par défaut par le compilateur.

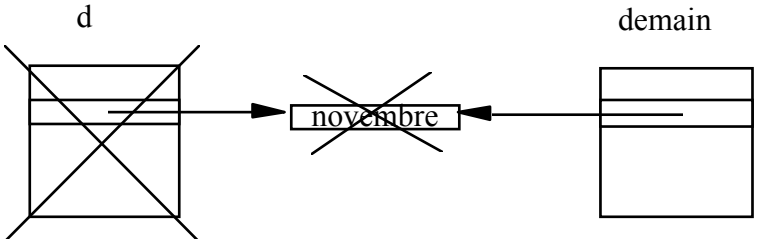
3.4.4.1 Constructeur par copie fourni par défaut

Si le programmeur de la classe ne fournit pas de constructeur par copie, le système en fournit un par défaut qui se contente de copier la valeur de tous les attributs. Cela peut entraîner des problèmes dans le cas ou l'instance contient des pointeurs sur des zones réservées dynamiquement, comme le montre l'exemple suivant.



Le constructeur Date (int, char*, int), appelé pour initialiser la variable d, réserve dynamiquement un zone pour stocker la chaîne "novembre". Le constructeur par copie fourni par défaut est appelé pour initialiser la variable demain : il produit une copie bit à bit de d. Les instances demain et d partagent donc la même zone mémoire pour la chaîne "novembre".

Si d est détruit, la chaîne "novembre" est libérée par le destructeur et demain pointe sur une zone mémoire qui ne lui appartient pas et qui risque d'être réallouée par le système lors des prochains appels à **new**.



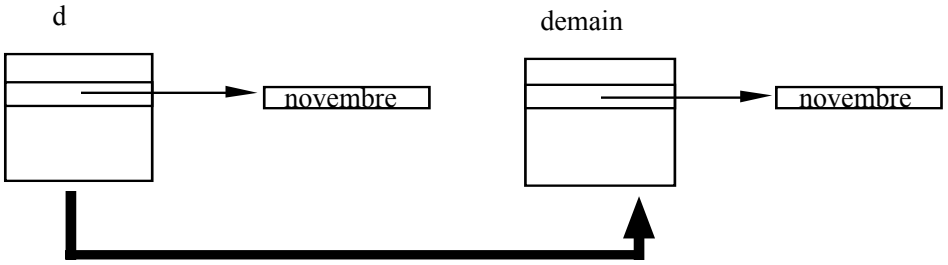
3.4.4.2 Ecriture d'un constructeur par copie

```
class Date
{
    int jour, an;
    char *mois;
public:
    Date (int, char *, int);    // constructeur
    . . . . .
    Date (const Date &);        // constructeur par copie
};

Date::Date (const date & autre_date)
{
    jour = autre_date.jour;
    an = autre_date.an;
    mois = new char[strlen(autre_date.mois)+1];
    strcpy (mois, autre_date.mois);
}
```

```
Date d("25 novembre 1994");
Date demain = d;                        // appel constructeur par copie
```

NB1: La syntaxe Date::Date (const date & autredate) signifie "le constructeur Date de la classe Date avec comme argument autredate qui est une référence de Date intangible"

NB2: autredate est une référence ce qui évite un passage par valeur et donc une duplication de la variable mais le constructeur n'a pas le droit de modifier autredate à cause du const. De toute façon, on ne pourrait pas écrire Date::Date (Date autredate) car autredate étant passé par valeur, il y aurait appel implicite du constructeur pas copie... ce qui conduirait à un appel récursif infini !!



Dans le constructeur par copie on donne à jour et à an les valeurs extraites de autredate, puis on réserve de la place dynamiquement pour dupliquer la chaîne pointée par mois. Si la variable d est détruite son destructeur détruit la chaîne pointée par mois mais la chaîne pointée par l'attribut mois de la variable demain reste intacte.

On remarquera que comme le constructeur par copie est surchargé, on ne peut plus compter sur le constructeur par copie fourni par défaut et il faut donc recopier tous les attributs explicitement même ceux qui ne posaient pas de problèmes particuliers comme jour et mois.

3.4.4.3 Cas d'appel d'un constructeur par copie

Un constructeur par copie est appelé dans les cas suivants :

- création d'une instance copie d'une autre

```
Date hier = maintenant;         // appel constructeur par copie
Date hier (maintenant);         // appel constructeur par copie
```

- passage de paramètre par valeur à une fonction

```
int f(Date d)       int main()
{                   {
. . . . .               Date hier (1, "janvier", 1970);
. . . . .               . . . . .
. . . . .               y = f (hier);   // appel constructeur par copie
}                   }
```

- retour d'une valeur d'une fonction

```
Date g(int x, int y, int z)
{
    Date d;
    . . . . .
    return  d;      // appel du constructeur par copie (indispensable
}                   // car la variable locale d est détruite ici !)
```

3.4.5 Cas des tableaux d'objets

On peut utiliser des tableaux d'objets de la même manière que des tableaux d'éléments de types de base.

3.4.5.1 Définition d'un tableau

```
int t[10];          // définition d'un tableau de 10 int
Date td[10];        // définition d'un tableau de 10 dates
int * pt;           // définition  d'un pointeur sur int
date * ptd;         // définition d'un  pointeur sur date
pt = new int[20];   // réservation dynamique d'un tableau de 20 int
ptd = new Date[20]; // réservation dynamique d'un tableau de 20 dates
```

```
t[i] = 5;       // accès à l'élément i du tableau t
pt[i] = 6;      // accès à l'élément i du tableau dynamique pointé
                // par pt
td[i] = d1;     // accès à l'élément i du tableau td
ptd[i] = d2;    // accès à l'élément i du tableau dynamique pointé par ptd
```

NB1: on remarque que l'accès à un élément d'un tableau se note toujours de la même manière même lorsque le tableau est alloué dynamiquement (pas de déréférencement explicite du pointeur). C'est une "bizarrerie" héritée du langage C.

NB2: l'affectation de dates ci-dessus peut nécessiter la surcharge de l'opérateur = pour la classe Date (voir § 5.3.1)

3.4.5.2 Appel de constructeurs pour un tableau d'objets

```
Date t[2] = { Date(1,2,89), Date(2,5,95) };
Date u[3] = { Date(1,2,89), Date(2,5,89) };
```

NB1: Le premier exemple définit un tableau de 2 dates initialisé par 2 dates construites en appelant un constructeur de la forme Date(int, int, int) qui doit donc exister.

NB2: Le deuxième exemple définit un tableau de 3 dates initialisé par 2 dates construites avec un constructeur de la forme Date(int, int, int), on doit donc avoir ce constructeur mais on doit disposer aussi d'un constructeur sans argument Date() pour construire le troisième élément du tableau. Faute de disposer de ce constructeur le compilateur indique une erreur.

3.4.5.3 Appel des destructeurs pour un tableau d'objets

Un tableau local est entièrement détruit à la sortie du bloc dans lequel il a été défini, un tableau global (ou static) est détruit à la fin de l'exécution du programme. Un tableau réservé dynamiquement doit être libéré par le programmeur à l'aide de **delete[]**. La libération du tableau appelle le destructeur de chacun des éléments du tableau.

3.5 Attribut caché this

Toutes les instances de toutes les classes possèdent un attribut caché du nom de **this**, cet attribut n'est pas défini par le programmeur mais fourni automatiquement par le compilateur. this est un pointeur sur l'instance en cours de la classe c'est à dire qu'il contient l'adresse de l'instance L'intérêt de this c'est qu'une méthode peut utiliser sa valeur pour déterminer où se trouve l'instance manipulée.

Pour un exemple de l'usage de this, voir le § 5.3.1, sur la surcharge de l'opérateur '='.

3.6 L'opérateur d'accès '::'

L'opérateur d'accès permet de désigner une méthode ou une donnée d'une classe

nom_classe::nom_fonction(. . . .)

nom_classe::nom_attribut

Il permet de désigner d'une manière non ambiguë un élément d'une classe dans le cas de classe imbriquées (classes définies à l'intérieur d'une classe, voir §7.3) et de résoudre des conflits de noms de variables comme le montre l'exemple suivant

```
int mois;               // variable globale

class Date {
    int jour, mois, an;
public:
    Date (int, int, int);
    . . . . .
};

Date::Date (int j, int mois, int a)
{
    jour = j; an = a;
    Date::mois = mois;
    ::mois = mois;
}
```

On a ici une ambiguïté entre 3 données de même nom, toutes dans la portée du constructeur Date. En effet Date voit une variable globale de nom mois, un attribut de sa classe de nom mois et un argument

de nom mois. L'ambiguïté est levée en utilisant des noms complets (qualified names) de la manière suivante :

- Date::mois désigne le membre mois de la classe Date
- mois désigne le paramètre d'appel du constructeur.
- ::mois désigne la variable globale mois

L'opérateur d'accès est également utilisé par les **namespaces**. Par exemple lorsqu'on utilise la STL (Standard Template Library), toutes les classes sont définies dans le namespace **std**. Pour déclarer une liste de Truc, il faut donc écrire :

```
#include <list>
class Truc {  . . . . . };

int main (void)
{
    std::list<Truc> maliste;
    . . . . .
}
```

Cependant, pour simplifier l'écriture, on peut placer en début de fichier la directive **using namespace std**, et on peut écrire plus simplement :

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

class Truc {  . . . . . };

int main (void)
{
    list<Truc> maliste;
    . . . . .
}
```

3.7 Fonctions, classes et méthodes amies

3.7.1 Note sur méthodes et fonctions classiques du C

Comme C++ est une extension du langage C, il peut coexister des fonctions membres de classe (ou méthodes) et des fonctions classiques C n'appartenant à aucune classe.

```
class Date
{
    int jour, mois, annee;
public:
    Date (int, int, int);
    . . . . .
    void afficher () const;
};

Date::Date (int j, int m, int a)
{    jour = j; an = a; m = mois;  }

void Date::afficher ()  const
{    cout << jour << "/" << mois << "/" << annee; }

int somme (int x, int y)
{     return x + y;  }

int main()
{
    int a = 2, b = 5, d;
    Date hier (2,3,1995);
    d = somme (a,b);
    hier.afficher();
}
```

NB1: somme est une fonction C classique.

NB2: afficher est une fonction membre qui ne peut être appelée qu'à travers une instance.

3.7.2 Les fonctions amies

Un fonction amie d'une classe est une fonction C classique mais ayant accès à la partie privée de la classe.

```
class Truc                      // déclaration de la classe Truc
{
    int x;
public:
    Truc(int);
    void afficher() const;            // fonction membre afficher
    friend  int f(const Truc &);      // fonction amie f
};

void Truc::afficher() const     // définition fonction membre afficher
{
    cout << x << endl;
}
```

```
int f(const Truc & t)    // définition fonction  externe f
{
    return t.x;
}                        // autorisé car f est amie de Truc

int main()         // fonction main
{
    int y;
    Truc a(5);
    a.afficher();        // appel fonction membre
    y = f(a);            // appel fonction externe
}
```

NB: f accède à la partie privée x de la variable a instance de la classe truc. Sans la déclaration "friend", cette fonction aurait provoqué une erreur à la compilation.

3.7.3 Exemple d'utilisation des fonctions amies

On dispose d'une classe Vecteur4 correspondant à un vecteur à 4 composantes int et d'une classe Matrice4 correspondant à une matrice d'int de dimension 4x4. Les éléments du vecteur et de la matrice sont dans les zones privées des classes et l'on dispose de fonctions membres publiques elem() pour accéder aux éléments.

```
class Vecteur4
{
    int v[4];
public:
    int elem (int i);           // accès à l'élément d'indice i
};

class Matrice4
{
    int m[4][4];
public:
    int elem (int i, int j);    // accès à l'élément d'indices i,j
};

int Vecteur4::elem (int i)
{    return v[i]; }

int Matrice4::elem (int i, int j)
{    return m[i][j]; }
```

On veut écrire une fonction compare qui vérifie si une ligne d'une instance de la classe Matrice4 est identique à une instance de la classe Vecteur4. La fonction compare est externe, elle n'appartient ni à la classe Vecteur4 ni à la classe Matrice4.

```
bool compare (Matrice4 mat, Vecteur4 vect)
{
    bool trouve = false;
    for (int i=0; (i<4) && (!trouve); i++)
    {
        bool concorde = true;
        for (int j=0; (j<4) && (concorde); j++)
            concorde = (vect.elem(j) == mat.elem(i,j));
        trouve = concorde;
    }
    return trouve;
}
```

La fonction compare fait de très nombreux appels aux fonctions membres elem() de Vecteur4 et de Matrice4 uniquement pour atteindre les parties privées de ces classes, il y a donc du temps perdu en appels et retours de fonctions Une simplification peut être introduite en rendant publiques les éléments de Vecteur4 et de Matrice4 mais on perd l'intérêt de la protection.

Il faudrait pouvoir écrire une fonction compare qui soit membre de Vecteur4 et membre de Matrice4 pour quelle ait accès aux zones privées, ceci est impossible car une fonction ne peut pas être membre de 2 classes. Par contre la fonction compare peut être amie des classes Vecteur4 et Matrice4.

```
class Matrice4;

class Vecteur4
{
    int v[4];
public:
    int elem (int i);           // accès à l'élément d'indice i
    friend bool compare (Matrice4, Vecteur4);
};

class Matrice4
{
    int m[4][4];
public:
    int elem (int i, int j);    // accès à l'élément d'indices i,j
    friend bool compare (Matrice4, Vecteur4);
};

bool compare (Matrice4 mat, Vecteur4 vect)
{
    bool trouve = false;
    for (int i=0; (i<4) && (!trouve); i++)
    {
        bool concorde = true;
        for (int j=0; (j<4) && (concorde); j++)
            concorde = (vect.v[j] == mat.m[i][j]);
        trouve = concorde;
    }
return trouve;
}
```

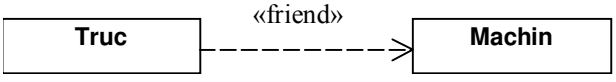
NB: On utilise une prédéclaration de la classe Matrice4 car elle doit être connue avant la classe Vecteur4 pour pouvoir écrire le prototype de la fonction amie compare.

Le prototype d'une fonction amie d'une classe est annoncé dans la classe par le mot clé **friend**, la définition de la fonction amie n'est pas différente de celle d'une fonction externe classique.

Dans la fonction compare on a maintenant un accès direct aux éléments des zones privés sans passer par des fonctions d'accès.

3.7.4 Classe amie

Une classe peut être déclarée amie d'une autre classe, ce qui permet à toutes les fonctions membres de la première classe d'avoir accès à la zone privée de la deuxième.

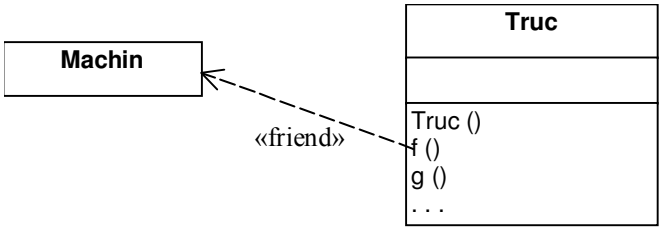


```
class Machin
{
private:
    . . . . .
public:
    . . . . .
friend Truc;       // la classe truc est amie de la classe machin
    . . . . .
};
```

Dans cet exemple toutes les méthodes de la classe Truc peuvent accéder librement à la partie privée (attributs et méthodes) de la classe Machin.

3.7.5 Fonction membre amie

Une fonction membre d'une classe peut être déclarée comme amie d'une autre classe, la fonction a alors accès à la zone privée de la classe dont elle est amie.




```
class Machin
{
    . . . . .
public:
    . . . . .
    friend Truc::f (int);  // la méthode f de la classe Truc est amie
    . . . . .
};
```

Chapitre 4
Réutilisation de classes

Quand on a construit et testé une classe on est sûr qu'elle marche on peut donc la réutiliser pour construire d'autre classes plus complexes. Il existe deux manières de réutiliser une classe, soit en l'utilisant comme partie d'une nouvelle classe, soit comme base pour développer une nouvelle classe.

4.1 Composition de classes

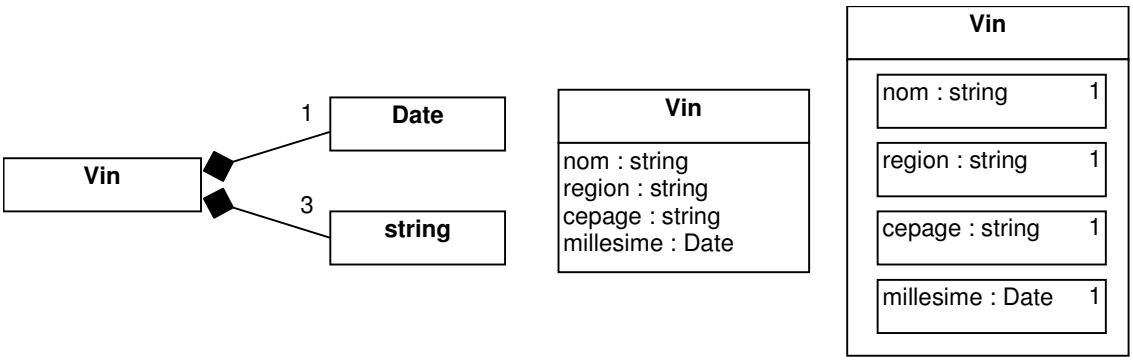
Dans une classe, un membre peut être soit une fonction (méthode), soit une donnée d'un type de base (attribut), soit une instance d'une autre classe (attribut), c'est ce dernier cas qui est utilisé dans la composition de classes

Une classe A construite avec un membre instance d'une classe B, définit une relation **"partie de"** entre B et A, B est une partie de A. On peut ainsi définir des concepts d'un certain niveau, les "matérialiser" sous forme de classes puis définir des concepts de niveau supérieur en composant les concepts du premier niveau.

