

## Sommaire

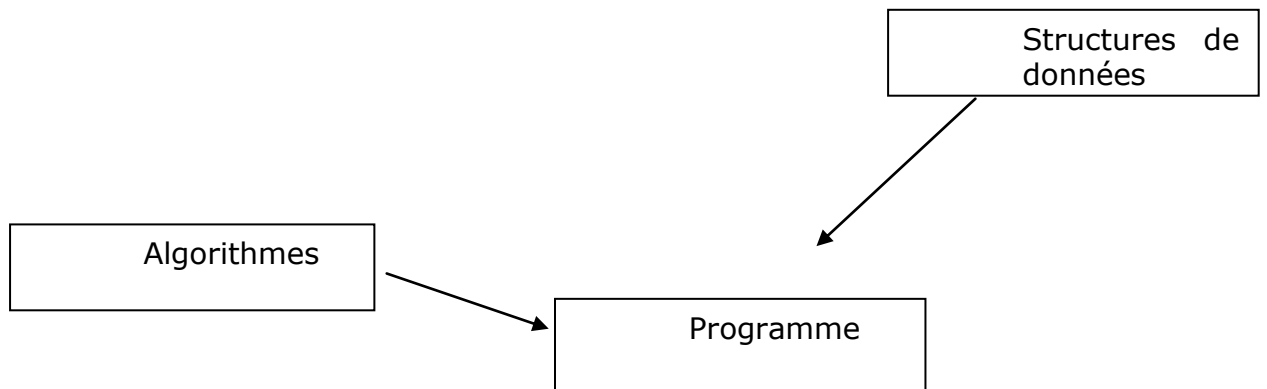
1.	Introduction à la P.O.O. ....	4
1.1.	La programmation classique : .....	4
1.2.	La programmation objet ou P.O.O. ....	4
1.3.	Avantages de la P.O.O. ....	5
1.4.	Les concepts fondamentaux de la P.O.O. ....	5
1.4.1.	Notion d'objet .....	5
1.4.2.	Une classe .....	5
1.4.3.	L'instanciation .....	6
1.4.4.	Le principe d'encapsulation.....	6
2.	Introduction au langage C++ .....	7
2.1.	Placement des déclarations de variables .....	7
2.2.	Le type bool .....	7
2.3.	Les références .....	7
	Les références vers des structures .....	9
	Les références lors d'un appel de fonction: .....	9
2.4.	Fonctions en ligne .....	10
2.5.	Valeurs par défaut des arguments des fonctions.....	11
2.6.	Surcharge des noms de fonctions .....	11
2.7.	Entrées-sorties simples en C++ .....	12
2.7.1.	Le flux de sortie cout.....	13
2.7.2.	Le flux d'entrée cin .....	13
2.8.	Allocation dynamique de mémoire.....	14
2.8.1.	Allocation dynamique en C .....	14
2.8.2.	Allocation dynamique en C++ .....	15
3.	Les classes en C++.....	15
3.1.	Classes et objets.....	15
3.2.	Accès aux membres d'un objet.....	16
3.3.	Accès à ses propres membres, accès à soi-même(this).....	16
3.4.	Membres publics et privés: groupe A .....	17
3.5.	Encapsulation au niveau de la classe .....	18
3.6.	Structures.....	18
3.7.	Définition des classes.....	19
3.7.1.	Définition séparée et opérateur de résolution de portée .....	19
3.7.2.	Fichier d'en-tête et fichier d'implémentation .....	19
3.8.	Constructeurs.....	20
3.8.1.	Définition de constructeurs.....	20

3.8.2.	Appel des constructeurs.....	22
3.8.3.	Constructeur par défaut.....	22
3.9.	Construction des objets membres .....	22
3.10.	Destructeurs .....	23
3.11.	Membres constants .....	24
3.11.1.	Données membres constantes .....	24
3.11.2.	Fonctions membres constantes .....	24
3.12.	Membres statiques .....	25
3.12.1.	Données membres statiques.....	25
3.12.2.	Fonctions membres statiques.....	26
<b>3.13.</b>	<b>Amis;groupe A</b> .....	27
3.13.1.	Fonctions amies.....	27
3.13.2.	Classes amies .....	28
4.	Surcharge des opérateurs .....	28
4.1.	Principe .....	29
4.2.	Les opérateurs arithmétiques (+, -, *, /, %)......	29
4.3.	Les opérateurs de comparaison (==, >, <, ...) .....	31
4.3.1.	L'opérateur == .....	31
4.3.2.	L'opérateur <.....	32
4.3.3.	Les autres opérateurs de comparaison .....	32
4.4.	L'opérateur d'affectation (=).....	32
4.4.1.	Rappel : le constructeur de copie .....	33
4.4.2.	Le rapport avec la surcharge de l'opérateur = ?.....	33
4.4.3.	Implémentation de la méthode operator= pour la classe Duree ...	34
5.	L'héritage.....	34
5.1.	Définitions: .....	34
5.2.	Notation: .....	35
5.3.	Exemple .....	35
5.4.	Héritage et accessibilité des membres .....	36
5.4.1.	Membres protégés .....	36
5.4.2.	Héritage privé, protégé, public.....	36
➤	Héritage privé .....	36
➤	Héritage protected .....	37
➤	Héritage public .....	37
5.5.	Redéfinition des fonctions membres .....	38
5.6.	Construction et destruction des objets dérivés .....	38
6.	Le polymorphisme .....	40
6.1.	Conversion standard vers une classe de base.....	40

6.2.	Type statique, type dynamique, généralisation.....	42
7.	Fonctions virtuelles .....	44
8.	Les classes abstraites.....	45
9.	La classe string de la librairie standard (STL).....	47
9.1.	La classe string en C++ .....	47
9.2.	Les constructeurs.....	47
9.3.	Les méthodes.....	47
9.4.	Les méthodes de recherche: .....	48
10.	La classe vector de la librairie standard (STL).....	51
10.1.	La classe vector en C++.....	51
10.2.	Les constructeurs.....	51
10.3.	Les accesseurs.....	52
10.4.	Les méthodes.....	52
10.5.	Les méthodes spécifiques .....	53

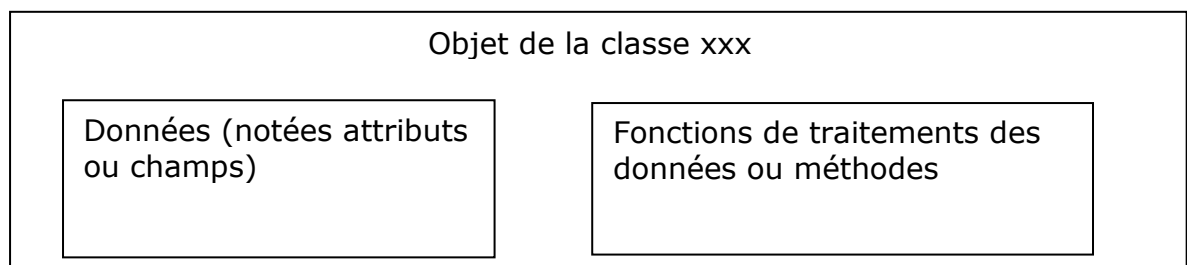
# 1. Introduction à la P.O.O.

## 1.1. La programmation classique :



Dans la programmation traditionnelle (années 70 et 80) un programme faisait appel à des fonctions ou procédures (en PASCAL) qui traitaient des données ou des structures de données (plus généralement). Le principe était de trouver l’algorithme puis ensuite de trouver la structure de donnée la mieux appropriée pour le passage de paramètre aux différentes fonctions. Cette programmation rendait la structure de données « fragile » car n’importe quelle fonction pouvait accéder et changer les champs de cette structure. De plus la réutilisation du code de certaines fonctions était difficile car rien n’était compartimenté.

## 1.2. La programmation objet ou P.O.O.



Dans la programmation objet chaque paquets de données (qui aurait fait parti d’une structure en programmation classique) est dans une classe. Les différents traitements à réaliser sur ces données sont faits par des fonctions appartenant à la classe. L’accès aux données ne se fera que par des fonctions de la classe. Le fait de compartimenter le code et les données dans une classe permet la réutilisation du code.

### Exemples :

Classe Bateau : champs : Capitaine, type, couleur, position, vitesse, ...  
Fonctions ou méthodes : marche, stop, plus\_vite, moins\_vite

Classe Fenetre champs : positionX, positionY, TailleX, TailleY, Couleur, Nom, ...  
Fonctions ou méthodes : Créer, Réduire, Fermer, Déplacer, ...

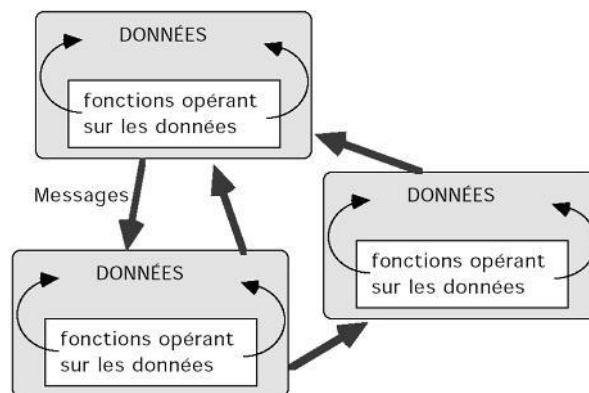
### 1.3. Avantages de la P.O.O.

- La découpe du problème en classes et donc en objets (instances de la classe) permet la réutilisation du code, chose plus difficile avec un programme classique.
- Une classe peut hériter d'une autre classe ce qui permet d'utiliser les fonctions et les données de la classe parent et de rajouter les données et fonctions de la nouvelle classe (en programmation graphique toute nouvelle fenêtre windows est un objet hérité de la classe Fenêtre dans laquelle on va rajouter le code et les données de l'application).
- Les fonctions de la classe n'ont pas un grand nombre d'arguments puisque les données sont lisibles par les fonctions de la classe.
- Les données de la classe sont protégées de l'extérieur : sécurité.
- Grâce à la P.O.O., on gagne en rapidité de fabrication du code, puisqu'il est possible d'utiliser un grand nombre de classe déjà existante.

### 1.4. Les concepts fondamentaux de la P.O.O

#### 1.4.1. Notion d'objet

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



#### 1.4.2. Une classe

Une classe est une description abstraite d'un objet. Elle peut être considérée en quelque sorte comme un moule...

Véhicule
+Marque : string(idl)
+Puissance fiscale : int
+Vitesse maximale : int
+Vitesse courante : int
+Créer un véhicule()
+Détruire un véhicule()
+Démarrer()
+Accélérer(entrée Taux : int)

Nom de la classe

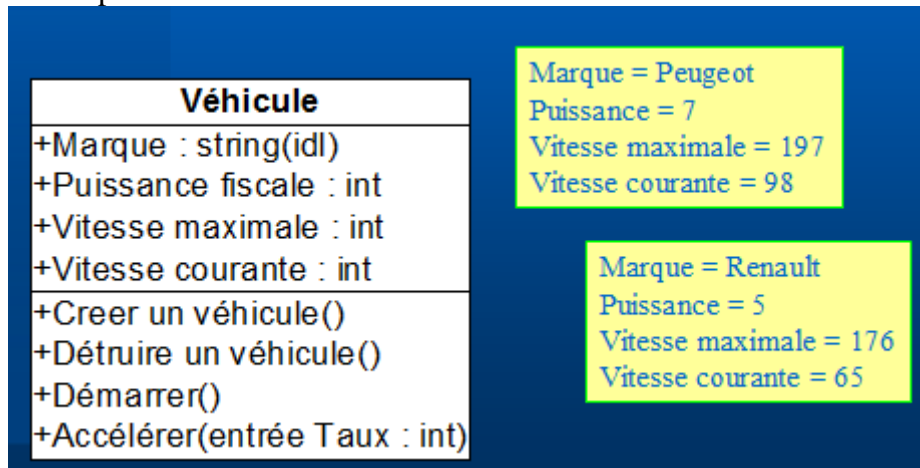
Description des attributs ou données membres

Description des méthodes = code associé

### 1.4.3. L'instanciation

- Chaque objet correspond à une instance de la classe à laquelle il fait référence.
- L'instanciation est le processus par lequel on crée de nouveaux objets, à partir du modèle défini par une classe.

Exemple:



La création d'un objet est constituée de deux phases :

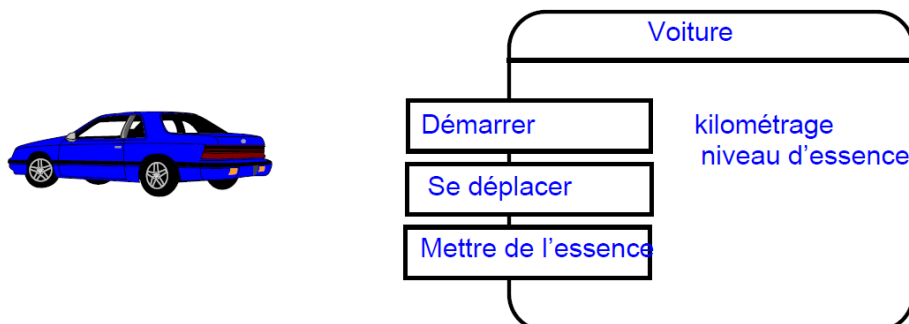
- Une phase du ressort de la classe : allouer de la mémoire et un contexte d'exécution minimaliste. Méthode de classe
- Une phase du ressort de l'objet : initialiser ses attributs d'instance. Méthode d'instance

Dans les langages tels que Java, C++, VB ou C#, ces deux phases ne sont pas différenciées. On fait appel à une méthode spéciale appelée : le constructeur

### 1.4.4. Le principe d'encapsulation

L'encapsulation : regroupement des informations d'état (attributs) et de comportement (méthodes ou opérations) sous un nom unique

L'encapsulation consiste à masquer aux utilisateurs d'un objet les détails relevant de son implémentation (vue interne ou réalisation), et de ne laisser accessible que la vue externe ou interface.



Principe : les attributs ne doivent pas être accessibles de l'extérieur. Seules les opérations sont habilitées à modifier les valeurs des attributs. En conséquence, un changement de format de ces attributs ne se répercute pas à l'extérieur.

### **Avantages de l'encapsulation:**

- **L'intérieur de l'objet est protégé.** L'accès direct aux données est impossible. Il faut passer par une méthode (publique) constituant l'interface de l'objet pour avoir accès aux données. Ceci garantit la sécurité et l'intégrité des données.
- **La complexité est dissimulée.** Le programmeur n'a plus à se préoccuper ni du détail de la structure des données, ni de la manière dont sont effectués les traitements.
- **La maintenance est simplifiée.** La portée des modifications est limitée à l'intérieur de l'objet concerné. L'implémentation d'une opération peut évoluer sans modifier l'interface de l'objet.
- **Les échanges avec l'extérieur sont codifiés.** Seules les caractéristiques que l'on souhaite offrir aux utilisateurs potentiels de l'objet figurent dans la partie externe. Cette interface constitue un contrat entre l'objet et ses clients, c'est-à-dire entre le concepteur et ses utilisateurs.
- .....héritage, surcharge,...polymorphisme

## **2. Introduction au langage C++**

Cette première section expose un certain nombre de notions qui, sans être directement liés à la méthodologie objets, font déjà apparaître C++ comme une amélioration notable de C.

### ***2.1.Placement des déclarations de variables***

En C les déclarations de variables doivent apparaître au début d'un bloc. En C++, au contraire, on peut mettre une déclaration de variable partout où on peut mettre une instruction.

Cette différence paraît mineure, mais elle est importante par son esprit. Elle permet de repousser la déclaration d'une variable jusqu'à l'endroit du programme où l'on dispose d'assez éléments pour l'initialiser. On lutte aussi contre les variables « déjà déclarées mais non encore initialisées » qui sont un important vivier de bugs dans les programmes.

### ***2.2.Le type bool***

En plus des types définis par l'utilisateur (ou classes, une des notions fondamentales de la programmation orientée objets), C++ possède quelques types qui manquaient à C, notamment le type booléen et les types références.

Le type bool (pour booléen) comporte deux valeurs : false et true. Contrairement à C :

- le résultat d'une opération logique (&&, ||, etc.) est de type booléen,
- là où une condition est attendue on doit mettre une expression de type booléen et
- il est déconseillé de prendre les entiers pour des booléens et réciproquement.

Exemple de code C++:

```
bool trouv= true;
int age;
```

### ***2.3.Les références***

Le C++ introduit un nouveau concept : **les références**. Une référence est un synonyme d'une autre variable

A côté des pointeurs, les références sont une autre manière de manipuler les adresses des objets placés dans la mémoire. Une référence est un pointeur géré de manière interne par la machine. Si T est un type donné, le type «référence sur T » se note T&.

Exemple :

**int i;**

**int & r = i;** // r est une référence sur i

Une valeur de type référence est une adresse mais, hormis lors de son initialisation, toute opération effectuée sur la référence agit sur l'objet référencé, non sur l'adresse. Il en découle qu'il est obligatoire d'initialiser une référence lors de sa création ;

`r = j;` // ceci ne transforme pas r en une référence

// sur j mais copie la valeur de j dans i

Il y a donc deux règles qu'il faut retenir :

- **Règle 1** : une référence doit être initialisée dès sa déclaration.
- **Règle 2** : une fois initialisée, une référence ne peut plus être modifiée.

Exemple en C++:

```
int main()
{
    int age = 21;
    int &referenceSurAge = age;

    cout << referenceSurAge << endl;
    cout << age << endl;

    referenceSurAge = 40;

    cout << referenceSurAge << endl;
    cout << age << endl;

    return 0;
}
```

Résultat :

21

21

40

40

Comme vous pouvez le voir, une référence s'utilise exactement comme la variable d'origine. C'est le compilateur qui fait la "conversion" et qui sait qu'il doit affecter la variable "age" lorsqu'on travaille avec la référence.

### *Comparatif pointeur / référence*

En C++, les pointeurs existent toujours. Les références sont juste une alternative aux pointeurs. Elles ont surtout l'avantage d'être plus simples à utiliser, mais elles ne peuvent pas les remplacer complètement. Pourquoi ? Une référence ne peut pas faire référence à une nouvelle variable une fois qu'elle a été initialisée. Un pointeur, lui, peut toujours pointer vers une nouvelle variable au cours de l'exécution du programme.

----- Code d'exemple avec un pointeur -----	----- Code d'exemple avec une référence -----
Code : C++ - Sélectionner	Code : C++ - Sélectionner
<pre> 1  int main() 2  { 3      int age = 21; 4      int *pointeurSurAge = &amp;age; 5 6      cout &lt;&lt; *pointeurSurAge; 7 8      *pointeurSurAge = 40; 9 10     cout &lt;&lt; *pointeurSurAge; 11 12 13     return 0; 14 }</pre>	<pre> 1  int main() 2  { 3      int age = 21; 4      int &amp;referenceSurAge = age; 5 6      cout &lt;&lt; referenceSurAge; 7 8      referenceSurAge = 40; 9 10     cout &lt;&lt; referenceSurAge; 11 12 13     return 0; 14 }</pre>

## Les références vers des structures

Si vous faites une référence vers une structure, il faudra utiliser le symbole point "." et non le symbole flèche "->" lorsque vous voulez accéder à un élément d'une structure.

### Code : C++

```

struct Coordonnees
{
    int x;
    int y;
};

int main()
{
    Coordonnees point;
    Coordonnees &referenceSurPoint = point;

    referenceSurPoint.x = 10;
    referenceSurPoint.y = 5;

    cout << "x : " << referenceSurPoint.x << endl;
    cout << "y : " << referenceSurPoint.y << endl;

    return 0;
}
```

## Les références lors d'un appel de fonction:

En fait, comme pour les pointeurs, les références révèlent toute leur utilité lorsqu'on appelle une fonction

### Code : C++

```

struct Coordonnees
{
    int x;
    int y;
};
```

```

void remiseAZero(Coordonnees &pointAModifier);

int main()
{
    Coordonnees point;

    remiseAZero(point); // Pas besoin d'indiquer l'adresse de point avec un
//& lors de l'appel

    return 0;
}

void remiseAZero(Coordonnees &pointAModifier) // La fonction indique
//qu'elle récupère une référence
{
    // La référence s'utilise exactement comme une variable
    // On utilise donc des points "." et non des flèches "->"
    pointAModifier.x = 0;
    pointAModifier.y = 0;
}

```

### *Comparaison pointeur / référence*

Une fois de plus, voici un comparatif du même code utilisant d'un côté un pointeur, de l'autre une référence.

#### **Code d'exemple avec un pointeur**

```

1 int main()
2 {
3     Coordonnees point;
4     remiseAZero(&point);
5
6     return 0;
7 }
8
9 void remiseAZero(Coordonnees
10 *pointAModifier)
11 {
12     pointAModifier->x = 0;
13     pointAModifier->y = 0;
14 }

```

#### **Code d'exemple avec une référence**

```

1 int main()
2 {
3     Coordonnees point;
4     remiseAZero(point);
5
6     return 0;
7 }
8
9 void remiseAZero(Coordonnees
10 &pointAModifier)
11 {
12     pointAModifier.x = 0;
13     pointAModifier.y = 0;
14 }

```

## **2.4. Fonctions en ligne**

Normalement, un appel de fonction est une rupture de séquence : à l'endroit où un appel figure, la machine cesse d'exécuter séquentiellement les instructions en cours ;

les arguments de l'appel sont disposés sur la pile d'exécution, et l'exécution continue ailleurs, là où se trouve le code de la fonction.

Une fonction en ligne est le contraire de cela : là où l'appel d'une telle fonction apparaît il n'y a pas de rupture de séquence. Au lieu de cela, le compilateur remplace l'appel de la fonction par le corps de celle-ci, en mettant les arguments effectifs à la place des arguments formels.

Cela se fait dans un but d'efficacité : puisqu'il n'y a pas d'appel, pas de préparation des arguments, pas de rupture de séquence, pas de retour, etc. Mais il est clair qu'un tel traitement ne peut convenir qu'à des fonctions fréquemment appelées (si une fonction ne sert pas souvent, à quoi bon la rendre plus rapide ?) de petite taille (sinon le code compilé risque de devenir démesurément volumineux) et rapides (si une fonction effectue une opération lente, le gain de temps obtenu en supprimant l'appel est négligeable).

En C++ on indique qu'une fonction doit être traitée en ligne en faisant précéder sa déclaration par le mot **inline** :

Exemple:

```
inline int abs(int x)
{
return x >= 0 ? x : -x;
}
```

## 2.5. Valeurs par défaut des arguments des fonctions

Les paramètres formels d'une fonction peuvent avoir des valeurs par défaut.

Exemple :

```
void trier(void *table, int nbr, int taille = sizeof(void *), bool croissant = true);
```

Lors de l'appel d'une telle fonction, les paramètres effectifs correspondants peuvent alors être omis (ainsi que les virgules correspondantes), les paramètres formels seront initialisés avec les valeurs par défaut. Exemples :

```
trier(t, n, sizeof(int), false);
trier(t, n, sizeof(int)); // croissant = true
trier(t, n); // taille = sizeof(void *), croissant = true
```

## 2.6. Surcharge des noms de fonctions

La *signature d'une fonction* est la suite des types de ses arguments formels (et quelques éléments supplémentaires, que nous verrons plus loin). Le type du résultat rendu par la fonction ne fait pas partie de sa signature.

La *surcharge des noms des fonctions* consiste en ceci : en C++ des fonctions différentes peuvent avoir le même nom, à la condition que leurs signatures soient assez différentes pour que, lors de chaque appel, le nombre et les types des arguments effectifs permettent de choisir sans ambiguïté la fonction à appeler.

Exemple :

```
int puissance(int x, int n)
{
calcul de  $x^n$  avec x et n entiers
}
```

```

double puissance(double x, int n)
{
calcul de  $x^n$  avec x flottant et n entier
}
double puissance(double x, double y)
{
calcul de  $x^y$  avec x et y flottants
}

```

On voit sur cet exemple l'intérêt de la surcharge des noms des fonctions : la même notion abstraite « x à la puissance n » se traduit par des algorithmes très différents selon que n est entier ( $x^n = x \cdot x \dots x$ ) ou réel ( $x^n = e^{n \log x}$ ) ; de plus, pour n entier, si on veut que  $x^n$  soit entier lorsque x est entier, on doit écrire deux fonctions distinctes, une pour x entier et une pour x réel.

Or le programmeur n'aura qu'un nom à connaître, puissance. Il écrira dans tous les cas `c = puissance(a, b)`; le compilateur se chargeant de choisir la fonction la plus adaptée, selon les types de a et b.

#### Notes.

- Le type du résultat rendu par la fonction ne fait pas partie de la signature. Par conséquent, on ne peut pas donner le même nom à deux fonctions qui ne diffèrent que par le type du résultat qu'elles rendent.
- Lors de l'appel d'une fonction surchargée, la fonction effectivement appelée est celle dont la signature correspond avec les types des arguments effectifs de l'appel. S'il y a une correspondance exacte, pas de problème. S'il n'y a pas de correspondance exacte, alors des règles complexes s'appliquent, pour déterminer la fonction à appeler. Malgré ces règles, il existe de nombreux cas de figure ambigus, que le compilateur ne peut pas résoudre.