```
class Date
{
    . . . . .
};

class Vin
{
    string nom;          // membre instance de string
    string region;       // membre instance de string
    string cepage;       // membre instance de string
    Date millesime;      // membre instance de Date
public:
    . . . . .
};
```

Ici la classe vin contient des attributs instances de la classe de bibliothèque string et un attribut millésime instance de la classe date définie plus haut.

En UML, il existe 3 représentations possibles de la relation de composition, voir figure ci-dessous.



Les relations entre les fonctions membres des classe initiales et des classes composées sont mises en évidence par l'exemple suivant.

```
class Composant
{
    int x;
public:
    Composant (int y) {  x = y; }     // constructeur de composant
    void ecrit() const {  cout << x << endl;  }
};
```

```
class Composee
{
    float z;
    Composant u;                // membre instance de composant
    Composant v;                // membre instance de composant
public:
    Composee (float, int, int); // constructeur de composé
    void ecrit() const;
};
```

La classe composant ne présente aucune particularité liée au fait qu'elle sera ou non utilisée dans une autre classe.

4.1.1 Constructeur d'une classe composée

Le constructeur de la classe composée **appelle systématiquement** les **constructeurs** de chacun de ses **composants** avant d'exécuter le code situé entre les accolades. Comme il ne "sait" pas à quel constructeur s'adresser, il appelle un constructeur de composant sans argument. Si ce constructeur existe il n'y a pas de problème sinon il y a erreur à la compilation. Pour éviter une telle erreur il faut soit avoir des **constructeurs sans arguments pour toutes les classes** pour préparer leur réutilisation éventuelle, soit indiquer dans le constructeur de la classe composée qu'il faut appeler d'autres constructeurs que les constructeurs sans argument.

```
Composee::Composee (float a1, int a2, int a3): z(a1), u(a1), v(a2)
{
    . . . . .     // constructeur de composée
}
```

Dans le constructeur de composée les deux points introduisent un séquence d'appel des constructeurs des composants (le constructeur du composant u avec comme argument a2, puis le constructeur du composant v avec comme argument a3) avant le corps du constructeur entre accolades.

Les attributs correspondant à des types de base disposent de constructeurs prédéfinis ce qui permet de les initialiser comme les attributs instances d'autres classes (cas de "z(a1)" dans l'exemple ci-dessus).

4.1.2 Destructeur d'une classe composée

Les destructeurs des membres composants sont appelés automatiquement par le destructeur de la classe composée ce qui permet de détruire les composants avant de détruire le composé.

4.1.3 Utilisation des fonctions membres d'une classe composant

Une fonction membre de la classe composée qui doit effectuer un travail sur un de ses membres instance d'une classe composant, peut appeler une fonction membre de classe composant par l'intermédiaire d'un membre concerné.

```
void Composee::ecrit() const    // fonction membre ecrit
{
cout << z << endl;
u.ecrit();                      // appel de Composant::ecrit()
v.ecrit();                      // appel de Composant::ecrit()
}
```

La fonction "ecrit" de la classe composée appelle la fonction "ecrit" de la classe composant par l'intermédiaire des instances u et v.

4.2 Héritage de classe

Quand une classe a été testée et validée on peut la réutiliser telle quelle dans une autre classe par composition. Un autre cas de réutilisation à envisager est l'extension ou l'adaptation d'une classe existante pour traiter un problème analogue mais légèrement différent. Une classe testée et utilisée est généralement compilée ce qui interdit toute modification et garantit que les applications qui l'utilisent pourront continuer à l'utiliser sans problème. Une telle classe constitue de ce point de vue un **logiciel fermé**.

Tout logiciel est amené à évoluer pour s'adapter à de nouvelles situations et prendre en compte de nouvelles applications, de ce point de vue il doit être **ouvert**.

Cette contradiction du logiciel à la fois ouvert et fermé est impossible à résoudre proprement avec des fonctions ou des procédures. En effet une fonction est fermée si on la conserve sous forme compilée, par ailleurs on doit garder le code source pour la maintenir ouverte, on est conduit à avoir plusieurs formes pour le même logiciel puis plusieurs versions au fur et à mesure que l'on fait des modifications sur la fonction. On aboutit à un ensemble complexe de versions très difficile à maintenir.

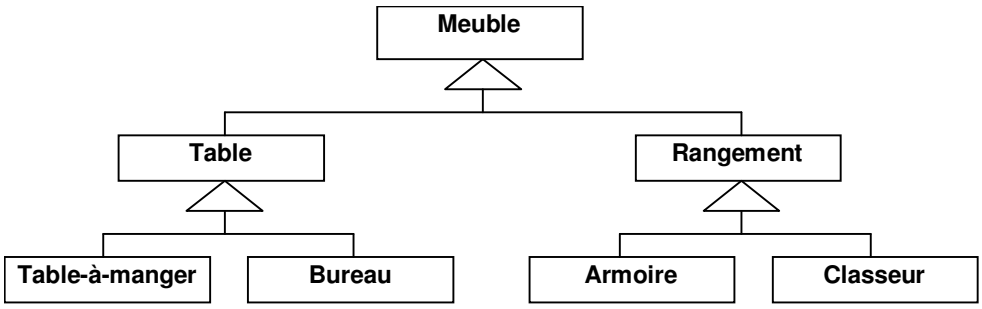
Une classe compilée est fermée, elle est donc stable pour toutes les applications qui l'utilisent. par contre il est possible, même **sans avoir le code source** de définir une nouvelle classe héritière de cette classe en indiquant seulement les différences entre la classe mère et la classe héritière. On a alors du logiciel à la fois **fermé et ouvert** puisqu'il est stable et évolutif. Le travail de réalisation des nouvelle classe est faible puisqu'on conserve l'acquis des classes existantes.

4.2.1 Héritage simple

Si on doit réaliser une classe voisine d'une classe existante, on ne définit pas complètement une nouvelle classe qui aurait une grande partie commune avec la première, on crée une classe dérivée (héritière) de la première (classe mère) en indiquant seulement les différences entre classe de base et classe dérivée

La classe dérivée hérite des membres de la classe mère, elle apparaît donc comme une copie de cette classe mère. La classe dérivée a ses propres membres, attributs et méthodes ce qui permet d'introduire les différences entre elle et la classe mère. Un objet instance de la classe dérivée a donc accès à ses membres propres ainsi qu'à ceux de la classe mère.

La définition d'une classe B dérivée d'une classe A représente la relation **"sorte-de"** entre B et A, B est une sorte de A, B est une spécialisation de A. Si toutes les propriétés de A sont vérifiées par B et que B n'introduit que des ajouts par rapport à A, on dit que B est ou **sous type** de A.

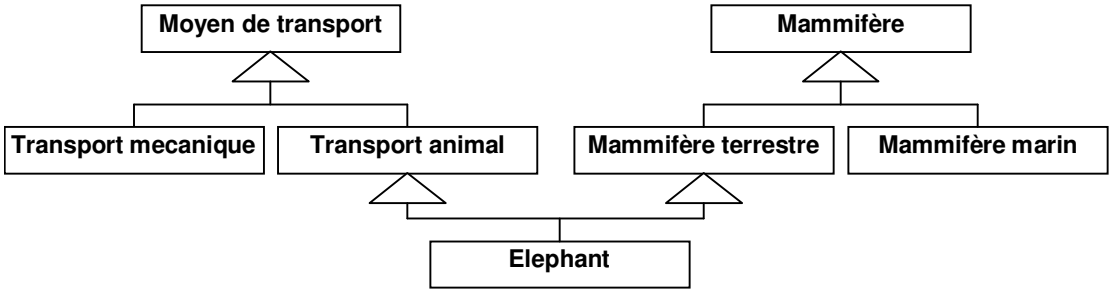


L'utilisation répétée de l'héritage de classe permet de constituer une hiérarchie de classes représentant une relation de spécialisation entre ces classes. L'héritage est transitif, les classes héritent des membres de tous leurs ancêtres.

Le mécanisme d'accès aux membres d'une classe fonctionne par recherche en **largeur d'abord**. Quand un membre d'une classe est invoqué, il y a d'abord recherche dans la classe de ce membre, s'il n'est pas trouvé il y a recherche dans la ou les classes mères, puis dans les classes grand mères, et ainsi de suite.

4.2.2 Héritage multiple

C++ permet de définir une classe dérivée de plusieurs classes (**héritage multiple**) ce qui permet de prendre en compte différents points de vue pour définir un concept à l'aide d'une classe.

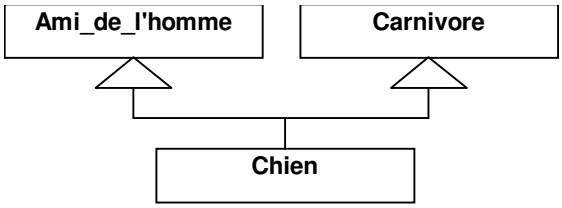


Du point de vue zoologique un éléphant est une sorte de mammifère terrestre, du point de vue utilisation c'est une sorte de moyen de transport animal.

4.2.2.1 Problème de contradiction sémantique

L'héritage multiple peut introduire des conflits avec des valeurs contradictoires de certains attributs et des problème de choix de méthodes quand plusieurs méthodes de même nom existent chez les ancêtres d'une classe.

Comme Ami_de_l'homme, le Chien peut hériter d'une fonction ayant un comportement amical vis-à-vis d'un Homme, comme Carnivore le Chien peut hériter d'une fonction ayant un comportement agressif vis à vis d'un homme. Le langage ne peut pas résoudre de telles ambiguïtés qui existent au départ dans le monde à modéliser, c'est le programmeur qui doit résoudre ces problèmes.



L'ambiguïté entre plusieurs générations est résolue par le mécanisme d'héritage en largeur d'abord. L'ambiguïté au sein d'une même génération doit être résolue explicitement par le programmeur qui doit définir un membre adéquat dans la classe concernée à l'aide d'un nom complet.

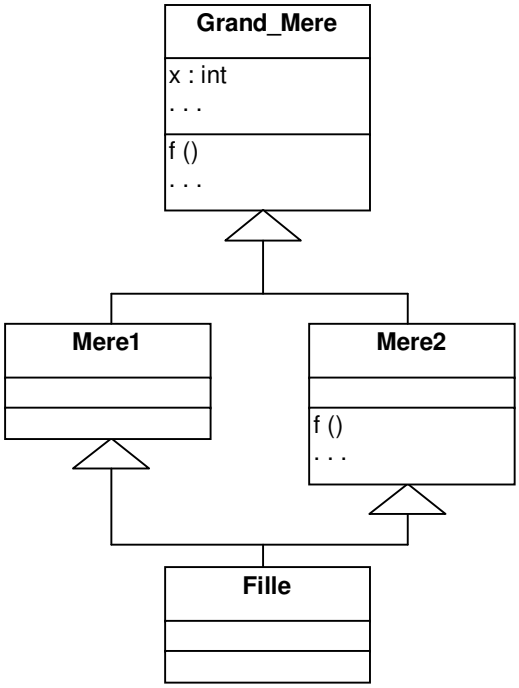
4.2.2.2 Héritage en diamant

Un configuration d'héritage en losange (diamond) introduit mécaniquement des problèmes de duplication et d'ambiguïté.

Mere1 et Mere2, héritant de Grand_Mere comportent chacune l'attribut x et la méthode f(). Fille héritant de Mere1 et Mere2, contient 2 exemplaires de l'attribut x et deux méthodes f().

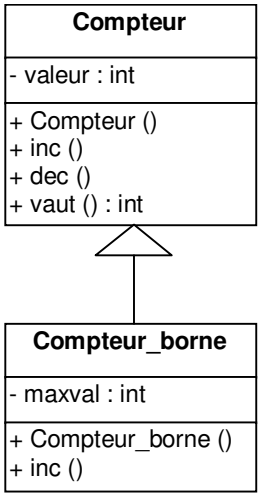
Comme f() est surchargée par Mere2, les 2 fonctions f() sont différentes. Lors d'un appel "standard" à la fonction f() on ne sait pas qu'elle la version utilisée. Lors d'une modification de l'attribut x dans la classe Fille on ne sait pas quel est l'exemplaire utilisé.

Les ambiguïtés peuvent être levées en utilisant des noms qualifiés (Mere1::x et Mere2::x).



4.2.3 Exemple d'héritage

On définit la classe compteur avec un attribut valeur représentant la valeur du compteur et des méthodes pour initialiser, incrémenter, décrémenter et connaître la valeur du compteur.




```
class Compteur
{
    int valeur;
public:
    Compteur() {  valeur = 0; }
    void inc() {  ++valeur; }
    void dec() {  if (valeur>0) --valeur; }
    int vaut() {  return valeur; }
};
```

On définit ensuite une nouvelle classe pour représenter un compteur dont la valeur maximale est paramétrée. Un compteur_bornéé est une "sorte de" compteur avec quelques particularités en plus (attribut maxval et gestion d'une valeur max).

```
class Compteur_borne : public Compteur
{
    int maxval;
public:
    Compteur_borne (int max) : Compteur() {  maxval = max; }
    void inc()  {  if (vaut()<maxval) Compteur::inc(); }
};
```

En pratique :

- Tous les attributs (valeur) et toutes les méthodes (inc, dec, vaut) de Compteur sont héritées par Compteur_borme.
- Dans la classe Compteur_borne on ajoute un attribut (maxval) et une méthode (inc).
- L'attribut valeur de Compteur étant privé, il n'est pas accessible directement par Compteur_borne (voir détails plus loin). C'est pourquoi on appelle Compteur::inc() dans la méthode inc() de Compteur_borne et que l'on ne fait pas ++valeur directement.
- La méthode inc() de Compteur_borne surcharge la méthode inc() de Compteur. Elle n'est donc plus directement accessible ce qui explique la forme complète Compteur::inc() dans la méthode inc() de Compteur_borne.

4.2.4 Héritage et accessibilité

Les constructeurs, le destructeur, et l'opérateur = de la classe de base ne sont pas hérités par la classe dérivée.

Dans une classe on a considéré jusqu'à présent 2 zones (privée et publique), il y a en fait 3 possibilités (publique, protégée, privée) :

- la zone privée est celle qui existe par défaut, elle est introduite par **private:**,
- la zone publique est introduite par **public:**
- la zone protégée est introduite par le mot clé **protected**:

```
class Truc
{
private:
    int x;              // attribut x privé
protected:
    int y;              // attribut y protégé
public:
    int z;              // attribut z public
    void f() { . . . . . }
};
Truc u;                 // u instance de truc
```

Un élément de la zone **private** est :

- inaccessible par une instance `u.x = 35;  // ERREUR`
- accessible par une fonction membre
- inaccessible par héritage

Un élément de la zone **protected** est :

- inaccessible par une instance `u.y = 45;  // ERREUR`
- accessible par une fonction membre
- potentiellement accessible par héritage

Un élément de la zone **public** est :

- accessible par une instance `u.z = 55;  // OK`
- accessible par une fonction membre
- potentiellement accessible par héritage

4.2.5 Modes d'héritage

L'accessibilité par héritage est liée au mode d'héritage que l'on présente ci-dessous, selon le mode d'héritage une donnée publique ou protégée sera ou non accessible par les classes héritières.

Les trois modes d'héritage possibles sont héritage privé, héritage protégé et héritage public respectivement indiqués par les mots clés **private**, **protected**, **public.**

```
class Mere
{ . . . . . };

class Derivee1 : private Mere          // héritage privé
{ . . . . . };

class Derivee2 : protected Mere        // héritage protégé
{ . . . . . };

class Derivee3 : public Mere           // héritage publique
{ . . . . . };
```

Le tableau ci-dessous donne le résultat de **l'accessibilité des zones dans la classe dérivée** :

		Zones de la classe de base		
		Zone *private*	Zone *protected*	Zone *public*
Modes d'héritage	Héritage *public*	inaccessible	protected	public
	Héritage *protected*	inaccessible	protected	protected
	Héritage *private*	inaccessible	private	private

Dans les héritages autres que public une classe dérivée n'a pas les mêmes droits sur les membres de ses classes ancêtres que les classes ancêtres elles mêmes. On ne peut donc pas dire qu'une classe dérivée est une copie complétée de la classe mère. Seul le mode d'héritage public assure une stabilité des droits d'accès dans la transitivité de l'héritage.

NB1: Malgré l'existence de ces 3 modes d'héritage en C++, le mode public est de loin le plus utilisé et les deux autres modes sont à réserver à des cas très particuliers.

NB2: Par défaut l'héritage est privé!

L'exemple ci-dessous présente le mode d'héritage le plus couramment utilisé :

```
class Mere
{
protected:
    int x;
public:
    vaut() {  return x; }
    . . . . .
};
class Derivee : public Mere
{
public:
    affiche() {  cout << x; }
};
int main()
{
    Mere A;
    Derivee B;
    B.affiche();         // affiche() a accès à x de la classe Mere
    cout << A.x;         // ERREUR COMPILATION
    cout << B.x;         // ERREUR COMPILATION
}
```

Pratiquement on utilise peu la zone private, on la remplace par la zone protected ce qui ne change rien à l'intérieur de la classe mais conserve l'accessibilité pour les classes descendantes. Les héritages s'effectuent en mode public ce qui assure la stabilité des droits d'accès. Toutefois il peut être intéressant d'utiliser une zone private quand on veut contrôler définitivement l'accès à des informations quelque soit l'héritage que l'on puisse faire.

4.2.6 Modifications par rapport à une classe de base

Une classe dérivée est définie relativement à une classe de base en introduisant trois types de différences:

- l'ajout d'attributs ou de fonctions membres,
- la modification d'attributs ou de fonctions membres,
- la suppression d'attributs ou de fonctions membres.

L'**ajout** de nouveaux membres s'effectue en définissant ces membres dans la classe dérivée, cette dernière a donc ses propres membres plus ceux des classes ancêtres.

La **modification** de fonctions membres s'effectue en redéfinissant dans la classe dérivée des fonctions déjà existantes dans les classes ancêtres. Lors de l'accès à une fonction c'est celle située dans la classe de l'instance concernée qui sera invoquée sans mettre en œuvre l'héritage.

```
class Mere
{
    . . . . .
public:
    void f0(){. . .}
    void f1(){. . .}
};
class Derivee : public Dere
{

public:
    void f2() { . . . }
    void f1() { . . . }
};
```

```
int main()
{
    Mere A;
    Derivee B;
    A.f0();              // appel de f0 de mere
    B.f0();              // appel de f0 de mere
    A.f1();              // appel de f1 de mere
    B.f1();              // appel de f1 de derivee (modification)
    B.f2();              // appel de f2 de derivee (ajout)
    A.f2();              // ERREUR COMPILATION
}
```

NB: La classe Mere possède deux méthodes f0 et f1, la classe Derivee définit une méthode f1 qui masque la fonction f1 de mère et une fonction f2 qui représente un ajout. Pour une instance de la classe Derivee les fonctions Mere::f0, Derivee::f1 et Derivee::f2 sont accessibles.

La **suppression** de fonctions membres s'effectue indirectement en définissant un **héritage privé** à partir de la classe de base ce qui **masque** toutes les fonctions de la classe de base puis en **exportant explicitement** les fonctions qui doivent être conservées dans la classe dérivée.

```
class Mere
{
    int i,j;
public:
    Mere (int ii=0, int jj=0)         // constructeur Mere(int,int)
        { i = ii; j = jj; }
    void init (int val) { i = j = val; }     // Mere::init(int)
    void annuler() { i = j = 0; }            // Mere::annuler()
};
class Derivee : Mere
{
public:
    Derivee (int x, int y) : Mere(x,y) {}    // constructeur de Derivee
    void init (int val)               // Derivee::init(int)
        { Mere::init (val); }         // appel explicite à Mere::init
};
```

```
int main()
{
    Mere A (0, 1);
    Derivee B(2, 3);
    A.init(1);
    A.annuler();
    B.init(2);
    B.annuler();   // ERREUR COMPILATION, annuler inaccessible par B
}
```

NB1: Par défaut l'héritage est privé.

NB2: Seule la méthode init() de la classe mère est utilisable pour une instance de la classe dérivée, car elle est redéfinie comme méthode publique dans Derivee.

NB3: Dans la classe Dérivee on peut utiliser une syntaxe plus simple en remplaçant :

```
    void init (int val) { Mere::init (val); }
```

par :

```
    Mere::init;
```

4.2.7 Appel de constructeurs

Pour construire une instance de la classe dérivée, un constructeur commence systématiquement par appeler le constructeur de la classe mère puis effectue un traitement spécifique situé entre les accolades pour prendre en compte les modifications par rapport à la classe mère.

Comme on l'a vu pour la composition de classes, le constructeur appelé automatiquement est un constructeur sans argument, il est donc nécessaire que la classe mère dispose d'un constructeur sans argument sinon il y a erreur de compilation.

Pour éviter l'appel automatique du constructeur sans argument de la classe mère on peut indiquer dans le constructeur de la classe dérivée un constructeur particulier de la classe mère à appeler. C'est ce qui a été fait dans l'exemple ci-dessus (§ 4.2.6), le constructeur Derivee(int,int) appelle le constructeur Mere(int, int) avec les paramètres x et y.

NB1: L'appel du constructeur de la classe mere est situé entre l'entête et le corps du constructeur de la classe dérivée et il est introduit par ':'

NB2: Le constructeur est identifié par le nom de la classe ce qui permet d'en appeler plusieurs dans le cas d'héritage multiple.

NB3: Dans le cas d'une hiérarchie de classes on peut avoir une série d'appels en cascade de constructeurs.

NB4: Un constructeur peut appeler d'une part les constructeurs des classes mères dont il dérive, et d'autre part les constructeurs de membres instances d'autres classes

4.2.8 Appel du destructeur

De manière symétrique aux constructeurs, lors de la destruction d'une instance, le destructeur de la classe dérivée est appelé pour effectuer le traitement spécifique, puis le destructeur de la classe mère est appelé automatiquement .

4.2.9 Accès aux fonctions membres de la classe mère

Une fonction membre de la classe dérivée peut appeler une fonction d'une classe ancêtre. S'il y a ambiguïté entre noms de fonctions identiques situées à différents niveau de la hiérarchie de classes, on utilise le nom complet. Cela est en particulier nécessaire lorsque la méthode a été surchargée dans la classe dérivée, comme l'appel à Compteur::inc dans l'exemple du Compteur_borne du § 4.2.3.

Chapitre 5
Les opérateurs

5.1 Surcharge d'un opérateur

Dans la plupart des langages de programmation certains opérateurs ont un sens différent selon le type des opérandes.

```
int a, b = 4, c= 2;
double ad, bd = 4.3, cd = 2.5;
a = b + c;
ad = bd + cd;
```

L'opérateur '+' correspond ici à 2 traitements différents selon qu'il est utilisé pour une addition d'entiers ou pour une addition de réels, on dit que l'opérateur + est surchargé.

5.1.1 Surcharge d'opérateurs pour des nouveaux types en C++

En C++, il est possible de surcharger un opérateur portant sur des types de base pour qu'il s'applique à des nouveaux types créés (classes). Pour cela, on doit spécifier dans ces classes les traitements à effectuer sur les opérandes quand on utilise l'opérateur. Quand un opérateur a été surchargé son utilisation pour des instances de classe est aussi simple que pour des instances de types de base.

Si on définit une classe complexe et que l'on surcharge l'opérateur '+' pour effectuer des sommes de complexes on peut écrire les lignes suivantes :

```
complexe a, b(4.3 , 3.4), c(2.5 , 5.2);
a = b + c;
```

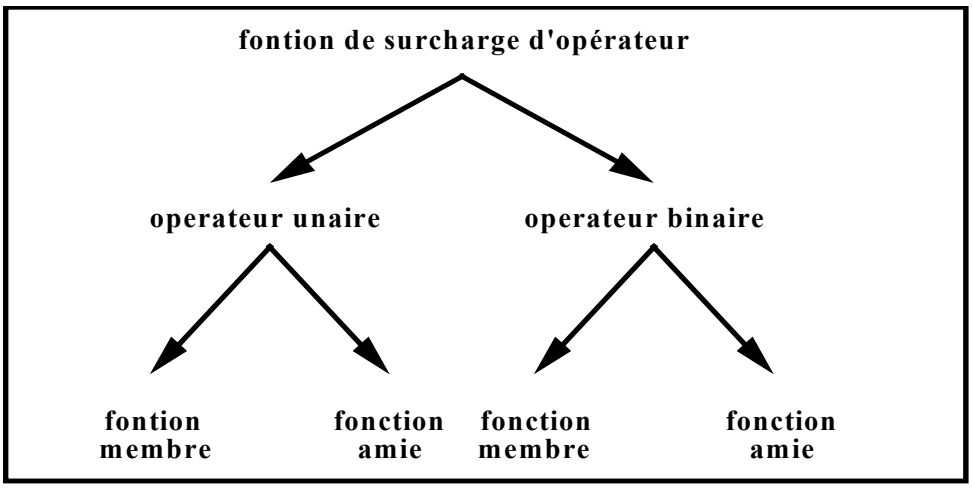
NB. a et b et c sont des instances de complexe, b et c sont initialisées par un constructeur complexe(double, double)

5.1.2 Règles pour la surcharge

- On ne peut surcharger que des **opérateurs existant en C++ pour les types de base**, on ne peut donc pas créer un nouvel opérateur.
- On ne peut **pas changer le gabarit d'un opérateur**, c'est à dire son arité, sa priorité et son associativité. Un opérateur surchargé pour une classe a donc le même nombre d'opérandes, la même priorité et la même associativité (gauche ou droite) que celles qu'il a pour les types de base.
- Un opérateur surchargé doit avoir obligatoirement **au moins un opérande instance d'une classe**, il n'est donc pas possible de surcharger un opérateur pour qu'il s'applique à des types C comme par exemple des tableaux.

5.2 Mise en oeuvre de la surcharge d'un opérateur

Pour surcharger un opérateur il faut considérer deux cas : les opérateurs unaires qui n'ont qu'un seul opérande (exemple: !x) et les opérateurs binaires qui ont deux opérandes (exemple : x + y). Pour surcharger un opérateur on doit définir une fonction spécifiant le traitement à effectuer, cette fonction peut être une fonction membre (méthode) ou une fonction globale C amie de la classe concernée.



5.2.1 Surcharge d'un opérateur unaire avec une fonction membre

Quand le compilateur rencontre une expression de la forme **∇x** où :

- **∇** est un opérateur C++ unaire
- x est une instance d'une classe C

il transforme cette expression en **x.operator∇(),** ce qui nécessite que la fonction membre operator**∇** () soit définie dans la classe C concernée.

∇x ⇒ x.operator∇( )

∇ operateur unaire C++ de base
x instance d'une classe **C**
T C::operator∇( ) fonction membre de la classe **C** avec retour de type **T**

On peut écrire une classe pour représenter les nombres complexes et surcharger l'opérateur ! avec pour sens l'obtention du nombre complexe conjugué (On prend pour convention que **!a** désigne le conjugué de a)

```
complexe a(1.2, 3.3);      // a = 1.2 + i * 3.3
complexe b;
b = !a;                    // b = 1.2 - i * 3.3
```

La classe complexe peut être défini de la manière suivante

```
class complexe
{
    double reel, imaginaire;
public:
    complexe(double,double);
    complexe operator!();
};
```

```
complexe complexe::operator!()
{
    complexe r;
    r.reel = reel; r.imaginaire = - imaginaire;
    return r;
}
```

5.2.2 Surcharge d'un opérateur unaire avec une fonction amie

S'il n'existe pas de fonction membre pour surcharger un opérateur, le compilateur remplace l'expression **∇x** par **operator∇(x)** dans laquelle :

- **∇** est un opérateur unaire de base C++
- **x** est une instance d'une classe C

- **operator∇(x)** est l'appel a une fonction globale C++ amie de la classe C concernée, définissant le traitement associé à l'opérateur.

  ∇x ⇒ operator∇ (x)

  ∇ operateur unaire C++ de base
  x instance d'une classe **C**
  T operator∇(x) fonction amie de la classe **C** avec retour de type **T**

En reprenant le modèle de la classe complexe

```
complexe a(1.2, 3.3);                  // a = 1.2 + i * 3.3
complexe b;
b = !a;                          // b = 1.2 - i * 3.3
```

La classe complexe peut être définie de la manière suivante

```
class complexe
{
    double reel, imaginaire;
public:
    complexe(double,double);
    friend complexe operator!(complexe);
};
```

```
complexe operator!(complexe c)
{
    complexe r;
    r.reel = c.reel; r.imaginaire = - c.imaginaire;
    return r;
}
```

5.2.3 Surcharge d'un opérateur binaire avec une fonction membre

Quand le compilateur rencontre une expression de la forme **x ∇ y** où :
- **∇** est un opérateur binaire de base C++
- **x** est une instance d'une classe C
- **y** une variable de type quelconque T

il la remplace par l'expression **x.operator∇(y)** ce qui nécessite de définir dans la classe concernée une fonction membre operator∇ () avec un argument de type T

x∇ y ⇒ x.operator∇ (y)

∇ operateur binaire C++ de base
x instance d'une classe **C**
y de type **T**
T1 C::operator∇ (T) fonction membre de la classe **C,** avec retour de type **T1**

On peut ainsi définir l'addition de complexes

```
complexe a(1.2, 2.3), b(3.4, 4.5);
complexe c;
c = a + b;               // c = 4.6 + i * 6.8
```

```
class complexe
{
    double reel, imaginaire;
public:
    complexe(double, double);
    complexe operator+(complexe);
};
```

```
complexe complexe::operator+(complexe y)
{
    complexe r;
    r.reel = reel + y.reel;
    r.imaginaire = imaginaire + y.imaginaire;
    return r;
}
```

NB1: Ici le deuxième opérande est de type complexe mais il pourrait être de n'importe quel type.

NB2: Avec une surcharge d'opérateur binaire par fonction membre le premier opérande est obligatoirement instance de la classe concernée.

5.2.4 Surchage d'un opérateur binaire avec une fonction amie

De la même manière que pour les opérateurs unaires, on peut surcharger un opérateur binaire en utilisant une fonction globale amie de la classe.

Quand le compilateur rencontre une expression de la forme **x ∇ y** où :

- **∇** est un opérateur binaire C++ de base
- x, ou y , ou les deux sont des instances d'une classe
- il existe une fonction amie de la classe de forme **operator∇(…)**

il remplace l'expression par **operator∇(x, y)**

x∇ y ⇒ operator∇(x, y)

∇ operateur binaire C++ de base
x instance d'une classe **C**
y de type **T**
T1 operator∇(C, T) fonction amie de la classe **C** avec retour de type **T1**

ou bien

∇ operateur binaire C++ de base
x de type **T**
y instance d'une classe **C**
T1 operator∇ (T, C) fonction amie de la classe **C** avec retour de type **T1**

On peut ainsi définir l'addition de deux complexes qui rend un complexe

```
complexe a(1.2,2.3), b(3.4, 4.5);
complexe c;
c = a + b :        // c = 4.6 + i * 6.8
```

```
class complexe
{
    double reel, imaginaire;
public:
    complexe(double, double);
    friend complexe operator+(complexe, complexe);
};
```

```
complexe operator+(complexe x, complexe y)
{
    complexe r;
    r.reel= x.reel + y.reel;
    r.imaginaire = x.imaginaire + y.imaginaire;
    return r;
}
```

NB: Dans le cas de l'addition de deux complexes il est indifférent de réaliser la surcharge avec une fonction membre ou avec une fonction amie. On peut cependant préférer la fonction membre à la fonction amie qui constitue une violation du mécanisme de protection des données privées. Par contre la syntaxe d'une fonction amie reflète la symétrie des opérandes de l'expression.

Dans le cas d'une surcharge d'opérateur binaire avec le premier opérande de type quelconque et le deuxième opérande instance d'une classe on doit forcément utiliser une fonction amie de la classe. C'est le cas pour une surcharge de l'opérateur + pour des expressions de la forme réel + complexe.

```
complexe a(1.2,3.4),b;
double d(5.2);
. . . . .
b = d + a;          // somme double + complexe
                    // b = 6.4 + i * 3.4
```

```
class complexe
{
    double reel;
    double imaginaire;
public:
    complexe(double, double);
    friend complexe operator+(double, complexe)
};
```

```
complexe operator+(double d, complexe c)
{
    complexe r;
    r.reel = d + c.reel; r.imaginaire = c.imaginaire;
    return r;
}
```

5.3 Surcharge d'opérateurs "exotiques"

Certains opérateurs C/C++ ont une syntaxe ou des propriétés qui posent quelques difficultés pour les surcharger, c'est pourquoi il est nécessaire d'utiliser des astuces ou au moins une programmation non régulière.

5.3.1 Surcharge de l'opérateur =

L'opérateur = pose un problème spécifique car il doit d'une part **renvoyer une valeur** comme tout opérateur à cause de la possibilité d'effectuer des affectations en cascade (a = b = c;), d'autre part il a un **"effet de bord"** c'est à dire qu'il modifie un de ses opérandes. L'opérateur effectue trois opérations distinctes :

- libération de la mémoire éventuellement acquise dynamiquement par l'opérande de gauche,
- affectation de la valeur de l'opérande de droite à l'opérande de gauche
- renvoi de la valeur de l'opérande de gauche

On peut remarquer que la première opération est exactement le travail d'un destructeur et que la deuxième opération est exactement le travail d'un constructeur par copie.

Dans l'exemple suivant on reprend la classe Date (Cf. §3.4.2) et on ajoute la possibilité d'affecter une variable date à une autre.