## 2.7. Entrées-sorties simples en C++

Cette section traite de l'utilisation simple des flux standard d'entrée-sortie, c'est-à-dire la manière de faire en C++ les opérations qu'on fait habituellement en C avec les fonctions `printf` et `scanf`.

Un programme qui utilise les flux standard d'entrée-sortie doit comporter la directive

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

Ou bien, si vous utilisez un compilateur récent :

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

Les flux d'entrée-sortie sont représentés dans les programmes par les trois variables, pré déclarées et pré initialisées, suivantes :

- `cin`, le flux standard d'entrée (l'équivalent du `stdin` de C), qui est habituellement associé au clavier du poste de travail,
- `cout`, le flux standard de sortie (l'équivalent du `stdout` de C), qui est habituellement associé à l'écran du poste de travail,
- `cerr`, le flux standard pour la sortie des messages d'erreur (l'équivalent du `stderr` de C), également associé à l'écran du poste de travail.

### 2.7.1. Le flux de sortie cout

cout n'est pas une fonction mais un flux, un élément nouveau introduit en C++. Notez que ça n'a rien à voir avec la POO pour le moment

Le flux cout est l'équivalent de la fonction printf... mais en mieux, en plus simple

Exemple:

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```

**endl** : est un mot-clé qui signifie "fin de ligne" (end line en anglais). En fait, c'est un mot qui remplace le \n qu'on utilise en C.

Mais on peut toujours utiliser \n en C++

Exemple;

```
cout << "Hello world!\n";
```

L'affichage d'une variable ne nécessite pas l'utilisation des spécificateurs des formats comme dans le cas de printf.

Exemple1:

```
int main()
{
    int age = 21;
    cout << "Salut, j'ai " << age << " ans" << endl;
    return 0;
}
```

Exemple2:

```
int main()
{
    int age = 21;
    char pseudo[] = "Natalie";
    cout << "Salut, j'ai " << age << " ans et je m'appelle " << pseudo << endl;
    return 0;
}
```

### 2.7.2. Le flux d'entrée cin

cin représente l'entrée en C++.

cin représente donc le clavier et permet de récupérer du texte saisi par l'utilisateur.

Exemple1:

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

int main()
```

```

{
int age = 0;
cout << "Quel age avez-vous ?" << endl;
cin >> age;
cout << "Ah ! Vous avez donc " << age << " ans !" << endl;
return 0;
}

```

Exemple2:

```

#include <iostream>
using namespace std;

int main()
{
char pseudo[50];
cout << "Quel est votre pseudo ?" << endl;
cin >> pseudo;
cout << "Salut " << pseudo << endl;
return 0;
}

```

## 2.8. Allocation dynamique de mémoire

### 2.8.1. Allocation dynamique en C

- La déclaration d'un tableau définit un tableau "statique"(il possède un nombre figé d'emplacements). Il y'a donc un gaspillage d'espace en mémoire en réservant toujours l'espace maximal prévisible.
- Il serait souhaitable que l'allocation de la mémoire dépende du nombre d'éléments à saisir. Ce nombre ne sera connu qu'à l'exécution: c'est l'allocation dynamique.

Les fonctions d'allocation de la mémoire sont: **malloc**, **calloc** et **realloc**.

- Chaque fonction prend une zone d'une taille donnée dans l'espace mémoire libre réservé pour le programme et affecte l'adresse du début de la zone à une variable pointeur.
- S'il n'y a pas assez de mémoire libre à allouer, la fonction renvoie le pointeur NULL.

#### Fonction malloc

<pointeur>= <type> malloc(<taille>);

- <type> est un type pointeur définissant l'objet pointé par <pointeur>
- <taille> est le nombre d'octets alloués pour <pointeur>
- <taille> est du type unsigned int, donc on ne peut pas réserver plus de 65535 octets à la fois

Exemple:

Char \*pc;

Pc= (char \*) malloc(4000);

Soit à pc est affecté l'adresse d'un bloc mémoire de 4000 octets. Soit pc contient la valeur 0 s'il n'y a pas assez de mémoire libre.

.....suite voir cours de Ahiod

## 2.8.2. Allocation dynamique en C++

Des différences entre C et C++ existent aussi au niveau de l'allocation et de la restitution dynamique de mémoire.

Les fonctions **malloc** et **free** de la bibliothèque standard C sont disponibles en C++. Mais il est fortement conseillé de leur préférer les opérateurs **new** et **delete**. La raison principale est la suivante : les objets créés à l'aide de **new** sont initialisés à l'aide des constructeurs (définis dans la section suivante du cours) correspondants, ce que ne fait pas **malloc**. De même, les objets libérés en utilisant **delete** sont finalisés en utilisant le destructeur (voir section suivante) de la classe correspondante, contrairement à ce que fait **free**.

- Pour allouer un unique objet : - `new type`
- Pour allouer un tableau de n objets : - `new type[n]`

Exemple:

```
type *ptr = new type; // un objet
```

```
int *tab = new int[n]; // allocation d'un tableau de n entiers
```

## 3. Les classes en C++

### 3.1. Classes et objets

Un objet est constitué par l'association d'une certaine quantité de mémoire, organisée en champs, et d'un ensemble de fonctions, destinées principalement à la consultation et la modification des valeurs de ces champs. La définition d'un type objet s'appelle une classe. D'un point de vue syntaxique, cela ressemble beaucoup à une définition de structure, sauf que

- le mot réservé **class** remplace le mot **struct**,
- certains champs de la classe sont des fonctions.

Par exemple, le programme suivant est une première version d'une classe `Point` destinée à représenter les points affichés dans une fenêtre graphique :

```
class Point {  
public:  
void afficher() {  
cout << '(' << x << ',' << y << ')';  
}  
void placer(int a, int b) {  
x = a; y = b;  
}  
private:  
int x, y;  
};
```

Chaque objet de la classe `Point` comporte un peu de mémoire, composée de deux entiers `x` et `y`, et de deux fonctions : `afficher`, qui accède à `x` et `y` sans les modifier, et `placer`, qui change les valeurs de `x` et `y`.

L'association de membres et fonctions au sein d'une classe, avec la possibilité de rendre privés certains d'entre eux, s'appelle l'*encapsulation* des données. Intérêt de la démarche : puisqu'elles ont été déclarées privées, les coordonnées `x` et `y` d'un point ne peuvent être modifiées autrement que par un appel de la fonction `placer` sur ce point.

Autre avantage important : on pourra à tout moment changer l'implémentation (i.e. les détails internes) de la classe tout en ayant la certitude qu'il n'y aura rien à changer dans les programmes qui l'utilisent.

*Note.* Dans une classe, les déclarations des membres peuvent se trouver dans un ordre quelconque, même lorsque ces membres se référencent mutuellement. Dans l'exemple précédent, le membre `afficher` mentionne les membres `x` et `y`, dont la définition se trouve après celle de `afficher`.

### Vocabulaire: On appelle

- **objet** : une donnée d'un type classe ou structure,
  - **fonction membre** : une fonction déclarée dans une classe.
  - **donnée membre** : un membre qui est une variable.
- Dans la plupart des langages orientés objets, les fonctions membres sont appelées méthodes.
  - Dans beaucoup de langages orientés objets, les données membres sont appelées variables d'instance et aussi, sous certaines conditions, propriétés

## 3.2. Accès aux membres d'un objet

On accède aux membres des objets en C++ comme on accède aux membres des structures en C. Par exemple, à la suite de la définition de la classe `Point` donnée précédemment on peut déclarer des variables de cette classe en écrivant :

```
Point a, b, *pt; // deux points et un pointeur de point
```

Dans un contexte où le droit de faire un tel accès est acquis. L'accès aux membres du point `a` s'écrit :

```
a.x = 0; // un accès bien écrit au membre x du point a
```

```
a.afficher(); // un appel bien écrit de la fonction afficher() de l'objet a
```

Si on suppose que le pointeur `pt` a été initialisé, par exemple par une expression telle que

```
pt = new Point; // allocation dynamique d'un point alors des accès analogues aux précédents s'écrivent :
```

```
pt->x = 0; // un accès bien écrit au membre x du point pointé par pt
```

```
pt->afficher(); // un appel de la fonction distance de l'objet pointé par pt
```

## 3.3. Accès à ses propres membres, accès à soi-même(*this*)

Quand des membres d'un objet apparaissent dans une expression écrite dans une fonction *du même objet* on dit que ce dernier fait un *accès à ses propres membres*. On a droit dans ce cas à une notation simplifiée : on écrit le membre tout seul, sans expliciter l'objet en question. C'est ce que nous avons fait dans les fonctions de la classe `Point` :

```
class Point {  
    ...  
    void afficher() {  
        cout << '(' << x << ', ' << y << ')';  
    }  
    ...  
};
```

Dans la fonction `afficher`, les membres `x` et `y` dont il est question sont ceux de l'objet à travers lequel on aura appelé cette fonction. Autrement dit, lors d'un appel comme `unPoint.afficher()`;

Le corps de cette fonction sera équivalent à

```
cout << '(' << unPoint.x << ',' << unPoint.y << ')';
```

**Accès à soi-même.** Il arrive que dans une fonction membre d'un objet on doit faire référence à l'objet (tout entier) à travers lequel on a appelé la fonction. Il faut savoir que dans une fonction membre on dispose de la pseudo variable **this** qui représente un pointeur vers l'objet en question. Par exemple, la fonction afficher peut s'écrire de manière équivalente, *mais cela n'a aucun intérêt* :

```
void afficher() {  
    cout << '(' << this->x << ',' << this->y << ')';  
}
```

Le mot clé *this* permet de désigner l'objet dans lequel on se trouve, c'est-à-dire que lorsque l'on désire faire référence dans une fonction membre à l'objet dans lequel elle se trouve, on utilise *this*.

Pour voir un exemple plus utile d'utilisation de **this** imaginons qu'on nous demande d'ajouter à la classe Point deux fonctions booléennes, une pour dire si deux points sont égaux, une autre pour dire si deux points sont le même objet. Dans les deux cas le deuxième point est donné par un pointeur :

```
class Point {  
    ...  
    bool pointEgal(Point *pt) {  
        if (pt->x == x && pt->y == y)  
            return true;  
    }  
    bool memePoint(Point *pt) {  
        return pt == this;  
    }  
    ...  
}
```

### 3.4. Membres publics et privés: groupe A

Par défaut, les membres des classes sont privés. Les mots clés public et private permettent de modifier les droits d'accès des membres :

```
class nom {  
    les membres déclarés ici sont privés  
    public:  
    les membres déclarés ici sont publics  
    private:  
    les membres déclarés ici sont privés  
    etc.  
};
```

Les expressions **public:** et **private:** peuvent apparaître un nombre quelconque de fois dans une classe. Les membres déclarés après private: (resp. public:) sont privés (resp. publics) jusqu'à la fin de la classe, ou jusqu'à la rencontre d'une expression public: (resp. private:).

Un membre public d'une classe peut être accédé partout où il est visible ; un membre privé ne peut être accédé que depuis une fonction membre de la classe.

Exemple:

```
Point p;  
p.x=3; // erreur car accès interdit  
p.y=4; // erreur car accès interdit  
p.placer(3,4); // correct car accès légal
```

### 3.5. Encapsulation au niveau de la classe

Les fonctions membres d'une classe ont le droit d'accéder à tous les membres de la classe : deux objets de la même classe ne peuvent rien se cacher. Par exemple, le programme suivant montre notre classe Point augmentée d'une fonction pour calculer la distance d'un point à un autre :

```
class Point {  
public:  
    void afficher() {  
        cout << '(' << x << ',' << y << ')';  
    }  
    void placer(int a, int b) {  
        validation des valeurs de a et b;  
        x = a; y = b;  
    }  
    double distance(Point autrePoint) {  
        int dx = x - autrePoint.x;  
        int dy = y - autrePoint.y;  
        return sqrt(dx * dx + dy * dy);  
    }  
private:  
    int x, y;  
};
```

Lors d'un appel tel que `p.distance(q)` l'objet `p` accède aux membres privés `x` et `y` de l'objet `q`. On dit que C++ *pratique l'encapsulation au niveau de la classe*, non au niveau de l'objet. En C++ encapsuler n'est pas *cache*: mais *interdire*. Les usagers d'une classe voient les membres privés de cette dernière, mais ne peuvent pas les utiliser.

### 3.6. Structures

Une structure est la même chose qu'une classe mais, par défaut, les membres y sont publics.