```
Date d1(12, "decembre", 1928);
Date d2(25, "juin", 1998);
d1 = d2;                        // utilisation de l'opérateur =
```

```
class Date
{
    int jour, an;
    char *mois;
public:
    Date(int, char *,int);
    ~Date();
    Date &operator=(const Date &);
};
```

```
Date & Date::operator=(const Date & droite)
{
    if(this != &droite)         // test du cas a = a
    {
        delete mois[];
        mois = new char[strlen(droite.mois)+1];
        strcpy(mois, droite.mois);
        jour = droite.jour; an = droite.an;
        return *this;
    }
}
```

NB1: On retrouve bien dans la définition de operator=() les instructions figurant dans le destructeur suivies des instructions figurant dans le constructeur par copie.

NB2: On teste si les deux opérandes de = sont la même variable pour éviter de détruire une instance avant de la reconstruire avec elle même, c'est a dire avec une information peut être inexistante. Si on est dans le cas a = a, on ne fait rien.

NB3: Le return *this est nécessaire pour pouvoir écrire une expression contenant de multiples affectations comme d1 = d2 = d3 = d4;

NB4: Il ne faut pas confondre le constructeur par copie utilisé lors de la définition d'une variable comme **Date d1 = d2;** et l'opérateur d'affectation utilisé après définition des variables.

NB5: operator=() renvoie une référence de date pour éviter un temps de recopie d'une date

```
Date d1(1,"janvier",2020);      // constructeur Date(int, char*, int)
Date d2 = d1;                   // constructeur par copie
Date d3;                        // constructeur Date()

d3 = d2;                        // opérateur =
```

5.3.2 Surcharge de l'opérateur [ ]

L'opérateur[] d'accès à un élément d'un tableau peut être surchargé pour alléger les écritures dans l'utilisation d'une classe vecteur ou même pour mettre en place un accès par indice dans une structure non prévue pour cela (liste linéaire)

```
class vecteur_int
{
    int *pt;
public:
    vecteur_int(int taille)   {  pt = new int[taille];  }
    int elem(int ind)         {  return pt[ind];  }
    int &operator[](int ind)  {  return pt[ind];  }
    . . . . .
};
```

```
vecteur_int v(20);
int x;
x = v.elem(3);     // sans utiliser l'opérateur []
x = v[3];          // en utilisant l'opérateur []
```

NB1: L'opérateur permet un accès simple aux éléments du vecteur situés en zone privée. Sans opérateur [] on doit utiliser la fonction d'accès elem() et écrire x = v.elem(3) au lieu de x = v[3]. Si l'on renonce à la protection des données en mettant les éléments du vecteur en zone publique on n'a plus à appeler la fonction d'accès elem() mais l'écriture reste lourde x = v.pt[3]

NB2: La méthode opérator[]() renvoie un int par référence ce qui permet d'utiliser cet opérateur à gauche d'un opérateur =

```
x = v[i];   // OK
v[i] = x;   // erreur si on a int vecteur_int::operator[](int)
v[i] = x;   // OK     si on a int & vecteur_int::operator[](int)
```

Il est possible de surcharger l'opérateur [] avec comme deuxième opérande n'importe quel type de base ou classe ce qui permet de réaliser facilement des tables associatives dans lesquelles t[clé] correspond à l'information associée à la clé.

5.3.3 Surcharge de l'opérateur new

En C++ l'opérateur new permet de réserver de la place dans le TAS pour un type de base ou pour une classe. Le mécanisme d'allocation mémoire prend en charge toutes les tailles d'objets possibles et gère au mieux l'espace mémoire en récupérant les "trous".

Si on fait beaucoup d'allocations et de libérations d'espace pour des instances d'une classe particulière et que l'on désire optimiser ces opérations, on peut gérer soi même une zone de stockage (TAS personnel) et surcharger l'opérateur new. L'opérateur new appelé pour les instance de la classe dans laquelle il est défini utilisera notre mécanisme de gestion de mémoire. Le mécanisme système général sera utilisé pour les instances d'autres classes et pour les types de base.

L'opérateur new est défini avec un argument de type size_t (défini dans le fichier entête stdlib) et il doit renvoyer un pointeur générique de type void *

```
class truc
{
    truc();
    void *operator new(size_t);
    . . . . . .
};
truc * pt;         // pt pointeur sur truc
pt = new truc;     // réservation de la place pour un  truc
```

NB1: On peut noter l'espace entre operator et new dans le nom de la fonction, cet espace est nécessaire

NB2: Bien que cela n'apparaisse pas de manière explicite dans la déclaration de la fonction membre operator new( ), cette fonction est statique et ne peut donc accéder qu'aux membres statiques de la classes où ils sont définis (attributs ou méthodes).

L'opérateur delete est surchargé en réalisant une fonction membre de prototype **void operator delete(void *)** pour lequel les remarques concernant new s'appliquent (espace entre operator et delete, caractère implicitement statique.

5.3.4 Surcharge de l'opérateur ()

Si l'on surcharge l'opérateur () pour une classe, les instances de cette classe sont appelés des objets fonctions car ils peuvent être appelés comme des fonctions . Un des intérêts des objets fonctions est de les passer comme argument d'un autre fonction, cela constitue alors un des moyen de passer une fonction en argument d'une fonction;

```
class truc
{
    . . . . . .
public:
    . . . . .
    int operator()(int n, int m)
       { . . . . . .}
};
truc x;
int y;
y = x(3,8);
```

5.3.5 Surcharge des opérateurs d'incrémentation et décrémentation (++, --)

Il est possible et souvent utile de surcharger les opérateurs ++ et -- pour les appliquer à des instances de classes. Une utilisation classique est la progression et régression d'une variable iterator à l'intérieur d'une structure de données. L'opération ++ permet d'avancer une sorte de pointeur à l'intérieur d'une liste ou d'un vecteur en veillant par exemple à ne pas déborder de la structure, l'opérateur -- effectue le travail inverse.

Les opérateurs ++ et -- ont une particularité dans le fait qu'il peuvent être préfixés ou post fixés, c'est à dire placés avant ou après leur opérande. Pour le type de base int la version préfixée de ++ effectue une incrémentation avant utilisation, la version post fixée effectue l'incrémentation après utilisation (v[++i] et v[i++] ne désignent pas le même élément)

Pour distinguer les opérateurs pré et post fixés, les fonctions operator++() et opérateur--() utilisent un argument supplémentaire factice.

```
class ptr_sur_int
{
    . . . . .
public:
    . . . . .
    int * operator++ ()    { . . . . . }     // préfixé
    int * operator++ (int) { . . . . . }     // post fixé
    int * operator-- ()    { . . . . . }     // préfixé
    int * operator-- (int) { . . . . . }     // post fixé
};
```

5.3.6 Opérateurs spécifiques ou opérateurs polyvalents

Quand on doit surcharger un opérateur pour différents types d'opérandes on peut écrire une série de fonctions **operator∇()** avec des arguments correspondant aux différents types. Dans le cas d'une classe complexe on peut avoir la définition suivante :

```
class complexe
{
    double reel, im;
public:
    complexe(double, double);
    complexe();
    complexe operator+(const complexe &);
    complexe operator+(double);
    complexe operator+(int);
    . . . . . .
};
complexe a(1.2,2.3), b(3.4, 4.5), c;
double d(5.6);
int e(5);
c = a + b;
c = a + d;
c = a + e;
```

Si on doit aussi utiliser l'opérateur - il faut déclarer une série de fonctions operator-() pour prendre en compte les différents types.

Il est possible de simplifier la programmation en définissant une seule fonction operator+(complexe) et une seule fonction operator-(complexe). On définira une série de constructeurs pour la classe complexe, un constructeur avec un arguments complexe, un constructeur avec un argument double, un constructeur avec un argument int.

Les différents constructeurs interviennent comme **convertisseurs implicites** entre les types double et complexe, int et complexe ce qui permet de n'avoir qu'une seule déclaration pour chaque opérateur.

```
class complexe
{
    double reel, im;
public:
    complexe();
    complexe(double, double);
    complexe(double);
    complexe(int);
    complexe operator+(const complexe&);
    complexe operator-(const complexe&);
. . . . . .
};
```

Chapitre 6
Les entrées-sorties sur flots en C++

6.1 Entrées-sorties sur clavier et écran (flots C++)

6.1.1 Flots d'entrée et de sortie

Un programme C ou C++ connaît un certain nombre de canaux d'entrée-sortie : **stdin** : canal standard d'entrée, **stdout** : canal standard de sortie, **stderr** canal standard d'erreur. Par défaut stdin correspond au clavier, stdout et stderr correspondent à l'écran. Le langage C dispose d'un certain nombres de fonctions de bibliothèque permettant d'effectuer des entrées sorties sur les canaux standard (printf, scanf, etc.)

Les concepteurs du langage C++ on développé quelques classes permettant de représenter les canaux d'entrée-sortie et d'encapsuler les opérations sur ces canaux afin de simplifier la programmation, ces classes sont disponibles dans la bibliothèque standard. La classe **ostream** représente un flot de sortie, et la classe **istream** représente un flot d'entrée.

6.1.2 La classe ostream

La classe ostream représente un flot en sortie, elle dispose d'un certain nombre d'attributs caractérisant le flot et d'un certain nombre de méthodes pour manipuler ce flot. Parmi ces méthodes on trouves des méthodes classique comme put(), write() et aussi des méthodes **operator<<()** pour surcharger l'opérateur '<<' . On peut alors utiliser '<<' pour effectuer une écriture.

```
class ostream
{
    . . . . .
public:
    ostream& operator<<(char *);
    ostream& operator(int);
    ostream& operator<<(long);
    ostream& operator<<(double)
    ostream& operator<<(char);
    ostream& operator<<(float);
    . . . . .
    ostream& operator<<(long double);
    ostream& put(char);
    ostream& put(unsigned char);
    . . . . .
    ostream& write(const char *,int);
    . . . . .
};
```

C++ dispose par ailleurs de **variables prédéfinies instances de la classe ostream**, ce sont **cout**, **cerr** et **clog**. Ces variables désignent respectivement le flot de sortie, le flot d'erreur et le flot de journalisation, ces trois flots sont assignés par défaut à l'écran. Une écriture s'effectue alors simplement en utilisant la variable prédéfinie et l'opérateur d'écriture '<<'

6.1.3 Ecriture sur cout, cerr ou clog

Un transfert vers cout s'effectue à l'aide de l'opérateur << en utilisant une expression de la forme

```
cout << donnée;    // écriture de donnée sur l'écran
```

L'opérateur **<<** a été choisi parce qu'il présente les bonnes priorité et associativité et que le symbole suggère le sens du transfert de la donnée vers cout.

La donnée est d'abord convertie en une suite de caractères puis transmise à cout (écran). L'utilisation de l'opérateur '<<' permet indiquer une écriture avec une syntaxe simple que l'on peut comparer à

l'utilisation de la fonction printf() du C de base, qui nécessite de fournir des codes de conversion %d, %f, %c, %s dépendant des types des variables.

Il est aussi possible d'empiler les transferts en utilisant plusieurs opérateurs << dans une même expression.

```
int x = 23;
double y = 24.36;
string ch("bonjour");
char t[] = "une chaine";
cout << "salut" ;              //  → salut
cout << t ;                    //  → une chaine
cout << x ;                    //  → 23
cout << y ;                    //  → 24.36
cout << ch ;                   //  → bonjour
cout << "x = " << x << endl;   //  → x = 23
```

NB1: endl est une constante correspondant au caractère de passage à la ligne (line feed)

NB2: Le dernier exemple montre un empilement de sorties sur l'écran

L'opérateur '<<' permet de transférer sur le canal concerné (ici cout) tous les types de base C et un certain nombre des types de données de la bibliothèque normalisée C++ (string par exemple).

6.1.4 La classe istream

La classe istream est analogue à la classe ostream, elle possèdes des méthodes comme get(), getline(), read(), putback(), mais aussi des méthodes **operator>>()** pour surcharger l'opérateur '>>' pour tous les types de base.

C++ dispose d'une **variable prédéfinie, instance de istream**, appelée **cin**. Par défaut cin est assignée au clavier.

6.1.5 Lecture sur cin

Un transfert depuis cin vers une variable s'effectue à l 'aide de l'opérateur '>>' en utilisant une expression de la forme

```
cin >> variable ;    // lecture au clavier, stockage dans variable
```

NB: L'opérateur >> indique que le transfert s'effectue depuis cin vers la variable.

L'opérateur >> effectue la conversion de la suite de caractères tapés au clavier en une représentation binaire correspondant au type de la variable de destination. Cette conversion est possible pour tous les types de base ainsi que pour certains types de la bibliothèque C++ (string par exemple)

```
int x ;
double y ;
string ch;                // ch string vide
sting ch1("hop");         // ch1 string contenant hop
cin >> x;                 // 12        →     x contient 12
cin >> y ;                // 3.55      →     y contient 3.55
cin >> ch ;               // toto      →     ch contient toto
cin >> ch1;               // bidule    →     ch1 contient bidule
```

NB: Lors de la lecture d'une string la taille est ajustée automatiquement. Ici la taille de ch1 passe de 3 à 6

On peut empiler les entrées saisies au clavier pour stocker les valeurs entrées dans les différentes variables.

```
int x, y ;
double z ;
cin >> x >> y >> z ;      // 1 22  3.5    →    x = 1, y = 22, z = 3.5
```

6.1.6 Lecture de caractères (filtrage)

L'utilisation de l'opérateur >> associé à cin et à une variable de type caractère filtre les caractères **espaces blancs**, c'est a dire que les caractères <**espace**>**,** <**enter**>, <**vtab**>, <**htab**> sont retirés du flot et non stockés dans les variables de réception.

```
char t[12] ;       // t tableau de 12 char
int i ;
for (i=0; i<10; i++)
    cin >> t[i];         // lecture d'un caractère dans t[i]
t[i] = '\0';             // marque fin de chaine
cout << t << endl;

bon j
  ou
r
  abc              // → bonjourabc
```

Pour prendre en compte les espaces blancs lors d'une lecture à partir de cin dans une variable de **type caractère** on utilise la fonction **get()** de la classe istream.

```
char t[12] ;
int i ;
for (i=0; i<10; i++)
    cin.get (t[i]);      // lecture d'un caractère dans t[i]
t[i] = '\0';             // marque fin de chaine
cout << t << endl;

bon j  our abc           // → bon j  our
```

NB: Lors de la lecture d'entiers il y a systématiquement filtrage des espaces blancs avant détection du premier chiffre.

6.1.7 Test de fin de lecture

Quand on doit lire au clavier une série de caractères ou un séries de nombres dont on ne connaît pas la longueur a priori, on doit disposer d'un moyen pour indiquer que la saisie est terminée (signal de fin de lecture). Par ailleurs les instructions de lecture doivent pouvoir détecter une saisie normale et un signal de fin de lecture.

Pour un PC le signal de fin est engendré au clavier par la frappe simultanée des touches <CTRL> et Z, suivie du caractère <ENTER>. <CTRL>Z et <ENTER> doivent être seuls sur la ligne.

La variable cin a une valeur booléenne que l'on peut tester pour savoir si le flot d'entré est dans l'état "bon". Quand un signal de fin de lecture a été envoyé, le flot n'est plus dans l'état bon et sa valeur est alors false. Pour déterminer que la lecture ne peut pas se poursuivre on peut tester la variable cin qui a pour valeur true tant que l'entrée est correcte.

```
// lecture d'une série de caractères
char c;
while (cin)        // test de la variable cin (flot d'entrée)
    {
    cin.get(c);    // lecture d'un caractère sans filtrage
    if(cin)        // vérification après saisie
        {
        traitement de c
        . . . . .
        }
    }
```

```
// lecture d'une série d'entiers
int x;
while (cin)        // test de la variable cin (flot d'entrée)
    {
    cin >> x;      // lecture d'un entier
    if(cin)        // vérification après saisie
        {
        traitement de x
        . . . . .
        }
    }
```

6.1.8 Fichiers système à inclure

Pour utiliser les classes istream, ostream et les variables prédéfinies cin, cout, cerr, clog il faut inclure dans le programme un certain nombre de fichiers système

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

NB1: *iostream* permet d'utiliser les entrées-sorties sur les flots classiques cin, cout, cerr et clog.

NB2: *iomanip* permet d'utiliser les fonctions de formatage des entrées-sorties

NB3: *using namespace std;* permet d'utiliser des noms courts pour toutes les classes istream ostream, et les variables prédéfinies cin, cout, etc (sinon il faudrait écrire std::cin, std::cout, etc.)

6.2 Entrées-sorties sur fichiers disque

Pour effectuer des entrées-sorties sur disque on utilise les classes **ifstream** et **ofstream** analogues aux classes **istream** et **ostream**.

6.2.1 Lecture sur fichier disque

6.2.1.1 Ouverture du fichier en lecture

Pour lire sur un fichier disque on doit d'abord l'ouvrir. L'opération d'ouverture s'effectue en créant une variable du type prédéfini ifstream, en la liant à un fichier disque et en indiquant que cette variable sera utilisée en mode lecture de la manière suivante :

```
string nomfichier("fich1.txt");
ifstream fi(nomfichier.c_str(), ios::in);
if (fi)
    {
    instruction de lecture
    }
else
    cout << "Erreur ouverture fichier" << endl;
```

NB1: nomfichier est une variable de type string, on peut aussi utiliser une chaîne C de base stockée dans un tableau de char

NB2: La fonction c_str() est nécessaire pour convertir la string en une chaîne C de base

NB3: ios::in indique que le fichier sera utilisé en lecture

NB4: Si le fichier est ouvert correctement la variable fichier prend pour valeur true

On peut aussi utiliser une chaîne C de base comme nom de fichier :

```
char nomfichier[] = "fich1.txt";
ifstream fi(nomfichier, ios::in);
```

NB: On n'utilise pas la fonction c_str() car la chaîne est une chaîne C de base et non une string

6.2.1.2 Lecture sur le fichier

La lecture sur le fichier s'effectue à l'aide de l'opérateur '>>' comme pour la lecture au clavier mais la variable cin est remplacée par la variable de type ifstream créée ci-dessus

```
fi >> variable     // lecture sur fichier disque et stockage
```

```
string nomfichier("fich1.txt");
double x;
ifstream fi(nomfichier.c_str(), ios::in);    // ouverture du fichier
if (fi)
    {
    fi >> x;             // lecture d'un double sur le fichier
    }                    // et stockage dans la variable x
else
    cout << "Erreur ouverture fichier" << endl;
```

6.2.1.3 Fermeture du fichier

Apres utilisation du fichier disque on doit le fermer à l'aide de la fonction **fclose()** de la classe ifstream de la manière suivante :

```
fi.fclose();       // fermeture du fichier
```

6.2.1.4 Lecture de caractères sur un fichier disque

La lecture de caractères sur fichiers disque s'effectue de la même manière que pour la lecture de caractères sur le clavier. Il suffit de remplacer la variable prédéfinie cin par la variable de type ifstream (voir 6.1.4)

6.2.1.5 Test de fin de fichier

La détection de fin de fichier sur disque s'effectue de la même manière que pour la lecture sur le clavier.

```
ifstream fi(nomfichier2.c_str(), ios::in);
double x;
if (fi)                  // test ouverture correcte de fichier
    {
    while (fi)           // test de fin de fichier
        fi >> x;         // lecture d'un double
    fi.close();
    }
```

NB: La variable fichier fi de type ifstream garde la valeur true tant que l'entrée est correcte. Elle prend la valeur false quand il y a fin de fichier.

6.2.2 Ecriture sur un fichier disque

6.2.2.1 Ouverture du fichier en écriture

L'ouverture en écriture est analogue à l'ouverture en lecture. On crée une variable de type prédéfini ofstream, on la lie à un fichier disque et on fixe le mode d'ouverture en écriture

```
string nomfichier("fichier1.txt");
ofstream fo(nomfichier1.c_str(), ios::out);
if (fo)
    {
    instruction d'écriture
    }
else
    cout << "erreur ouverture fichier" << endl;
```

6.2.2.2 Ecriture sur le fichier

On utilise une forme semblable à celle mise en œuvre pour l'écriture sur l'écran dans laquelle on remplace la variable cout par la variable fichier de type ofstream

```
string nomfichier("fichier1.txt");
double x = 3.1416;
ofstream fo(nomfichier1.c_str(), ios::out);
if (fo)
    fo << x;       // ecriture de x sur le fichier
else
    cout << "erreur ouverture fichier" << endl;
```

6.2.2.3 Fermeture du fichier

On utilise la même instruction pour fermer un fichier utilisé en lecture ou en écriture

```
fo.fclose()                    // fermeture du fichier
```

6.2.3 Fichiers systèmes a inclure

Pour utiliser les classes iftream, ofstream il faut inclure dans le programme le fichier système fstream

```
#include <fstream>
```

6.3 Ecriture/lecture de variables de type défini par l'utilisateur

6.3.1 Ecriture de variables de type défini par l'utilisateur

La facilité d'écriture de variables des types de base conduit à vouloir écrire des instances de classes définies par l'utilisateur avec les mêmes outils. On peut, par exemple, définir une classe complexe, créer une instance c de complexe et vouloir écrire un complexe avec une instruction de la forme cout << c. Pour cela il faudrait surcharger l'opérateur << dans la classe ostream pour écrire des complexes, ceci n'est pas possible du fait que la classe ostream et l'instance cout sont prédéfinies par le concepteur du langage C++ et donc non modifiables.

L'opérateur '<<' a 2 opérandes, un ostream et une instance de classe que l'on veut écrire. Comme il n'est pas possible de surcharger '<<' dans la classe ostream, on le surcharge dans la classe définie par l'utilisateur c'est à dire la classe complexe. Le premier opérande de '<<' n'est pas un complexe, on doit donc utiliser une fonction amie.

```
#include <iostream>
using namespace std;
class complexe
{
    double re,im;
public:
    friend ostream& operator<< (ostream &, const complexe &);
    . . . . .
};
ostream & operator<< (ostream &f, const complexe &c)
{
    return f << "(" << c.re << "," << c.im << ")\n";
}
```

NB: Il ne faut pas oublier de renvoyer le flot dans operator<<() car c'est ce qui permet d'enchaîner des sorties dans une même instruction.

```
void main(void)
{
    complexe x(1.5, 3.0);
    . . . . .
    cout << "x vaut : " << x;          //  →  x vaut : (1.5,3.0)
}
```

6.3.2 Lecture de variables de type défini par l'utilisateur

Comme pour les écritures il est possible d'effectuer des lectures d'instances de classes définie par l'utilisateur à l'aide de l'opérateur '>>'. Il faut surcharger cet opérateur dans la classe utilisateur à l'aide d'une fonction amie.

Le code de operator>>() est voisin de celui de operator<<() mais dans le cas ou la lecture d'une instance correspond à plusieurs données, comme les parties réelle et imaginaire d'un complexe, il faut éventuellement indiquer à l'utilisateur comment entrer les données

```
#include <iostream>
using namespace std;
class complexe
{
    double re,im;
public:
    friend istream& operator<< (istream &, complexe &);
    . . . . .
};
```

```
istream & operator<< (istream &f, complexe &c)
{
    cout << "partie réelle : ";       f >> c.re;
    cout << "partie imaginaire : ";   f >> c.im;
    return f;
}
```

Chapitre 7
Notions avancées sur les classes

Ce chapitre présente quelques éléments avancés concernant les classes, ce sont essentiellement les attributs et méthodes constants (**const**), les attributs et méthodes de classes (**static**), et enfin les classes imbriquées.

7.1 Variables, attributs, méthodes const

En C/C++ on peut utiliser le modificateur **const** dans un certain nombre de situations pour empêcher les modifications des éléments concernés. Si le programme contient des instructions de modification d'un élément déclaré **const** on a une erreur à la compilation.

7.1.1 Variable const

Une variable déclarée const ne peut pas voir sa valeur modifiée par le programme

```
int main()
{
    const int x = 2;
    x = 3;               // ERREUR MODIFICATION INTERDITE
}
```

7.1.2 Pointeur constant, pointeur sur constante

Un pointeur peut être déclaré constant pour empêcher qu'il change de valeur c'est à dire qu'il pointe sur autre chose que ce sur quoi il pointait initialement.

```
int main()
{
    int x = 2;
    int y =3;
    int * const px = &x; // px pointeur constant pointe sur x
    px = &y;             // ERREUR px NE PEUT PAS POINTER SUR y
}
```

Un pointeur sur une information constante, à ne pas confondre avec un pointeur constant, est tel qu'on ne peut pas changer l'information sur laquelle il pointe.

```
int main()
{
    int x = 2;
    const int * px = &x;  // px pointe sur x
    *px = 4;              // ERREUR L'INFO POINTEE PAR PX NE PEUT PAS
}                         // CHANGER
```

On peut cependant avoir l'accès en modification en utilisant un autre pointeur qui n'est pas du type pointeur sur information constante (ici de type int *)

```
int main()
{
    int x = 2;
    const int * px = &x;  // px pointeur pointe sur x constant
    int * py;             // py pointeur sur int non constant
    py = (int *) px;      // recopie avec conversion de px dans py
    *py = 4;              // accès pour modification à l'info pointée
}
```

NB. On effectue ici une conversion de type du pointeur px avant de l'affecter à py

7.1.3 Attributs constants d'un objet

On peut avoir des attributs d'une classe dont on désire fixer la valeur une fois pour toutes, ces attributs intangibles sont introduits par le mot clé **const**. Le fait qu'un attribut soit intangible empêche de lui affecter une valeur initiale d'une manière simple, on doit l'initialiser dans le constructeur de la classe en utilisant la même syntaxe que pour les classes composées présentées au Chapitre 4.

```
class Truc
{
    const int x;                        // attribut intangible
    float y;
public :
    Truc(int i, int j): x(i) { y=j; } // constructeur inline avec
                                        // initialisation
    . . . . .
    void f(int a) { x = a; }            // ERREUR x NON MODIFIABLE
};
```

NB: L'attribut intangible x est initialisé dans le constructeur avant même d'exécuter le corps du constructeur entre accolades.

7.1.4 Méthodes de consultation

Les méthodes d'une classe ont accès aux attributs et effectuent généralement des modifications des valeurs de ces attributs. Les attributs d'une classe représentent l'état de l'objet instance et les méthodes représentent le comportement de l'objet, il est donc normal que l'état de l'objet change quand on demande un travail à ce dernier en invoquant une méthode.

Il peut cependant exister des méthodes qui effectuent un travail sans modifier les attributs de l'objet, ces méthodes de consultation des attributs sont aussi appelées des **observateurs**.

Si une méthode ne doit pas changer les valeurs des attributs de l'instance pour laquelle est invoquée, on peut en avertir le compilateur afin qu'il vérifie qu'il n'y a pas de tentative de modification. Cet avertissement se fait à l'aide du mot clé **const** que l'on place à la fin de la déclaration de la méthode

```
class Truc
{
    int x;
public:
    Truc(int i)           // constructeur
        {  x = i; }
    void voir() const;    // méthode voir() déclarée const
    void regarder();      // méthode regarder() pas déclarée const
};
void Truc::voir() const
    {   cout << "x vaut : " << x << endl; }
void Truc::regarder()
    {   cout << "x vaut : " << x << endl; }
```

Une méthode déclarée const est utilisable pour un objet const alors qu'une méthode classique ne l'est pas. Le compilateur vérifie cette situation et indique une erreur même si la méthode concernée ne tente pas de modifier les attributs.

```
const Truc x;       // x objet constant
. . . . .
x.voir();           // appel d'une méthode const pour un objet const
x.regarder();       // ERREUR APPEL D'UNE METHODE NON const SUR OBJET
                    // const
```

7.2 Attributs et méthodes de classe

7.2.1 Attributs static

Par défaut, les attributs sont définis dans les classes puis dupliqués dans les instances, c'est pour cela que les diverses instances peuvent avoir des valeurs d'attributs différentes. Les données situées dans les instances sont appelées **attributs** ou **variables d'instance**. Il peut être intéressant d'avoir certaines

données communes à toutes les instances (**variables de classe**) et donc pas dupliquées dans les instances, C++ résout ce problème à l'aide des attributs introduits par le mot clé **static**.

```
class Truc
{
public:
    int x;              // attribut d'instance
    static int y;       // attribut de classe
    Truc(int i)         // constructeur
        { x = i; }
    void f()            // méthode d'instance
        { y = 4; }      // accès à un attribut de classe à partir d'une
};                      // méthode

int Truc::y = 5;        // définition de l'attribut de classe

int main()
{
    Truc t(10);         // définition d'une instance de la classe truc
    t.x = 7;            // accès à un attribut d'instance
    Truc::y = 8;        // accès à un attribut de classe
    t.f();
}
```

NB1: L'attribut **static y** est déclaré dans la classe truc, mais il doit ensuite être défini comme une variable globale dont le nom est préfixé par le nom de la classe avec l'opérateur d'accès. Dans cette définition on peut fixer une valeur initiale.

NB2: L'accès à un attribut **static** depuis l'extérieur de la classe s'effectue en utilisant le nom préfixé par le nom de la classe avec l'opérateur d'accès **::** (à condition que l'attribut soit déclaré **public** !)

NB3: Dans une méthode de la classe on peut accéder à un attribut **static** en utilisant un nom simple puisque la préfixe de classe est implicite.

Un exemple d'utilisation d'attribut static est celui du comptage du nombre d'instances d'une classe:

- on déclare un attribut **static compteur** initialisé à 0,
- on incrémente cet attribut dans les constructeurs de la classe,
- on décrémente le compteur dans le destructeur.

Ainsi à tout instant l'attribut de classe **compteur** reflète le nombre d'instances de cette classe.

7.2.2 Méthode static

Une méthode peut être déclarée static de la même manière que l'on déclare un attribut static. Une méthode static peut être appelée indépendamment de toute instance de la classe, ce qui peut être utile lorsque l'on doit effectuer un travail lié à la classe avant toute création d'instance de la classe. Une méthode statique ne permet d'accéder qu'aux attributs static car les attributs d'instances (non static) sont par définition liés aux instances. Une méthode static permet d'atteindre les attributs static privés avant toute création d'instance.

```
class Truc
{
    static int x;         // attribut static
public:
    Truc(){}
    static int f(int i)   // methode static
        { x = i;}
};

int Truc::x;            // définition de l'attribut de classe

int main()
{
    Truc::f(5);           // accès à un attribut static privé
}                         // avant création d'instance
```

NB: L'opérateur surchargé new est implicitement static car il est là pour créer une instance, il doit donc être appelable avant création d'une instance (Cf. §5.3.3).

7.3 Classes imbriquées

Une classe imbriquée est une classe déclarée à l'intérieur d'une autre classe. Une classe imbriquée est locale à la classe englobante

7.3.1 Classe imbriquée publique

Une classe imbriquée publique est déclarée dans la zone publique de la classe englobante. La classe intérieure est visible de l'extérieur et l'on peut donc créer des instances de la classe imbriquée

```
class exterieure
{
    int ve;
public:
    class interieure
    {
    public:
        int vi;
        interieure(int i) {  vi = i; }
        void affiche()    {  cout << vi; }
    };
    void f()                                  // f crée une instance x
        {  interieure x(2); x.affiche(); }    // de la classe intérieure
    exterieure(int i)
        {  ve = i; }
};

int main()
{
    exterieure e(10);                 // e instance de extérieure
    exterieure::interieure z(5);      // z instance de intérieure
    e.f();                            // création d'une instance locale
}                                     // de intérieure dans f
```

NB1: z est une instance de la classe intérieure initialisée avec la valeur 5.

NB2: On utilise un nom complet (qualified name) avec l'opérateur d'accès :: pour désigner la classe dont z est une instance.

7.3.2 Classe imbriquée privée

La classe imbriquée est déclarée dans la partie privée de la classe englobante, il n'est pas possible de créer des instances de la classe intérieure à l'extérieur de la classe extérieure.