```
struct nom {  
    les membres déclarés ici sont publics  
private:  
    les membres déclarés ici sont privés  
public:  
    les membres déclarés ici sont publics  
    etc.  
};
```

### 3.7. Définition des classes

#### 3.7.1. Définition séparée et opérateur de résolution de portée

Tous les membres d'une classe doivent être au moins déclarés à l'intérieur de la formule `class nom{...}` ; qui constitue la déclaration de la classe.

Cependant, dans le cas des fonctions, aussi bien publiques que privées, on peut se limiter à n'écrire que leur en-tête à l'intérieur de la classe et définir le corps ailleurs, plus loin dans le même fichier ou bien dans un autre fichier.

Il faut alors un moyen pour indiquer qu'une définition de fonction, écrite en dehors de toute classe, est en réalité la définition d'une fonction membre d'une classe. Ce moyen est l'*opérateur de résolution de portée*, dont la syntaxe est

```
NomDeClasse::
```

Par exemple, voici notre classe Point avec la fonction distance définie séparément :

```
class Point {  
public:  
...  
double distance(Point autrePoint);  
...  
}
```

Il faut alors, plus loin dans le même fichier ou bien dans un autre fichier, donner la définition de la fonction « promise » dans la classe Point. Cela s'écrit :

```
double Point::distance(Point autrePoint) {  
int dx = x - autrePoint.x;  
int dy = y - autrePoint.y;  
return sqrt(dx * dx + dy * dy);  
};
```

#### 3.7.2. Fichier d'en-tête et fichier d'implémentation

En programmation orientée objets, « programmer » c'est définir des classes. Le plus souvent ces classes sont destinées à être utilisées dans plusieurs programmes, présents et à venir. Se pose alors la question : comment disposer le code d'une classe pour faciliter son utilisation ?

**Voici comment on procède généralement :**

- les définitions des classes se trouvent dans des fichiers en-tête (fichiers « .h », « .hpp », etc.),
- chacun de ces fichiers en-tête contient la définition d'une seule classe ou d'un groupe de classes intimement liées ; par exemple, la définition de notre classe Point pourrait constituer un fichier Point.h
- les définitions des fonctions membres qui ne sont pas définies à l'intérieur de leurs classes sont écrites dans des fichiers sources (fichiers « .cpp » ou « .cp »),
- aux programmeurs utilisateurs de ces classes sont distribués :
  - les fichiers « .h »
  - le fichiers objets résultant de la compilation des fichiers « .cpp »

Par exemple, voici les fichiers correspondants à notre classe Point (toujours très modeste) :

```
Fichier Point.h :  
class Point {  
public:  
void placer(int a, int b) {  
validation de a et b
```

```
x = a; y = b;
}
double distance(Point autrePoint);
private:
int x, y;
};
```

```
Fichier Point.cpp :
#include "Point.h"
#include <math.h>
double Point::distance(Point autrePoint) {
int dx = x - autrePoint.x;
int dy = y - autrePoint.y;
return sqrt(dx * dx + dy * dy);
}
```

### 3.8. Constructeurs

#### 3.8.1. Définition de constructeurs

**Un constructeur d'une classe est une fonction membre spéciale qui :**

- a le même nom que la classe,
- n'indique pas de type de retour,
- ne contient pas d'instruction return.

Le rôle d'un constructeur est d'initialiser un objet, notamment en donnant des valeurs à ses données membres.

Exemple :

```
class Point {
public:
Point(int a, int b) {
x = a; y = b;
}
... autres fonctions membres ...
private:
int x, y;
};
```

Un constructeur de la classe est toujours appelé, explicitement ou implicitement, lorsqu'un objet de cette classe est créé, et en particulier chaque fois qu'une variable ayant cette classe pour type est définie.

Une classe peut posséder plusieurs constructeurs, qui doivent alors avoir des signatures différentes :

```
class Point {
public:
Point(int a, int b) // constructeur normal
{
validation de a et b
x = a; y = b;
}
```

```

Point(int a) //constructeur normal
{
x = a; y = 0;
}
Point() // constructeur par défaut
{
x = y = 0;
}

Point(Point &p) // constructeur par copie
{
x =p.x;
y = p.y;
}

...
private:
int x, y;
};

```

L'emploi de paramètres avec des valeurs par défaut permet de grouper des constructeurs. La classe suivante possède les mêmes constructeurs que la précédente :

```

class Point {
public:
Point(int a = 0, int b = 0) {
validation de a et b
x = a; y = b;
}
...
private:
int x, y;
};

```

Comme les autres fonctions membres, les constructeurs peuvent être déclarés dans la classe et définis ailleurs. Ainsi, la classe précédente pourrait s'écrire également

```

class Point {
public:
Point(int a = 0, int b = 0);
...
private:
int x, y;
};

```

et, ailleurs :

```

Point::Point(int a, int b) {
x = a; y = b;
}

```

### 3.8.2. Appel des constructeurs

Un constructeur est *toujours* appelé lorsqu'un objet est créé, soit explicitement, soit implicitement. Les appels explicites peuvent être écrits sous deux formes :

```
Point a(3, 4);  
Point b = Point(5, 6);
```

Dans le cas d'un constructeur avec un seul paramètre, on peut aussi adopter une forme qui rappelle l'initialisation des variables de types primitifs:

```
Point e = 7; // équivaut à : Point e = Point(7)
```

Un objet alloué dynamiquement est lui aussi toujours initialisé, au moins implicitement. Dans beaucoup de cas il peut, ou doit, être initialisé explicitement. Cela s'écrit :

```
Point *pt;  
...  
pt = new Point(1, 2);
```

### 3.8.3. Constructeur par défaut

Le constructeur par défaut est un constructeur qui peut être appelé sans paramètres : ou bien il n'en a pas, ou bien tous ses paramètres ont des valeurs par défaut. Il joue un rôle remarquable, car *il est appelé chaque fois qu'un objet est créé sans qu'il y ait appel explicite d'un constructeur*, soit que le programmeur ne le juge pas utile, soit qu'il n'en a pas la possibilité :

```
Point x; // équivaut à : Point x = Point()  
Point t[10]; // produit 10 appels de Point()  
Point *p = new Point; // équivaut à : p = new Point()  
Point *q = new Point[10]; // produit 10 appels de Point()
```

## 3.9. Construction des objets membres

Lorsque des membres d'une classe sont à leur tour d'un type classe on dit que la classe a des *objets membres*.

L'initialisation d'un objet de la classe nécessite alors l'initialisation de ses objets membres.

Lorsque les objets membres n'ont pas de constructeurs par défaut, une syntaxe spéciale permet de préciser les arguments des constructeurs des membres :

```
NomDeLaClasse(paramètres)  
: membre(paramètres), ... membre(paramètres) {  
corps du constructeur  
}
```

A titre d'exemple, imaginons que notre classe Point ne possède pas de constructeur sans arguments, et qu'on doive définir une classe Segment ayant deux points pour membres (un segment est déterminé par deux points). Voici comment on devra écrire son constructeur :

```
class Segment {  
Point origine, extremite;  
int epaisseur;
```

```

public:
Segment(int ox, int oy, int ex, int ey, int ep)
: origine(ox, oy), extremite(ex, ey) {
epaisseur = ep;
}
...
};

```

*Note.* La syntaxe spéciale pour l'initialisation des objets membres peut être utilisée aussi pour initialiser les données membres de types primitifs. Par exemple, le constructeur de *Segment* précédent peut aussi s'écrire :

```

class Segment {
...
Segment(int ox, int oy, int ex, int ey, int ep)
: origine(ox, oy), extremite(ex, ey), epaisseur(ep) {
}
...
}

```

### 3.10. Destructeurs

De la même manière qu'il y a des choses à faire pour initialiser un objet qui commence à exister, il y a parfois des dispositions à prendre lorsqu'un objet va disparaître.

Un destructeur est une fonction membre spéciale. Il a le même nom que la classe, précédé du caractère ~. Il n'a pas de paramètres, ni de type de retour. Il y a donc au plus un destructeur par classe. Le destructeur d'une classe est appelé lorsqu'un objet de la classe est détruit, juste avant que la mémoire occupée par l'objet soit récupérée par le système.

Par exemple, voici le destructeur qu'il faut ajouter à la classe *PointNomme*. Sans ce destructeur, la destruction d'un point n'entraînerait pas la libération de l'espace alloué pour son étiquette :

```

class PointNomme {
int x,y;
char *label;
PointNomme(int a, int b, char *s) {
x = a; y = b;
label = new char[strlen(s) + 1];
strcpy(label, p.label);
}
...
~PointNomme() {
delete [] label;
}
...
};

```

**NB:** Le destructeur n'est jamais appelé explicitement dans un programme. Il est utilisé automatiquement par le compilateur au moment de la destruction des objets.

Exemple:

```

PointNomme *p=new PointNomme(2,4,"A");
Delete p;// destrucion de l'objet

```

### 3.11. Membres constants

#### 3.11.1. Données membres constantes

Une donnée membre d'une classe peut être qualifiée `const`. Il est alors obligatoire de l'initialiser lors de la construction d'un objet, et sa valeur ne pourra par la suite plus être modifiée.

A titre d'exemple voici une nouvelle version de la classe `Segment`, dans laquelle chaque objet reçoit, lors de sa création, un « numéro de série » qui ne doit plus changer au cours de la vie de l'objet :

```
class Segment {
    Point origine, extremite;
    int epaisseur;
    const int numeroDeSerie;
public:
    Segment(int x1, int y1, int x2, int y2, int ep, int num);
};
```

#### Constructeur, version erronée :

```
Segment::Segment(int x1, int y1, int x2, int y2, int ep, int num)
: origine(x1, y1), extremite(x2, y2) {
    epaisseur = ep;
    numeroDeSerie = num; // ERREUR : tentative de modification d'une constante
}
```

#### Constructeur, version correcte, en utilisant la syntaxe de l'initialisation des objets membres :

```
Segment::Segment(int x1, int y1, int x2, int y2, int ep, int num)
: origine(x1, y1), extremite(x2, y2), numeroDeSerie(num) {
    epaisseur = ep;
}
```

#### 3.11.2. Fonctions membres constantes

Le mot `const` placé à la fin de l'en-tête d'une fonction membre indique que l'état de l'objet à travers lequel la fonction est appelée n'est pas changé du fait de l'appel. C'est une manière de déclarer qu'il s'agit d'une fonction de *consultation* de l'objet, non d'une fonction de *modification* :

```
class Point {
    ...
    void placer(int a, int b); // modifie l'objet
    float distance(Point p) const; // ne modifie pas l'objet
    ...
};
```

#### Exemple:

```
void uneFonction(const Point a) {
    Point b;
```

```
...
double d = a.distance(b);
...
}
```

La qualification **const** de la fonction `distance` est indispensable pour que l'expression précédente soit acceptée par le compilateur. C'est elle seule, en effet, qui garantit que le point `a`, contraint à rester constant, ne sera pas modifié par l'appel de `distance`.

### 3.12. Membres statiques

Chaque objet d'une classe possède son propre exemplaire de chaque membre ordinaire (bientôt nous dirons membre non statique) de la classe :

- pour les données membres, cela signifie que de la mémoire nouvelle est allouée lors de la création de chaque objet ;
- pour les fonctions membres, cela veut dire qu'elles ne peuvent être appelées qu'en association avec un objet (on n'appelle pas « la fonction `f` » mais « la fonction `f` sur l'objet `x` »).

A l'opposé de cela, les membres statiques, signalés par la qualification **static** précédant leur déclaration, sont partagés par tous les objets de la classe. De chacun il n'existe qu'un seul exemplaire par classe, quel que soit le nombre d'objets de la classe.

Les données et fonctions membres non statiques sont donc ce que dans d'autres langages orientés objets on appelle *variables d'instance* et *méthodes d'instance*, tandis que les données et fonctions statiques sont appelées dans ces langages *variables de classe* et *méthodes de classe*.

#### 3.12.1. Données membres statiques

```
class Point {
int x, y;
public:
static int nombreDePoints;
Point(int a, int b) {
x = a; y = b;
nombreDePoints++;
}
};
```

Chaque objet `Point` possède ses propres exemplaires des membres `x` et `y` mais, quel que soit le nombre de points existants à un moment donné, il existe un seul exemplaire du membre `nombreDePoints`.

La visibilité et les droits d'accès des membres statiques sont régis par les mêmes règles que les membres ordinaires.

**Initialisation:** La ligne mentionnant `nombreDePoints` dans la classe `Point` est une simple « annonce ». Il faut encore créer et initialiser cette donnée membre (ce qui, pour une donnée membre non statique, est fait par le constructeur lors de la création de chaque objet). Cela se fait par une formule analogue à une définition de variable, écrite dans la portée globale, même s'il s'agit de membres privés :

```
int Point::nombreDePoints = 0;
```

(la ligne ci-dessus doit être écrite dans un fichier « .cpp », non dans un fichier « .h ») L'accès à un membre statique depuis une fonction membre de la même classe s'écrit comme l'accès à un membre ordinaire (voyez l'accès à nombreDePoints fait dans le constructeur Point ci-dessus).

L'accès à un membre statique depuis une fonction non membre peut se faire à travers un objet, n'importe lequel, de la classe :