```
class exterieure
{
    int ve;
    class interieure
    {
    public:
        int vi;
        interieure(int i) {  vi = i; }
        void affiche()    {  cout << vi; }
    };
public:
    void f()
        {  interieure x(2); x.affiche(); }
    exterieure(int i)
        {  ve = i; }
};

int main()
{
    exterieure e(10);             // création e instance de exterieure
    exterieure::interieure z(5);  // ERREUR création impossible
    e.f();
}
```

NB1: La classe intérieure est inaccessible directement, on ne peut pas créer l'instance z de la classe intérieure.

NB2: La fonction f, membre de la classe extérieure, peut créer et utiliser des instances de la classe intérieure comme x, dans l'exemple.

7.3.3 Définition des méthodes des classes imbriquées

Dans l'exemple précédent les méthodes des classes extérieure et intérieure étaient définies inline, il est aussi possible de les définir à l'extérieur des classes comme le montre l'exemple suivant :

```
class ext
{
    int ve;
public:
    class int1
    {
    public:
        int v1;
    };
    class int2
    {
    public:
        int v2;
    };
    int1 f(int2);
};

ext::int1 ext::f(ext::int2 x)
{    . . . . .   }

int main()
{
    ext ve;                     // ve instance de ext
    ext::int1 vi1;              // vi1 instance de int1
    ext::int2 vi2;              // vi2 instance de int2
    vi1 = ve.f(vi2);
}
```

NB: On doit utiliser les noms complets pour les classes afin de lever toute ambiguïté.

Chapitre 8
Gestion d'objets dynamiques et polymorphisme

8.1 Données dynamiques et pointeurs

Les données dynamiques sont crées pendant l'exécution à partir du tas (heap) et sont manipulées à travers des pointeurs. C++ fournit deux opérateurs (new et delete) pour réserver et libérer des données dynamiques

```
char *p, *q;              // p, q pointeurs sur char
struct machin
{
    int x; double y;
} *r;                     // r pointeur sur struct machin
. . . . .
p = new char;             // réservation d'un char
q = new char[10];         // réservation de 10 char
r = new machin;           // réservation d'un struct machin
. . . . .
*p = 'a';          // accès à l'élément pointé par p
(*r).x = 2;        // accès aux membres de la structure pointée par r
r -> y = 3.14;     // idem
. . . . .
delete p;          // libération des zones pointées par p, q et r
delete[] q;
delete r;
```

8.2 Objets dynamiques

Les objets, comme les variables des types de base peuvent être réservés et libérés pendant l'exécution. Ils sont manipulés à travers des pointeurs, et sont réservés et libérés par les opérateurs new et delete

```
class stock
{
    string nom;                          // nom du produit
    int q;                               // quantité de produit
public:
    stock(){}                            // constructeur
    stock(string p, int i);              // constructeur
    stock(const stock &s);               // constructeur
};
. . . . .
int main()
{
    stock s1("tomates",5000);            // instance de stock
    stock *ptr1, *ptr2, *ptr3;           // pointeurs sur stock
    ptr1 = new stock("pommes", 34);      // création d'un stock
    ptr2 = new stock(s1);                // création d'un stock
    ptr3 = new stock();
    . . . . .
}
```

Où:

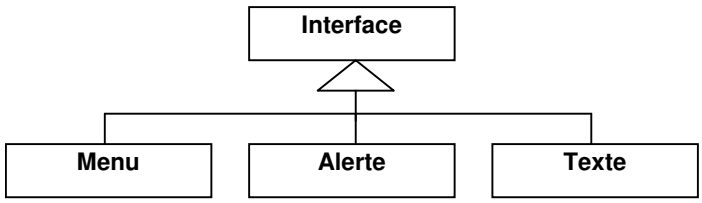
- s1 est un objet instance de stock
- ptr1, ptr2 et ptr3 sont des pointeurs sur objet stock
- ptr1 pointe sur un objet dynamique créé avec le deuxième constructeur
- ptr2 pointe sur un objet dynamique crée avec le constructeur par copie
- ptr3 pointe sur un objet dynamique créé avec le constructeur sans argument

NB: L'opérateur new appelle explicitement le constructeur de stock

8.3 Polymorphisme et fonctions virtuelles

8.3.1 Polymorphisme

Un objet polymorphe est un objet susceptible de prendre plusieurs formes (changement de type) pendant l'exécution du programme qui le manipule. La notion de polymorphisme s'appuie sur **l'héritage de classe** comme le montre l'exemple suivant qui concerne des objets d'interface



Les classes menu, alerte et texte représentent les concepts de menu, de fenêtre d'alerte et de zone de texte. Ces 3 classes sont dérivées de la classe interface qui représente un objet d'interface homme-machine et regroupe les attributs et les méthodes communes.

Une variable polymorphe instance de la classe interface peut, durant l'exécution, prendre la forme d'un menu, d'une alerte ou d'un texte, ceci est correct puisque les menus, les alertes, les textes sont des "sortes de" interface. Cette possibilité est exploitée généralement pour constituer des structures de données contenant à un même instant des éléments de types différents (tableau polymorphe, liste polymorphe, etc.). Un tableau d'interfaces peut contenir des éléments instances de menu, d'alerte ou de texte.

tableau d'interfaces

alerte
alerte
menu
alerte
texte
menu
menu

Le problème des objets polymorphes est la sélection des fonctions membres correspondant à la forme de l'objet au moment de l'appel. Par exemple si l'on appelle la méthode affiche() pour un objet instance d'interface, selon qu'il contient en fait un menu, une alerte ou un texte, il faut que la méthode affiche() spécifique de menu, alerte ou texte soit appelée.

8.3.2 Mise en oeuvre du polymorphisme en C++

En C++, le polymorphisme ne s'applique qu'à des objets dynamiques manipulés à travers des **pointeurs** qui sont transtypés lors des changements de forme.

```
class interface
{  . . . . .  };

class menu: public interface          // menu hérite d'interface
{  . . . . .  };

class alerte: public interface        // alerte hérite d'interface
{  . . . . .  };

class texte: public interface         // texte hérite d'interface
{  . . . . .  };
```

```
int main()
{
    interface *ptr;             // ptr sur interface
    menu *ptr1;                 // ptr sur menu
    alerte *ptr2;               // ptr sur alerte
    texte *ptr3;                // ptr sur texte
    ptr2 = new alerte(. . .);
    ptr = ptr2;                 // *ptr est un objet polymorphe
    . . . . .                   // ayant ici la forme d'une alerte
}
```

NB: ptr est un pointeur sur interface, connu comme tel par le compilateur, mais qui pointe sur une alerte pendant l'exécution, ptr peut pointer plus tard sur un menu ou un texte.

Si l'on utilise ptr comme dans l'exemple ci-dessus on rencontre un problème de sélection des méthodes. En effet si l'on a une méthode de nom affiche() dans interface et des méthodes affiche() dans menu, alerte et texte et si on a les instructions suivantes

```
interface *ptr;                 // ptr pointeur sur interface
ptr = new alerte(. . .);        // ptr pointe sur une alerte
ptr->affiche();                 // appel de affiche de interface
```

Le compilateur génère un appel à la méthode affiche() de la classe interface car ptr est défini comme un pointeur sur interface. Le traitement exécuté n'est pas le traitement attendu car il faudrait invoquer la méthode affiche() de la classe alerte puisque ptr pointe maintenant sur une alerte.

Ce mauvais choix de fonction membre est dû à la **liaison statique** effectuée lors de la compilation (**early binding**), ptr est lié définitivement à la classe interface, même si, à l'exécution ptr pointe sur un objet alerte

Pour avoir un fonctionnement correct il faut mettre en oeuvre, pendant l'exécution, une **liaison dynamique** entre un objet et une fonction membre (**late binding**). Dans le cas de liaison dynamique, la fonction membre appelée sera celle correspondant à la forme de l'objet à cet instant. La liaison dynamique est mise en oeuvre en C++ par l'intermédiaire des **fonctions virtuelles**.

8.3.3 Fonctions virtuelles

Dans la classe mère dont dérivent les différentes formes d'un objet polymorphe on déclare comme virtuelles les fonctions membres pour lesquelles on veut réaliser une liaison dynamique. Le compilateur fabrique une table de branchement (VTABLE) vers les fonctions identiques dans les classes descendantes, et génère le code de la méthode pour que celle-ci effectue un branchement à travers cette table. L'appel d'une méthode pour un pointeur polymorphe est toujours compilé comme une invocation de la méthode de la classe mère mais cette méthode effectue un branchement vers la méthode adéquate, à travers la table.

Une méthode est déclarée comme virtuelle en la faisant précéder du mot clé **virtual**.

```
class interface
{
public:
    virtual void affiche() { }
    virtual void deplace() { }
    . . . . .
};

class menu: public interface
{
public:
    menu();
    ~menu();
    void affiche();
    void deplace();
};

class alerte: public interface
{
public:
    alerte();
    ~alerte();
    void affiche();
    void deplace();
};
```

```
int main()
{
    interface *ptr;             // ptr pointeur sur interface
    menu *ptr1;
    alerte *ptr2;
    ptr2 = new alerte(. . .);
    ptr = ptr2;                 // ptr pointe sur une alerte
    . . . . .
    ptr -> affiche();           // appel fonction affiche de alerte
}
```

NB1: Les fonctions virtuelles doivent **avoir un corps dans la classe mère** même si celui-ci n'est pas utilisé (couple d'accolades éventuellement vide)

NB2: Les fonctions virtuelles doivent avoir exactement le même prototype dans la classe mère et dans les classes dérivées sinon elles sont considérées comme différentes (surcharge de fonction) et le mécanisme de liaison dynamique n'est pas utilisé.

8.3.3.1 Méthodes virtuelles pures, classes abstraites

Les méthodes virtuelles vues plus haut ont une implémentation définie dans la classe mère (éventuellement { }). La classe mère peut donc avoir des instances au même titre que les classes dérivées.

Il peut arriver qu'une classe n'existe que pour mettre en facteur des caractéristiques communes à un certain nombre de classes dérivées, sans qu'elle corresponde à un concept dont il peut exister des représentants, la classe mère n'aura donc jamais d'instances. une telle classe est dite **classe abstraite**.

Une méthode peut être **virtuelle pure**, c'est à dire qu'elle n'a pas d'implémentation définie dans la classe mère. Elle doit obligatoirement être redéfinie dans les classes héritières. La classe mère ne peut donc pas avoir d'instances, elle est abstraite.

Pour déclarer virtuelle pure une méthode on fait précéder son prototype du mot clé virtual et on le fait suivre de '= 0'

```
class mere                          // classe abstraite
{
    . . . . .
public:
    virtual void f(int x) = 0;      // f() est virtuelle pure
    . . . . .
};
class derivee: public mere
{
    . . . . .
public:
    void f(int x) { . . . . . }  // redéfinition de f() dans dérivée
        . . . . .
};
```

NB: Une classe abstraite contient au moins une méthode virtuelle pure

8.4 Pointeurs, polymorphisme et gestion de mémoire dynamique

La gestion de mémoire dynamique en utilisant les opérateurs new et delete est en général mal maîtrisée par les programmeurs débutants, c'est pourquoi on a développé la **bibliothèque STL** qui propose des objets simples à utiliser en masquant les problèmes de gestion de mémoire.

L'utilisation systématique des classes de bibliothèque (string, list, vector, etc.) permet de s'affranchir de tous les problèmes de gestion de la mémoire dynamique (réservation par new, libération par delete), elle permet aussi d'effectuer simplement des recopies et des comparaison à l'aide des opérateurs =, ==, != <, >, etc.

Cette programmation plus facile grâce à l'utilisation de la bibliothèque STL est remise en cause par le polymorphisme. En effet le polymorphisme nécessite l'utilisation de pointeurs pour sa mise en œuvre. Si l'on reprend l'exemple des classes interface, alerte, menu et texte du début du chapitre, on constate qu'un tableau polymorphe doit être un tableau de pointeurs sur interface. Chaque case du tableau pourra alors contenir un pointeur sur menu, alerte ou texte.

Le fait de manipuler explicitement des données à travers des pointeurs conduit à utiliser les opérateurs new et delete pour respectivement réserver et libérer de l'espace dans la zone de mémoire dynamique.

Chapitre 9
Généricité

9.1 Fonctions génériques

9.1.1 Problème posé

Un programmeur doit souvent réaliser un logiciel pour effectuer un même traitement sur des données de types divers (tri d'*int*, tri de *double*, etc.). Pour cela il utilise le même algorithme mais comme les types de données sont différents un seul programme ne peut pas suffire, plusieurs solutions sont alors envisageables :

- l'écriture de plusieurs fonctions différentes, mais le problème se pose des noms différents (résolu par la surcharge), des versions différentes, de la cohérence des évolutions, de la maintenance, etc.
- l'écriture d'une fonction avec des arguments pointeurs génériques, mais il n'y a pas de contrôle de type et la conversion des pointeurs est à la charge du programmeur.
- l'écriture d'une macro générique avec le type de données comme paramètre, c'est alors le préprocesseur que génère automatiquement les différentes versions de la fonction, cependant il n'y a pas contrôle du compilateur. C'est typiquement ce que les programmeurs faisaient en langage C mais que l'on cherche à éviter pour des raisons évidentes de qualité du logiciel.
- la constitution d'une hiérarchie de classes dont les feuilles correspondent aux différents types de données, mais le problème se pose de la maintenance des diverses implémentations de la fonction dans les classes dérivées.

On voit donc que l'on aimerait avoir la souplesse fournies par les macros pour paramétrer les algorithmes tout en gardant les possibilités de contrôle de type à la compilation. C'est ce que propose le mécanisme des fonctions génériques et l'instanciation automatique pour divers types grâce au mécanisme fourni par C++.

9.1.2 Mise en œuvre d'une fonction générique

9.1.2.1 Définition d'une fonction générique

Un fonction générique est introduite par la séquence **template<class T>** où T représente un paramètre de type. La définition de la fonction est ensuite classique si ce n'est que le type à paramétrer est remplacé par T.

```
template<class T>        // fonction maximum générique
T maxi(T x, T y)
{
    return (x > y ? x : y);
};
```

La fonction maxi renvoie le maximum de deux valeurs d'un type T à spécifer au moment de l'utilisation. Cette fonction est un modèle (template) qui servira de base pour la génération automatique de versions adaptées à des types précis.

NB1: Le texte de la fonction générique doit avoir un sens pour tous les types susceptibles de remplacer T. Ici, l'opérateur > doit être défini pour ces types.

NB2: On peut avoir plusieurs paramètres de généricité comme T, chacun doit apparaître au moins une fois dans les paramètres de la fonction :

```
template<class T, class X>      // fonction générique parametrée
int f(T a, X b)                 // par deux types
{ . . . . . };
```

9.1.2.2 Instanciation et utilisation d'une fonction générique

```
class Truc
{
    . . . . .
public:
    bool operator> (Truc, Truc);
};
. . . . .
int main ()
{
    Truc a,b,c;
    int x,y,z;
    . . . . .
    a = maxi(b,c);       // maxi de deux Truc
    x = maxi(y,z);       // maxi de deux int
}
```

9.1.2.3 Instances explicites d'une fonction générique

Dans le cas où la fonction générique ne peut pas être appliquée à un type particulier, on peut fournir une instance particulière de la fonction pour le type en question. Dans le cas de la fonction maxi ci-dessus l'opérateur > ne s'applique pas aux chaînes de caractères (il faut utiliser strcmp) on fournit donc l'instance spécialisée pour les char*

```
char * maxi(char *x, char *y)
{
    return ( (strcmp(x,y)>0) ? x : y);
}
```

Lors d'un appel de fonction, il y a d'abord recherche parmi les instances explicites, puis si aucune ne convient, il y a recherche d'une fonction générique. Quand il existe une instance explicite, les conversions implicites de paramètres sont effectuées. Dans le cas d'instanciation automatique d'une fonction générique ces conversions ne sont pas mises en oeuvre.

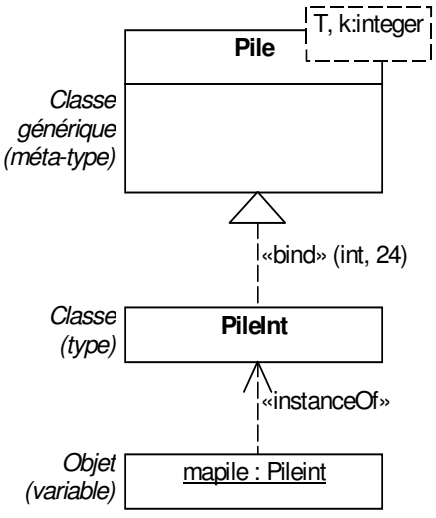
9.2 Classes génériques

9.2.1 Problème posé

On a souvent à réaliser des structures de données s'appuyant sur des types variés (pile d'*int*, pile de *double*, etc.) il serait donc intéressant d'écrire une seule fois un classe générique pile et de la spécialiser pour des int, des double, ou d'autres types.

Il est possible de mettre en oeuvre des solutions diverses dans l'esprit de celles proposées pour les fonctions. Les problèmes posés étant pratiquement les mêmes, C++ propose la notion de classe générique (modèle de classe) que l'on peut instancier pour divers types de données.

On peut considérer les classes génériques comme des méta-types : par analogie avec les instances de classes, on peut dire que l'on peut "instancier" des classes génériques pour former des classes véritables qui à leur tour sont instanciables sous forme d'objets.



9.2.2 Mise en oeuvre d'une classe générique

9.2.2.1 Définition d'une classe générique

Une classe générique est introduite par la séquence **template<class T>** où T est un paramètre représentant le type générique. La suite de la définition de classe est classique mais avec utilisation de T comme type générique. L'exemple ci-dessous correspond à une pile générique. Comme pour les fonctions, il est posssible d'avoir plusieurs paramètres de type dans la définition d'une classe générique.

```
template<class T>
class Pile
{
    T *lapile;
    int taille;
    int top;
public:
    Pile(int t)   {  lapile = new T[taille = t]; top = -1; }
    ~Pile()   {  delete[] lapile; }
    void empile(const T &e)   {  lapile[++top] = e; }
    T& depile()   {  return[top--]; }
};
```

NB1: Une classe générique peut être paramétrée par plusieurs types, ces types qui apparaissent entre les <> séparés par des virgules doivent figurer dans la définition de la classe.

NB2: Il faut que les opérations utilisées dans la classe générique aient un sens pour tous les types utilisés. Ici l'opérateur = doit être défini pour tous les types qui remplaceront T dans les instances de la classe générique.

9.2.2.2 Instanciation et utilisation d'une classe générique

Pour instancier une classe générique pour un type particulier on fait suivre son nom du type entre <>.