```
Point a, b, c;  
...  
cout << a.nombreDePoints << "\n";
```

Mais, puisqu'il y a un seul exemplaire de chaque membre statique, l'accès peut s'écrire aussi indépendamment de tout objet, par une expression qui met bien en évidence l'aspect « variable de classe » des données membres statiques :

```
cout << Point::nombreDePoints << "\n";
```

### 3.12.2. Fonctions membres statiques

Une fonction membre statique n'est pas attachée à un objet. Par conséquent :

- elle ne dispose pas du pointeur this,
- de sa classe, elle ne peut référencer que les fonctions et les membres statiques.

Par exemple, voici la classe Point précédente, dans laquelle le membre nombreDePoints a été rendu privé pour en empêchant toute modification intempestive. Il faut donc fournir une fonction pour en consulter la valeur, nous l'avons appelée combien :

```
class Point {  
    int x, y;  
    static int nombreDePoints;  
public:  
    static int combien() {  
        return nombreDePoints;  
    }  
    Point(int a, int b) {  
        x = a; y = b;  
        nombreDePoints++;  
    }  
};
```

Pour afficher le nombre de points existants on devra maintenant écrire une expression comme (a étant de type Point) :

```
cout << a.combien() << "\n";
```

ou, encore mieux, une expression qui ne fait pas intervenir de point particulier :

```
cout << Point::combien() << "\n";
```

### 3.13. Amis;groupe A

#### 3.13.1. Fonctions amies

Une fonction amie d'une classe C est une fonction qui, sans être membre de cette classe, a le droit d'accéder à tous ses membres, aussi bien publics que privés.

Une fonction amie doit être déclarée ou définie dans la classe qui accorde le droit d'accès, précédée du mot réservé **friend**. Cette déclaration doit être écrite indifféremment parmi les membres publics ou parmi les membres privés :

Exemple:

```
class Tableau {  
  int *tab, taille;  
  friend void afficher (Tableau &);  
public:  
  Tableau(int nbrElements);  
  ...  
};
```

Et, plus loin, ou bien dans un autre fichier :

```
void afficher(Tableau &t) {  
  cout << '[';  
  for (int i = 0; i < t.taille; i++)  
    cout << t.tab[i];  
  cout << "]" ;  
}
```

**Note:** Notez cet effet de la qualification friend : bien que déclarée à l'intérieur de la classe Tableau, la fonction afficher n'est pas membre de cette classe ; en particulier, elle n'est pas attachée à un objet, et le pointeur this n'est pas défini dans cette fonction.

Il y a des cas où une fonction amie d'une classe est une fonction membre d'une autre classe. Imaginons, par exemple, que la fonction afficher doive écrire les éléments d'un objet Tableau dans un certain objet Fenêtre :

```
class Tableau;  
class Fenetre {  
  int num;  
public:  
  void afficher(Tableau );  
  ...  
};
```

```
class Tableau {  
  int *tab, taille;  
public:  
  friend void Fenetre::afficher(Tableau );  
  ...  
};
```

Maintenant, la fonction afficher est membre de la classe Fenetre et amie de la classe Tableau : elle a tous les droits sur les membres privés de ces deux classes, et son écriture s'en trouve facilitée :

```
void Fenetre::afficher(Tableau t) {
cout << "Numero de la fenetre :"<<num<< endl;

for (int i = 0; i < t.taille; i++)
cout << t.tab[i];
}
```

### 3.13.2. Classes amies

Une classe amie d'une classe C est une classe qui a le droit d'accéder à tous les membres de C. Une telle classe doit être déclarée dans la classe C (la classe qui accorde le droit d'accès), précédée du mot réservé **friend**, indifféremment parmi les membres privés ou parmi les membres publics de C.

Exemple:

```
Class Moi{
Public:
    Friend class MonAmi;
.....
};
```

Exemple 2: la classe Cercle est amie de la classe Point

```
class Point {
int x, y;
public:
friend class Cercle;
Point(int a, int b) {
x = a; y = b;
}
};
class Cercle {
Point centre;
int rayon
public:
cercle(int a, int b,int r):centre(a,b) {
rayon=r;
}
Void afficher(){cout<<centre.x<<centre.y<<rayon<<;}
};
```

Remarque:

La relation d'amitié n'est pas transitive, « les amis de mes amis *ne sont pas* mes amis ».

## 4. Surcharge des opérateurs

On l'a vu, le langage C++ propose beaucoup de nouveautés qui peuvent se révéler très utiles, si on arrive à s'en servir correctement (par exemple la surcharge de fonctions).

Une des nouveautés les plus étonnantes est la **surcharge des opérateurs**, que nous allons étudier dans ce chapitre.

C'est une technique qui permet de réaliser des opérations mathématiques intelligentes entre vos objets lorsque vous utilisez dans votre code des symboles comme +, -, \*, etc. Au final, votre code sera plus court et plus clair, et gagnera donc en lisibilité.

#### 4.1.Principe

Le principe est très simple. Supposons que vous ayez créé une classe pour stocker une durée (ex. : 4h23m), et que vous avez 2 objets de type Duree. Vous voulez les additionner entre eux pour connaître la durée totale

En temps normal, il faudrait créer une fonction "additionner" :

```
resultat = additionner(duree1, duree2);
```

La fonction additionner ferait ici la somme de duree1 et duree2 et stockerait ça dans resultat. Ça fonctionne, mais ce n'est pas franchement lisible. Ce que je vous propose dans ce chapitre, c'est d'être capable d'écrire ça :

```
resultat = duree1 + duree2;
```

#### 4.2.Les opérateurs arithmétiques (+, -, \*, /, %)

Pour être capable d'utiliser le symbole "+" entre 2 objets, il faut créer une méthode ayant précisément pour nom **operator+** qui a pour prototype :

```
Objet operator+(const Objet &monObjet);
```

#### Exemple:

```
class Duree
{
public:

    Duree(int heures = 0, int minutes = 0, int secondes = 0);
    Duree operator+(const Duree &d); // surcharge de l'opérateur +

private:

    int m_heures;
    int m_minutes;
    int m_secondes;
};
```

L'implémentation de la méthode operator+ est la suivante:

```
Duree Duree::operator+(const Duree &d)
{
    int heures = m_heures;
    int minutes = m_minutes;
    int secondes = m_secondes;

    // 1 : ajout des secondes
    secondes += d.m_secondes;
```

```

// Si le nombre de secondes dépasse 60, on rajoute des minutes et on met un nombre de secondes inférieur à 60
minutes += secondes / 60;
secondes %= 60;

// 2 : ajout des minutes
minutes += d.m_minutes;
// Si le nombre de minutes dépasse 60, on rajoute des heures et on met un nombre de minutes inférieur à 60
heures += minutes / 60;
minutes %= 60;

// 3 : ajout des heures
heures += d.m_heures;

// Création de l'objet resultat et retour
Duree resultat(heures, minutes, secondes);
return resultat;
}

```

Exemple d'utilisation:

```

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

using namespace std;

int main()
{
    Duree duree1(0, 10, 28), duree2(0, 15, 2);
    Duree resultat;

    duree1.afficher();
    cout << "+" << endl;
    duree2.afficher();

    resultat = duree1 + duree2; //equivalent à resultat = duree1.operator+(duree2);

    cout << "=" << endl;
    resultat.afficher();

    return 0;
}

```

Exemple 2:

```

int main()
{
    Duree duree1(0, 10, 28), duree2(0, 15, 2), duree3(1, 10, 2);
}

```

```
Duree resultat;
```

```
resultat = duree1 + duree2+duree3;// resultat = duree1.operator+(duree2.operator+(duree3));
```

Pour surcharger les autres opérateurs arithmétiques, rien de plus simple : créez une méthode dont le nom commence par operator suivi de l'opérateur en question. Cela donne donc :

- operator-
- operator\*
- operator/
- operator%

Pour notre classe Duree, il peut être intéressant de définir la soustraction (operator-). Je vous laisse le soin de le faire, en vous basant sur l'addition ça ne devrait pas être trop compliqué.

En revanche, les autres opérateurs ne servent a priori à rien : en effet, on ne multiplie pas des durées entre elles, et on les divise encore moins. Comme quoi, tous les opérateurs ne sont pas utiles à toutes les classes : ne définissez donc que ceux qui vous seront vraiment utiles.

Si multiplier une Duree par une Duree n'a pas de sens, en revanche on peut imaginer que l'on multiplie une Duree par un nombre entier. Ainsi, l'opération 2h25m50s \* 3 est envisageable. Attention à utiliser le bon prototype, en l'occurrence :

```
Duree operator*(int nombre);
```

### 4.3. Les opérateurs de comparaison (==, >, <, ...)

Ces opérateurs vont vous permettre de comparer des objets entre eux. Le plus utilisé d'entre eux est probablement l'opérateur d'égalité (==) qui permet de vérifier si 2 objets sont égaux. C'est à vous d'écrire le code de la méthode qui détermine si les objets sont identiques, l'ordinateur ne peut pas le deviner pour vous car il ne connaît pas la "logique" de vos objets.

Tous ces opérateurs de comparaison ont un point en commun particulier : ils renvoient un booléen (**bool**) et non un objet comme c'était le cas des autres opérateurs.

#### 4.3.1. L'opérateur ==

On va écrire l'implémentation de l'opérateur d'égalité :

```
bool Duree::operator==(const Duree &duree)
{
    if (m_heures == duree.m_heures && m_minutes == duree.m_minutes &&
m_secondes == duree.m_secondes)
        return true;
    else
        return false;
}
```

On compare à chaque fois un attribut de l'objet dans lequel on se trouve avec un attribut de l'objet auquel on se compare (les heures avec les heures, les minutes avec les minutes...). Si ces 3 valeurs sont identiques alors on peut considérer que les objets sont identiques et renvoyer true (vrai).

Dans le main, on peut faire un simple test de comparaison pour vérifier si on a fait les choses correctement :

```
int main()
{
    Duree duree1(0, 10, 28), duree2(0, 10, 28);

    if (duree1 == duree2)
        cout << "Les durees sont identiques";
    else
        cout << "Les durees sont differentes";

    return 0;
}
```

### 4.3.2. L'opérateur <

Voici une implémentation pour l'opérateur "est strictement inférieur à" (<) :

```
bool Duree::operator<(const Duree &duree)
{
    if (m_heures < duree.m_heures)
        return true;
    else if (m_heures == duree.m_heures && m_minutes < duree.m_minutes)
        return true;
    else if (m_heures == duree.m_heures && m_minutes == duree.m_minutes &&
m_secondes < duree.m_secondes)
        return true;
    else
        return false;
}
```

### 4.3.3. Les autres opérateurs de comparaison

On ne va pas les implémenter ici, ça surchargerait inutilement. Par contre, je vous invite à essayer de les implémenter pour notre classe Duree, ça fera un bon exercice d'algorithmique. Il reste notamment :

**operator>**  
**operator<=**  
**operator>=**  
**operator!=**

### 4.4. L'opérateur d'affectation (=)

Un des principaux pièges de ce chapitre vient de l'opérateur "=" que l'on peut lui aussi surcharger. C'est l'opérateur d'affectation, qui permet donc de donner une valeur à un objet.

En fait, le "piège" vient du fait que vous risquez de le confondre avec le constructeur de copie.

C'est pourquoi je vais insister plus précisément sur cet opérateur pour que vous voyiez bien la différence et ce à quoi il sert.

#### 4.4.1. Rappel : le constructeur de copie

Quand le constructeur de copie est-il appelé ? Vous vous en souvenez ? Il y a en fait plusieurs cas, les 3 principaux étant :

##### *1/ Lors de l'appel explicite au constructeur de copie*

Lorsque vous déclarez un objet et que vous indiquez en paramètre un autre objet, le constructeur de copie est appelé :

```
Objet monObjet;  
Objet copieObjet(monObjet); // Appel du constructeur de copie
```

##### *2/ Lors d'une affectation au moment de la déclaration*

C'est pareil, sauf qu'on utilise le signe "=" ce qui rend le code plus lisible :

```
Objet monObjet;  
Objet copieObjet = monObjet; // Appel du constructeur de copie
```

##### *3/ Lors d'un appel de fonction qui prend un objet en paramètre*

Si la fonction prend un objet (et non pas un pointeur ni une référence) en paramètre, l'objet est "copié" spécialement pour la fonction. Il y a appel du constructeur de copie avant le début de la fonction.

```
void maFonction (Objet copieObjet) // Appel du constructeur de  
copie  
{  
  
}
```

#### 4.4.2. Le rapport avec la surcharge de l'opérateur = ?

Si elle existe, la méthode **operator=** sera appelée dès qu'on essaie d'affecter une valeur à notre objet.

Par exemple, si à un moment dans le code on affecte à notre objet la valeur d'un autre objet :

```
Point A(2,3),B;  
  
B=A;
```

**Mais attends... C'est pas le constructeur de copie qui sera appelé là ?**

Non justement, c'est là qu'est le piège. Dans le cas n°2 vu plus haut, c'est lors de la déclaration de l'objet qu'on fait une affectation.

Dans ce cas, c'est le constructeur de copie qui est appelé.

En revanche, dans tout le reste du code, si on affecte une valeur à notre objet, c'est cette fois la méthode `operator=` qui sera appelée.

Donc en résumé :

- Lors de la déclaration (création) de notre objet, si vous utilisez le signe "=" pour lui affecter immédiatement la valeur d'un autre objet, c'est le **constructeur de copie** qui est appelé.

Exemple:

```
Objet copieObjet = monObjet; // Déclaration de l'objet
Point A=B;// appel du constructeur par défaut
```

- Après, à n'importe quel autre moment, si vous décidez d'affecter la valeur d'un autre objet à votre objet, c'est la méthode surchargeant l'opérateur = (**operator=**) qui sera appelée.

Exemple:

```
copieObjet = monObjet; // Affectation APRES la déclaration
Point A,B;
A=B;// appel de la methode operator=
```

#### 4.4.3. Implémentation de la méthode `operator=` pour la classe `Duree`

```
Duree Duree::operator=(const Duree &duree)
{
    m_heures = duree.m_heures;
    m_minutes = duree.m_minutes;
    m_secondes = duree.m_secondes;

    return *this;
}
```

Vous noterez que la méthode renvoie l'objet lui-même. En effet, souvenez-vous, `this` est un pointeur vers l'objet. Si on écrit `*this`, c'est donc l'objet lui-même que l'on renvoie. Cela permet de traiter le cas où on chaîne les affectations :

```
objet1 = objet2 = objet3;
```

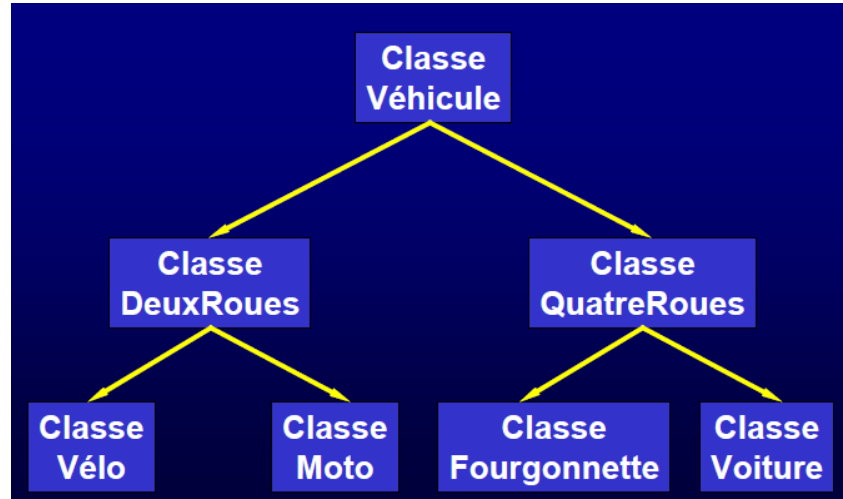
## 5. L'héritage

### 5.1. Définitions:

- L'héritage consiste, à partir d'une classe existante A, à définir une nouvelle classe B.
- La classe existante A est appelée classe mère ou classe de base, la nouvelle classe B est appelée classe fille ou classe dérivée.
- On dit que la classe B hérite ou dérive de la classe A.
- Une classe fille hérite automatiquement les attributs et les méthodes de sa classe mère, sans avoir à les réécrire.

- Une classe mère peut avoir plusieurs classes filles.
- On parle d'héritage simple, quand une classe fille hérite d'une seule classe mère.
- On parle d'héritage multiple, quand une classe fille hérite simultanément de plusieurs classes mères.

Exemples:



**5.2. Notation:**

La syntaxe de l'héritage est la suivante:

```
class nomclasse : Typedérivation classe1, Typedérivation classe2, ... typedérivation classeN
    {déclarations et définitions des membres spécifiques de la nouvelle classe};
```

où Typedérivation est un des mots-clés: **private, protected ou public**

**5.3. Exemple**

Voici un exemple de la classe Pixel qui hérite de la classe Point:

```
class Point {
int x, y;
public:
Point(int, int);
void afficher() { // affichage d'un Point
cout << '(' << x << ',' << y << ')';
}
...
};
class Pixel : public Point {
char *couleur;
public:
Pixel(int, int, char *);
void afficher() // affichage d'un Pixel
... };
```

## 5.4. Héritage et accessibilité des membres

### 5.4.1. Membres protégés

En plus des membres publics et privés, une classe C peut avoir des membres *protégés*. Annoncés par le mot clé **protected**, ils représentent une accessibilité intermédiaire car ils sont accessibles par les fonctions membres et amies de C et aussi par *les fonctions membres et amies des classes directement dérivées* de C.

Les membres protégés sont donc des membres qui ne font pas partie de l'interface de la classe, mais dont on a jugé que le droit d'accès serait nécessaire ou utile aux concepteurs des classes dérivées.

Modificateur d'accès	Visibilité dans les classes filles	Visibilité depuis l'extérieur
<b>private</b>	Non	Non
<b>protected</b>	Oui	Non
<b>public</b>	Oui	Oui

### 5.4.2. Héritage privé, protégé, public

En choisissant quel mot-clé indique la dérivation, parmi *private*, *protected* ou *public*, le programmeur détermine l'accessibilité dans la classe dérivée des membres de la classe de base. On notera que :

- cela concerne l'accessibilité des membres, non leur présence (les objets de la classe dérivée contiennent toujours tous les membres de toutes les classes de base),
- la dérivation ne peut jamais servir à *augmenter* l'accessibilité d'un membre.

Conséquence de ces deux points, les objets des classes dérivées ont généralement des membres inaccessibles :

Les membres privés de la classe de base sont présents dans les objets de la classe dérivée, mais il n'y a aucun moyen d'y faire référence.

#### ➤ Héritage privé

Syntaxe :

```
class classeDérivée : privateoptionnel classeDeBase { ... }
```

Le mot clé **private** est optionnel car l'héritage privé est l'héritage par défaut.

Statut dans la classe mère	Nouveau statut dans la classe fille	Accès par les méthodes de la classe fille	Accès par des entités externes
Public	Private	Oui	Non
Protected	Private	Oui	Non
Private	Private	Non	Non

Dans l'héritage privé, l'interface de la classe de base disparaît (l'ensemble des membres publics cessent d'être publics). Autrement dit, on utilise la classe de base pour réaliser l'implémentation de la classe dérivée, mais on s'oblige à écrire une nouvelle interface pour la classe dérivée.

### ➤ Héritage protected

**Syntaxe :**

```
class classeDérivée : protected classeDeBase { ... }
```

Cette forme d'héritage, moins souvent utilisée que les deux autres, est similaire à l'héritage privé (la classe de base fournit l'implémentation de la classe dérivée, non son interface) mais on considère ici que les détails de l'implémentation, c.-à-d. les membres publics et protégés de la classe de base, doivent rester accessibles aux concepteurs d'éventuelles classes dérivées de la classe dérivée.

Statut dans la classe mère	Nouveau statut dans la classe fille	Accès par les méthodes de la classe fille	Accès par des entités externes
Public	Protected	Oui	Non
Protected	Protected	Oui	Non
Private	Private	Non	Non

### ➤ Héritage public

**Syntaxe :**

```
class classeDérivée : public classeDeBase { ... }
```

Dans l'héritage public, l'interface est conservée : tous les éléments du comportement public de la classe de base font partie du comportement de la classe dérivée. C'est la forme d'héritage la plus fréquemment utilisée, car la plus utile et la plus facile à comprendre : dire que D dérive publiquement de B c'est dire que, vu de l'extérieur, tout D est une sorte de B, ou encore que *tout ce qu'on peut demander à un B on peut aussi le demander à un D*.

La plupart des exemples d'héritage que nous verrons seront des cas de dérivation publique.

Statut dans la classe mère	Nouveau statut dans la classe fille	Accès par les méthodes de la classe fille	Accès par des entités externes
Public	Public	Oui	Oui
Protected	Protected	Oui	Non
Private	Private	Non	Non

## 5.5. Redéfinition des fonctions membres

Lorsqu'un membre spécifique d'une classe dérivée a le même nom qu'un membre d'une classe de base, le premier masque le second, et cela quels que soient les rôles syntaxiques (constante, type, variable, fonction, etc.) de ces membres.

Si la classe D dérive publiquement de la classe B, tout D peut être vu comme une sorte de B, c'est-à-dire un B « amélioré » (augmenté des membres d'autres classes de base, ou de membres spécifiques) ; il est donc naturel qu'un D réponde aux requêtes qu'on peut soumettre à un B, et qu'il y réponde de façon améliorée.

D'où l'intérêt de la redéfinition : la classe dérivée donne des versions analogues mais enrichies des fonctions de la classe de base.

Souvent, cet enrichissement concerne le fait que les fonctions redéfinies accèdent à ce que la classe dérivée a de plus que la classe de base. Exemple : un Pixel est un point amélioré (c.-à-d. augmenté d'un membre supplémentaire, sa couleur). L'affichage d'un pixel consiste à l'afficher en tant que point, avec des informations additionnelles. Le code suivant reflète tout cela :

```
class Point {
int x, y;
public:
Point(int, int);
void afficher() { // affichage d'un Point
cout << '(' << x << ',' << y << ')';
}
...
};
class Pixel : public Point {
char *couleur;
public:
Pixel(int, int, char *);
void afficher() { // affichage d'un Pixel
cout << '[';
Point::afficher(); // affichage en tant que Point
cout << ';' << couleur << ']';
}
...
};
```

### Utilisation :

```
Point pt(2, 3);
Pixel px(4, 5, "rouge");
...
pt.afficher(); // affichage obtenu : (2,3)
px.afficher(); // affichage obtenu : [(4,5) ;rouge]
```

## 5.6. Construction et destruction des objets dérivés

Lors de la construction d'un objet, ses sous-objets hérités sont initialisés selon les constructeurs des classes de base correspondantes. S'il n'y a pas d'autre indication, il s'agit des constructeurs par défaut.

Si des arguments sont requis, il faut les signaler dans le constructeur de l'objet selon une syntaxe voisine de celle qui sert à l'initialisation des objets membres:

```
classe(paramètres)
: classeDeBase(paramètres), ... classeDeBase(paramètres) {
corps du constructeur
}
```

De la même manière, lors de la destruction d'un objet, les destructeurs des classes de base directes sont toujours appelés. Il n'y a rien à écrire, cet appel est toujours implicite.

**Exemple1:**

```
Pixel::pixel(int a,int b,char*c):Point(a,b),couleur(c){};
Ou bien
Pixel::pixel(int a,int b,char*c):Point(a,b){
Couleur= new char[strlen(c)+1];
Stcpy(couleur,c);}
```

**Exemple2:**

Prenons le cas de la classe **ObjetGraphique** telle qu'elle a été définie ci-dessous. Nous allons dériver deux nouvelles classes respectivement nommées **Ligne** et **Cercle**. Pour l'instant, nous ne spécifions que les constructeurs et les attributs. La classe **Cercle** rajoute un attribut nommé **rayon** alors que la classe **Ligne** rajout un **angle** et une **longueur**. Les codes résultants sont :

```
class ObjetGraphique
{
public:
    ObjetGraphique(int x, int y, int couleur=0, int epaisseur=0) :
        pointBase_(x,y),
        couleur_(couleur),
        epaisseur_(epaisseur)
    {}
protected:
    Point pointBase_;
    int couleur_;
    int epaisseur_;
};
```

```
class Ligne : public ObjetGraphique
{
public:
    Ligne (int x, int y, int longueur, double angle, int couleur=0, int epaisseur=0) :
        ObjetGraphique(x, y, couleur, epaisseur),
        longueur_(longueur),
        angle_(angle)
    {}
...
private:
    int longueur_;
```

```
double angle_;  
};
```

```
class Cercle : public ObjetGraphique  
{  
public:  
    Cercle (int x, int y, int rayon, int couleur=0, int epaisseur=0) :  
        ObjetGraphique(x, y, couleur, epaisseur),  
        rayon_(rayon)  
    {}  
    ...  
private:  
    int rayon_;  
}
```

Première chose que l'on remarque immédiatement : les attributs présents dans la classe **ObjetGraphique** n'ont pas à être redéclarés dans les classes dérivées. Etant à accès **protected**, ils seront transférés immédiatement vers les classes filles.

L'appel au constructeur de la classe mère se fait en première position dans la liste d'initialisation, avant l'appel des constructeurs des attributs. Cette stratégie est cohérente : on commence par initialiser les attributs en provenance de la classe mère avant d'initialiser les derniers introduits.

## 6. Le polymorphisme

### *6.1. Conversion standard vers une classe de base*

Si la classe D dérive publiquement de la classe B alors les membres de B sont membres de D. Autrement dit, les membres publics de B peuvent être atteints à travers un objet D. Ou encore : tous les services offerts par un B sont offerts par un D.

Par conséquent, là où un B est prévu, on doit pouvoir mettre un D. C'est la raison pour laquelle la conversion (explicite ou implicite) d'un objet de type D vers le type B, une classe de base accessible de D, est définie, et a le statut de conversion standard. Exemple :