```
int main(void)
{
    Pile<int> p1(10);    // définition d'une pile de 10 int
    Pile<double> p2(5) ; // définition d'une pile de 5 double
    . . . . .
    p1.empile(20);
    p2.empile(5.78);
    . . . . .
    y = p2.depile();
    . . . . .
}
```

L'utilisation de typedef permet d'alléger les notations

```
typedef Pile<int> Pileint;
Pileint p21(20);
```

9.2.2.3 Méthodes d'une classe générique à l'extérieur de la classe

Les méthodes d'une classe générique peuvent ne pas être inline. On les définit alors à l'extérieur de la classe, ce qui oblige à utiliser la syntaxe (lourde) suivante :

```
template<classT> void pile<T>::empile(const T &e)
{
    lapile[++top] = e;
}
```

NB: Les opérations mises en jeu dans la programmation d'une classe générique (opérateurs, fonctions) doivent avoir un sens pour tous les types que l'on désire utiliser, ce qui nécessite généralement des surcharges des opérateurs =, >, <, ==, etc.

9.2.2.4 Paramétrage d'une classe par des variables

Une classe générique peut être paramétrée par des types et par des variables :

```
template<class T, int t> class Pile
{
    T *lapile;
    int taille;
    int top;
public:
    Pile()   {  lapile = new T[taille = t]; top = -1; }
    ~Pile()  {   delete[] lapile; }
    void empile(const T & e)   {  lapile[++top] = e; }
    T& depile()   {  return[top--]; }
};
. . . . .
Pile<int,20> mapile;           // création d'une pile de 20 int
```

NB: La classe Pile est paramétrée par le type T et par la variable entière t, dans cet exemple la variable t est utilisé dans le constructeur pour spécifier la taille de la Pile.

9.2.2.5 Instances explicites d'une classe générique

L'instanciation d'une classe générique peut être impossible pour certains types (opérateurs sans signification, traitements spécifiques nécessaires, etc.). Le programmeur peut alors introduire des instances spécifiques de la classe.

```
class Pile<char*>
{
    . . . . .
    // traitements spécifiques
    . . . . .
};
```

NB: Les instances explicites sont d'abord prise en compte avant d'effectuer des instanciations automatiques.

9.2.2.6 Classe générique et fonction amie

Une fonction friend qui fait référence à un paramètre de généricité doit elle aussi être générique. Une fonction friend générique n'est amie que des instances de la classe générique qui ont les mêmes paramètres de généricité.

9.2.2.7 Classe générique et attribut de classe

Les attributs de classe, déclarés avec le mot clé static sont communs à tous les objets instances d'une instance particulière d'une classe générique.

9.2.3 La bibliothèque standard C++

La bibliothèque standard C++, STL (Standard Template Library) est, comme son nom l'indique, entièrement basé sur les classes génériques.

Exemple:

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

list<int> l;       // l instance de la classe list<int> générée
                   // à partir de la classe générique list de la STL
```

Chapitre 10
Traitements des exceptions

10.1 Problème posé

Quand une erreur se produit pendant l'exécution d'un programme, plusieurs stratégies des traitement sont envisageables :

- Le traitement du problème dans le programme qui détecte l'erreur, mais l'erreur et son traitement doivent être prévues au moment de l'écriture du code ce qui donne des programmes lourds et complexes.
- L'émission d'un message d'erreur et l'arrêt du programme, mais cette solution n'est peut être pas souhaitable pour des systèmes de contrôle (vols habités, pilotage d'usine) pour lesquels l'utilisateur ne pourrait que constater l'arrêt du programme.
- Le signalement de l'erreur détectée dans une fonction par un code de retour que le programme appelant peut traiter comme il le veut, mais le programmeur n'est pas obligé de tester le code d'erreur. Par ailleurs une fonction a généralement déjà une valeur de retour représentant le résultat de l'opération, il est alors difficile d'avoir en plus un code d'erreur.

Une solution plus élégante est introduite par le traitement des exceptions qui apporte toute la souplesse nécessaire. Avec le mécanisme d'exception que l'on trouve dans tous les langages modernes (C++, Java, Ada, etc.) on dispose de trois éléments permettant de traiter facilement les erreurs, ce sont : les séquences de gestion d'erreur (**traitement d'exception**), le mécanisme de surveillance d'erreur (**capture d'exception**) et le mécanisme de signalisation d'erreur (**levée d'exception**). Le principe des exception est de séparer complètement la détection de l'erreur (levée et capture) de son traitement :

- Les **traitements d'exception** sont des séquences associées à un type d'erreur (type pris au sens de type de données, c'est à dire type de base ou classe). Une séquence est exécutée quand une erreur a été détectée et que le mécanisme de surveillance d'exception a fait le lien avec un gestionnaire d'exception.
- Le mécanisme de **capture d'exception** permet la surveillance d'une séquence de code délimitée par des mots clés, elle associe à chaque type d'erreur possible une séquence de traitement d'erreur. Quand il y a erreur, le déroulement normal du programme est arrêté et il y a branchement automatique à la séquence de traitement correspondante. La séquence de traitement n'est pas considérée comme un sous programme mais plutôt comme un traitement alternatif du traitement normal.
- Le mécanisme de **levée d'exception** est mis en œuvre soit automatiquement par le matériel ou par le système d'exploitation, soit explicitement par le programmeur. Quand le matériel ou le système détecte une erreur du genre "division par zéro" il lève une exception. Si la séquence d'instructions était surveillée et qu'il y a un traitement d'exception, ce traitement est activé. Si dans un programme on détecte une erreur on peut lever un exception à l'aide d'une instruction spéciale du langage. Cette exception sera traitée si le code était surveillé et s'il existe une séquence de traitement adaptée

10.2 Mise en œuvre

10.2.1 Association de traitements d'exceptions à une séquence d'instructions

L'instruction **try catch** permet de contrôler l'exécution d'une séquence d'instructions en spécifiant des gestionnaires d'exception (**handlers**) pour traiter un certain nombre d'erreurs pouvant survenir.

Chaque gestionnaire possède un argument qui spécifie l'exception prise en compte, c'est le type de l'exception qui permet de différencier les gestionnaires. A un type d'erreur particulier est associé un type d'exception et donc un gestionnaire particulier d'exception. Une exception est un objet de type quelconque (type de base ou classe définie par l'utilisateur).

La syntaxe de try catch est illustrée par l'exemple suivant :

```
try
{
    . . . . .
    . . . . .                    // instructions C++ à contrôler
    . . . . .
}
catch(exception1) { traitement1 }      // gestionnaire d'exception1
catch(exception2) { traitement2 }      // gestionnaire d'exception2
catch(exception3) { traitement3 }      // gestionnaire d'exception3
. . . . .
catch(exceptionN) { traitementN }      // gestionnaire d'exceptionN
catch(...)  { traitement commun }      // gestionnaire de toutes
                                       // les exceptions
```

NB1: Les instructions à surveiller sont dans la paire d'accolades suivant le mot clé try et chaque gestionnaire d'exception est introduit par le mot clé catch.

NB2: Si une erreur d'exécution est détectée dans la séquence surveillée, une exception sera "levée" ce qui passera automatiquement le contrôle au traitement approprié dans le gestionnaire correspondant à cette exception.

NB3: Un gestionnaire d'exception introduit par catch suivi de 3 points entre parenthèses traite toutes les exceptions.

10.2.2 Traitement d'une exception

Quand une exception est levée (voir §10.2.3) dans une séquence d'instructions située entre try et catch ou dans les fonctions appelées par cette séquence, il y a abandon de la séquence et passage direct aux traitements d'exception introduits par catch. Les différents catch sont essayés dans l'ordre où ils sont écrits et le premier catch de même type que l'exception est exécuté. Si le traitement d'exception ne met pas fin au programme le contrôle est passé à l'instruction suivant les traitements d'exception.

S'il n'y a pas d'exception levée dans la séquence surveillée, les catch sont sautés et l'exécution se poursuit.

10.2.3 Levée d'exception

Dans une séquence sous surveillance, lors d'une détection d'erreur, on peut lever une exception pour que le contrôle soit passé à un gestionnaire approprié, s'il existe. La levée d'une exception s'effectue à l'aide de l'instruction **throw**, avec comme paramètre un objet exception, c'est à dire une constante, une variable ou un type.

```
Truc t;                          // objet  de type Truc
. . . .
if(erreur1) throw 4;             // levée exception int
. . . .
if(erreur2) throw t;             // levée exception Truc
```

NB1: Dans le cas de la première erreur le contrôle est passé à un gestionnaire d'exception de type int, s'il existe à la suite de la séquence surveillée par try.

NB2: Dans le cas de la deuxième erreur le contrôle sera passé à un gestionnaire d'exception de type Truc, s'il existe à la suite de la séquence surveillée par try.

10.2.4 Levée d'exception dans une fonction

On peut spécifier dans l'entête d'une fonction les types d'exceptions qu'elle a le droit de lever. Ceci permet de placer des levées d'exception pour un certain nombre d'erreurs dans une séquence de code puis de se limiter à un plus petit nombre selon le contexte d'exécution de la séquence. L'entête de la fonction indique les exceptions autorisées à l'aide du mot throw suivi des types d'exception entre parenthèses.

```
class divzero            // une classe exception
{
     . . . . .
};

void div(int n, int m) throw(int, float, divzero)
{
    . . . . .
}
```

NB: La fonction div ne peut lever que des exceptions int, float et divzero

En l'absence d'une telle déclaration concernant les exceptions, la fonction peut lever tout type d'exception :

```
void div(int n, int m)
{
    . . . . .
}
```

On peut refuser à une fonction la levée de toute exception en mettant le mot throw suivi de parenthèses vides dans l'entête de la fonction:

```
void div(int n, int m) throw()
{
    . . . . .
}
```

10.3 Exemples

10.3.1 Exemple complet de traitement d'exception

On écrit ici une classe pile et on prévoit dans les fonctions membres empile et depile de lever une exception de type chaîne de caractères si la pile est vide dans depile et si elle est pleine dans empile. L'exception est une chaîne qui contient libellé de l'erreur.

```
class pile
{                                    // pile d'int de taille 3
private:
    int lapile[3];
    int top;
public:
    pile()  {  top = -1; }
    void empile(int e)
    {
        if (top < 2)
            lapile[++top] = e;
        else
            throw "saturation";      // levée d'une exception de
    }                                // type chaîne de caractères
    int depile()
    {
        if (top>-1)
            return lapile[top--];
        else
            throw "vacuité";         // levée d'une exception de
    }                                // type chaîne de caractères
};
```

Dans le programme utilisateur de la classe pile on définit une instance de pile et on l'utilise. On place sous surveillance la séquence d'appels à empile et on fournit un gestionnaire pour traiter l'erreur.

```
int main()
{
    pile mapile;
    int i=10, j=50;
    try                                 // début surveillance
    {
        mapile.empile(i);
        mapile.empile(j);
        mapile.empile(i);
    }                                   // fin surveillance
    catch (char *s)                     // gestionnaire d'erreur
    {
        cout << "erreur pile : " << s << endl;
    }
};
```

10.3.2 Exemple complet avec classes exceptions imbriquées

```
class pile                              // pile d'int de taille 3
{
    int lapile[3];
    int top;
public:
    class saturation            // classe imbriquée exception
    {
    public:
        int val;
        saturation(int i) { val = i;}
    };
    classe vacuite              // classe imbriquée exception
    {
    public:
        vacuite(){ }
    };
    pile()  {  top = -1; }
    void empile(int e)
    {
        if (top < 2) lapile[++top] = e;
        else         throw saturation(e);  // levée d'exception saturation
    }
    int depile()
    {
        if (top>-1)  return lapile[top--];
        else         throw vacuite();    // levée d'exception vacuité
    }
};
```

```
int main()
{
    pile mapile;
    int i = 10, j = 50,x;
    try
    {
        mapile.empile(i);
        mapile.empile(j);
        . . . . .
        x = mapile.depile();
    }
    catch (pile::saturation &s)
    {
        cout << "saturation en tentant d'empiler " << s.val << endl;
    }
    catch (pile::vacuite &s)
    {
        cout << "pile vide" << endl;
    }
}
```

10.4 Imbrication de séquences de surveillance d'erreur

Il est possible d'avoir des imbrications de séquences contrôlées par try catch avec remontée d'une exception d'une structure try catch vers une structure try catch englobante.

Dans un gestionnaire d'exception introduit par catch on peut lever une autre exception ou passer la même exception à un gestionnaire englobant. La levée d'une autre exception s'effectue en utilisant throw suivi de la nouvelle exception entre parenthèses. Le passage de l'exception en cours de traitement à un gestionnaire englobant s'effectue en utilisant **throw sans opérande**.

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

void f(int x)
{
    cout << "sequence try catch de f" << endl;
    try
    {
        if(x == 1) throw 1;
        if(x == 2) throw 3.14f;
        cout << "fin de sequence try catch de f" << endl;
    }
    catch(int v)
    {
        cout << "traitement exception int de f" << endl;
    }
    catch(float v)
    {
        cout << "traitement exception float de f" << endl;
        throw;
    }
    cout << "apres sequence try catch de f" <<endl;
    return;
}
```

NB1: Dans la fonction f on surveille les valeurs de x et on lève une exception int ou float selon que x vaut 1 ou 2.

NB2: Le traitement de l'exception de type float passe l'exception à un gestionnaire englobant (s'il existe).

```
int main()
{
    int a;
    cout << "entrez a"; cin >> a;
    cout << "sequence try catch de main" << endl;
    try
    {
        f(a);
        cout << "fin sequence try catch de main" << endl;
    }
    catch(int v)
    {
        cout << traitement exception int de main" << endl;
    }
    catch(float v)
    {
        cout << traitement exception float de main" << endl;
    }
    catch(...)
    {
        cout << "traitement de toutes les exceptions de main" << endl;
    }
    cout << "apres sequence try catch de main" << endl;
}
```

NB1: Dans la fonction main on surveille une séquence d'instructions comportant un appel à la fonction f qui est elle même surveillée.

NB2: catch(...) est un gestionnaire pour tout type d'exception.

L'exécution du programme ci-dessus donne les résultats :

```
entrez a : 0
sequence try catch de main
sequence try catch de f
fin sequence try catch de f
apres sequence try catch de f
fin sequence try catch de main
apres sequence try catch de main
```

NB: La variable a vaut 0 on traverse donc main et la fonction f appelée sans lever d'exception

```
entrez a : 1
sequence try catch de main
sequence try catch de f
traitement exception int de f
apres sequence try catch de f
fin sequence try catch de main
apres sequence try catch de main
```

NB: La variable a vaut 1, on entre dans la séquence surveillée de main, puis dans la séquence surveillée de f. Une exception de type int est levée dans f, le traitement d'exception int de f est exécuté puis on revient dans main qui finit normalement son exécution sans exception.