```
class Point { // point « géométrique »  
int x, y;  
public:  
    Point(int, int);  
    ...  
};  
class Pixel : public Point { // pixel = point coloré  
char *couleur;  
public:  
    Pixel(int, int, char *);
```

```
...  
};
```

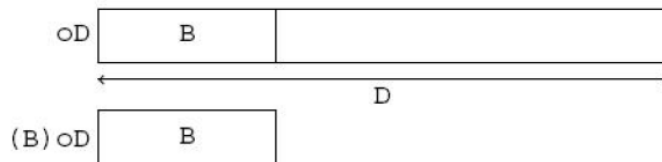
Utilisation :

```
Pixel px(1, 2, "rouge");  
Point pt = px; // un pixel a été mis là où un point était attendu, il y a conversion implicite
```

La conversion d'un objet de la classe dérivée vers la classe de base recouvre deux réalités très différentes :

1. Convertir un objet D vers le type B c'est lui enlever tous les membres qui ne font pas partie de B (les membres spécifiques et ceux hérités d'autres classes). Dans cette conversion il y a perte effective d'information :

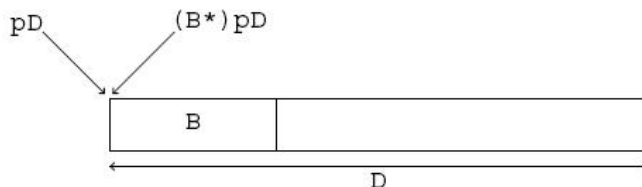
```
D oD;  
B oB=oD; (conversion implicite de l'objet oD en un objet de type B)
```



**Conversion de oD, un objet D, en un objet B**

2. Au contraire, convertir un D\* en un B\* (c'est-à-dire un pointeur sur D en un pointeur sur B) ne fait perdre aucune information. Pointé à travers une expression de type B\*, l'objet n'offre que les services de la classe B, mais il ne cesse pas d'être un D avec tous ses membres :

```
D *pD;  
B *pB;  
pB= pD;
```



**Conversion de pD, un pointeur sur un D, en un pointeur sur un B**

Dans le cas des pointeurs, c'est seulement une conversion de type qui est réalisée: l'objet en lui-même n'est pas affecté. Ce qui implique qu'un pointeur sur le type B peut pointer vers un objet de type D. c'est ce qu'on appelle le polymorphisme.

Par exemple, supposons posséder deux fonctions de prototypes :

```
void fon1(Point pt);  
void fon2(Point *pt);
```

et supposons qu'elles sont appelées de la manière suivante :

```
Pixel pix = Pixel(2, 3, "rouge")  
...  
fon1(pix);  
fon2(&pix);
```

Le statut de l'objet pix en tant que paramètre de ces deux fonctions est tout à fait différent. L'objet passé à fon1 comme argument est une version tronquée de pix, alors que le pointeur passé à fon2 est l'adresse de l'objet pix tout entier :

```
void fon1(Point pt) {  
La valeur de pt est un Point ; il n'a pas de couleur. Il y avait peut-être à l'origine un Pixel mais, ici, aucune conversion ne peut faire un Pixel à partir de pt.  
}
```

```
void fon2(Point *pt) {  
Il n'est rien arrivé à l'objet dont l'adresse a servi à initialiser pt lors de l'appel de cette fonction. Si on peut garantir que cet objet est un Pixel, l'expression suivante a un sens :  
((Pixel *) pt)->couleur ... }
```

Les références (pointeurs gérés de manière interne par le compilateur) doivent être considérées comme des pointeurs. Nous pouvons donc ajouter un troisième cas :

```
void fon3(Point &rf) {  
Il n'est rien arrivé à l'objet qui a servi à initialiser rf lors de l'appel de cette fonction. Si on peut garantir que cet objet est un Pixel, l'expression suivante a un sens :  
((Pixel &) rf).couleur ...  
}
```

## 6.2. Type statique, type dynamique, généralisation

Considérons la situation suivante :

```
class B {  
...  
};  
class D : public B {  
...  
};  
D unD;  
...  
B unB = unD;  
B *ptrB = &unD;  
B &refB = unD;
```

Les trois affectations ci-dessus sont légitimes ; elles font jouer la conversion standard d'une classe vers une classe de base. Le type de unB ne soulève aucune question (c'est un B), mais les choses sont moins claires pour les types de ptrB et refB. Ainsi ptrB, par exemple, pointe-t-il un B ou un D ?

- on dit que le *type statique* de \*ptrB (ce que ptrB pointe) et de refB est B, car c'est ainsi que ptrB et refB ont été déclarées ;
- on dit que le *type dynamique* de \*ptrB et de refB est D, car tel est le type des valeurs effectives de ces variables.

- Le type statique d'une expression découle de l'analyse du texte du programme ; il est connu à la compilation.

- Le type dynamique, au contraire, est déterminé par la valeur courante de l'expression, il peut changer durant l'exécution du programme.

La notion de type dynamique va nous amener assez loin. Pour commencer, remarquons ceci : grâce aux conversions implicites de pointeurs et références vers des pointeurs et références sur des classes de base, tout objet peut être momentanément tenu pour plus général qu'il n'est. Cela permet les traitements génériques, comme dans l'exemple suivant, qui présente un ensemble de classes conçues pour gérer le stock d'un supermarché

```
class Article {  
protected:char *denom; // denomination  
double prix; // prix HT  
int quant;// quantité  
Article(char *d, int q, double p)  
: denom(d), quant(q), prix(p) { }  
void afficher();  
...  
};
```

```
class Habillement : public Article {  
int taille;  
Habillement(char *d, int q, double p, int t)  
: Article(d, q, p), taille(t) { }  
...  
};
```

```
class Informatique : public Article {  
char garantie;  
Informatique (char *d, int q, double p, char g)  
: Article(d, q, p), garantie(g) { }  
...  
};
```

Définissons une structure (un tableau de pointeurs) pour mémoriser *tous* les articles du stock :

```
Article *stock[maxStock];
```

```
int n= 0;  
stock[n++] = new Habillement("Tenue gendarme fluo rose", 250, 200.0, 52);  
stock[n++] = new Informatique("Pentium III 800 Mhz Diesel HDI", 500, 1000.0,  
'C');  
stock[n++] = new Habillement("pantalons", 250, 200.0, 52);
```

#### **Exemple d'exécution:**

```
double valeurStock = 0;  
for (int i = 0; i < n; i++)  
valeurStock += stock[i]->quant * stock[i]->prix;
```

Ou bien:

```
for (int i = 0; i < n; i++)
stock[i]->afficher();
```

Dans cet exemple nous avons utilisé la généralisation des pointeurs pour accéder à des membres dont les objets héritent sans les modifier : les notions de quantité disponible et de prix unitaire sont les mêmes pour toutes les sortes d'articles du stock. La question devient beaucoup plus intéressante lorsqu'on accède à des membres de la classe de base qui sont redéfinis dans les classes dérivées. Voici un autre exemple, classique : la classe Figure est destinée à représenter les éléments communs à tout un ensemble de figures géométriques, telles que des triangles, des ellipses, des rectangles, etc. :

```
class Figure {
...
};
class Triangle : public Figure {
...
};
class Ellipse : public Figure {
...
};

Figure *image[maxFigure];
int n = 0;
image[n++] = new Triangle(...);
image[n++] = new Figure(...);
image[n++] = new Ellipse(...);
...
for (int i = 0; i < n; i++)
image[i]->seDessiner();
```

Si on suppose que la fonction seDessiner (qui doit nécessairement être déclarée dans la classe Figure, sinon la ligne ci-dessus serait trouvée incorrecte à la compilation) est redéfinie dans les classes Triangle, Ellipse, etc., il est intéressant de se demander quelle est la fonction qui sera effectivement appelée par l'expression: image[i]->seDessiner() écrite ci-dessus.

Par défaut, la détermination du membre accédé à travers une expression comme la précédente se fait durant la compilation, c'est-à-dire d'après le *type statique* du pointeur ou de la référence. Ainsi, dans l'expression image[i]->seDessiner(); c'est la fonction seDessiner de la classe Figure qui est appelée. Cette réalité décevante obéit à des considérations d'efficacité et de compatibilité avec le langage C; elle semble limiter considérablement l'intérêt de la généralisation des pointeurs...

...heureusement, la section suivante présente un comportement beaucoup plus intéressant ! c'est l'utilisation des classes virtuelles.

## 7. Fonctions virtuelles

Soit f une fonction membre d'une classe C. Si les conditions suivantes sont réunies :

- f est redéfinie dans des classes dérivées (directement ou indirectement) de C,

- f est souvent appelée à travers des pointeurs ou des références sur des objets de C ou de classes dérivées de C, alors f mérite d'être une *fonction virtuelle*. On exprime cela en faisant précéder sa déclaration du mot réservé **virtual**.

L'important service obtenu est celui-ci : si f est appelée à travers un pointeur ou une référence sur un objet de C, le choix de la fonction effectivement activée, parmi les diverses redéfinitions de f, se fera d'après le type dynamique de cet objet.

Une classe possédant des fonctions virtuelles est dite classe *polymorphe*. La qualification virtual devant la redéfinition d'une fonction virtuelle est facultative : les redéfinitions d'une fonction virtuelle sont virtuelles d'office.

Exemple :

```
class Figure {
public:
virtual void seDessiner() {
fonction passe-partout, probablement sans grand intérêt
}
...
};
class Triangle : public Figure {
public:
void seDessiner() {
implémentation du tracé d'un triangle
}
...
};
class Ellipse : public Figure {
public:
void seDessiner() {
implémentation du tracé d'une ellipse
...
};
Figure *image[maxFigure];
int n = 0;
image[n++] = new Triangle(...);
image[n++] = new Figure(...);
image[n++] = new Ellipse(...);
...
for (int i = 0; i < N ; i++)
image[i]->seDessiner();
```

Maintenant, l'appel `image[i]->seDessiner()` active la fonction redéfinie dans la classe de l'objet effectivement pointé par `image[i]`. Cela constitue, bien entendu, le comportement intéressant, celui qu'on recherche presque toujours. On notera que cet appel est légal parce que la fonction `seDessiner` a été déclarée une première fois dans la classe qui correspond au type statique de `image[i]`.

## 8. Les classes abstraites

Dans le paragraphe précédent, nous avons déclaré des fonctions virtuelles. C++ autorise la déclaration des fonctions virtuelles pures dont la définition n'est pas donnée.

Exemple:

```
class Figure {
public:
virtual void seDessiner() = 0; // Une fonction virtuelle pure
...
};
```

Une fonction virtuelle pure ne peut pas, à ce niveau (dans la classe Figure), posséder une implémentation. En plus, le programmeur, est obligé de définir cette implémentation dans une classe dérivée ultérieure, Une fonction virtuelle pure reste virtuelle pure dans les classes dérivées, aussi longtemps qu'elle ne fait pas l'objet d'une redéfinition (autre que « = 0 »).

Pour le compilateur, une *classe abstraite* est une classe qui a des fonctions virtuelles pures. Tenter de créer des objets d'une classe abstraite est une erreur, qu'il signale :

```
class Figure {
public:
virtual void seDessiner() = 0;
...
};
class Rectangle : public Figure {
int x1, y1, x2, y2;
public:
void seDessiner() {
trait(x1, y1, x2, y1);
trait(x2, y1, x2, y2);
trait(x2, y2, x1, y2);
trait(x1, y2, x1, y1);
}
...
};
```

Utilisation :

```
Figure f; // ERREUR : Création d'une instance de la
// classe abstraite Figure
Figure *pt; // Oui : c'est un pointeur
pt = new Figure; // ERREUR : Création d'une instance de la
// classe abstraite Figure
Rectangle r; // Oui : la classe Rectangle n'est pas abstraite
pt = new Rectangle; // Oui
```

## 9. La classe string de la librairie standard (STL)

### 9.1. La classe string en C++

La classe string fait partie de la bibliothèque STL. Pour utiliser la classe string, il faut inclure le fichier string.

```
#include <string>
```

Les strings sont des chaînes de caractères avec une gestion de la mémoire et des méthodes de gestion intégrés. Voici les principales méthodes de cette classe.

### 9.2. Les constructeurs

`string();`

`string(const char *s);`  
A partir d'un char \*.

`string(const char *s, size_t n);`  
A partir d'un char \* avec au plus n caractères.

`string(size_t n, char c);`  
Répète le caractère 'c' n fois.

`string(const string &str, size_t pos=0, size_t n=-1);`  
A partir d'une sous chaîne, avec n caractères à partir de pos.

#### **Exemples:**

```
string s1; // Chaîne vide s1
string s2("Clermont-Ferrand") // Initialisation avec un char *
char tab[]="Auvergne";
string s3(tab,4); // s3=="Auve"
string s4(s2); // constructeur par copie
string s5(s2,8); // s5=="Ferrand"
string s6(s2,16); // s6=="Clermont-Ferrand"
string s7(s2,18); // s7=="Clermont-Ferrand"
string s8(4,'c'); // s8=="cccc"
```

### 9.3. Les méthodes

`const char *c_str() const;`  
Conversion du string vers un char\*.

`size_t length() const;`  
Renvoie la longueur de la chaîne.

`bool empty() const;`  
Indique si la chaîne est vide. (Vrai = vide).

`string &assign(const string &str, size_t pos, size_t n);`

string &assign(size\_t rep, char c);  
Affecte une sous chaîne ou une succession du même caractère.

size\_t copy(char \*s, size\_t n, size\_t pos=0);  
Copie n caractères à partir de pos de s.

string substr(size\_t pos=0, size\_t n=-1) const;  
Extrait une sous chaîne à partir de pos avec au plus n caractères.

int compare(size\_t po1, size\_t n1, const string &str, size\_t po2, size\_t n2) const;  
Compare la sous chaîne de n1 caractères à partir de po1 avec une sous chaîne str composé de n2 caractères à partir de pos2. La valeur retournée est: <0, =0 ou >0. Respectivement: La première sschaîne est avant la seconde, la première est égale à la seconde sschaîne et la première sschaîne est après la seconde.

void swap(string &str);  
Permute.

string &append(const string &str, size\_t pos, size\_t n);  
string &append(size\_t rep, char c);  
Ajoute une sschaîne ou une succession de caractères.

string &insert(size\_t pos1, const string &str, size\_t pos2, size\_t n);  
string &insert(size\_t pos, size\_t rep, char c);  
Insère une sschaîne dans une chaîne.

string &replace(size\_t pos1, size\_t n1, const string &str, size\_t pos2, size\_t n2);  
string &replace(size\_t pos, size\_t n1, size\_t n2, char c);  
Remplace une sschaîne par une autre sschaîne.

string &erase(size\_t pos, size\_t n);  
Efface n caractères de la chaîne à partir de pos.

#### ***9.4. Les méthodes de recherche:***

Les méthodes de recherches renvoie -1 si l'occurrence n'est pas trouvée.

size\_t find(const string &str, size\_t pos=0) const;  
size\_t find(char c, size\_t pos=0) const;  
size\_t find\_first\_of(const string &str, size\_t pos=0) const;  
size\_t find\_first\_of(char c, size\_t pos=0) const;  
Recherche de la première occurrence de str ou c à partir de pos.  
Renvoie le rang de l'occurrence.

size\_t find\_first\_not\_of(const string &str, size\_t pos=0) const;  
size\_t find\_first\_not\_of(char c, size\_t pos=0) const;  
Recherche de la première occurrence d'un caractère NON SPECIFIE de str ou c à partir de pos. Renvoie le rang de l'occurrence.

size\_t **rfind**(const string &str, size\_t pos=-1) const;  
size\_t **rfind**(char c, size\_t pos=-1) const;  
size\_t **find\_last\_of**(const string &str, size\_t pos=-1) const;  
size\_t **find\_last\_of**(char c, size\_t pos=-1) const;  
Recherche de la dernière occurrence de str ou c à partir de pos.  
Renvoie le rang de l'occurrence.

size\_t **find\_last\_not\_of**(const string &str, size\_t pos=-1) const;  
size\_t **find\_last\_not\_of**(char c, size\_t pos=-1) const;  
Recherche de la dernière occurrence d'un caractère NON SPECIFIE de str ou c à partir de pos. Renvoie le rang de l'occurrence.

**Exemples:**

```
#include <cstdlib>
#include <iostream>
#include <string>

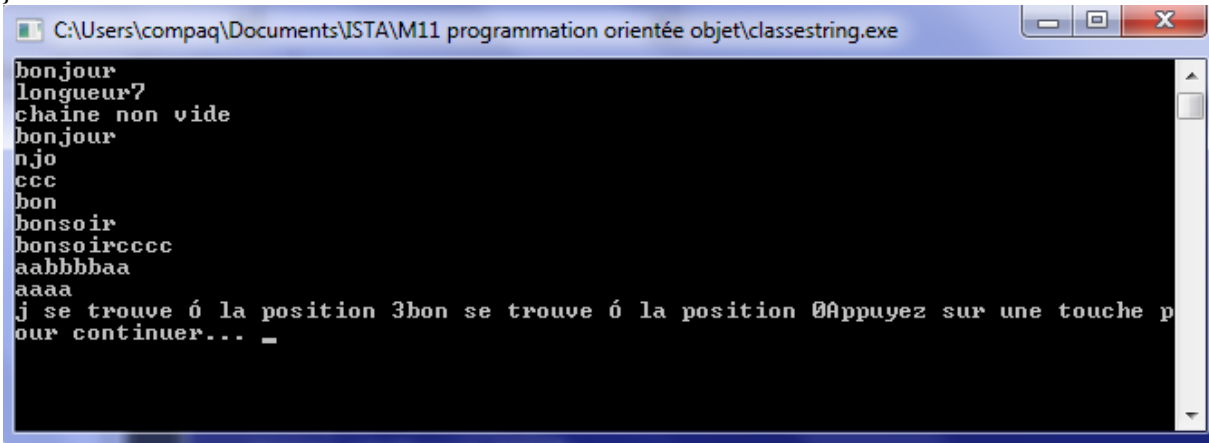
using namespace std;

int main(int argc, char *argv[])
{
    string s="bonjour";
    cout<<s<<endl;
    cout<<"longueur"<<s.length()<<endl;
    if(s.empty()) cout<<"chaine vide"<<endl; else cout<<"chaine non vide"<<endl;
    const char *c=s.c_str();

    cout<<c<<endl;
    string s2=s.assign(s,2,3);
    cout<<s2<<endl;
    string s3;
    s3.assign(3,'c');
    cout<<s3<<endl;
    s="bonjour";
    string s4=s.substr(0,3);
    cout<<s4<<endl;
    string s5="bonsoir";
    string s6="bonne nuit";
    s6.swap(s5);// permute s6 avec s5

    cout<<s6<<endl;
    s6.append("cccc");
    cout<<s6<<endl;
    string s8="aaaa";
    s8.insert(2,"bbbb");
    cout<<s8<<endl;
    s8.erase(2,4);
    cout<<s8<<endl;
    string s9="bonjour";
```

```
cout<<"j se trouve à la position"<<" "<< s9.find("j");  
cout<<"bon se trouve à la position"<<" "<< s9.find("bon");  
  
system("PAUSE");  
return EXIT_SUCCESS;  
}
```



```
C:\Users\compaq\Documents\ISTA\M11 programmation orientée objet\classestring.exe  
bonjour  
longueur?  
chaine non vide  
bonjour  
njo  
ccc  
bon  
bonsoir  
bonsoircccc  
aabbbaa  
aaaa  
j se trouve à la position 3bon se trouve à la position 0Appuyez sur une touche p  
our continuer... _
```

## 10. La classe vector de la librairie standard (STL)

### 10.1. La classe vector en C++

La classe vector fait partie de la bibliothèque STL. Pour utiliser la classe vector, il faut inclure le fichier vector.

```
#include <vector>
```

Les vectors sont des tableaux avec une gestion de la mémoire et des méthodes de gestion intégrés. Voici les principales méthodes de cette classe.

#### Exemples:

```
#include <vector>
#include <string>
using namespace std;

vector<int> monTableau; // Crée un vector d'entiers vide.

vector<string> monTableau2; // Crée un vector de strings vide.

vector<Point> monTableau3; // Crée un vector pouvant contenir des Points.
```

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

vector<double> monTableau(10); // Crée un tableau de 10 doubles.

vector<int> monTableau2(5); // Crée un tableau de 5 entiers.

vector<Point> tab(10); // Création d'un tableau de 10 Points en utilisant le constructeur par défaut de la classe Point.
```

```
vector<double> monTableau(10,3.14); // Crée un tableau de 10 doubles dont la valeur est 3.14.

vector<int> monTableau2(5,4); // Crée un tableau de 5 entiers dont la valeur est 4.
```

### 10.2. Les constructeurs

```
vector();
```

```
explicit vector(size_t n);
Vecteur de n éléments.
```

`vector(const vector<T>& x);`  
Constructeur par copie.

`vector(const_iterator first, const_iterator last);`  
Construction par extraction d'un sous-vecteur.

### 10.3. *Les accesseurs*

iterator `begin()`;  
const\_iterator `begin() const`;  
Itérateur sur le premier élément.

iterator `end()`;  
const\_iterator `end() const`;  
Itérateur au-delà du dernier élément.

reverse\_iterator `rbegin()`;  
const\_reverse\_iterator `rbegin() const`;  
Itérateur rétrograde sur le dernier élément.

reverse\_iterator `rend()`;  
const\_reverse\_iterator `rend() const`;  
Itérateur rétrograde au-delà du premier élément.

reference `front()`;  
const\_reference `front() const`;  
Référence au premier élément.

reference `back()`;  
const\_reference `back() const`;  
Référence au dernier élément.

### 10.4. *Les méthodes*

size\_t `size() const`;  
Nombre d'élément.

void `push_back(const T& x);`  
Ajoute l'élément x en fin de vecteur.

void `pop_back()`;  
Supprime le dernier élément du vecteur.

iterator `insert(iterator position, const T& x);`  
Insert l'élément x devant l'élément désigné par position.

iterator `insert(iterator position);`  
Insert un élément construit par défaut devant l'élément désigné par position.

void `insert(iterator position, const_iterator first, const_iterator last);`

Insert une séquence d'éléments désigné par first et last devant l'élément désigné par position.

iterator **erase**(iterator position);  
Efface l'élément à la position donnée.

iterator **erase**(iterator first, iterator last);  
Efface les éléments entre first et last exclus.

void **clear**();  
Efface tous les éléments du vecteur.

### 10.5. *Les méthodes spécifiques*

T **back**()  
Valeur du dernier élément .

T **front**()  
Valeur du premier élément.

#### Exemples:

```
#include <cstdlib>
#include <iostream>
#include <vector>

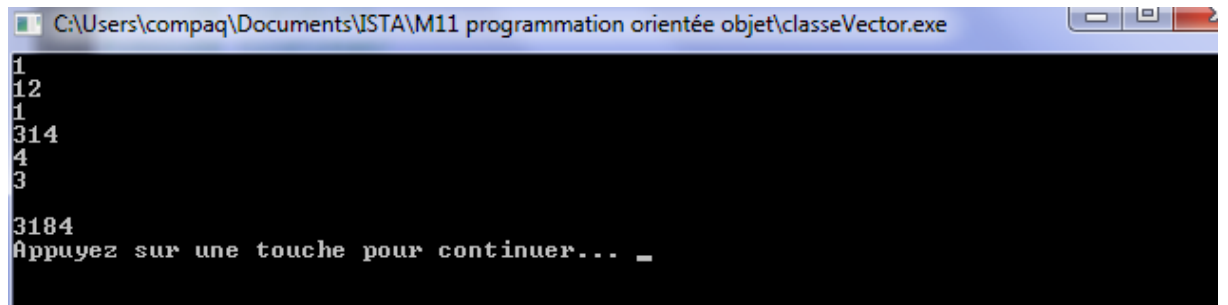
using namespace std;

int main(int argc, char *argv[])
{
    vector<int> t;
    vector<int>::iterator it;
    t.push_back(1);
    t.push_back(2);
    cout<<t.at(0)<<endl;
    for(int i=0;i<t.size();i++)
        cout<<t[i];
    cout<<endl;
    t.pop_back();
    for(int i=0;i<t.size();i++)
        cout<<t[i];

    // t.insert(it,0);
    t.insert(t.begin(),3);
    t.insert(t.end(),4);
    cout<<endl;
    for(int i=0;i<t.size();i++)
        cout<<t[i];
    cout<<endl;
    cout<<t.back()<<endl;
    cout<<t.front()<<endl;
```

```
cout<<endl;
it=t.begin();
t.insert(it+2,8);
for(int i=0;i<t.size();i++)
    cout<<t[i];

cout<<endl;
system("PAUSE");
return EXIT_SUCCESS;}
```



```
C:\Users\compaq\Documents\ISTA\M11 programmation orientée objet\classeVector.exe
1
12
1
314
4
3
3184
Appuyez sur une touche pour continuer... _
```