```
entrez a : 2
sequence try catch de main
sequence try catch de f
traitement exception float de f
traitement exception float de main
apres sequence try catch de main
```

NB: La variable a vaut 2, on entre dans la séquence surveillée de main, puis dans la séquence surveillée de f. Une exception de type float est levée dans f, le traitement d'exception float de f est exécuté et il relève la même exception. Le traitement d'exception float de main est alors exécuté.

Annexe A
Introduction à UML

A.1 Pourquoi UML

UML (Unified Modeling Language = "langage de modélisation objet unifié") résulte de la fusion de trois méthodes qui ont le plus influencé la modélisation objet au milieu des années 90 : OMT, Booch et OOSE. Il a été normalisé en 1997 par l'OMG (Object Management Group) qui est aussi à l'origine de CORBA.

UML est un langage graphique (supporté par un méta-modèle) et qui propose un ensemble de vues (appelées diagrammes) modélisant les aspects statiques et dynamiques des applications.

Les diagrammes sont nombreux et se divisent en 2 grandes catégories:

Les diagrammes structurels (vues statiques d'un système) :
- Les diagrammes de classes
- Les diagrammes d'objets
- Les diagrammes de composants
- Les diagrammes de déploiement

Les diagrammes comportementaux (vues dynamiques d'un système)
- les cas d'utilisation
- le diagramme de collaboration
- le diagramme de séquence
- le diagramme d'états-transitions
- le diagramme d'activités

Dans le cadre de ce cours nous utiliserons essentiellement les diagrammes de classes (et un peu les diagrammes d'objets).

UML ne propose pas de méthodologie d'analyse et de conception en propre. Cependant il peut les supporter toutes ! Parmi celles-ci nous citerons RUP (Rational Unified Process) et sa version "éducation", UPEDU qui est abordée dans un cours d'approfondissement.

A.2 Diagramme de Classes

Un diagramme de classes est une collection d'éléments de modélisation qui montre la structure d'un modèle. Il ne tient pas compte des aspects dynamiques et temporels.

Pour un modèle complexe, plusieurs diagrammes de classes complémentaires doivent être construits. Il est donc tout à fait possible qu'une même classe apparaisse sur plusieurs diagrammes avec différents niveaux de détails.

Dans un système donné, certaines classes peuvent avoir 0, 1, quelques ou de nombreuses instances. Pour spécifier certains aspects, il peut donc être nécessaire d'instancier certeines classes à titre d'exemple voire de construire des diagrammes d'objets (Cf. §A.3)

Remarque importante : seuls les éléments d'UML utilisés dans le cadre de ce cours sont présentés ici. Il existe de nombreux détails ou compléments qui sont délibérément passés sous silence !

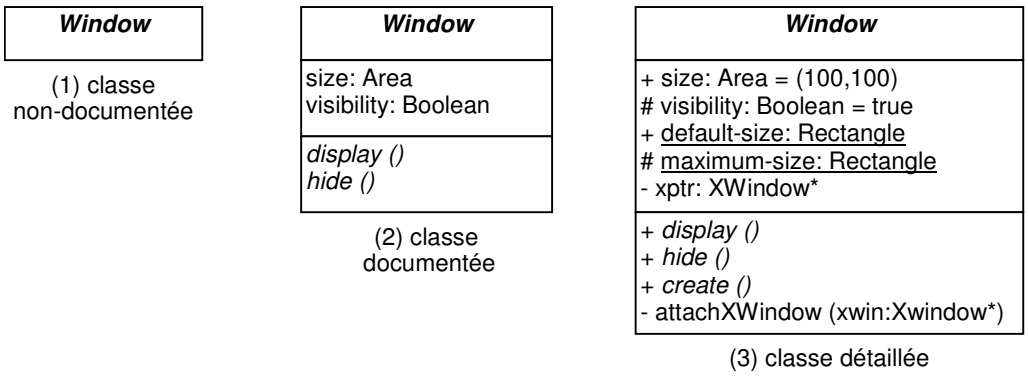
A.2.1 Les classes

Les classes ont 3 représentations possibles selon le niveau de détail souhaité:
- (1) **classe non-documentée** (details suppressed) : seul le nom de la classe apparaît dans un compartiment unique.
- (2) **classe documentée** (analysis-level details) : il y a deux compartiments supplémentaires qui comportent respectivement les attributs et les méthodes de la classe. Seuls les noms et les types des attributs sont précisés. Les noms des méthodes apparaissent mais pas les paramètres ni le type retourné. Ce niveau de détail ne cherche pas l'exhaustivité tant dans la

liste des attributs que des méthodes ; pour bien le montrer, il est possible de faire suivre la liste par des points de suspension (...).

- (3) **classe détaillée** (implementation-level details) : dans cette représentation on cherche à être exhaustif. Le niveau de protection des attributs et des méthodes est spécifié. Les paramètres des méthodes sont détaillés et le type de retour (s'il existe) précisé.

Le nom de la classe est écrit en gras. Pour les classe abstraites, le nom est écrit en italiques.



Remarque: on conseille généralement de commencer le nom des classes par une majuscule et le nom des attributs et des méthodes par une minuscule

Attributs :

Les attributs sont décrits par :

- La visibilité : public ('+'), privé ('-') ou protégé ('#') ; non-présentée dans les classes documentées
- Le nom de l'attribut : un identificateur
- Le symbole ':'
- Le type de l'attribut: soit un type primitif, soit un type construit.
- Une multiplicité (facultative) : nombre ou intervalle placé entre crochets '[' et ']'
- Le symbole '=' suivi d'une valeur par défaut pour cet attribut (facultatif)

Les attributs de classe sont soulignés (membres statiques en C++)

NB: pour les types primitifs nous utiliserons les types de C++ car c'est notre langage d'implémentation.

Méthodes :

Les méthodes sont décrites par :

- La visibilité : public ('+'), privé ('-') ou protégé ('#')
- Le nom de la méthode : un identificateur
- Une liste d'arguments (ou paramètres formels) placés entre parenthèses '(' et ')'
- Le symbole ':' suivi du type retourné par la méthode (omis si pas de valeur de retour)

Les méthodes abstraites (c-à-d non implémentées dans la classe) sont écrites en italiques.

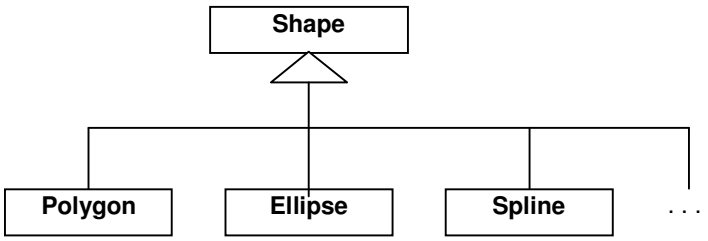
Arguments d'un méthode :

Les arguments sont :

- La direction de l'argument: 'in' (en entrée), 'out' (en sortie), 'inout' (en entrée/sortie). Si ce n'est pas précisé alors la valeur par défaut est 'in'.
- Le nom de l'argument : un identificateur
- Le symbole ':'
- Le type de l'argument: soit un type primitif, soit un type construit.
- Le symbole '=' suivi d'une valeur par défaut pour cet argument (facultatif)

A.2.2 Héritage de classes

L'héritage entre classes est notée en UML par une relation de généralisation : ligne pleine terminée par une flèche triangulaire évidée qui pointe vers la classe parent. L'héritage multiple est possible même s'il est souvent déconseillé. Des points de suspension peuvent être utilisés pour montrer qu'il existe d'autres classes dérivées qui ne sont pas représentées sur le diagramme.



A.2.3 Associations entre classes

Les associations binaires sont représentées sous forme de trait plein entre deux classes. Un certain nombre de décorations peuvent être ajoutées pour préciser leurs propriétés. Il existe également des association ternaires (ou de plus haut degré) qui ne seront pas vues dans ce cours.

Les associations peuvent être réflexives. Dans ce cas, il s'agit le plus souvent d'un lien entre 2 instances distinctes de la même classe.

L'association peut recevoir **un nom** qui est placé près de la ligne mais pas vers les extrémités (pour ne pas le confondre avec les rôles). Ce nom sert juste à préciser la sémantique de l'association. Un petit triangle noir peut également préciser le sens de lecture de ce nom.

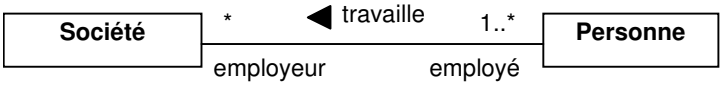
Navigabilité restreinte d'une association :

Par défaut une association est navigable dans les deux sens. Si la navigabilité est restreinte à un seul sens, celui-ci est précisé en mettant une flèche simple du côté de la classe destination (Cf. §A.2.4).

Décorations d'une association :

Une association peut avoir des décorations à chacune de ses extrémités :

- **une cardinalité** : elle spécifie le nombre d'instances possibles de la classe placée à cette extrémité pour cette association. Cela peut être un nombre exact (0, 1, 2, ...) ou un intervalle (0..1, 0..3, 1..2). Il est également possible d'utiliser le symbole '*' pour un nombre indéterminé (exemple: * ou 0..*), ou dans un intervalle sans borne supérieure (1..*, 2..*).
- **un rôle** : c'est un nom qui exprime le rôle joué par la classe qui est placé à cette extrémité de l'association. Ce rôle peut être qualifié par une visibilité ('+' = public, '-' = privé, '#" = protégé)

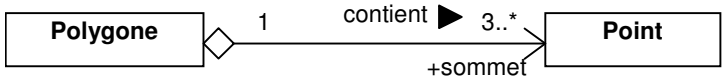


Cet exemple se lit de cette manière:
- les instances de la classe Personne sont liées à plusieurs instances (éventuellement aucune) de la classe Société qui jouent le rôle d'employeur.
- les instances de la classe Société sont liées à une ou plusieurs instances de la Classe Personne qui jouent le rôle d'employé.

A.2.4 Agrégation

L'agrégation est une forme d'association où l'on précise que l'instance de l'une des classes est un agrégat d'instances de l'autre classe. Elle se note par un petit losange évidé du côté de la classe agrégat.

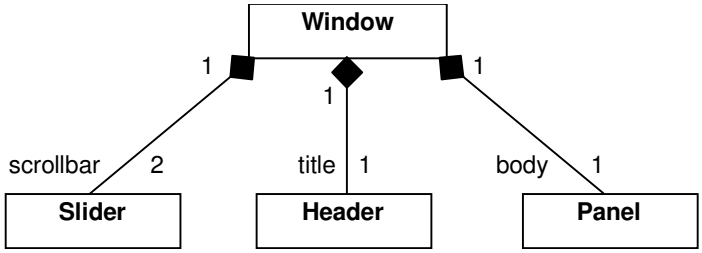
La cardinalité du côté de l'agrégat est généralement de 1 mais il est tout à fait possible que des instances de la classe agrégée soient partagées par plusieurs agrégats.



Cet exemple se lit de cette manière:
Un polygone est l'agrégation de 3 points ou plus qui jouent le rôle de sommet pour ce polygone. Les points ne peuvent pas "voir" le polygone auquel ils participent car le lien est à navigabilité restreinte.

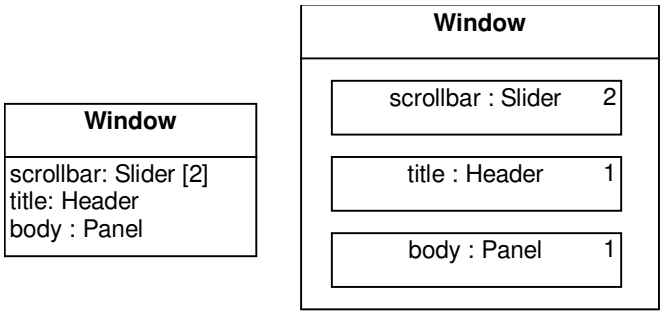
A.2.5 Composition

La composition est une forme d'agrégation forte où les instances des classes composants ne peuvent appartenir qu'à une seule instance de la classe composée (cardinalité 1 au maximum du côté de la classe composée). Elle se note comme une association en ajoutant un petit losange plein du côté de la classe composée.



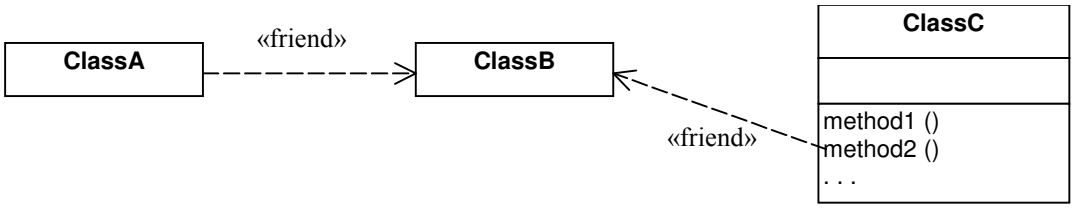
La composition est très forte est implique que les instances de la classe composée ont la même durée de vie que la classe composante.

Il existe également deux autres manières de la représenter comme le montre l'exemple ci-dessous.



A.2.6 Relations de dépendance

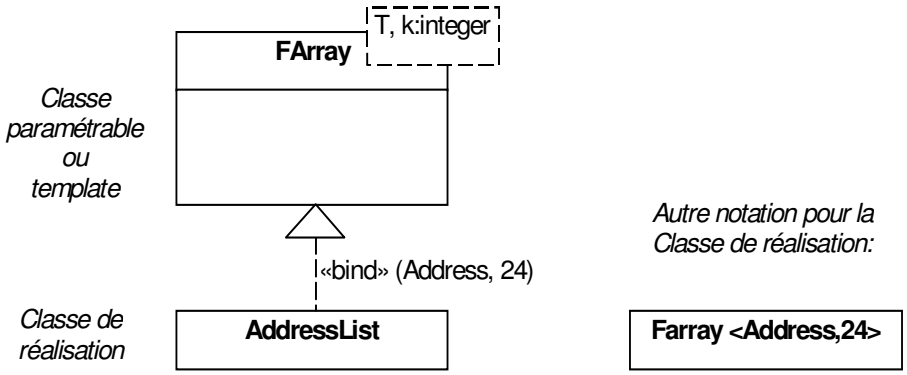
La relation de dépendance se note avec un trait pointillé terminée par une flèche simple. Un mot-clé placé entre guillemets ('«' et '»') à côté du trait précise la nature de la dépendance. Parmi toutes les relations possibles nous ne parlerons ici que de la dépendance «friend». Un autre dépendance est présentée au §A.3.4 («instanceOf»).



A.2.7 Classes paramétrables (templates)

Un **template** est une classe paramétrée par des paramètres formels. Il définit une famille de classes, chaque classe étant spécifiée en instanciant les paramètres formels par des valeurs. De manière classique les paramètres représentent des types des attributs de la classe ; cependant ils peuvent aussi être des entiers, d'autres types et voire même des méthodes.

Un template se définit comme une classe en ajoutant dans le coin haut-droit un rectangle en traits pointillés qui contient la liste des paramètres du template.



Les paramètres sont définis par :
- Le nom du paramètre: un identificateur
- Le symbole ':' suivi du type du paramètre (facultatif)
- Le symbole '=' suivi d'une valeur par défaut pour ce paramètre (facultatif)

Le type du paramètre peut être omis. Dans ce cas, le paramètre est supposé être une classe ou un type de donnée.

Un template ne peut-être utilisé directement. Ses paramètres doivent être instanciés pour pouvoir définir des classes appelées **classes de réalisation**.

La classe de réalisation se note en mettant le nom du template suivi de la liste des paramètres effectifs placés entre '<' et '>'.

Un template ne peut pas être directement utilisé dans des relations d'héritage ou d'association sauf dans deux cas particuliers :
- pour une relation d'héritage, le template peut hériter d'une classe ordinaire,
- pour une association, le template peut être la source d'une association à navigabilité restreinte.

A.3 Diagrammes d'objets

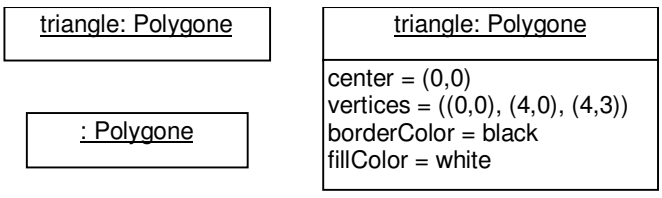
Le diagramme d'objets contient des instances de classes (objets) et leurs relations (appelés liens). Il peut être vu comme un instantané de l'état du système permettant de préciser des détails qui n'apparaissent pas clairement sur le diagramme de classes. Il permettent essentiellement de présenter des exemples et sont donc d'un usage assez limité.

En pratique des objets peuvent être mélangés à un diagramme de classes et on ne parle de diagramme d'objets qui s'il ne contient que des objets et pas de classes.

A.3.1 Objets

Les objets ont une représentation proche des classes. Le premier compartiment comporte :
- le nom de l'objet : un identificateur (facultatif)
- le symbole ':' (obligatoire !),
- le nom de classe dont l'objet est l'instance.

L'ensemble est souligné pour bien montrer qu'il s'agit d'une instance de la classe.



Le second compartiment (facultatif) contient la liste des attributs et des valeurs associées :

- le nom de l'attribut,
- le symbole ':' suivi du type de l'attribut (facultatif car redondant avec la définition de la classe)
- le symbole '=' suivi de la valeur de l'attribut.

Il n'y a pas de compartiment prévu pour les méthodes puisqu'elles sont déclarées dans la classe.

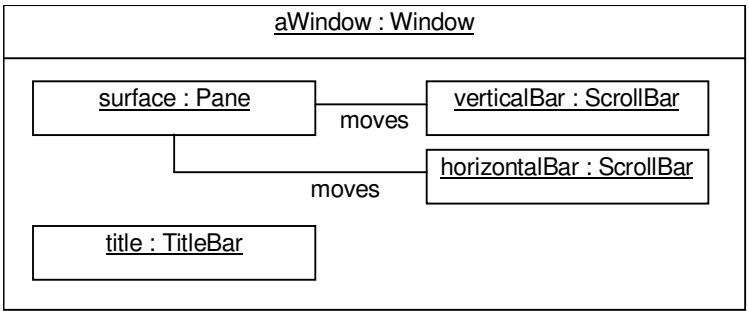
A.3.2 Liens

Un **lien** est une relation entre deux objets. Comme nous l'avons vu dans le diagramme de classes, les associations (comme les agrégations et les compositions) sont représentées sur ce diagramme comme des relations entre des classes. En pratique, ce sont les instances de ces classes qui sont reliées à travers ces associations. On peut donc dire finalement que:

Un objet est une instance d'une classe
Un lien est une instance d'une association

A.3.3 Objets composites

Un objet composite est une instance d'un classe composite, ce qui implique qu'il existe des relations de composition entre la classe et ses composants.



A.3.4 Relations "instanceOf"

Sur une diagramme mélangeant des classes et des objets il peut être nécessaire de faire apparaître les relations d'instances. Elle est représentée comme une relation de dépendance entre un objet et sa classe avec comme libellé «instanceOf».